ESTRUTURA DE DADOS

UNIDADE 2 – ORDENAÇÃO DE DADOS

Ana Paula dos Santos Braatz Vieira

Introdução

A estrutura de dados se trata de um método utilizado para organizar os dados de forma adequada, para o computador processá-los de maneira eficiente, além de também definirem os métodos de acesso e o modo como serão manipulados. No entanto, como organizamos os dados?

Nas aplicações, encontramos problemas que, muitas vezes, requerem ordenação para facilitar o acesso e o tempo, como colocar os números em ordem crescente ou decrescente e os nomes em ordem alfabética. Assim, para organizar os elementos de forma eficiente, utilizamos métodos, como os algoritmos de ordenação.

Atualmente, existem diversos algoritmos de ordenação, por isso, o conceito deles e sua aplicação são essenciais para o desenvolvimento de sistemas. Dessa forma, podemos escolher o melhor método, de acordo com a necessidade, melhorando o desempenho das aplicações.

Nesse contexto, como decidimos qual método de ordenação aplicar? Como entendemos a complexidade de um algoritmo e como devemos analisá-la?

A escolha da estrutura adequada para cada sistema depende diretamente do conhecimento dos algoritmos e do que o sistema necessita. Para escolher um método de ordenação que seja eficiente, temos que entender a complexidade dos algoritmos, analisando o desempenho do que está sendo desenvolvido. A complexidade é utilizada para entender e avaliar se um algoritmo é eficiente, considerando o espaço de memória utilizado e o tempo para resolver os problemas. Assim, analisa-se o algoritmo com hipóteses comuns e inusitadas, a fim de descobrir qual será o desempenho nos melhores e piores casos.

Ao longo desta unidade, vamos nos aprofundar nesses pontos de estudos. Acompanhe!

2.1 Algoritmos

Algoritmos de ordenação são exemplos de algoritmos criados para resolverem problemas de forma mais rápida, com o auxílio de um computador. Existem várias técnicas que podem ser utilizadas, sendo que cada uma tem procedimentos distintos para resolver a mesma tarefa. Algumas são mais adequadas para serem usadas com uma base de dados pequena, enquanto outras para uma base de dados maior, assim como umas são mais simples que outras. O fato é que todas ilustram regras básicas para a manipulação de estruturas de dados.

De acordo com Ziviani (2012, p. 111), ordenar "[...] corresponde ao processo de rearranjar um conjunto de objetos em ordem ascendente ou descendente. O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado".

VOCÊ SABIA?

Existe uma ONG educacional chamada Khan Academy, criada em 2008, com a missão de oferecer conteúdo sobre diversas matérias. Entre elas, há alguns métodos de ordenação e a descrição da notação de complexidade de algoritmos. Para saber mais sobre ela, podemos acessar o *site* por meio do *link*: https://pt.khanacademy.org/computing/computer-science/algorithms#quick-sort.

Para ordenar os dados, utilizamos métodos de ordenação, que podem ser classificados em ordenação interna e ordenação externa. Clique nas abas para conhecer.

Ordenação interna

Na ordenação interna, os elementos a serem ordenados cabem na memória principal, sendo que qualquer registro pode ser imediatamente acessado.

Ordenação externa

Na ordenação externa, os elementos a serem ordenados não cabem na memória principal e os registros são acessados sequencialmente ou em grandes blocos.

CASO

Imagine que temos uma agenda telefônica e desejamos buscar algum contato (pode ser uma agenda *on-line* ou uma lista telefônica). É claro que a nossa busca se tornaria mais rápida se os nomes presentes na agenda estivessem em ordem alfabética, não é? Nessa perspectiva, uma lista telefônica possui pelo menos dois atributos fundamentais, como nome e número do telefone. Agora, pense, por exemplo, que temos uma agenda telefônica com os contatos de 100 pessoas, mas precisamos buscar o número do João. Se a agenda estiver ordenada pelo nome em ordem alfabética, a busca será mais fácil, certo?

E se essa agenda estiver organizada em ordem crescente pelos números de telefone, será que conseguiríamos achar o número do João de forma rápida? Sabemos que a resposta é "não", pois, se estamos procurando o número do João, quer dizer que não sabemos qual é, por isso, uma lista com os números em ordem crescente não nos ajudaria.

O exemplo da agenda telefônica é trivial, porém é importante ressaltar que, quando desejamos fazer uma ordenação com os dados, precisamos analisar quais dados realmente necessitam ser ordenados ou qual o atributo que irá auxiliar para que possamos recuperar as informações de forma mais rápida, ou seja, mais eficiente.

Assim, a partir de agora, vamos aprender os conceitos dos principais métodos de ordenação e um pouco sobre a complexidade de cada algoritmo.

2.1.1 Complexidade de algoritmos

Uma característica importante de qualquer algoritmo é o tempo de execução. É possível determiná-lo por meio de métodos baseados na observação da execução. Assim, podemos obter uma ordem de grandeza do tempo de execução por meio de expressões matemáticas que traduzam o comportamento dos algoritmos. De acordo com Szwarcfiter e Markenzon (2010), a tarefa de obter uma expressão matemática para avaliar o tempo de um algoritmo, em geral, não é simples, mesmo considerando uma expressão aproximada.

Para gerar essas expressões, é necessário o uso de funções de complexidade *f*, em que $f(n)$ é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n. Nesse caso, então f será chamado de **função de complexidade de tempo**. Caso seja utilizado para calcular a quantidade de memória usada pelo algoritmo, f será chamado de **função de complexidade de espaço** (ZIVIANI, 2012).

VOCÊ QUER LER?

Existem funções que auxiliam na avaliação do tempo de um algoritmo. A função mais simples é a constante ($f(n) = c$). Depois, temos a função logaritmo ($log\ n$), a função linear ($f(n) = n$), a função quadrática ($f(n) = n^2$), a função cúbica ($f(n) = n^3$), entre outras. Elas podem ser estudadas com mais intensidade no livro "Estrutura de dados e algoritmos em Java", dos autores Roberto Tamassia e Michel T. Goodrich. Na obra, os autores destacam as funções, fazem comparações das taxas de crescimento e relatam sobre a análise de algoritmos utilizando as funções. Vale a pena tirar um tempinho para a leitura!

