

# **Computabilidade e Complexidade de Algoritmos**



**Cruzeiro do Sul Virtual**  
Educação a distância



# Material Teórico



Complexidade de Algoritmos

**Responsável pelo Conteúdo:**

Prof. Dr. Luciano Rossi

**Revisão Textual:**

Prof.<sup>a</sup> Esp. Kelciane da Rocha Campos



# UNIDADE

## Complexidade de Algoritmos



- **Tempo de Execução em Função do Tamanho da Instância;**
- **Melhor Caso, Caso Médio e Pior Caso;**
- **Notação Assintótica.**



### OBJETIVO DE APRENDIZADO

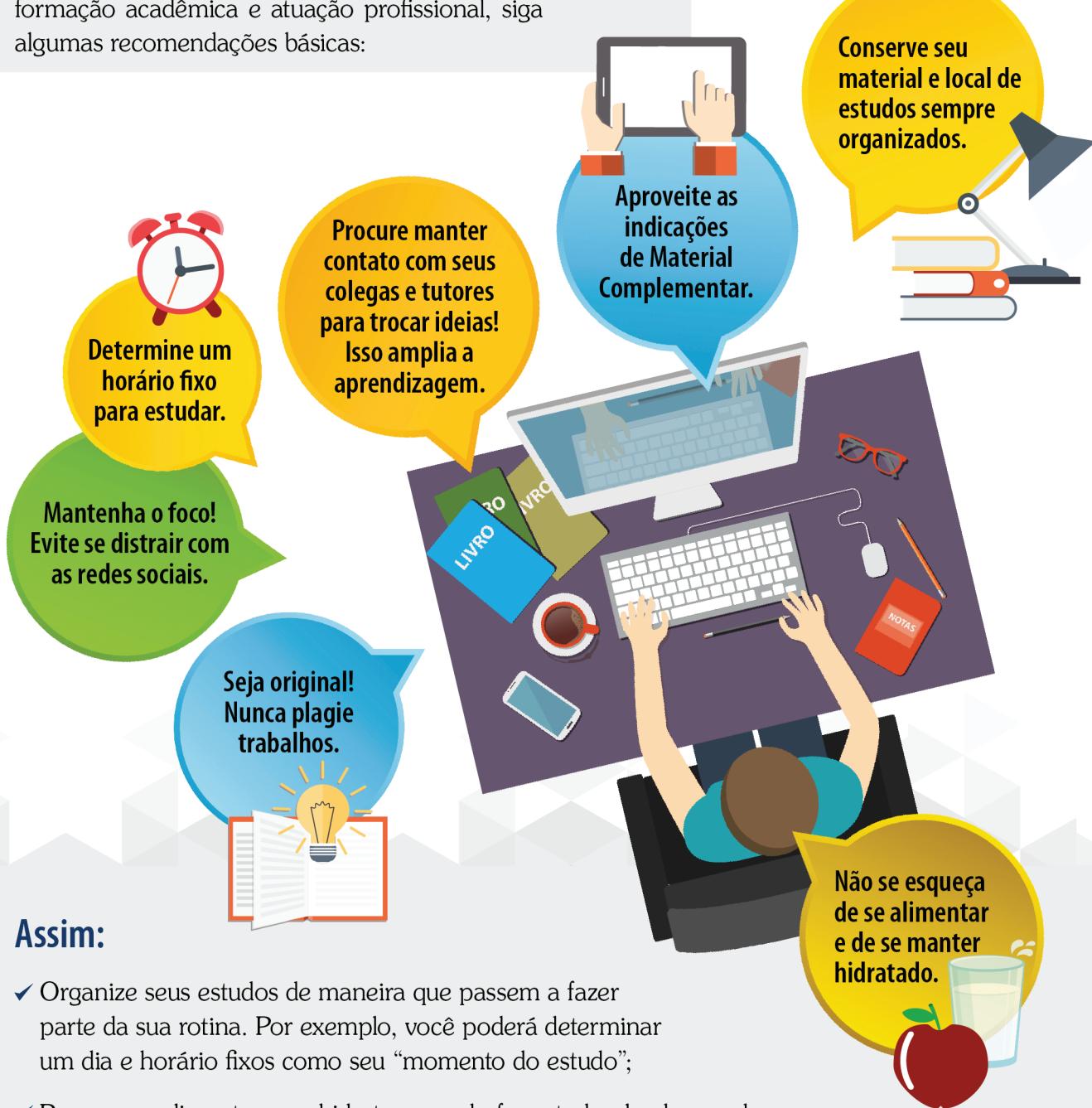
- Apresentar ao aluno a forma de classificação de algoritmos de acordo com o respectivo tempo de execução, considerando o tamanho da instância, bem como a utilização da notação assintótica a partir da definição de limites inferiores e superiores e as respectivas técnicas de análise.





# Orientações de estudo

Para que o conteúdo desta Disciplina seja bem aproveitado e haja maior aplicabilidade na sua formação acadêmica e atuação profissional, siga algumas recomendações básicas:



## Assim:

- ✓ Organize seus estudos de maneira que passem a fazer parte da sua rotina. Por exemplo, você poderá determinar um dia e horário fixos como seu “momento do estudo”;
- ✓ Procure se alimentar e se hidratar quando for estudar; lembre-se de que uma alimentação saudável pode proporcionar melhor aproveitamento do estudo;
- ✓ No material de cada Unidade, há leituras indicadas e, entre elas, artigos científicos, livros, vídeos e sites para aprofundar os conhecimentos adquiridos ao longo da Unidade. Além disso, você também encontrará sugestões de conteúdo extra no item **Material Complementar**, que ampliarão sua interpretação e auxiliarão no pleno entendimento dos temas abordados;
- ✓ Após o contato com o conteúdo proposto, participe dos debates mediados em fóruns de discussão, pois irão auxiliar a verificar o quanto você absorveu de conhecimento, além de propiciar o contato com seus colegas e tutores, o que se apresenta como rico espaço de troca de ideias e de aprendizagem.

