ESTUDO ANALÍTICO DO DESEMPENHO DE ALGORITMOS DE ORDENAÇÃO

Alexandre da Silva Pedroso[1]

Fausto Gonçalves Cintra[2]

Resumo

Neste estudo, busca-se inicialmente entender o conceito de algoritmos e para que eles são utilizados, compreendendo que eles são largamente utilizados por qualquer pessoa na realização várias tarefas do dia a dia. Na Ciência da Computação, algoritmos são utilizados na descrição detalhada das etapas para se resolver um problema dado. Dentre esses problemas, a ordenação de dados é de extrema importância para que cientistas da computação sejam capazes de desenvolver soluções eficientes para a localização de um dado em um banco de dados, entre outras aplicações. Visto que a ordenação pode ser conseguida por meio de diferentes algoritmos, este estudo traz como objetivo realizar a análise da complexidade temporal de três dos algoritmos de ordenação mais conhecidos: *bubble sort*, *selection sort* e *quick sort*, por meio da utilização de funções de grandeza assintótica. Conclui que, dentre os algoritmos analisados, o mais eficiente temporalmente é o *quick sort*, cuja complexidade é da ordem $O(n \log n)$ no caso médio.

Palavras-chave: *Bubble sort*. Complexidade temporal. Notação assintótica. *Quick sort*. *Selection sort*.

Abstract

In this study, we, initially, seek for understanding the concept of algorithms and what they are used for, recognizing that they are widely used by anyone in performing various daily tasks. In Computer Science, algorithms are used in the detailed description of the steps to solve a given problem. Among these problems, data sorting is extremely important to enable computer scientists to develop efficient solutions for locating data in a database, among other applications. Since sorting can be achieved through different algorithms, this study aims to analyze the temporal complexity of three of the most popular sorting algorithms: *bubble sort, selection sort* and *quick sort*, using asymptotic magnitude functions. It concludes that, among the analyzed algorithms, the most temporally efficient is the quick sort, whose complexity is of the order $O(n \log n)$ in the average case.

Keywords: Bubble sort. Temporal complexity. Asymptotic notation. Quick sort. Selection sort.

[1] Graduando em Análise e Desenvolvimento de Sistemas pela Fatec Franca “Dr. Thomaz Novelino”. Endereço eletrônico: alexandre.pedroso01@fatec.sp.gov.br.
[2] Docente da Fatec Franca “Dr. Thomaz Novelino”, Mestre em Desenvolvimento Regional pelo Centro Universitário Municipal de Franca, Bacharel em Ciência da Computação com ênfase em Análise de Sistemas pela Universidade de Franca. Endereço eletrônico: fausto.cintra@fatec.sp.gov.br.

1 Introdução

Atualmente, pode-se observar que muito se tem falado, nas mídias sociais, sobre o uso de algoritmos por essas mesmas mídias e por diversas aplicações que são utilizadas todos os dias na Internet. Apesar de esse assunto permear o cotidiano, muitas pessoas não têm o entendimento claro do que seja um algoritmo e do que ele pode fazer.

Pensando nessa problemática, buscou-se a definição de algoritmo em diversos autores. Segundo Cormen et al. (2012), um algoritmo é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída. Portanto, um algoritmo é uma sequência de etapas computacionais que transformam a entrada em uma saída. Szwarcfiter e Markenzon (2010), de forma semelhante, informam que um algoritmo é um processo sistemático para a resolução de um problema.

Lipschutz e Lipson (2004) escrevem que um algoritmo M é uma lista finita de passos com instruções bem definidas para resolver um problema particular, ou seja, para determinar a saída $f(X)$ de uma dada função f com entrada X (aqui, X pode ser uma lista ou conjunto de valores). Frequentemente, pode existir mais de uma maneira de obter $f(X)$. A escolha particular de algoritmo M para obter $f(X)$ pode depender da “eficiência” ou da “complexidade” do algoritmo.

Diante de tais definições, pode-se observar a importância do estudo e desenvolvimento de algoritmos para solução dos diversos problemas que se apresentam para desenvolvedores de *software* e aplicações computacionais.

De acordo com Pierro (2018), são necessárias três etapas para se construir um algoritmo. A primeira consiste em identificar o problema a ser resolvido, ou seja, é preciso definir o objetivo do algoritmo. Se o desafio for usar imagens para detectar algum tipo de câncer de forma mais precisa, o cientista da computação poderá criar uma estratégia levando em conta as características dos tumores, as bases de dados disponíveis e os métodos possíveis de diagnóstico.