Temos, então, que a complexidade de um algoritmo pode ser calculada por meio da complexidade temporal e da espacial. Na primeira, analisamos o tempo que as instruções levam para serem executadas (a partir das funções) e, na sequência, o espaço na memória que o algoritmo usa. O esforço realizado pelo algoritmo também deve ser observado, calculado a partir da quantidade de vezes que a operação fundamental (método de comparação) é executada.

Szwarcfiter e Markenzon (2010) destacam que a complexidade de tempo pode ser descrita com o esquema: seja A um algoritmo e $\{E1, \ldots, Em\}$ o conjunto de todas as entradas possíveis de A, devemos denotar por t^i o número de passos efetuados por A, quando a entrada for E^i. Define-se, então, que:

- complexidade do pior caso: $\max\ E^i \in E\ \{t^i\}$;
- complexidade do melhor caso: $\min\ E^i \in E\ \{t^i\}$;
- complexidade do caso médio: $\Sigma\ 1 \leq i \leq m\ p^i\ t^i$ em que p^i é a probabilidade de ocorrência da entrada E^i.

A complexidade de tempo de pior caso é uma das mais importantes, pois corresponde ao número de passos que levará o tempo em que o algoritmo efetua seu pior caso de execução, isto é, nos casos mais extremos. A complexidade de melhor caso, por outro lado, é de uso menos frequente, empregado em situações específicas, pois mostra o menor tempo de execução de um sistema. Por fim, a complexidade de caso médio, apesar de importante, é menos utilizada que a de pior caso, uma vez que exige o prévio conhecimento da distribuição de probabilidades das diferentes entradas do algoritmo, sendo a complexidade mais difícil de determinar.

Agora que conhecemos um pouco sobre a complexidade dos algoritmos e como é definida, vamos aprender sobre alguns métodos de ordenação. Na sequência, iremos estudar a respeito do método *Bubble-Sort*. Vejamos!

2.2 Método Bubble-Sort

O método de ordenação bolha ou *Bubble-Sort* é simples e um dos mais difundidos, geralmente utilizado para ordenação de uma base de dados pequena. Sua interação se limita em percorrer diversas vezes o *array* do início ao fim, sem interrupção, trocando a posição de dois elementos sempre que estiverem desordenados. Caso queiramos organizar os dados em ordem crescente, podemos ordenar do menor (início do *array*) para o maior (fim do *array*). No final da primeira interação, ao percorrer todo o *array*, podemos assumir que o último elemento já está na sua posição final.

A seguir, temos a ordenação crescente de um *array* unidimensional com o algoritmo de ordenação *Bubble-Sort*.

Figura 1 - Funcionamento da ordenação crescente de um array unidimensional
Fonte: Elaborada pela autora, 2019.

Considere o exemplo anterior como *array* = {5, 3, 8, 2, 6}.
Primeiramente, são analisados os dois elementos iniciais do *array*, que seriam o *array* [0] e o *array* [1]. Depois de analisar e ver qual dos elementos é o maior, o algoritmo vai trocar os elementos caso o segundo seja menor que o primeiro. Temos, então *array* = {3, 5, 8, 2, 6}. Assim, o algoritmo alterou os números 5 e 3, colocando o 3 no *array* [0] e o 5 no *array* [1].
Na próxima interação, o algoritmo irá comparar o elemento do *array* [1] com o *array* [2] e verificar qual deles é menor. No caso, o 5 é menor que o 8, então, o algoritmo não altera nada, ficando em *array* = {3, 5, 8, 2, 6}.
Depois, o algoritmo vai analisar o *array* [2] com o *array* [3]. Como o número 8 é maior que o 2, haverá a troca dos elementos reordenando o *array* em *array* = {3, 5, 2, 8, 6}. A interação na sequência irá analisar o *array* [3] e o *array* [4]. Como 8 é maior que 6, também haverá a troca dos elementos, reordenando novamente o *array* em *array* = {3, 5, 2, 6, 8}.
Assim, foi realizada a primeira interação por completo pelo algoritmo. Com ela, podemos verificar que o maior número ficou na última posição do *array*. Essas interações serão repetidas até que todo o *array* seja ordenado.
A seguir, vamos demonstrar a ordenação utilizando apenas o exemplo, mostrando quando há e quando não há a alteração dos elementos. Na tabela, temos as operações e, caso seja verdade, vamos realizar a troca dos elementos e reordenar o vetor. Em caso falso, não vamos fazer nada no *array*. Vejamos!

Array	Comparação	Verdade (troca)	Falso (mantêm)
{**5**, **3**, 8, 2, 6}	5 > 3 ?	x	
{3, **5**, **8**, 2, 6}	5 > 8 ?		x
{3, 5, **8**, **2**, 6}	8 > 2 ?	x	
{3, 5, 2, **8**, **6**}	8 > 6 ?	x	
{**3**, **5**, 2, 6, 8}	3 > 5 ?		x
{3, **5**, **2**, 6, 8}	5 > 2 ?	x	
{3, 2, **5**, **6**, 8}	5 > 6 ?		x
{3, 2, 5, **6**, **8**}	6 > 8 ?		x
{**3**, **2**, 5, 6, 8}	3 > 2 ?	x	
{2, **3**, **5**, 6, 8}	3 > 5 ?		x
{2, 3, **5**, **6**, 8}	5 > 6 ?		x
{2, 3, 5, **6**, **8**}	6 > 8 ?		x

Tabela 1 - Ordenação dos números contidos no array pelo método Bubble-Sort
Fonte: Elaborada pela autora, 2019.

Ao terminarmos de percorrer o *array*, podemos perceber que ele está organizado em ordem crescente. Note que o algoritmo fez 12 comparações e alterou os dados cinco vezes, pois ele só altera quando a operação (comparação) for verdade.

Dado o *array* = [9, 5, 1, 3, 4, 7, 2], ordene os números em ordem crescente com o *Bubble-Sort* e, depois, descreva quantas comparações e trocas foram realizadas.