# Tempo de Execução em Função do Tamanho da Instância

Anteriormente tivemos um contato inicial com a área de análise de algoritmos. Vimos que podemos nos deparar com algoritmos que têm o mesmo objetivo, mas que realizam seus passos em tempos diferentes.

O estudo do tempo que os algoritmos gastam para produzir uma saída é denominado de **análise de complexidade**. Por outro lado, considerando os problemas que podem ser solucionados por algoritmos, vimos que há duas classes fundamentais, uma composta por problemas solúveis computacionalmente e outra que agrupa os problemas insolúveis.

O estudo das classes de problemas que podem ser solucionados computacionalmente é denominado de **análise de computabilidade**.

Nesta unidade, nós aprofundaremos nossos estudos sobre a análise dos algoritmos. Inicialmente, vamos nos concentrar em analisar como os tamanhos das instâncias dos problemas afetam o tempo que um algoritmo demora em produzir um resultado. Esse tipo de análise é importante, pois, como bem sabemos, os computadores não possuem capacidade de processamento infinita, tão pouco são capazes de armazenar dados infinitamente. Por mais avançados que os computadores possam ser, a capacidade de processamento e a respectiva memória são recursos finitos e não são gratuitos.

Projetar algoritmos eficientes é um desafio que deve ser considerado por todos aqueles que se propõem a desenvolver aplicações. Atualmente, é muito comum a utilização de bibliotecas com funções que podem ser utilizadas no desenvolvimento de aplicações. A maioria dos desenvolvedores não se importa em como essas funções executam seu processamento, desde que sejam eficientes em atender às necessidades de sua aplicação.

Considerando uma analogia, um engenheiro civil pode utilizar um programa de computador que o auxilie nos cálculos para o dimensionamento de estruturas de concreto. Nesse sentido, não há nada de errado na utilização de ferramentas que auxiliem em determinadas tarefas, mas um bom engenheiro é capaz de realizar os cálculos sem o computador, mesmo que ele não os faça a todo momento. Nesse mesmo contexto, os desenvolvedores utilizam um conjunto de procedimentos ou funções computacionais que foram previamente construídas e não há nada errado nisso; ao contrário, esse comportamento permite um aumento de produtividade. Porém, é importante que os desenvolvedores sejam capazes de projetar seus próprios algoritmos de forma eficiente, visto que não há garantias de que para todos os problemas sempre haverá soluções prontas para o uso.

Conforme discutimos na unidade anterior, um algoritmo pode ser definido como um conjunto de passos bem definidos que transformam uma entrada em uma saída pretendida. Comumente, denominamos os dados de entrada do algoritmo de instância. Assim, uma instância é uma das diversas possibilidades de entrada. Uma característica pertinente às instâncias de um problema que é importante no contexto da análise de algoritmos é o seu respectivo tamanho.

A análise de complexidade de um algoritmo considera o tempo que cada um dos passos que o compõem gasta para executar a instrução e, também, a quantidade de vezes

que cada passo é executado. Desse modo, cada passo individual consiste de uma instrução que será executada pelo computador. Considere, por exemplo, uma instrução que estabelece que um determinado valor seja atribuído a uma variável específica. Veja que o tempo necessário para executar essa operação de atribuição é constante e dependente do *hardware*. Todas as vezes que esse tipo de operação for executado em um mesmo computador, ela gastará o mesmo tempo, por isso tempo constante. Por outro lado, se essa operação for executada em computadores com características diferentes, o tempo gasto em cada operação, também, será diferente, por isso a dependência do *hardware*.

O tamanho da instância de um problema impacta diretamente na eficiência dos algoritmos. Como descrito anteriormente, a eficiência é medida em função do custo (tempo de processamento) de uma instrução (passo do algoritmo) e da quantidade de vezes que a instrução é executada. Nesse sentido, o que determina quantas vezes a instrução é executada é o tamanho da instância. Assim, instâncias maiores levarão o algoritmo a executar as instruções mais vezes; consequentemente, o tempo total gasto pelo algoritmo também será maior.

Um exemplo clássico, sempre utilizado em cursos de computação, é o conjunto de algoritmos de ordenação. O problema de ordenação é definido formalmente por Cormen (2009) da seguinte forma:

- **Entrada:** uma sequência de  $n$  valores  $[x_1, x_2, \dots, x_n]$ ;
- **Saída:** uma permutação  $[x'_1, x'_2, \dots, x'_n]$  da sequência original de entrada, de modo que  $x'_1 \leq x'_2 \leq \dots \leq x'_n$ .

O problema de ordenação pertence à classe dos problemas solucionáveis computacionalmente e é um problema fácil, ou seja, existe uma solução em tempo polinomial. Além disso, há uma grande variedade de algoritmos corretos que solucionam o problema, considerando diferentes tempos de execução ou diferentes complexidades. Dessa forma, o problema de ordenação possibilita visualizar de maneira clara as diferenças existentes entre algoritmos mais ou menos eficientes. Nas unidades posteriores, faremos a análise de diferentes algoritmos que resolvem o problema de ordenação.

Vamos considerar o algoritmo de ordenação por inserção, conhecido como *Insertion Sort*, descrito no texto como Algoritmo 1. Trata-se de um algoritmo simples, cuja análise aqui descrita é fortemente baseada nas descrições feitas por Cormen (2009) no seu famoso livro denominado Algoritmos – Teoria e Prática.

## Algoritmo 1: algoritmo de ordenação por inserção (*Insertion Sort*)

<i>InsertionSort(V)</i>		
1. para $j \leftarrow 1$ até $ V  - 1$	$c_1$	$n$
2. $chave \leftarrow V[j]$	$c_2$	$n - 1$
3. $i \leftarrow j - 1$	$c_3$	$n - 1$
4. enquanto $i \geq 0$ e $V[i] > chave$	$c_4$	$\sum_{j=1}^{n-1} tj$
5. $V[i + 1] \leftarrow V[i]$	$c_5$	$\sum_{j=1}^{n-1} (tj - 1)$
6. $i \leftarrow i - 1$	$c_6$	$\sum_{j=1}^{n-1} (tj - 1)$
7. $V[i + 1] \leftarrow chave$	$c_7$	$n - 1$

Para cada linha de instrução, temos o respectivo tempo de execução ( $c_i$ ) (e o número de vezes que é executada (CORMEN, 2009).

