Capítulo 5

Eficiência de algoritmos

Quando é preciso encontrar a solução para algum problema, geralmente se constrói um algoritmo, e um algoritmo nada mais é do que o passo a passo que conduz o caminho para a solução. Entretanto, um problema pode conter diversas soluções (NECAISE, 2010). Com diversas possibilidades para solucionar um determinado problema, é necessário descobrir quais as características que podem classificar e avaliar se um determinado algoritmo pode transitar entre diversas aplicações, ou mesmo comparar quais algoritmos são melhores que outros dentro de um determinado cenário (CORMEN *et al*., 2009; NECAISE, 2010).

Algumas das principais características que guiam os estudos com algoritmos estão relacionadas a recursos computacionais como tempo de processamento e espaço de memória ocupado. Em geral, o foco das análises sempre gira em torno do tempo de processamento de um algoritmo comparado a outro dentro de um mesmo contexto (CORMEN *et al.*, 2009).

A análise do tempo de processamento de um algoritmo depende de alguns fatores importantes (NECAISE, 2010), como:

- A quantidade de informação que irá compor a entrada do algoritmo influencia diretamente a variável do tempo de processamento.
- O hardware que é utilizado na comparação também influencia o resultado do desempenho computacional do algoritmo. Condições como a realização de tarefas simultâneas e até mesmo a hora da execução pode proporcionar tendências nas comparações.
- Outra influência é a linguagem de programação escolhida para implementar os algoritmos, pois os resultados obtidos com uma linguagem totalmente compilada são diferentes dos obtidos com uma pré-compilada ou interpretada. Além disso, existem linguagens que possuem otimizações para certos cenários, transparentes ao desenvolvedor.

Outros interesses podem surgir ao comparar algoritmos, como consumo eficiente da bateria em dispositivos móveis; a acurácia do resultado obtido ao final do processamento das informações de um determinado algoritmo; ou até mesmo o consumo de memória em microprocessadores, devido às suas limitações ou por ser um dispositivo embarcado (CORMEN *et al.*, 2009).

Ao construir um algoritmo, é importante saber que, geralmente, ele não é capaz de ter um tempo de processamento bom e de ocupar uma quantidade baixa de memória. Em linhas gerais, pode-se dizer que

quanto mais espaço um algoritmo ocupar na memória, mais rápido será seu tempo de execução dado sua entrada. Isso quer dizer que o crescimento da ocupação de memória está ligado diretamente com o menor tempo de processamento do algoritmo (CORMEN *et al*., 2009).

Sendo assim, a análise de algoritmos tem como objetivo prover ferramentas para que os desenvolvedores sejam capazes de tomar decisões sobre qual o melhor algoritmo para o problema que deve ser solucionado. A ferramenta que encabeça toda a base no âmbito da análise de algoritmos é a matemática. Ela permite que os estudos possam ir de análises em linhas gerais, quando o problema é complexo, até resultados analíticos específicos a cada problema (CORMEN *et al.*, 2009).

Ao longo deste livro, o foco da comparação dos algoritmos terá como base a relação do tempo de processamento para a conclusão de uma tarefa. Nenhum cenário específico será utilizado para não gerar uma tendência na aplicação da análise de algoritmos.

1 Matemática como ferramenta para análise de algoritmos

Existem diversos casos em que a fronteira entre a matemática e a computação se estreita, desde a fabricação de hardwares até a construção de softwares. Na construção de softwares, esse estreitamento está relacionado tanto ao que existe por trás do código, a linguagem de máquina, como à lógica, e também representa as principais funções de comparação entre os algoritmos e suas definições formais.

Ao todo, pode-se dizer que existem sete funções comumente encontradas em toda a literatura de análise de algoritmos. Verifica-se cada uma delas e ao final um comparativo é feito sobre quais são as melhores e qual seria o alvo para buscar o resultado ótimo de acordo com o cenário de aplicação (GOODRICH; TAMASSIA, 2013).

1.1 Função constante

Com absoluta certeza, a função mais simples dentre todas as sete comuns é a função constante. Sua definição formal é dada através da equação:

$$f(n) = c$$

Em que *c* é um número fixo, não se alterando ao longo das variações de *n*.

Como o objetivo da análise é dado por funções que retornam um número inteiro, a função constante mais utilizada nas análises, nesse caso, é *f*(*n*) = 1 (GOODRICH; TAMASSIA, 2013). Além disso, a função constante pode ser relacionada a uma segunda função, como se fosse um peso para o resultado dela. Veja a representação formal:

$$f(n) = c \cdot g(n)$$

A função constante, então, costuma representar a quantidade de passos que um algoritmo executa em uma determinada aplicação.

1.2 Função logarítmica

Entre as sete funções, a que se demonstra mais onipresente é a função logarítmica (GOODRICH; TAMASSIA, 2013). Sua definição formal é dada pela seguinte equação:

$$f(n) = \log_b n$$

Para qualquer constante *b* > 1, sendo *b* definido como a base do logaritmo.