Agora que já sabemos como o nosso método de ordenação funciona, veja na figura a seguir o pseudocódigo do algoritmo.

```
for(i = 0; i < tamanho; i++){
    for(int j = 0; j< tamanho -1; j++){
        if(array [j] > array [j + 1]){
            aux = array [j];//aux = variável auxiliar que armazena o valor do array
            array [j] = array [j+1];
            array [j+1] = aux;
        }
    }
}
```

Figura 2 - Princípios básicos do método Bubble-Sort
Fonte: Elaborada pela autora, 2019.

Na figura a seguir, temos uma das possíveis formas de desenvolver a ordenação do algoritmo *Bubble-Sort*, com a implementação do código na linguagem Java. A parte em preto se trata da impressão do console.

```java
public class MyBubbleSort {
    public static void main(String args[]){
        int[] array = {5,3,8,2,6};
        int aux = 0;
        int i = 0;

        System.out.print("Array Desordenado = { ");
        for(i = 0; i<5; i++){
            if (i==4){
                System.out.print(array[i] + " }");
            }else{
                System.out.print(array[i] + ", ");
            }
        }
        System.out.println(" ");

        for(i = 0; i<5; i++){
            for(int j = 0; j<4; j++){
                if(array[j] > array[j + 1]){
                    aux = array[j];
                    array[j] = array[j+1];
                    array[j+1] = aux;
                }
            }
        }
        System.out.println(" ");
        System.out.print("Array Ordenado = { ");
        for(i = 0; i<5; i++){
            if (i==4){
                System.out.print(array[i] + " }");
            }else{
                System.out.print(array[i] + ", ");
            }
        }
    }
}
```

```
Array Desordenado = { 5, 3, 8, 2, 6 }

Array Ordenado = { 2, 3, 5, 6, 8 }
```

Figura 3 - Implementação em Java do algoritmo Bubble-Sort
Fonte: Elaborada pela autora, 2019.

Uma das vantagens do *Bubble-Sort* é o fato de ser um método mais simples e de fácil implementação. A desvantagem, por outro lado, está no fato da eficiência diminuir conforme os números dos elementos aumentam. Considerando um *array* com *n* elementos, a complexidade do algoritmo de ordenação *Bubble-Sort* é feita em:

- melhor caso: $O(n)$;
- médio caso: $O(n^2)$;
- pior caso: $O(n^2)$.

No tópico a seguir, vamos conhecer outro método de ordenação, seu pseudocódigo e sua complexidade. Vamos em frente!

2.3 Método *Insertion-Sort*

O método de ordenação por inserção ou *Insertion-Sort*, assim como o *Bubble-Sort*, também é considerado um método simples. Ele percorre os elementos de um *array* e, conforme avança, vai ordenando os elementos. Seu funcionamento consiste em percorrer os elementos da *array* da esquerda para a direita e para cada elemento.

Começando com o segundo, é realizada uma comparação para ordenar o elemento atual com o item anterior do *array*, de acordo com o critério de ordenação. Esse algoritmo é bastante eficaz para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos da ordem de classificação.

Muitos autores, como o Ziviani (2012), utilizam uma analogia para escrever o algoritmo: é o modo como algumas pessoas ordenam as cartas em um jogo de baralho. Imagine que você esteja jogando cartas e as cartas que estão na sua mão estão ordenadas. Você recebe uma nova carta e esta deve ser ordenada também, ou seja, é preciso colocá-la na posição correta. A carta pode ser maior ou menor que outras que você já possui, por isso, você começa a comparar sua nova carta com as cartas que estão na sua mão, até encontrar a posição correta. Encontrando-a, terá novamente uma mão com cartas ordenadas.

Vejamos como funciona a ordenação crescente de um *array* unidimensional com o algoritmo de ordenação *Insertion-Sort* a seguir.

Figura 4 - Ordenação crescente de um array unidimensional com Insertion-Sort
Fonte: Elaborada pela autora, 2019.

Considere o exemplo anterior como *array* = {5, 3, 8, 2, 6}.

Primeiramente, analisamos os dois elementos iniciais do *array*, ou seja, *array* [0] e *array* [1]. Depois de analisar e ver qual dos elementos é o maior, o algoritmo vai trocar os elementos, caso o segundo seja menor que o primeiro. Temos, então, *array* = {3, 5, 8, 2, 6}. Assim, o algoritmo alterou os números 5 e 3, colocando o 3 no *array* [0] e o 5 no *array* [1].

Na próxima interação, o algoritmo irá comparar o elemento do *array* [1] com o *array* [2], a fim de verificar qual deles é menor. No caso, o 5 é menor que o 8, então, o algoritmo não altera nada, ficando com *array* = {3, 5, 8, 2, 6}. Depois, o algoritmo vai analisar o *array* [2] com o *array* [3]. Como o número 8 é maior que o 2, haverá a troca dos elementos, reordenando o *array*. Assim, *array* = {3, 5, 2, 8, 6}.

A interação a seguir, diferente do que ocorre no *Bubble-Sort*, irá continuar analisando o número 2 até encontrar seu lugar final. Dessa forma, irá comparar o *array* [2] = 2 com o *array* [1] = 5. Como 5 é maior que 2, haverá a troca novamente, ficando com *array* = {3, 2, 5, 8, 6}.

A próxima interação também continuará analisando o número 2, então, irá comparar o *array* [1] = 2 com o *array* [0] = 3. Como 3 é maior que 2, haverá a troca novamente, em que teremos *array* = {2, 3, 5, 8, 6}.

Assim, foi realizada a primeira interação pelo algoritmo, em que podemos verificar que ele foi comparando os *arrays* da esquerda para a direita, até encontrar o local final do menor número, que, no caso, foi o 2. A interação ainda será repetida mesmo que o conjunto esteja ordenado, pois o algoritmo ainda não percorreu o *array* inteiro e não sabe que ele já está ordenado.

Na tabela a seguir, temos o passo a passo desde o início de como é realizada a ordenação utilizando o *Insertion-Sort*.