Inicialmente, é importante esclarecer a notação utilizada para a descrição do algoritmo *Insertion Sort*, visto que ela será utilizada nesse curso sempre que houver a necessidade de se representar um algoritmo. A entrada, ou entradas, do algoritmo sempre será representada entre parênteses associada ao nome do algoritmo. No caso deste exemplo, a entrada é o arranjo  $V$ , cujo tamanho é denotado por  $|V|$ , o que indica o número de elemento em  $V$ . As atribuições são representadas da direita para a esquerda, nas quais a variável de atribuição é separada do valor (variável ou expressão) atribuído por uma seta neste sentido ( $\leftarrow$ ). Veja que a instrução  $a \leftarrow 5$  indica que o valor 5 está sendo atribuído à variável  $a$ . A estruturação aninhada, observada entre as instruções, evidencia sua respectiva hierarquia. Assim, considerando o Algoritmo 1, observamos que as instruções descritas pelas linhas 5 e 6 fazem parte do laço da linha 4.

Cada linha, ou instrução, descrita no Algoritmo 1 é associada a um custo ( $c_i$ ), o qual representa o tempo que se gasta para a execução daquele tipo específico de instrução e é dependente do *hardware*, conforme descrito anteriormente. Além disso, há, também, a indicação da quantidade de vezes que a instrução será executada. Essa indicação é feita em função da variável  $n$ , a qual representa o tamanho da instância ( $n = |V|$ ).

Considere a linha 1 do Algoritmo 1, a instrução descrita nessa linha é um laço no qual a variável  $j$  assumirá os valores de 1 até  $|V|-1$ . Os valores de  $j$  representam as posições no arranjo  $V$ , com exceção da primeira posição. Veja que esse laço sempre verifica o valor incrementado de  $j$  para assegurar que o mesmo não extrapole o limite definido para o laço. Assim, a última verificação do valor de  $j$  atesta que o limite foi extrapolado, o que impacta em não executar as instruções aninhadas, o que nos leva a observar que o teste do laço sempre executará uma vez mais que as demais instruções aninhadas, no nosso caso  $n$ . Além disso, há a variável  $c_1$ , que representa o tempo necessário para a execução desse tipo de instrução, o qual é dependente do *hardware* e independente de  $n$ .

De forma similar, o número de vezes que o laço da linha 4 será executado é dado pela somatória do número de execuções para cada  $t_j$ . Assim,  $t_1$  é o número de vezes que a instrução da linha 4 foi executada para  $j = 1$ ,  $t_2$  se refere às execuções para  $j = 2$  e assim sucessivamente. Veja que as instruções das linhas 5 e 6 irão ser executadas, em cada iteração, uma vez menos que a quantidade de vezes do laço (linha 4). A lógica aqui é a mesma descrita anteriormente para o laço da linha 1.

O tempo de execução do algoritmo *Insertion Sort* ( $T(n)$ ) será a somatória dos produtos entre os custos das instruções e a quantidade de vezes que a respectiva instrução será executada. Assim, temos o seguinte:

$$T(n) = c_1 n + c_2 (n - 1) + c_3 (n - 1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1} (t_j - 1) + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7 (n - 1)$$

Veja que o tempo de execução do algoritmo *Insertion Sort* em função do tamanho de sua entrada não é fácil de ser interpretado e, consequentemente, pouco útil para a classificação do algoritmo segundo sua eficiência.

A classificação de algoritmos, em função de sua respectiva eficiência, deve levar em conta qual entrada está sendo considerada e não somente seu tamanho. Por exemplo, um algoritmo de ordenação pode ser mais ou menos eficiente de acordo com a organização prévia dos elementos a serem ordenados. Se considerarmos duas instâncias de mesmo tamanho, sendo uma já ordenada e outra na qual os elementos estão em ordem inversa, podemos ter tempos diferentes de execução do algoritmo sobre essas duas entradas.

Na seção seguinte, veremos como a organização da instância de entrada pode interferir no tempo de execução do algoritmo. Trata-se dos conceitos de melhor e pior caso, além do caso médio.

## Melhor Caso, Caso Médio e Pior Caso

Vimos anteriormente que a identificação do tempo de execução de um algoritmo, particularmente o algoritmo *Insertion Sort*, é obtida por meio da análise de cada passo ou instrução que faz parte do algoritmo. Fundamentalmente, devemos considerar a somatória dos produtos dos custos individuais de cada instrução pelo número de vezes que a instrução é executada. Esse método resulta na identificação de  $T(n)$  como uma expressão complexa pouco útil para a classificação do algoritmo, segundo seu respectivo desempenho.

Veja que, para que seja possível descrever  $T(n)$  de uma maneira mais efetiva, devemos levar em consideração a organização prévia da instância de entrada. Essa organização terá um impacto importante na definição da complexidade do algoritmo em função do tamanho da entrada.

Para os algoritmos de ordenação, o que se denomina de melhor caso é quando a instância de entrada já está ordenada e não será necessário realizar nenhuma troca de posições entre os elementos no arranjo. Dessa forma, o algoritmo irá finalizar sua execução mais rapidamente, visto que os custos referentes às trocas de posições não serão realizados.

Considerando o Algoritmo 1, observa-se que as trocas são realizadas nas instruções que compõem o laço da linha 4, por meio do deslocamento à direita daqueles elementos maiores que o conteúdo da **chave**. Veja que o teste lógico do laço sempre vai verificar que  $V[i] \leq \text{chave}$ , sendo que  $i = j - 1$ , para todo  $j = 1, 2, 3, \dots, n - 1$ . Em outras palavras, o algoritmo seleciona um elemento no arranjo, inicialmente o segundo elemento, e compara-o com os elementos anteriores. Caso algum dos elementos anteriores seja maior que o elemento de comparação, então é feito o deslocamento desse elemento para a posição posterior. Como estamos tratando do melhor caso, os elementos estão previamente ordenados e não ocorrerá nenhum deslocamento dos elementos.