A segunda etapa consiste na organização da solução, estabelecendo a sequência de passos para resolver o problema. No caso de diagnóstico de câncer, vasculhar imagens médicas disponíveis, comparar tumores e seus volumes e levantar dados sobre a evolução da doença e sua mortalidade.

ISSN: 2595-5942

Por último, na terceira etapa, é construído o algoritmo utilizando se de uma linguagem de programação. Cada passo é traduzido em linhas de código, com os comandos necessários para sua execução.

De acordo com Szawarcfiter e Markenzon (2014), o estudo de algoritmos aborda dois aspectos básicos: a correção e a análise. No que tange à correção, tenciona-se que um algoritmo forneça uma solução correta para um problema dado, qualquer que seja o parâmetro de entrada fornecido. Quanto à análise de algoritmos, deseja-se observar a eficiência do algoritmo em termos de tempo de execução e de consumo de memória durante a solução do problema dado.

Verifica-se ainda na bibliografia de referência que existem uma infinidade de problemas que podem ser resolvidos com o auxílio de algoritmos usados em programas de computador, dentre os quais é possível citar como exemplos (CORMEN et al., 2012):

- O projeto Genoma Humano vem alcançando grande progresso no cumprimento de suas metas de identificar todos os 100.000 genes do DNA humano, determinar as sequências dos três bilhões de ares de bases químicas que constituem o DNA humano, armazenar informações em bancos de dados e desenvolver ferramentas para análise de dados. Cada uma dessas etapas exige algoritmos sofisticados.
- A Internet possibilita que pessoas em todo mundo acessem e obtenham rapidamente grande quantidade de informações. Com o auxílio de algoritmos engenhosos, *sites* da internet conseguem gerenciar a manipular esse grande volume de dados. Cite-se aqui a determinação de boas rotas para transmissão de dados e a utilização de um mecanismo de busca para encontrar rapidamente páginas em que são apresentadas determinadas informações.
- A Projeta Sistemas, *startup* localizada em Vitória (ES), criou um sistema computacional com algoritmos que se baseiam em imagens 3D para estimar o peso de um boi.

Um problema mais comum a ser solucionado utilizando um algoritmo é a necessidade de se ordenar uma sequência de valores em ordem crescente.

Por exemplo, um algoritmo, ao receber como entrada uma sequência de n números

$$a_1, a_2, \dots, a_n$$

deveria permutar esses números de modo que a saída (solução) para o problema seja a sequência ordenada

$$a'_1, a'_2, \dots, a'_n$$

tal que

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n.$$

Este estudo se ocupará do aspecto analítico do estudo de três dos algoritmos de ordenação vistos nas aulas da disciplina de Estrutura de Dados do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Fatec Franca, quais sejam: *bubble sort*, *selection sort* e *quick sort*. Mais especificamente, objetiva-se aprofundar a compreensão sobre o tempo de execução dos algoritmos citados.

2 Análise assintótica de algoritmos

Para cumprir com o objetivo proposto, procedeu-se a um estudo analítico buscando compreender o comportamento de tempo de cada um dos algoritmos, a partir do estudo de comparação de ordens assintóticas de funções, a fim de chegar a uma estimativa acerca da eficiência dos algoritmos.

Segundo Feofiloff (2019), para o estudo de análise de algoritmos, ao utilizarmos, por exemplo, as expressões $n + 10$ ou $n^2 + 1$, deve-se concentrar nos valores enormes de n, pois, para tais valores enormes de n, algumas funções possuem a mesma “taxa de crescimento”, sendo, portanto, “equivalentes”.

O campo da matemática que se interessa por tais valores enormes de n é chamada análise assintótica. Nessa seara, as funções são classificadas em “ordens assintóticas”. A definição a seguir relaciona o comportamento assintótico de duas funções distintas.