Array	Comparação	Verdade	Falso
{**5**, **3**, 8, 2, 6}	5 > 3 ?	x	
{3, **5**, **8**, 2, 6}	5 > 8 ?		x
{3, 5, **8**, **2**, 6}	8 > 2 ?	x	
{3, **5**, **2**, 8, 6}	5 > 2 ?	x	
{**3**, **2**, 5, 8, 6}	3 > 2 ?	x	
{2, 3, 5, **8**, **6**}	8 > 6 ?	x	
{2, 3, **5**, **6**, 8}	5 > 6 ?		x

Figura 5 - Ordenação dos números contidos no array pelo método Insertion-Sort
Fonte: Elaborada pela autora, 2019.

Dessa forma, terminamos de percorrer o *array* e ele está organizado em ordem crescente. Note que o algoritmo fez sete comparações e alterou os dados cinco vezes, pois ele só altera quando a operação (comparação) for verdade.

Dado o *array* = [9, 5, 1, 3, 4, 7, 2], ordene os números em ordem crescente com o *Insertion-Sort* e descreva quantas comparações e trocas foram realizadas.

Agora que já sabemos como o método de ordenação funciona, vamos apresentar o pseudocódigo do algoritmo *Insertion-Sort*:

```
1  for (i=1; i < tamanho; i++){
2      aux = array [i];
3      j= i-1;
4      array [0] = aux;
5      while (x < array [j]){
6          array [j+1] = array [j];
7          j--;
8      }
9      array [j+1] = aux;
10 }
```

Figura 6 - Princípios básicos do método Insertion-Sort
Fonte: Elaborada pela autora, 2019.

A seguir, temos uma das possíveis formas de desenvolver o algoritmo de ordenação do *Insertion-Sort*, com a implementação do código na linguagem Java. A parte em preto se trata da impressão do console.

```java
public class MyInsertionSort {
    public static void main (String args[]){
        int[]array = {5,3,8,2,6};
        int i = 0;
        int aux = 0;
        int j = 0;

        System.out.print("Array Desordenado = { ");
        for( i = 0; i<5; i++){
            if (i==4){
                System.out.print(array [i] + " }");
            }else{
                System.out.print(array [i] + ", ");
            }
        }
        System.out.println(" ");

        for (i = 1; i < 5; i++) {
            aux = array[i];
            j = i-1;
            while ((j > -1) && (array [j] > aux ) ) {
                array [j+1] = array [j];
                j--;
            }
            array[j+1] = aux;
        }

        System.out.println(" ");
        System.out.print("Array Ordenado = { ");
        for(i = 0; i<5; i++){
            if (i==4){
                System.out.print(array[i] + " }");
            }else{
                System.out.print(array [i] + ", ");
            }
        }
    }
}
```

```
Array Desordenado = { 5, 3, 8, 2, 6 }

Array Ordenado = { 2, 3, 5, 6, 8 }
```

Figura 7 - Algoritmo de ordenação Insertion-Sort em Java
Fonte: Elaborada pela autora, 2019.

O *Insertion-Sort* tem como vantagem ser um algoritmo estável, que não altera a ordem dos dados iguais e pode organizar os elementos assim que os recebe, sem necessitar organizar depois de receber todos os dados. Porém, uma desvantagem é o alto custo de movimentação de elementos no *array*.
Considerando um *array* com *n* elementos, a complexidade do algoritmo de ordenação *Insertion-Sort* é dada por:

- melhor caso: $O(n)$;
- médio caso: $O(n^2)$;
- pior caso: $O(n^2)$.

No tópico a seguir, vamos conhecer o método *Selection-Sort*. Acompanhe!

2.4 Método *Selection-Sort*

O método ordenação por seleção ou *Selection-Sort*, assim como os algoritmos de ordenação anteriores, também é considerado um método simples. Este, no entanto, é baseado em selecionar o menor elemento e colocar na posição inicial do *array*, comparando-o com todos os elementos (se o objetivo for organizar em ordem crescente). Encontrando o menor valor, o algoritmo percorre novamente todo o vetor para encontrar o segundo menor valor, e assim sucessivamente.
A seguir, podemos ver como funciona a ordenação crescente de um *array* unidimensional com o algoritmo de ordenação *Selection-Sort*.

Figura 8 - Ordenação crescente de um array unidimensional com Selection-Sort
Fonte: Elaborada pela autora, 2019.

Considere o exemplo anterior como *array* = {5, 3, 8, 2, 6}.
Primeiramente, o algoritmo irá percorrer o *array* comparando todos os elementos até encontrar o menor elemento. Encontrando o menor valor, o algoritmo irá trocá-lo com o valor da posição inicial. No exemplo, ele irá comparar o *array* [0] com todos os elementos até o último *array*, que é o *array* [4]. Assim, perceberá que o valor que está no *array* [0] não é o de menor valor e irá realizar a troca,ficando com *array* = {2, 3, 8, 5, 6}.
Depois, ele fará todo o processo novamente, mas, agora, irá percorrer a partir do *array* [1], pois estamos na segunda interação.
Na tabela a seguir, temos o passo a passo desde o início de como é realizada a ordenação utilizando o *Selection-Sort*. Observe!

Array	Comparação
{5, 3, 8, 2, 6}	Compara todos os elementos até achar o menor número, faz a troca e coloca o menor elemento no *array* [0].
{2, 3, 8, 5, 6}	Compara todos os elementos a partir do *array* [1] até achar o menor número, faz a troca e coloca o menor elemento no *array* [1].
{2, 3, 8, 5, 6}	Compara todos os elementos a partir do *array* [2] até achar o menor número, faz a troca e coloca o menor elemento no *array* [2].
{2, 3, 5, 8, 6}	Compara todos os elementos a partir do *array* [3] até achar o menor número, faz a troca e coloca o menor elemento no *array* [3].
{2, 3, 5, 6, 8}	*Array* ordenado.

Figura 9 - Ordenação dos números contidos no array pelo método Selection-Sort
Fonte: Elaborada pela autora, 2019.

Dessa forma, terminamos de percorrer o *array* e ele está organizado em ordem crescente.

Dado o *array* = [9, 5, 1, 3, 4, 7, 2], ordene os números como em ordem crescente, conforme o método do *Selection-Sort*.