Nesse contexto, o número de vezes que a instrução da linha 4 será executada é  $t_j = 1$ , para  $j = 1, 2, 3, \dots, n - 1$ . Desse modo, temos que o tempo de execução do algoritmo *Insertion Sort* no melhor caso é:

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n-1) + c_7 (n-1).$$

Podemos separar os custos  $c_i$  da variável  $n$  aplicando algumas operações algébricas, da seguinte forma:

$$T(n) = c_1n + c_2n - c_2 + c_3n - c_3 + c_4n - c_4 + c_7n - c_7$$

$$T(n) = c_1n + c_2n + c_3n + c_4n + c_7n - c_2 - c_3 - c_4 - c_7$$

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7).$$

Veja que, se representarmos as somatórias dos custos por  $a = c_1 + c_2 + c_3 + c_4 + c_7$  e  $b = c_2 + c_3 + c_4 + c_7$ , podemos reescrever a equação substituindo os custos pelas variáveis  $a$  e  $b$ :

$$T(n) = an - b.$$

Finalmente, chegamos a uma função linear de  $n$ , a qual representa o tempo de execução do algoritmo *Insertion Sort* no melhor caso. Assim, podemos dizer que o tempo gasto pelo algoritmo para produzir a saída é linear ao tamanho da instância de entrada.

Os custos  $c_5$  e  $c_6$  não são considerados, quando tratamos do melhor caso, pois as instruções aninhadas ao laço da linha 4 no Algoritmo 1 nunca são executadas.

Analisamos o tempo de execução do algoritmo *Insertion Sort* quando ele recebe uma instância de entrada previamente ordenada, ou seja, o melhor caso. Vimos que nesse caso não há nenhuma troca a ser realizada, visto que todos os elementos do arranjo estão em seus devidos lugares. Por outro lado, podemos ter uma instância de entrada onde seus elementos estejam posicionados na ordem inversa daquela que desejamos, ou seja, os elementos estão organizados em ordem decrescente. Para essa configuração específica do arranjo, damos o nome de pior caso. Assim, no pior caso todos os elementos do arranjo deverão ser reposicionados.

Vimos que no melhor caso o controle do laço que contém as instruções de movimentação dos elementos é executado uma única vez; assim, para todo  $j$  temos que  $t_j = 1$ . No pior caso, temos a situação oposta, ou seja, teremos de executar o reposicionamento de todos os elementos; assim, para todo  $j$  temos que  $t_j = j$ , para  $j = 1, 2, \dots, n-1$ . Assim, cada elemento  $V[j]$  será comparado com todos os elementos do subarranjo ordenado à direita  $V[0, \dots, j-1]$ .

O próximo passo para reduzir a função  $T(n)$  a uma função que caracterize o tempo de execução do algoritmo é resolver as somatórias. Veja que, conforme descrito anteriormente, no pior caso  $t_j = j$ , assim podemos utilizar essa igualdade substituindo na primeira somatória  $\sum_{j=1}^{n-1} t_j$ , a qual fica da seguinte forma:

$$\sum_{j=1}^{n-1} j = 1 + 2 + \dots + n - 1.$$

Veja que essa somatória consiste da soma dos valores de  $j$ , os quais variam entre 1 e  $n-1$ . Podemos resolver essa somatória utilizando uma técnica que consiste em somar a sequência de soma com a mesma sequência invertida, da seguinte maneira:

$$1 + 2 + \dots + n - 1$$

$$n - 1 + n - 2 + \dots + 1.$$

Veja que se somarmos os termos correspondentes às suas respectivas posições, temos:  $(1+n-1) + (2+n-2) + \dots + (n-1+1)$ ; executando as somas dos termos, a somatória fica da seguinte forma:  $n + n + \dots + n$ . Estamos somando o valor de  $n$  um número de vezes equivalente a  $n-1$  e, por conta de utilizarmos duas vezes a sequência, temos que dividir a resultante por dois. Assim, podemos reescrever a somatória como segue:

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



Artigo sobre as somatórias, suas propriedades principais e somatórias de funções polinomiais, disponível em: <https://bit.ly/2VRGOYu>

Utilizaremos o mesmo método para resolver a somatória  $\sum_{j=1}^{n-1} t_j (t_j - 1)$ ; primeiro, substituiremos  $t_j$  por  $j$ . Assim, a somatória fica da seguinte forma:

$$\sum_{j=1}^{n-1} (j-1) = 0 + 1 + \dots + n - 2.$$

Somando a sequência original  $(0 + 1 + \dots + n - 2)$  com sua representação invertida  $(n - 2 + n - 3 + \dots + 0)$ , temos:  $(0 + n - 2) + (1 + n - 3) + \dots + (n - 2 + 0)$ . Nesse sentido, podemos dizer que temos a soma de  $n - 2$  uma quantidade de vezes igual a  $n - 1$ ; da mesma forma utilizada anteriormente, temos que dividir esse produto por dois. O resultado da resolução da somatória pode ser simplificado da seguinte maneira:

$$\begin{aligned} & \frac{(n-2)(n-1)}{2}, \\ & \frac{n^2 - n - 2n + 2}{2}, \\ & \frac{n^2 - 3n + 2}{2}, \\ & \frac{n(n-3)}{2} + 1. \end{aligned}$$

Desse modo, podemos observar que a somatória original pode ser reescrita como segue:

$$\sum_{j=1}^{n-1} (j-1) = \frac{n(n-3)}{2} + 1.$$

Após a resolução das somatórias, teremos alguns passos a serem seguidos para a simplificação da função de tempo de execução  $T(n)$ .