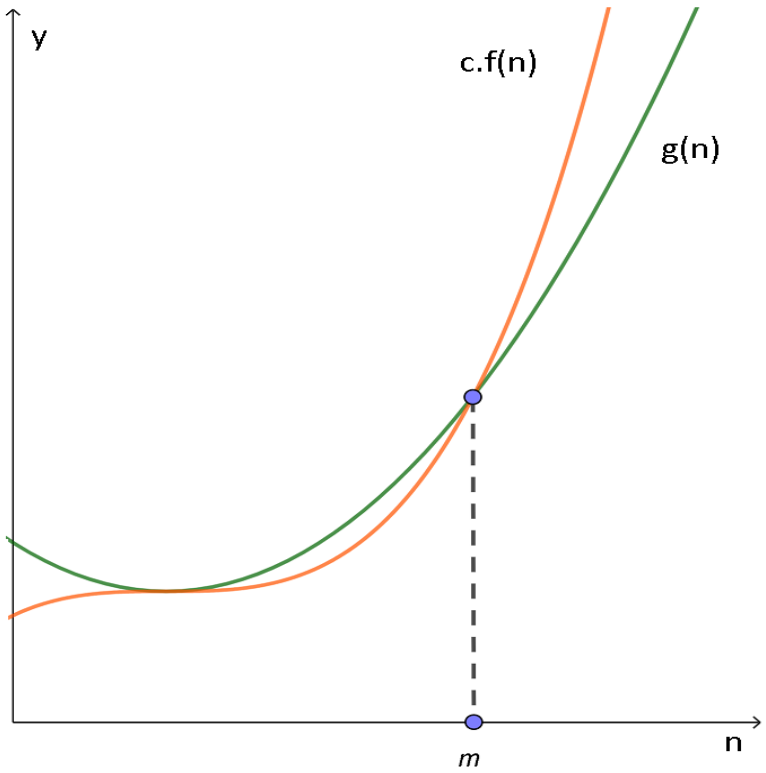
Definição 1: Uma função $f(n)$ **domina assintoticamente** outra função $g(n)$ se existem duas constantes positivas c e m tais que, para $n \geq m$, tem-se $|g(n)| \leq c \times |f(n)|$.

ISSN: 2595-5942

Por exemplo, dadas as funções $g(n) = n$ e $f(n) = -n^2$, pode-se verificar que $|n| \leq |-n^2|$ para todo n natural. Se fizermos $c = 1$ e $m = 0$, a definição 1 será satisfeita. Logo, conclui-se que $f(n)$ domina assintoticamente $g(n)$. Nesse caso $g(n)$ não domina assintoticamente $f(n)$, pois $|-n^2| > c|n|$ para todo $n > c$ e $n > 1$ para qualquer valor de c.

Segundo Ziviani (2015), foi Knuth quem, em 1968, sugeriu uma notação para dominação assintótica. Para expressar que uma função $f(n)$ domina assintoticamente $g(n)$ escreve-se $g(n) = O(f(n))$, em que se lê $g(n)$ é da ordem no máximo $f(n)$. Por exemplo, quando se diz que o tempo de execução $T(n)$ de um programa é $O(n^2)$, isso significa que existem constantes c e m tais que, para valores de n maiores ou iguais a m, $T(n) \leq cn^2$. A Figura 1 mostra um exemplo gráfico de dominação assintótica que ilustra a notação O.

Figura 1 - Dominação assintótica de f(n) sobre g(n)



Fonte: elaborado pelos autores.

2.1 Notação Big-O

Definição 2: Sejam f e g funções do conjunto de números inteiros ou dos números reais para o conjunto dos números reais. É dito que $f(x)$ é $O(g(x))$ (lê-se $f(x)$ é Big-O de $g(x)$) se houver constantes C e k, tal que

$$|f(x)| \leq C|g(x)|$$

sempre que $x > k$.

As constantes C e k na definição Big-O são chamadas de parâmetros da relação $f(x)$ é $O(g(x))$.

Para mostrar que $f(x)$ é $O(g(x))$ basta encontrar um par de constantes C e k, os parâmetros, tal que $f(x)$ é $O(g(x))$ sempre que $x > k$.

Um método útil para encontrar um par de parâmetros é primeiro selecionar um valor de k para o qual o tamanho de $|f(x)|$ pode ser rapidamente estimado quando $x > k$ e ver se é possível usar essa estimativa para encontrar um valor de C para o qual $f(x)$ é $O(g(x))$ para $x > k$. Observe-se o exemplo a seguir.

Exemplo 1: Mostre que $f(x) = x^2 + 2x + 1$ é $O(x^2)$.