Agora que já sabemos como o nosso método de ordenação funciona, vamos apresentar o pseudocódigo do algoritmo *Selection-Sort*.

```
for (i=0; i<tamanho-1; i++){
    min = i;
    for (j=i+1; j< tamanho; j++){
            if (array [j] < array [min])
            min = j;
    }
    aux = array [i];
    array [i] = array [min];
    array [min] = aux;
    }
}
```

Figura 10 - Princípios básicos do método Selection-Sort
Fonte: Elaborada pela autora, 2019.

Na sequência, temos uma das possíveis formas de desenvolver o algoritmo ordenação *Selection-Sort*, com a implementação do código na linguagem Java. A parte em preto se trata da impressão do console.

```java
public class MySelectionSort {
    public static void main(String[] args) {
        int[]array = {5,3,8,2,6};
        int i, j;
        int min, temp;

        System.out.print("Array Desordenado = { ");
        for(i = 0; i<5; i++)
        {
            if (i==4){
                System.out.print(array [i] + " }");
            }else{
                System.out.print(array [i] + ", ");
            }
        }
        System.out.println(" ");


        for (i = 0; i < 5-1; i++)
        {
            min = i;
            for (j = i+1; j < 5; j++)
            {
              if (array[j] < array[min])
                min = j;
            }
            temp = array[i];
            array[i] = array[min];
            array[min] = temp;
        }
        System.out.println(" ");
        System.out.print("Array Ordenado = { ");
        for(i = 0; i<5; i++){
            if (i==4){
                System.out.print(array[i] + " }");
            }else{
                System.out.print(array [i] + ", ");
            }
        }
    }
}
```

```
Array Desordenado = { 5, 3, 8, 2, 6 }

Array Ordenado = { 2, 3, 5, 6, 8 }
```

Figura 11 - Algoritmo de ordenação Selection-Sort em Java
Fonte: Elaborada pela autora, 2019.

O algoritmo é simples de ser implementado e ocupa menos memória por não utilizar um vetor auxiliar para a ordenação. Porém, uma desvantagem é o alto custo de movimentação de elementos no vetor e o fato de não ser estável.

Assim, considerando um *array* com *n* elementos, a complexidade do algoritmo de ordenação *Selection-Sort* é que:

- melhor caso: $O(n^2)$;
- médio caso: $O(n^2)$;
- pior caso: $O(n^2)$.

No tópico a seguir, será apresentado outro método de ordenação: *Quick-Sort*.

2.5 Método *Quick-Sort*

O método de ordenação rápida ou *Quick-Sort* é mais complexo que os algoritmos anteriores. É um método de ordenação interna rápido, pois utiliza a estratégia de dividir para conquistar. Assim, a ideia é escolher um elemento qualquer chamado **pivô**, sendo que, a partir desse pivô, o *array* é organizado (ZIVIANI, 2012).

VOCÊ O CONHECE?

Charles Antony Richard Hoare, nascido em 1934, formou-se na universidade de Oxford em 1956. É conhecido por desenvolver o método de ordenação *Quick-Sort* em 1960, ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras. O algoritmo de ordenação é um dos mais utilizados do mundo até hoje.

Para aprender mais sobre o tema, clique nas abas abaixo.

- **Pivô**

  Primeiramente, o algoritmo escolhe um elemento do *array* chamado pivô. Existem várias maneiras possíveis de se escolher um pivô, podendo ser o primeiro, o último ou o elemento central do *array*. Tal escolha depende do critério do programador ao implementar o código.

- **Ordenação crescente**

  Depois de escolhermos o pivô, reorganizamos o *array* a partir dele, de modo que os elementos menores fiquem a sua esquerda e os elementos maiores fiquem a sua direita, caso a ordenação dos dados seja crescente.

- **Fim do processo**

  Terminando esse processo, o pivô estará em sua posição final e haverá dois subconjuntos de *array*s não ordenados, um à direita e outro à esquerda do pivô. Depois, de maneira recursiva, o algoritmo ordena o subconjunto dos elementos menores e maiores que o pivô, até que o *array* seja ordenado.

Vamos ver como funciona a ordenação crescente de um *array* unidimensional com o algoritmo de ordenação *Quick-Sort* a seguir.

Figura 12 - Ordenação crescente de um array unidimensional com Quick-Sort
Fonte: Elaborada pela autora, 2019.

Considere o exemplo anterior como *array* = {5, 3, 8, 2, 6}.

Primeiramente, o algoritmo irá escolher um pivô. No caso, vamos pegar o número 5 para ser o pivô. A próxima etapa é procurar à direita do pivô um número menor que 5. Essa busca terá início no final do *array*. Nesse caso, comparamos o pivô (5) com o número 6 (*array* [4]). Como 6 não é menor que 5, o algoritmo não faz alterações: *array* = {5, 3, 8, 2, 6}.

O próximo passo é comparar o pivô (5) com o número 2 (*array* [3]). Como 3 é menor que 5, a troca é realizada, ficando com *array* = {2, 3, 8, 5, 6}.

O passo seguinte é procurar à esquerda do pivô um número maior que 5. Nesse caso, comparamos o pivô (5) com o número 3 (*array* [1]). Como 3 é menor que 5, o algoritmo não faz alterações, permanecendo em array = {2, 3, 8, 5, 6}.

Depois, precisamos continuar a procurar à esquerda do pivô por um número maior que 5. Como 8 é maior que 5, a troca é realizada e ficamos com *array* = {2, 3, 5, 8, 6}. Dessa forma, o número 5 já está na sua posição final e o algoritmo irá dividir o *array* em subconjuntos. O local em que o pivô foi alocado é que irá dividir os conjuntos. Teremos, então, que {2, 3} {5} {8, 6}. Sendo assim, o algoritmo faz a escolha de um novo pivô em cada subconjunto e percorre o algoritmo, reorganizando o *array* de maneira que os elementos menores que o pivô fiquem à sua esquerda, enquanto os elementos maiores fiquem à direita, até que o *array* seja totalmente ordenado.

A seguir, temos na tabela o passo a passo desde o início de como é realizada a ordenação utilizando o *Quick-Sort*.