- **Primeiro passo:** fazer a substituição das resoluções das somatórias na função  $T(n)$ ;

$$T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n-1)}{2}\right) + c_5\left(\frac{n(n-3)}{2} + 1\right) + c_6\left(\frac{n(n-3)}{2} + 1\right) + c_7(n-1),$$

- **Segundo passo:** multiplicar as constantes  $c_4$ ,  $c_5$  e  $c_6$  pelos conteúdos dos respectivos parênteses que os acompanham:

$$T(n) = c_1n + c_2n - c_2 + c_3n - c_3 + \frac{c_4n(n-1)}{2} + \frac{c_5n(n-3)}{2} + c_5 + \frac{c_6n(n-3)}{2} + c_6 + c_7n - c_7,$$

- **Terceiro passo:** multiplicar  $c_4n$ ,  $c_5n$  e  $c_6n$  pelos conteúdos dos respectivos parênteses que os acompanham, separando as frações correspondentes:

$$T(n) = c_1n + c_2n - c_2 + c_3n - c_3 + \frac{c_4n^2}{2} - \frac{c_4}{2} + \frac{c_5n^2}{2} - \frac{3c_5n}{2} + c_5 + \frac{c_6n^2}{2} - \frac{3c_6n}{2} + c_6 + c_7n - c_7,$$

- **Quarto passo:** agrupar os valores cujos expoentes da variável  $n$  tenham o mesmo valor ( $n^2, n^1, n^0$ ):

$$T(n) = \frac{c_4n^2}{2} + \frac{c_5n^2}{2} + \frac{c_6n^2}{2} + c_1n + c_2n + c_3n - \frac{c_4n}{2} - \frac{3c_5n}{2} - \frac{3c_6n}{2} + c_7n - c_2 - c_3 + c_5 + c_6 - c_7,$$

- **Quinto passo:** colocar a variável  $n$  em evidência, de acordo com o valor dos expoentes, separando, assim, os coeficientes das variáveis:

$$T(n) = \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 - \frac{c_4}{2} - \frac{3c_5}{2} - \frac{3c_6}{2} + c_7\right)n - (c_2 + c_3 - c_5 - c_6 + c_7),$$

- **Sexto passo:** substituir os grupos de valores que multiplicam a variável  $n$  por novas constantes ( $a$ ,  $b$  e  $c$ ) que representem os coeficientes:

$$\begin{aligned} a &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right), \\ b &= \left(c_1 + c_2 + c_3 - \frac{c_4}{2} - \frac{3c_5}{2} - \frac{3c_6}{2} + c_7\right), \\ c &= (c_2 + c_3 - c_5 - c_6 + c_7), \end{aligned}$$

- **Último passo:** obter a função  $T(n)$  que representa a complexidade computacional do algoritmo *Insertion Sort*:

$$T(n) = an^2 + bn - c.$$

As novas constantes, utilizadas na representação da função, agrupam os respectivos custos computacionais que dependem do tipo da instrução e do *hardware* utilizado. Veremos, mais à frente, que quando o tamanho da instância de entrada é suficientemente grande, essas constantes não são representativas para a classificação do algoritmo.

A obtenção de uma função que represente o tempo de execução no pior caso nos indica o estabelecimento de um limite superior. Essa informação é importante, pois ela garante que o algoritmo nunca demorará mais do que o tempo estabelecido pela função.

Estudamos, até aqui, a análise de complexidade de algoritmos considerando o melhor e o pior caso. Vale ressaltar que se o resultado da análise do pior caso nos fornece um limite superior de tempo de execução, a análise do melhor caso indica o limite inferior, ou seja, o algoritmo consumirá sempre, no mínimo, aquele tempo estabelecido.

A Figura 1 apresenta um diagrama que descreve as curvas referentes a ambos os casos analisados. Note que o crescimento da curva vermelha, que representa o tempo de execução do algoritmo *Insertion Sort* no pior caso, cresce mais rapidamente que a curva verde, a qual representa o melhor caso. Para os casos em que a organização da instância de entrada não se enquadra como melhor ou pior caso, as respectivas curvas que indicam a complexidade desses casos estarão em algum lugar na área hachurada, dentro dos limites estabelecidos pelas funções quadrática e linear.

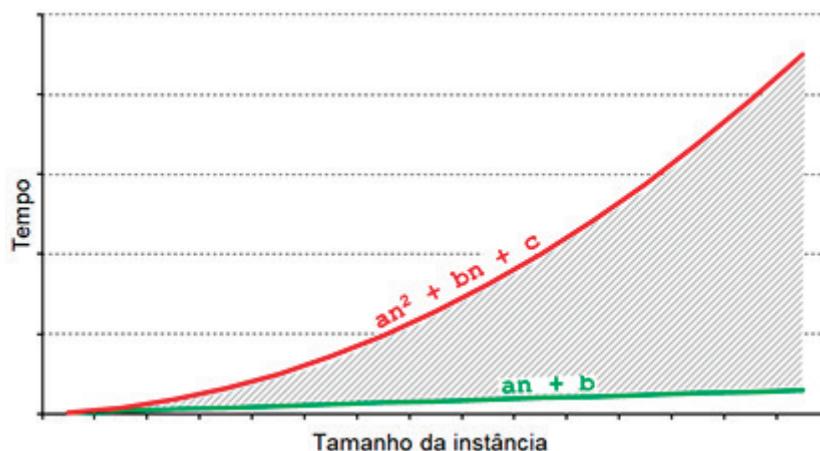


Figura 1

Diagrama com a representação do crescimento das curvas que descrevem as funções quadrática ( $an^2 + bn + c$ ) e linear ( $an + b$ ), as quais são, respectivamente, os limites superior e inferior do tempo de execução do algoritmo *Insertion Sort*.

As instâncias que não correspondem ao melhor ou pior caso são consideradas como caso médio e, em teoria, a curva traçada pela função será intermediária entre as duas anteriores. Porém, comumente o caso médio tem quase o mesmo comportamento do pior caso. Outra questão sobre o caso médio é a dificuldade em identificar uma instância que corresponda a esse caso. Por vezes, o caso médio é identificado por meio de análise estatística, considerando-se um conjunto aleatório de instâncias do problema.