Solução: Estima-se o tamanho de $f(x)$ quando $x > 1$, pois $x < x^2$ e $1 < x^2$ quando $x > 1$. Daí se obtém

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

sempre que $x > 1$. Consequentemente, tem-se $C = 4$ e $k = 1$ como parâmetros para mostrar que $f(x)$ é $O(x^2)$. Ou seja, $f(x) = x^2 + 2x + 1 < 4x^2$ para todo $x > 1$.

Exemplo 2: Mostre que $f(x) = 7x^2$ é $O(x^3)$.

Solução: Estima-se o tamanho de $f(x)$ para $x > 7$, pois multiplicando os dois lados de $x > 7$ por x^2 obtém-se $x^3 > 7x^2$. Consequentemente, tem-se que $C = 1$ e $k = 7$ como parâmetros para estabelecer a relação em que $7x^2$ é $O(x^3)$.

Exemplo 3: Suponha que $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_mx^m$ tem grau m. Prove que $f(x)$ é $O(x^m)$.

Solução: Seja $b_0 = |a_0|$, $b_1 = |a_1|$,..., $b_m = |a_m|$. Então, para $n \geq 1$,

$$f(x) \leq b_0 + b_1x + b_2x^2 + \cdots + b_mx^m = \left(\frac{b_0}{x^m} + \frac{b_1}{x^{m-1}} + \cdots + b_m\right)x^m$$

$$\leq (b_0 + b_1 + b_2 + \cdots + b_m)x^m = Mx^m$$

onde $M = |a_0| + |a_1| + |a_2| + \cdots + |a_m|$. Portanto, $f(x)$ é $O(x^m)$.

Generalizando essa solução, pode-se escrever $5x^3 + 3x$ é $O(x^3)$ e $x^5 - 4.000.000x^2$ é $O(x^5)$.

ISSN: 2595-5942

2.2 Notação Big-Ω

Definição 3: Considere f e g funções do conjunto de números inteiros ou dos números reais para o conjunto dos números reais. Diz-se que $f(x)$ é $\Omega(g(x))$ (lê-se $f(x)$ é Big Ômega de $g(x)$) se houver constantes C e k, tal que

$$|f(x)| \geq C|g(x)|$$

sempre que $x > k$.

Exemplo 4: A função $f(x) = 8x^3 + 5x^2 + 7$ é $\Omega(g(x))$, em que $g(x)$é a função $g(x) = x^3$. É fácil verificar isto, pois $f(x) = 8x^3 + 5x^2 + 7 \geq 8x^3$ para todo número real positivo x. É o mesmo que dizer que $g(x) = x^3$ é $O(8x^3 + 5x^2 + 7)$, que pode ser estabelecido diretamente virando a inequação.

2.3 Notação Big-Θ

Definição 4: Considere f e g funções do conjunto de números inteiros ou dos números reais para o conjunto dos números reais. Diz-se que $f(x)$ é $\Theta(g(x))$ (lê-se $f(x)$ é Big Theta de $g(x)$), se $f(x)$ é $O(g(x))$ e $f(x)$ é $\Omega\big(g(x)\big)$ e diz-se que $f(x)$ é da ordem de $g(x)$.

Exemplo 5: Mostre que $3x^2 + 8x \log x$ é $\Theta(x^2)$.

Solução: É fácil observar que $0 \leq 3x^2 + 8x \log x \leq 8x^2$, consequentemente $3x^2 + 8x \log x \leq 11x^2$ para todo $x > 1$. Desse modo, tem-se que $3x^2 + 8x \log x$ é $O(x^2)$ e como é fácil perceber que x^2 é $\Theta(3x^2 + 8x \log x)$ conclui-se que $3x^2 + 8x \log x$ é $\Theta(x^2)$.

2.4 Análise de algoritmos

Como escolher um algoritmo para resolver um problema? Quais critérios leva-se em consideração na hora da escolha? Segundo Gersting (2010), quando se compara algoritmos, vários critérios podem ser utilizados para julgarmos qual deles é o “melhor”. É necessário indagar, por exemplo, qual deles é mais fácil de ser entendido, ou qual é executado de forma mais eficiente.

No estudo analítico de algoritmos, deseja-se poder estimar a eficiência de algoritmos acerca do consumo de tempo do algoritmo em função do tamanho dos

ISSN: 2595-5942

dados de entrada. Para comparar o desempenho de dois algoritmos para solucionar um problema, suponha-se que $f(x)$ é o consumo de tempo de um algoritmo A e $g(x)$ é o consumo de tempo de um algoritmo B. Para compararmos a eficiência desses algoritmos A e B, é feito o comportamento de $f(x)$ e de $g(x)$ para valores enormes de x. Nesse caso se $f(x)$ é $O(g(x))$, o algoritmo A será considerado pelo menos tão eficiente quanto B. Se, além disso, $f(x)$ é $\Theta(g(x))$, o algoritmo A é considerado mais eficiente que B.