Array	Comparação	Pivô	Busca	Troca
{5, 3, 8, 2, **6**}	6 < 5 ?	5	Por um número menor à direita do pivô	Não
{5, 3, 8, **2**, 6}	2 < 5 ?	5	Por um número menor à direita do pivô	Sim
{2, **3**, 8, 5, 6}	3 > 5 ?	5	Por um número maior à esquerda do pivô	Não
{2, 3, **8**, 5, 6}	8 > 5 ?	5	Por um número maior à esquerda do pivô	Sim
{2, 3, 5, 8, 6}			O pivô (5) está na sua posição final	
{2, 3},{5},{8, 6}	3 < 2?	2	Por um número menor à direita do pivô	Não
{2},{3},{5},{8, 6}	Sem comparação	6	Por um número menor à direita do pivô	
{2},{3},{5},{8, 6}	8 > 6?	6	Por um número maior à esquerda do pivô	Sim
{2},{3},{5},{6},{8}				

Figura 13 - Ordenação dos números contidos no array pelo método Quick-Sort
Fonte: Elaborada pela autora, 2019.

Dessa forma, terminamos de percorrer o *array* e ele está organizado em ordem crescente.

Dado o *array* = [9, 5, 1, 3, 4, 7, 2], ordene os números corretamente em ordem crescente, pelo método *Quick-Sort*.

Agora que já sabemos como o método de ordenação funciona, vamos apresentar o pseudocódigo do algoritmo *Quick-Sort*. A escolha do pivô fica a critério do programador, sendo que podemos iniciar com o elemento mais à esquerda (como o *array* [0]) ou, até mesmo, pegar a quantidade de elementos e dividir por dois, pegando o número no meio do *array*, como no código a seguir.

```
void alg_quickSort(int array[], int esquerda, int direita) {
    int dir = direita;
        int esq = esquerda;
    int pivo = array [(esq + dir) / 2];
    int aux;

    while (esq <= dir) {
        while (array [esq] < pivo) {
            esq = esq + 1;
        }
        while (array [dir] > pivo) {
            dir = dir - 1;
        }
        if (esq <= dir) {
            aux = array [esq];
            array [esq] = array [dir];
            array [dir] = aux;
            esq = esq + 1;
            dir = dir - 1;
        }
    }
    if (dir > esquerda)
        quickSort(array, esquerda, dir);

    if (esq < direita)
        quickSort(array, esq, direita);

}
```

Figura 14 - Princípios básicos do método Quick-Sort
Fonte: Elaborada pela autora, 2019.

A seguir, temos uma das possíveis formas de desenvolver o algoritmo de ordenação *Quick-Sort*, com a implementação do código na linguagem Java. A parte em preto se trata da impressão do console.

```java
public class MyQuickSort {

    public static void quickSort(int array[], int esquerda, int direita) {
        int esq = esquerda;
        int dir = direita;
        int pivo = array[(esq + dir) / 2];
        int aux;

        while (esq <= dir) {
            while (array[esq] < pivo) {
                esq = esq + 1;
            }
            while (array[dir] > pivo) {
                dir = dir - 1;
            }
            if (esq <= dir) {
                aux = array[esq];
                array[esq] = array[dir];
                array[dir] = aux;
                esq = esq + 1;
                dir = dir - 1;
            }
        }
        if (dir > esquerda)
            quickSort(array, esquerda, dir);

        if (esq < direita)
            quickSort(array, esq, direita);

    }

    public static void main(String args[]) {

        int vetor[] = {5,3,8,2,6};
        System.out.print("Array Desordenado = { ");

        for (int i = 0; i < vetor.length; i++) {
            if (i==vetor.length -1){
            System.out.print(vetor[i] + " }");
            }else{
                System.out.print(vetor [i] + ", ");
            }
        }
        System.out.println(" ");
        System.out.print("Array Ordenado = { ");
        quickSort(vetor, 0, vetor.length - 1);

        for (int i = 0; i < vetor.length; i++) {
            if (i==vetor.length -1){
            System.out.print(vetor[i] + " }");
            }else{
                System.out.print(vetor [i] + ", ");
            }
        }
    }
}
```

```
Array Desordenado = { 5, 3, 8, 2, 6 }
Array Ordenado = { 2, 3, 5, 6, 8 }
```

Figura 15 - Algoritmo de ordenação Quick-Sort em Java
Fonte: Elaborada pela autora, 2019.

Quanto as vantagens do *Quick-Sor*t, este trabalha com eficiência com uma base de dados grande e divide o *array* em pequenos *arrays* a partir de um elemento pivô. Porém, uma desvantagem é o fato de a implementação ser mais difícil, de não ser um algoritmo estável, a decisão de escolha do pivô e que nem sempre o particionamento é balanceado.

Considerando um *array* com *n* elementos, a complexidade do algoritmo de ordenação *Quick-Sort* é dada por:

- melhor caso: $O(n \log n)$;
- médio caso: $O(n \log n)$;
- pior caso: $O(n^2)$.

Agora que entendemos sobre o método, no tópico a seguir conheceremos mais um: o *Merge-Sort*. Veremos sua complexidade e características. Acompanhe!

2.6 Método *Merge-Sort*

O método de ordenação intercalada ou *Merge-Sort* é complexo e, como o algoritmo anterior, faz uso da estratégia de dividir para conquistar. Assim, a ideia é dividir o conjunto de dados em subconjuntos, resolvendo cada subconjunto e, depois, juntando os resultados.

O algoritmo opera na estratégia de dividir os dados de entrada do *array* em dois subconjuntos com partes iguais para, em seguida, realizar o mesmo procedimento nos subconjuntos até ficar dois ou um elemento. Depois, o algoritmo une as partes ordenadas e os dois elementos de cada parte são separados. O menor deles é selecionado e retirado de sua parte. Assim, os menores entre os restantes são comparados e se prossegue dessa forma até serem unidas todas as partes. Observe a figura a seguir.