Nesta seção nós aprofundamos nossas análises de complexidade de algoritmos. Vimos que um algoritmo é composto por um conjunto de passos ou instruções e que cada um desses passos apresenta um custo computacional e é executado um número determinado de vezes. Além disso, vimos, também, que a soma dos produtos entre o custo e o número de execuções de cada instrução gera uma função que, a depender da organização prévia da instância, define o tempo de execução do algoritmo em função do tamanho da entrada.

Na próxima seção, estudaremos a notação assintótica, que se define como um padrão para referenciar o modo como o tempo de execução cresce sem limitações. Esse tipo de notação é amplamente utilizado para a caracterização de algoritmos de acordo com suas entradas, desde que sejam suficientemente grandes, visto que esse tipo de notação não se ajusta a entradas pequenas.

## Notação Assintótica

O estudo do tempo de execução de algoritmos, que realizamos até o presente momento, nos possibilita entender que o tempo gasto por um algoritmo pode crescer de forma mais ou menos rápida, de acordo com o tamanho da instância de entrada e, também, da configuração inicial dessa instância. Algo que é importante de ser acrescentado é que esse tipo de análise faz sentido para instâncias grandes, não sendo efetivo para a caracterização quando tratamos pequenas entradas.

Nesse contexto, a análise do tempo de execução de algoritmos, considerando-se instâncias de entrada suficientemente grandes, é denominada análise assintótica. Esse tipo de análise não é do domínio exclusivo da computação, mas sua aplicação é principalmente no âmbito das funções. Assim, dado que estamos considerando instâncias de entradas grandes, as constantes e termos de ordem mais baixa que observamos nas funções são dominados pelo termo de maior ordem.

Considere o exemplo anterior sobre o algoritmo *Insertion Sort*, para o pior caso; vimos que a função que representa o crescimento do tempo de execução sobre o tamanho da entrada  $n$  é uma função quadrática cuja forma geral é  $T(n) = an^2 + bn + c$ . Para instâncias de entrada grandes, o crescimento da função no limite é dominado pelo primeiro termo da função. Assim, podemos dizer que a função que descreve o crescimento do referido algoritmo é  $T(n) = n^2$ . Note que desconsideramos as constantes multiplicadoras:  $a$ ,  $b$  e  $c$ , bem como os termos de menor grau:  $n^1$  e  $n^0$ . Esses elementos, desconsiderados na representação da complexidade computacional, não são determinantes para o aumento do tempo de execução de um algoritmo quando o tamanho da entrada é suficientemente grande.

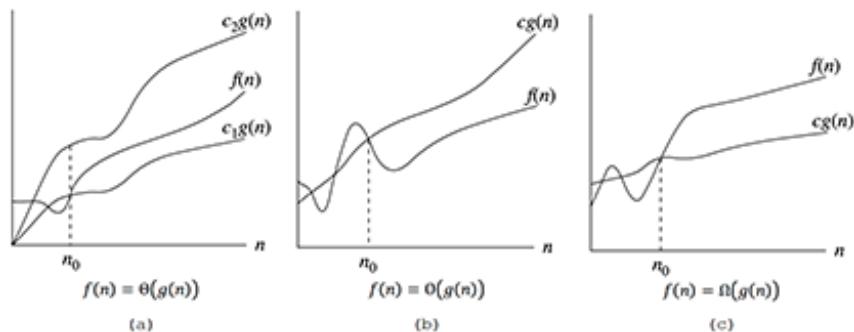


Figura 2 – Diagramas que representam o comportamento das funções  $f(n)$  em cada uma as notações assintóticas consideradas

Fonte: CORMEN, 2009

A análise da eficiência assintótica dos algoritmos considera uma notação particular que é útil para descrever o tempo de execução do algoritmo, seja qual for a organização

da instância de entrada. Assim, a notação assintótica fornece um conjunto de “categorias”, a partir das quais é possível caracterizar os algoritmos, de acordo com as respectivas eficiências computacionais, seja qual for a entrada considerada.

A notação  $\Theta$  (Theta) será a primeira que iremos estudar. Quando nos referimos à complexidade de tempo de um algoritmo, dizendo que seu tempo de execução é da ordem de  $T(n) = \Theta(g(n))$ , na verdade estamos afirmando que a função  $g(n)$  é um limite assintoticamente restrito para  $f(n)$ . Formalmente, Cormen (2009) define  $\Theta(g(n))$  como um conjunto de funções obtido da seguinte forma:

$$\Theta(g(n)) = \{f(x) | \exists c_1, c_2, n_0 > 0 \text{ sendo } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}.$$

A notação anterior define que o conjunto  $\Theta(g(n))$  é composto por funções ( $f(x)$ ) para as quais existem as constantes positivas  $c_1$ ,  $c_2$  e  $n_0$ , de modo que a taxa de crescimento das funções sejam maiores ou iguais à taxa de crescimento da função resultante do produto da constante  $c_1$  pela função  $g(n)$  e menores ou iguais à taxa de crescimento da função resultante do produto da constante  $c_2$  pela função  $g(n)$ , para quaisquer valores de  $n$  maiores ou iguais à constante  $n_0$ .

Considerando a Figura 2a, veja que a taxa de crescimento da função  $f(n)$  é menor que a taxa de  $c_2 g(n)$  e maior que a taxa de  $c_1 g(n)$  para quaisquer valores de  $n$  que sejam maiores que  $n_0$ . Existem valores à esquerda de  $n_0$  que são menores que  $c_1 g(n)$ ; assim, quando se diz que o tamanho da instância de entrada deve ser grande o suficiente, isso significa que as restrições descritas só são válidas a partir de  $n_0$ .