3 O problema da ordenação

O que é ordenar dados? Por que é tão importante ordenar dados em computação? Tome-se o caso de alguém que precise localizar o número de celular de um amigo na lista de contatos do seu celular, mas esses contatos não estão organizados sob nenhuma ordem. Caso a quantidade de contatos na lista seja elevada, essa busca pode se tornar uma atividade muito laboriosa e por vezes até mesmo infrutífera. Não por acaso, a lista de contatos dos celulares costuma estar organizada em ordem alfabética. Pode-se citar também a busca de informações sobre um cliente no banco de dados de uma empresa que possui milhares de cadastros. Se os dados se encontram ordenados, a busca pela informação se torna menos complexa. Por esses e outros motivos os algoritmos de ordenação se tornam um importante recurso utilizado por cientistas da computação.

Neste estudo, conforme antes anunciado, serão discutidas as características e a eficiência temporal dos algoritmos de ordenação *bubble sort*, *selection sort* e *quick sort*, tendo sido recolhidos dados a esse respeito por meio de pesquisa bibliográfica. Serão utilizadas as implementações dos algoritmos mencionados feitas na linguagem JavaScript no decurso das aulas da disciplina Estrutura de Dados do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Faculdade de Tecnologia de Franca “Dr. Thomaz Novelino” – Fatec Franca. Tais implementações, por seu turno, tiveram por base a obra de Groner (2018).

3.1 Bubble sort

O algoritmo *bubble sort* (ordenação por flutuação) compara cada dois valores adjacentes, realizando a permuta entre eles se o primeiro valor for maior do que o segundo. O processo se repete por várias passadas, até que, na última delas, nenhuma permuta tenha sido realizada.

A Figura 2 mostra a implementação do *bubble sort* utilizada na análise.

Figura 2 - Implementação do algoritmo *bubble sort*

```
let totTrocas, pass, comps

function bubbleSort(vetor, fnComp){
    totTrocas = 0, pass = 0, comps = 0
    let trocas
    do{
        trocas = 0  // {1}
        pass++
        for(let i = 0; i <= vetor.length - 2; i++){ // {2}
            if(fnComp(vetor[i], vetor[i + 1])) {
                [vetor[i], vetor[i + 1]] = [vetor[i + 1], vetor[i]] // {3}
                trocas++
            }
            comps++
        }
        totTrocas += trocas
    } while(trocas > 0)
}
```

Fonte: elaborado pelos autores.

O algoritmo é iniciado declarando-se três variáveis, `totTrocas`, `pass` e `comps`, que irão, respectivamente, contar o total de permutas, o número de passadas e as comparações realizadas na implementação do algoritmo para ordenar um vetor de números a ser ordenado, o qual a função recebe como entrada. Em `{1}` inicia-se uma nova passada, na etapa `{2}` percorre-se o vetor, da primeira até a penúltima posição, em `{3}` é realizada a troca de posição via desestruturação do vetor.

3.1.1 Complexidade temporal do *bubble sort*

Para determinar a complexidade temporal do *bubble sort*, calcula-se a média entre o melhor caso e o pior caso para um vetor com n elementos. Observe-se que a

ISSN: 2595-5942

execução da estrutura **`do..while`** das linha 6 a 17 depende do número de trocas realizadas.

O melhor caso ocorre quando o vetor a ser ordenado já está na ordem correta. Nesse caso, nenhuma troca será realizada e a estrutura **`for`** das linhas 9 a 15 será executada uma única vez, tendo sido realizadas $n - 1$ comparações pelo **`if`** na linha 10. Com isso diz-se que, no melhor caso, esse algoritmo possui complexidade $O(n)$.

O pior caso ocorre quando o vetor a ser ordenado está na ordem inversa à desejada. Nesse caso a estrutura **`for`** das linhas 9 a 15 será executada n vezes e serão realizadas $n - 1$ comparações pelo **`if`** na linha 10 para cada vez que a linha 9 for executada. Logo, o número de comparações realizadas será $n(n-1) = n^2 - n$. Isso leva a concluir que, no pior caso, o tempo de execução do algoritmo é $O(n^2)$.