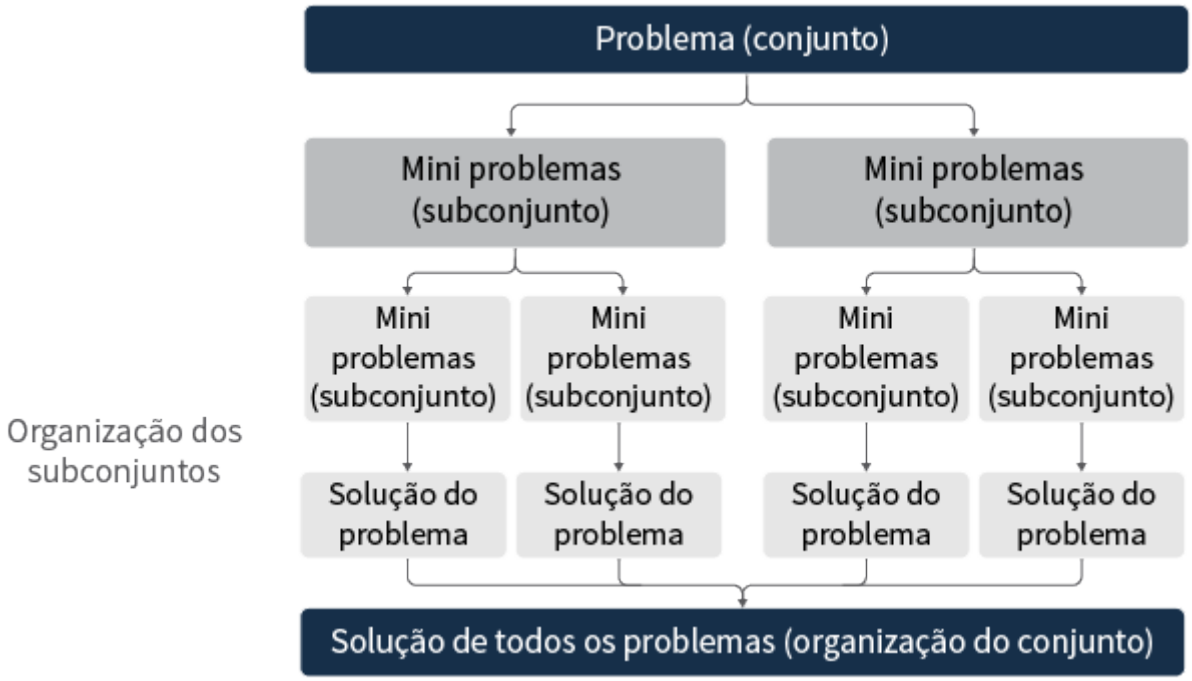

Figura 16 - Algoritmo Merge-Sort com estratégia de dividir para conquistar
Fonte: Elaborada pela autora, 2019.

Vamos ver como funciona a ordenação crescente de um *array* unidimensional com o algoritmo de ordenação *Merge-Sort*.

Figura 17 - Ordenação crescente de um array unidimensional com Merge-Sort
Fonte: Elaborada pela autora, 2019.

Considere o exemplo anterior como *array* = {5, 3, 8, 2, 6}.
A primeira coisa a ser feita é dividir o tamanho do *array* ao meio. Caso o número de elementos seja ímpar, um subconjunto terá um elemento a mais. Assim, os subconjuntos ficam divididos na esquerda e direita, desta maneira: *array* = {5, 3, 8},{2, 6}.
Depois, vamos dividir os subconjuntos da esquerda até que o subconjunto tenha tamanho 1, conforme vemos a seguir:
array = {5, 3},{8},{2, 6}
array = {5},{3},{8},{2, 6}
A seguir, intercalamos os subconjuntos e percorremos o *array* para verificar se existe troca. No caso, 3 é menor que 5, assim, trocamos a posição dos elementos, ficando com *array* = {3},{5},{8},{2, 6}.
Agora, vamos dividir os subconjuntos da direita até que o subconjunto tenha tamanho 1:
array = {3},{5},{8},{2, 6}
array = {3},{5},{8},{2},{6}
Depois que os elementos foram divididos, devemos comparar os subconjuntos. Nesse caso, comparamos se 3 é menor que 5. Como a comparação é verdade e já está em ordem, não ocorre uma troca, por isso, *array* = {3},{5},{8},{2},{6}.
Não temos, por enquanto, com o que compararmos o 8, pois o elemento pertence ao grupo da esquerda que foi dividido. Assim, vamos comparar se 2 é menor que 6. Como a comparação é verdade e já está em ordem, também não ocorre uma troca.
Agora, vamos comparar o primeiro elemento do primeiro subconjunto com o primeiro elemento do segundo subconjunto. No caso, 3 é menor que 8, então também não existe troca.
Se compararmos o segundo elemento do primeiro subconjunto com o primeiro elemento do segundo subconjunto, temos que 5 é menor que 8, também não ocorrendo uma troca.
Em seguida, vamos comparar o primeiro elemento do primeiro subconjunto com o primeiro elemento do primeiro subconjunto. Nesse caso, 2 é menor que 3, então reordenamos. Continuamos, então, a ordenação, comparando se o 3 é menor que o 5 (reordena), até ordenar todos os elementos do *array*.
Na tabela a seguir, temos o passo a passo desde o início de como é realizada a ordenação utilizando o *Merge-Sort*.

Array	Divide	Troca	Intercala	Troca
{5, 3, 8, 2, 6}	x			
{5, 3, 8},{2, 6}	x			
{5, 3},{8},{2, 6}	x			
{5},{3},{8},{2, 6}		x		Troca, pois 5 é maior que 3.
{3},{5},{8},**{2, 6}**	x			
{3},{5},{8},{2},{6}		x	x	Compara os elementos dos subconjuntos e intercala até que os elementos estejam ordenados.
{2},{3},{5},{6},{8}				

Figura 18 - Ordenação dos números contidos no array pelo método Merge-Sort
Fonte: Elaborada pela autora, 2019.

Dessa forma, terminamos de percorrer o *array* e ele está organizado em ordem crescente.

Dado o *array* = [9, 5, 1, 3, 4, 7, 2], ordene os números em ordem crescente pelo *Merge-Sort*. Depois, analise e compare o número de interações que cada algoritmo de ordenação fez para ordenar o *array*. Você conseguiu ver as diferenças? Elas influenciam no tempo de execução do nosso algoritmo.