Quando afirmamos que uma determinada função, que representa o crescimento do tempo de execução de um algoritmo, é  $\Theta$  de alguma outra função, estamos estabelecendo limites superior e inferior para a execução do algoritmo. Desse modo, sabemos quais são os tempos mínimo e máximo de execução para aquele algoritmo.

Considere um exemplo prático<sup>1</sup>, suponha que queremos mostrar que  $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ ; para alinharmos com o estabelecido, temos que  $f(n) = \frac{1}{2}n^2 - 3n$  e  $g(n) = n^2$ . Ainda de acordo com o estabelecido anteriormente, temos que encontrar as constantes positivas  $c_1$ ,  $c_2$  e  $n_0$  que satisfaçam a inequação:

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Dividindo cada termo por  $n^2$ , obteremos o seguinte:

$$\frac{c_1 n^2}{n^2} \leq \frac{\frac{1}{2}n^2 - 3n}{n^2} \leq \frac{c_2 n^2}{n^2}$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

<sup>1</sup> Exemplo elaborado pelo professor Moacir Ponti Jr. (ICMC – USP).

Deste modo, a desigualdade do lado direito é válida para  $n \geq 1$  selecionando  $c_2 \geq \frac{1}{2}$ . Para considerarmos a desigualdade do lado esquerdo válida, fazemos  $n \geq 7$  com  $c_1 \geq \frac{1}{14}$ . Assim, podemos afirmar que para  $c_2 = \frac{1}{2}$ ,  $n_0 = 7$  e  $c_1 = \frac{1}{14}$ , temos que:

$$\frac{1}{2}n^2 - 3n = \Theta(n^2).$$

Conforme descrito anteriormente, a notação assintótica  $\Theta$  define limites superior e inferior para uma função que define o tempo de execução de um algoritmo. Nesse mesmo contexto, a notação  $O$  (pode ser lida como “big oh”) é utilizada para expressar um limite assintótico superior para uma função.

Formalmente, a notação assintótica  $O(g(n))$  é um conjunto de funções  $f(n)$  definido da seguinte forma:

$$O(g(n)) = \{f(n) | \exists c, n_0 > 0 \text{ sendo } 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}.$$

A notação anterior define que o conjunto  $O(g(n))$  é composto por funções  $f(n)$ , para as quais existem as constantes positivas  $c$  e  $n_0$ , de modo que a taxa de crescimento de  $f(n)$  seja, no máximo, igual à taxa de crescimento da função resultante do produto de  $c$  pela função  $g(n)$ , para todo  $n$  maior que  $n_0$ .

Considere a Figura 2b, veja que para todos os valores de  $n$  à direita de  $n_0$  a taxa de crescimento da função  $f(n)$  é, no máximo, igual à taxa de crescimento da função  $g(n)$ . Assim, podemos afirmar que  $f(n) = O(g(n))$ .

A forma como foi escrita a afirmação anterior, apesar de comum, é um abuso da notação de igualdade, visto que  $f(n)$  não é igual a  $O(g(n))$ , formalmente correto seria grafar como  $f(n) \in O(g(n))$ .

Vamos analisar um novo exemplo<sup>2</sup> sobre a notação assintótica  $O$ , o objetivo aqui é verificar se  $2n + 10 \in O(n)$ . Para encontrar os valores para as constantes  $c$  e  $n_0$ , poderíamos proceder da seguinte forma:

$$2n + 10 \leq cn$$

$$cn - 2n \geq 10$$

$$(c - 2)n \geq 10$$

$$n \geq \frac{10}{c - 2}.$$

Assim, como  $c - 2 > 0$ , verificamos que a inequação inicial é válida para  $c = 3$  e  $n_0 = 10$ .

<sup>2</sup> Exemplo elaborado pelo professor Moacir Ponti Jr. (ICMC – USP).

A notação  $O$  é a mais comumente utilizada, pois, na maioria dos casos, estamos buscando a complexidade de tempo no pior caso, ou seja, queremos saber qual o limite superior para o tempo de execução do algoritmo. Porém, pode haver situações nas quais seja necessário saber qual é o limite inferior, nesses casos queremos saber o comportamento mínimo de uma dada função.

Para referenciar uma função que tenha um determinado limite assintótico inferior, utilizamos a notação  $\Omega$  (Ômega). O conjunto de funções  $\Omega(g(n))$  é definido da seguinte forma:

$$\Omega(g(n)) = \{f(n) | \exists c, n_0 > 0 \text{ sendo } 0 \leq cg(n) \leq f(n) \forall n \geq n_0\}.$$

Traduzindo a notação anterior, uma função  $f(n)$  pertence a  $\Omega(g(n))$  se existirem as constantes positivas  $c$  e  $n_0$ , de modo que  $f(n) \geq g(n)$  para um valor de  $n$  suficientemente grande ( $n \geq n_0$ ).

Analizando a Figura 2c, é possível observar que a função  $f(n)$  tem uma taxa de crescimento maior ou igual à taxa de  $g(n)$ , para todo o intervalo à direita de  $n_0$ . Vamos considerar um exemplo<sup>3</sup> para a notação  $\Omega$ , queremos verificar se a função  $f(x) = 3n^2 + n$  é, no mínimo, ômega de  $g(n) = n$ , para  $n \geq n_0$ .

$$3n^2 + n \in \Omega(n)$$

$$3n^2 + n \geq cn$$

$$\frac{3n^2 + n}{n} \geq \frac{cn}{n}$$

$$3n + 1 \geq c$$

$$n \geq \frac{c-1}{3},$$

Assim, para  $n_0 = 1$  podemos selecionar  $c = 4$ , confirmando que  $3n^2 + n \in \Omega(n)$ .