O caso médio para esse algoritmo será

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

e, portanto, a complexidade do caso médio para o algoritmo será $O(n^2)$.

3.2 *Selection sort*

O algoritmo *selection sort* (ordenação por seleção) encontra primeiro o valor mínimo entre os dados de entrada, coloca esse valor na primeira posição, depois encontra o segundo valor mínimo colocando-o na segunda posição, e faz isso sucessivamente até que o vetor esteja ordenado.

Inicialmente, conforme mostrado na Figura 3, são declaradas três variáveis: `trocas`, `pass`, e `comps`, para a contagem da quantidade de trocas, passadas e comparações realizadas durante a implementação do algoritmo, que recebe como entrada um vetor de números para ser ordenado. A função `encontrarMenor()` executa um *loop* em `{1}` que percorre o vetor até a última posição com o objetivo de encontrar o menor valor dentro do vetor. Já o *loop* executado em `{2}` percorre o vetor até a penúltima posição com o objetivo de realizar a troca em `{3}`, inserindo o número na sua posição.

ISSN: 2595-5942

Figura 3 - Implementação do algoritmo *selection sort*

```
1   let trocas, pass, comps
2
3   function selectionSort(vetor) {
4
5       trocas = 0, pass = 0, comps = 0
6
7       function encontrarMenor(inicio) {
8           let menor = inicio
9           for (let j = inicio + 1; j < vetor.length; j++) {   // {1}
10              if (vetor[j] < vetor[menor]) menor = j
11              comps++
12          }
13          return menor
14      }
15
16      for (let i = 0; i <= vetor.length - 2; i++) {   // {2}
17          pass++
18          let menor = encontrarMenor(i + 1)
19          if (vetor[menor] < vetor[i]) {
20              [vetor[menor], vetor[i]] = [vetor[i], vetor[menor]]     // {3}
21              trocas++
22          }
23          comps++
24      }
25  }
```

Fonte: elaborado pelos autores.

3.2.1 Complexidade temporal do *selection sort*

Para determinar a complexidade temporal do *selection sort* foi analisado o caso de ordenação de um vetor com n elementos.

Para esse algoritmo, ao utilizarmos um vetor com n posições, cada elemento i da primeira até a última posição é trocado de lugar com o menor elemento que se encontra entre as posições $(i+1)$ e $(n-1)$. Para encontrar o primeiro menor elemento e trocá-lo com o da posição 0, são realizadas $n-1$ comparações, considerando que a linha 10 será executada $n-2$ vezes e a linha 19 uma vez. Para encontrar o segundo menor elemento e trocá-lo com o da posição 1, são realizadas $n-2$ comparações em que a linha 10 é executada $n-3$ vezes e a linha 19 é executada uma vez. Esse padrão será repetido até que todo vetor seja ordenado, assim para encontrarmos o tempo de execução basta realizar a seguinte operação:

$$T(n) = 1 + 2 + 3 + \cdots + (n-1)$$

ISSN: 2595-5942

que é uma progressão aritmética de razão 1 e pode ser calculada aplicando-se a fórmula dos termos de uma progressão aritmética encontrando-se como resultado $T(n) = \frac{1}{2}(n^2 - n)$.

Sendo assim, é possível concluir que a complexidade temporal do algoritmo *selection sort* é $O(n^2)$.

3.3 *Quick sort*

O algoritmo *quick sort* (ordenação rápida), conforme mostrado na Figura 4, primeiramente seleciona um valor do vetor, o qual é designado como pivô (na implementação usada, o último elemento). Na primeira passada, divide o vetor, a partir da posição final do vetor, em um subvetor à esquerda contendo apenas valores menores que o pivô, e outro subvetor à direita, que contém apenas valores maiores que o pivô. Em seguida, recursivamente, repete o processo em cada um dos subvetores até que todo o vetor esteja ordenado.

No início, são declaradas três variáveis: `trocas`, `comps` e `pass`, para a contagem da quantidade de trocas, comparações e passadas realizadas durante a execução do algoritmo, que recebe como entrada um vetor de números para ser ordenado juntamente com as posições inicial e final a serem trabalhadas dentro do vetor. Em `{1}`, é realizado o percurso do vetor até a penúltima posição com o intuito de dividir o vetor em subvetores à esquerda com valores menores que o pivô e à direita com valores maiores que o pivô; em `{2}` o pivô é inserido em sua posição definitiva; e em `{3}` e `{4}` os subvetores à esquerda e à direita do pivô, respectivamente, são recursivamente ordenados.