Agora que já sabemos como o nosso método de ordenação funciona, vamos apresentar o pseudocódigo do algoritmo *Merge-Sort*.

```java
public static void merge(int array[], int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;

        //criação de arrays temporários
        int L[] = new int [n1];
        int R[] = new int [n2];

        //copia os dados para os array temporário
        for (int i=0; i<n1; ++i)
            L[i] = array[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = array[m + 1+ j];

        //mescla ou intercala os dados para arrays temporários
        //índices iniciais do primeiro e segundo subconjunto
        int i = 0, j = 0;

        //índice inicial do array mesclado
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                array[k] = L[i];
                i++;
            }
            else {
                array[k] = R[j];
                j++;
            }
            k++;
        }

        //Copia os elementos restantes de L [] se houver
        while (i < n1) {
            array[k] = L[i];
            i++;
            k++;
        }

        //Copia os elementos restantes de R[] se houver
        while (j < n2) {
            array[k] = R[j];
            j++;
            k++;
        }
    }
    // Ordena o array[l..r] usando o merge ()
    public static void sort(int array[], int l, int r) {
        if (l < r) {
            int m = (l+r)/2; // encontra o meio
            // sorteia a primeira e a segunda metade
            sort(array, l, m);
            sort(array , m+1, r);
            merge(array, l, m, r); // mescla as metades selecionadas
        }
    }
```

Figura 19 - Princípios básicos do método Merge-Sort
Fonte: Elaborada pela autora, 2019.

A seguir, vemos uma das possíveis formas de desenvolver o algoritmo de ordenação *Merge-Sort*, com a implementação do código na linguagem Java. A parte em preto se trata da impressão do console.

```java
public class MyMergeSort {
    public static void merge(int array[], int l, int m, int r) {

        int n1 = m - l + 1;
        int n2 = r - m;

        int L[] = new int [n1];
        int R[] = new int [n2];

        for (int i=0; i<n1; ++i)
            L[i] = array[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = array[m + 1+ j];

        int i = 0, j = 0;

        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                array[k] = L[i];
                i++;
            }
            else {
                array[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            array[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            array[k] = R[j];
            j++;
            k++;
        }
    }

    public static void sort(int array[], int l, int r) {
        if (l < r) {
            int m = (l+r)/2;

            sort(array, l, m);
            sort(array , m+1, r);

            merge(array, l, m, r);
        }
    }

    public static void mostraArray(int array[]) {
        for (int i=0; i<array.length; ++i) {
            if (i == array.length-1){
                System.out.print(array[i] + " }");
            }else{
                System.out.print(array[i] + ", ");
            }
        }
    }

```

```
Array Desordenado = { 5, 3, 8, 2, 6 }
Array Ordenado = { 2, 3, 5, 6, 8 }
```

```
64      public static void main(String args[]) {
65          int array[] = {5, 3, 8, 2, 6};
66
67          System.out.print("Array Desordenado = { ");
68          mostraArray (array);
69
70          MyMergeSort ob = new MyMergeSort();
71          ob.sort(array, 0, array.length-1);
72
73          System.out.println();
74          System.out.print("Array Ordenado = { ");
75          mostraArray (array);
76      }
77  }
```

Figura 20 - Algoritmo de ordenação Merge-Sort em Java
Fonte: Elaborada pela autora, 2019.

Temos, nesse caso, que o algoritmo é estável, em que não altera a ordem de dados iguais e é indicado para aplicações que possuem restrição de tempo. Porém, uma grande desvantagem é possuir um gasto extra de espaço de memória.
Considerando um *array* com *n* elementos, a complexidade do algoritmo de ordenação *Merge-Sort* é:

- Melhor Caso: O (n log n)
- Médio Caso: O (n log n)
- Pior Caso: O (n log n)

VOCÊ QUER VER?

Existe alguns vídeos que demostram o funcionamento dos algoritmos de ordenação em forma dançante. Em uma universidade na Romênia, alguns professores e um coreografo da *Sapientia University* trouxeram os conceitos sobre o funcionamento dos algoritmos de ordenação por meio das danças folclóricas, o que deixa o algoritmo bem mais divertido. Para assistir sobre o *Merge-Sort*, por exemplo, acesse o *link*: https://youtu.be/XaqR3G_NVoo. No canal, ainda temos apresentações sobre os demais métodos. Vale a pena conferir!

Sendo assim, pudemos aprender sobre alguns dos algoritmos mais utilizados, a complexidade de cada um e suas vantagens e desvantagens. Descobrimos, inclusive, que existem algoritmos que são mais utilizados para aumentar a eficiência, principalmente no fator tempo, como o *Quick-Sort* e o *Merge-Sort*, que são mais complexos.

Síntese

Chegamos ao fim de mais uma unidade de estudos. Aqui, vimos a respeito da ordenação de dados a partir de vários algoritmos, incluindo a complexidade e os exemplos de códigos implementados na linguagem de programação Java de cada um deles.

Nesta unidade, você teve a oportunidade de:

- assimilar os conceitos de métodos de ordenação;
- compreender o funcionamento dos algoritmos junto com vantagens e desvantagens;
- entender as diferenças entre os métodos;
- implementar os principais métodos de ordenação.

Bibliografia

ALGORYTHMICS. ***Merge-sort with Transylvanian-saxon (German) folk dance***. 23 abr. 2011. Disponível em: https://www.youtube.com/watch?v=XaqR3G_NVoo&feature=youtu.be. Acesso em: 29 jul. 2019.

KHAN ACADEMY. **Ciência da Computação**: algoritmos. 2019. Disponível em: https://pt.khanacademy.org/computing/computer-science/algorithms#quick-sort. Acesso em: 26 jul. 2019.

TAMASSIA, R.; GOODRICH, M. T. **Estrutura de dados e algoritmos em Java**. Porto Alegre: Bookman, 2006.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos.** 3. ed. Rio de Janeiro: LTC, 2010.

ZIVIANI, N. **Projeto de algoritmos**: com implementações em JAVA e C++. São Paulo: Cengage Learning Editores, 2012.