A notação  $o$  (leia oh pequeno) é muito parecida com a notação  $O$  (em concordância, leia oh grande). A diferença entre as duas notações está na sua definição de um limite superior que não é assintoticamente justo. O conjunto de funções  $o(g(n))$  é definido da seguinte forma:

$$o(g(n)) = \{f(n) | \forall c > 0 \exists n_0 > 0 \text{ sendo } 0 \leq f(n) < cg(n) \forall n \geq n_0\}.$$

A leitura do conjunto anterior pode ser feita da seguinte maneira: o conjunto  $o(g(n))$  é composto por funções  $f(n)$  em que para qualquer constante positiva  $c$  existe um valor positivo para  $n_0$ , tal que a função  $f(n)$  tem taxa de crescimento estritamente menor que a taxa de crescimento da função  $cg(n)$  para um valor de  $n$  suficientemente grande.

<sup>3</sup> Exemplo elaborado pelo professor Moacir Ponti Jr. (ICMC – USP).



### Importante!

Veja que a diferença entre  $O(g(n))$  e  $o(g(n))$  está nas seguintes desigualdades:  $f(n) \leq cg(n)$  para 0 e  $f(n) < cg(n)$  para o.

Finalmente, a notação  $\omega$  (leia ômega pequeno) tem significado similar à notação  $o$ . Nesse caso, a notação  $\omega$  refere-se a um limite inferior que não é assintoticamente preciso. O conjunto de funções  $\omega(g(n))$  é definido por:

$$\omega(g(n)) = \{f(n) | \forall c > 0 \exists n_0 > 0 \text{ sendo } 0 \leq cg(n) < f(n) \forall n \geq n_0\}.$$

Podemos considerar uma analogia para que o entendimento sobre as notações assintóticas apresentadas seja mais consistente. A analogia consiste entre a comparação assintótica de duas funções  $f$  e  $g$  e dois números reais  $a$  e  $b$ . Dessa forma, podemos considerar o seguinte:

$$f(n) = O(g(n)) \Rightarrow a \leq b,$$

$$f(n) = \Omega(g(n)) \Rightarrow a \geq b,$$

$$f(n) = \Theta(g(n)) \Rightarrow a = b,$$

$$f(n) = o(g(n)) \Rightarrow a < b,$$

$$f(n) = \omega(g(n)) \Rightarrow a > b.$$

Considerando a analogia anterior, podemos dizer que  $f(n)$  é assintoticamente menor que  $g(n)$  se  $f(n) = o(g(n))$ .

Os algoritmos podem ser analisados considerando cada instrução de forma individual. Assim, o importante é evidenciar o número de vezes que a respectiva instrução será executada. Como vimos, esse processo resultará em uma função no tamanho da instância de entrada que representará o tempo de execução do respectivo algoritmo.

Podemos obter diferentes funções a partir do processo anteriormente descrito. A seguir, vamos relembrar algumas funções importantes que podem representar a complexidade de tempo de algoritmos.

- **Função constante ( $f(n) = c$ ):** nesse tipo de função, independentemente do tamanho da entrada ( $n$ ), o número de operações executadas será sempre um valor constante;
- **Função logarítmica ( $f(n) = \log n$ ):** é uma função comumente observada para algoritmos que consideram a estratégia da divisão e conquista<sup>4</sup>;
- **Função linear ( $f(n) = n$ ):** também chamada de função afim ou do primeiro grau, indica que o número de operações realizadas é linearmente proporcional ao tamanho da instância de entrada;

<sup>4</sup> A estratégia de Divisão e Conquista será abordada nas outras unidades.

- **Função quadrática ( $f(n) = n^2$ ):** algoritmos cujo tempo de execução é representado por uma função quadrática normalmente apresentam dois laços de repetição aninhados. Nesses casos, sempre que o tamanho da entrada dobra, o tempo de execução é multiplicado por quatro;
- **Função cúbica ( $f(n) = n^3$ ):** também conhecida como função do terceiro grau, define o tempo de execução de algoritmos que, comumente, apresentam três laços aninhados, como, por exemplo, na multiplicação de matrizes. Nesse caso, sempre que o tamanho da entrada dobra o tempo de execução é multiplicado por oito;
- **Função exponencial ( $f(n) = a^n$ ):** algoritmos com o tempo de execução representado por esse tipo de função são pouco úteis, pois sua complexidade cresce muito rapidamente;
- **Função factorial ( $f(n) = n!$ ):** tem um comportamento pior que as funções exponenciais; portanto, algoritmos com essa complexidade são, igualmente, pouco úteis do ponto de vista prático.

As funções anteriormente descritas estão associadas com representações que não são precisas matematicamente. Veja que uma função quadrática tem sua forma geral representada por  $f(n) = an^2 + bn + c$  e não por  $f(n) = n^2$ . Porém, no contexto da análise de algoritmos, estamos tratando de análise assintótica, o que implica tamanhos grandes de instâncias de entrada. Assim, para entradas suficientemente grandes, as variáveis de menor grau são desconsideradas, bem como os coeficientes de multiplicação. Esses elementos são dominados pela variável de maior grau, neste contexto específico.



Para relembrar os conceitos de funções, assista às videoaulas sobre o tema,  
disponível em: <https://youtu.be/SPZqQ5qn3P0>

# Material Complementar

Indicações para saber mais sobre os assuntos abordados nesta Unidade:

## ▶ Vídeos

Dica rápida de como saber a complexidade de um algoritmo

<https://youtu.be/f--A1FK6KK4>

Fundamentos para notação assintótica e contagem de instruções | Análise de Algoritmos

<https://youtu.be/uxt09pBTnzw>

O Grande – Big O & Classes de Algoritmos | Análise de Algoritmos

<https://youtu.be/Qdf3SN2ucVw>

## 📄 Leitura

Notação assintótica

<https://bit.ly/2J0oX2o>

# Referências

CORMEN, T. H. *et al.* **Introduction to algorithms**. MIT press, 2009.

TOSCANI, L. V. **Complexidade de algoritmos**, v. 13: UFRGS. 3<sup>a</sup> edição. Porto Alegre: Bookman, 2012. (*e-book*)

ZIVIANI, N. **Projeto de algoritmos**: com implementações em JAVA e C. São Paulo: Cengage Learning, 2012. (*e-book*)



**Cruzeiro do Sul**  
Educacional