3.3.1 Complexidade temporal do *quick sort*

Para determinar a complexidade do *quick sort* serão analisados o pior caso e o melhor caso para ordenar um vetor com n elementos, segundo mostrado suscintamente em Ziviani (2015).

ISSN: 2595-5942

Figura 4 - Implementação do *quick sort*

```
1   let trocas, comps, pass
2
3   function quickSort(vetor, ini = 0, fim = vetor.length - 1) {
4       if (fim > ini) {
5           pass++
6           const pivot = fim
7           let div = ini - 1
8           for (let i = ini; i < fim; i++) {   // {1}
9               if (vetor[i] < vetor[pivot]) {
10                  comps++
11                  div++
12                  if (i != div) {
13                      [vetor[i], vetor[div]] = [vetor[div], vetor[i]]
14                      trocas++
15                  }
16              }
17          }
18          div++
19          if (vetor[pivot] < vetor[div]) {    // {2}
20              [vetor[pivot], vetor[div]] = [vetor[div], vetor[pivot]]
21              trocas++
22          }
23          comps++
24
25          quickSort(vetor, ini, div - 1)  // {3}
26
27          quickSort(vetor, div + 1, fim)  // {4}
28      }
29  }
```

Fonte: elaborado pelos autores

Uma característica peculiar do *quick sort* é sua ineficiência para vetores já ordenados quando a escolha do pivô é inadequada. Por exemplo, a sistemática dos extremos de um vetor já ordenado leva seu pior caso. Nessa situação, as partições serão extremamente desiguais e o procedimento de ordenação será chamado recursivamente n vezes, eliminando apenas um item em cada passada. Essa situação é desastrosa, pois o número de comparações passa a cerca de $\frac{n^2}{2}$, e o tamanho da pilha necessária para as chamadas recursivas é cerca de n.

A melhor situação possível ocorre quando cada partição divide o vetor em duas partes iguais. Logo,

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

ISSN: 2595-5942

onde $C\left(\frac{n}{2}\right)$ representa o custo de ordenar uma das metades e $n-1$ é o número de comparações realizadas. A solução para esta recorrência é:

$$T(n) = n\log n - n + 1.$$

No caso médio, de acordo com Sedgewick e Flajolet (apud ZIVIANI, 2015), o número de comparações realizadas é

$$T(n) \approx 1{,}386n\log n - 0{,}846n,$$

o que significa que em média o tempo de execução do *quick sort* é $O(n\log n)$.

3.4 Síntese

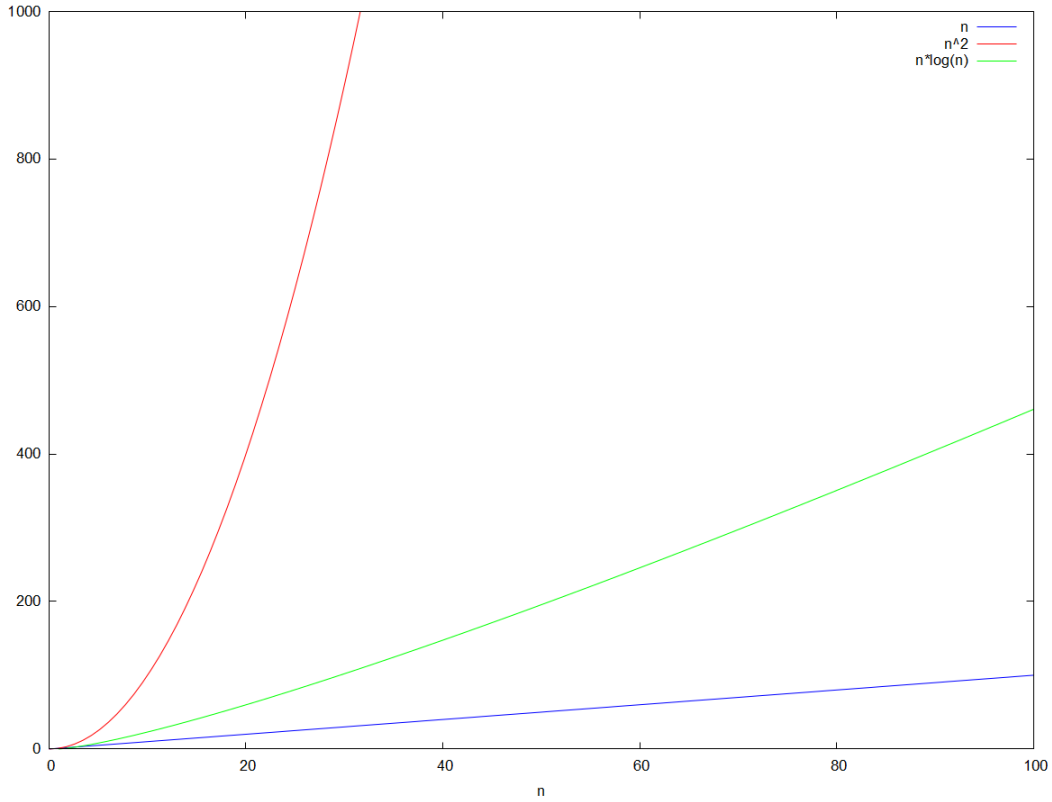
As complexidades temporais dos algoritmos de ordenação analisados são dadas em síntese na Tabela 1 e, para visualização mais intuitiva acerca do desempenho dos algoritmos, foi elaborado o gráfico da Figura 5.

Tabela 1 - Complexidade temporal dos algoritmos de ordenação analisados

Algoritmo	Complexidade do tempo		
	Melhores casos	**Casos médios**	**Piores casos**
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$

Fonte: elaborado pelos autores.

Figura 5 - Gráfico das funções de complexidade



Fonte: elaborado pelos autores.

Considerações finais

Tendo realizado o estudo, foi ampliada a compreensão de como é medido o desempenho de algoritmos utilizando-se da notação Big-O, com a utilização de funções assintóticas que são utilizadas para descrever analiticamente a complexidade dos algoritmos de ordenação abordados.

No estudo das funções de crescimento assintótico, foi possível verificar graficamente, conforme mostrado na Figura 5, como é o comportamento do consumo de tempo de cada algoritmo, de acordo com sua ordem de complexidade, conforme o tamanho do vetor a ser ordenado aumenta. Dentre os algoritmos de ordenação tratados neste estudo, para os melhores casos vimos que o algoritmo com complexidade temporal $O(n)$ possui melhor desempenho que os algoritmos com complexidades temporais $O(n \log n)$ e $O(n^2)$. Já para os casos médios, que ocorrem na maioria das vezes, o algoritmo que apresenta complexidade temporal

$O(n \log n)$ possui melhor desempenho que os algoritmos de complexidade temporal $O(n^2)$.

Registre-se o desejo de aprofundar o tema em futuros estudos em que, sob condições controladas de ambiente, torne possível verificar o que pode influenciar mais sobre o tempo de execução de algoritmos: a sua complexidade ou a configuração do *hardware* em que ele é executado.

Referências

CORMEN, T.H. et al. **Algoritmos**: Teoria e Prática. 3. ed. Rio de Janeiro, 2012.

FEOFILOFF, Paulo. **Minicurso de análise de algoritmos**. IME-USP, 2019. Disponível em: https://www.ime.usp.br/~pf/livrinho-AA/AA-BOOKLET.pdf. Acesso em: 6 jun. 2021.

GERSTING, Judith L. **Fundamentos Matemáticos para a Ciência da Computação**: Um Tratamento Moderno de Matemática discreta. 5. ed. Rio de Janeiro: LTC - Livros Técnicos e Científicos, 2010.

GRONER, Loiane. **Estrutura de dados e algoritmos em JavaScript**. 2. ed. São Paulo: Novatec, 2018.

LIPSCHUTZ, Seymour; LIPSON, Marc. **Teoria e problemas de matemática discreta**. 2. ed. Porto Alegre: Bookman, 2004.

PIERRO, B. **O mundo mediado por algoritmos**. Pesquisa Fapesp, São Paulo, n. 266, p. 18, 2018. Disponível em: https://revistapesquisa.fapesp.br/folheie-a-edicao-266/. Acesso em: 4 set. 2022.

SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. **Estruturas de Dados e Seus Algoritmos**. 3. ed. Rio de Janeiro: Ltc, 2010.

ZIVIANI, Nivio. **Projeto de algoritmos**: com implementações em Pascal e C. 3. Ed. São Paulo: Cengage Learning, 2015.