Uma regra importante para essa função é que, por definição, o $\log_b 1 = 0$. É possível efetuar uma aproximação efetiva para uma função de logaritmo. A função logarítmica é a quantidade de vezes que se pode dividir o *n* por *b* até chegar ao valor 1 ou próximo. Observe o exemplo elaborado por Goodrich e Tamassia (2013):

$$\log_3 27 \rightarrow 27 / 3 / 3 / 3 = 1$$

O resultado da função logarítmica de 27 na base 3 é igual a 3, uma vez que, para alcançar o valor 1, foram necessárias 3 divisões de 27 por 3. Veja outro exemplo:

$$\log_2 12 \rightarrow 12 / 2 / 2 / 2 / 2 = 0{,}75 \leq 1$$

Nesse caso, a divisão ficou abaixo de 1 e não exatamente 1. Por aproximação, considera-se que o resultado da função logarítmica de 12 na base 2 é igual a 4. Na base 2, aceita-se a aproximação, uma vez que diversos algoritmos utilizam a estratégia de divisão e conquista, quebrando sempre entre dois subproblemas o problema original (GOODRICH; TAMASSIA, 2013). Como a base 2 é a mais comum ao fazer uma análise de algoritmo, toda vez que é mencionada a função log *n* subentende-se que se trata de um log na base 2.

IMPORTANTE

O log é a quantidade de vezes que se pode dividir um determinado número pela base até que o resultado das divisões subsequentes seja próximo ou igual a 1.

Com esse pequeno truque matemático, é possível chegar ao valor de um logaritmo sem a necessidade de nenhum cálculo complexo.

1.3 Função linear

Assim como a função constante, a função linear também é uma função simples e igualmente importante no processo de análise de algoritmos (GOODRICH; TAMASSIA, 2013). Sua definição formal é dada pela seguinte equação:

$$f(n) = n$$

Isso significa que toda entrada *n* informada à função resultará em uma saída igual à entrada. Essa função é facilmente encontrada em diversos cenários da computação. Toda vez que é executada uma operação para *n* elementos, por exemplo, no caso de operações que percorrem um vetor por completo, a função que descreve essa operação é justamente a função linear (GOODRICH; TAMASSIA, 2013).

1.4 Função *n* log *n*

A função *n* log *n* é uma função que possui uma taxa de crescimento igual ao valor de sua entrada multiplicado pelo valor de log *n* na base 2, conforme a definição formal (GOODRICH; TAMASSIA, 2013):

$$f(n) = n \cdot \log n$$

Ela possui uma taxa de crescimento superior à função linear, mas é bem menor quando comparada a funções polinomiais de grau 2 ou mais. A função *n* log *n* é o alvo para pesquisadores que querem otimizar um problema de ordem quadrática (GOODRICH; TAMASSIA, 2013).

1.5 Função quadrática

Quando a entrada *n* de uma função *f* atribui à sua saída o produto de *n* pelo próprio *n*, obtém-se uma função quadrática. Confira sua definição formal na equação a seguir:

$$f(n) = n^2$$

Isso ocorre, dentro do cenário da computação, geralmente em algoritmos que possuem laços de repetição aninhados. Em outras palavras, isso quer dizer que existe um laço de repetição externo que executa *n* vezes, linearmente, e um outro laço de repetição interno que também executa linearmente *n* vezes (GOODRICH; TAMASSIA, 2013).

Apesar de ser uma função polinomial, assim como a linear, a função quadrática é recorrente dentro do processo de análise de algoritmos.

1.6 Função cúbica e demais polinomiais

Assim como a função quadrática, a função cúbica atribui o produto de uma entrada *n* pelo próprio *n*, só que, nesse caso, três vezes. A função cúbica é menos comum no cenário de análise de algoritmos. Sua definição formal está representada na equação a seguir:

$$f(n) = n^3$$

Contudo, durante a análise de algoritmos, outras funções polinomiais, com graus maiores que a linear, a quadrática e a cúbica, podem ser encontradas. De modo geral, um polinômio é definido formalmente da seguinte maneira:

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d$$

Em que:

- a_0, a_1, ..., a_d são os coeficientes do polinômio (as constantes) e $a_d \neq 0$.
- O inteiro *d* indica a maior potência de um termo no polinômio. Ele também representa o grau do polinômio.

Com base nessa definição, pode-se unificar as funções linear, quadrática e cúbica como funções polinomiais. Porém, devido ao contexto da aplicação – a análise de algoritmos –, é conveniente mantê-las separadas, tendo em vista que seu uso é mais comum assim (GOODRICH; TAMASSIA, 2013). Funções polinomiais com grau maior ou igual a 3 devem ser evitadas. Quando são encontradas em algoritmos, devem-se realizar estudos para que este seja otimizado.

1.7 Função exponencial

A última função mais comumente encontrada em análise de algoritmos é a função exponencial. Sua definição formal é apresentada pela equação a seguir:

$$f(n) = b^n$$

Em que:

- *b* é uma constante positiva, chamada de base; e
- *n* é o expoente recebido como argumento da função.

O resultado de *f*(*n*) é obtido através da multiplicação da base por ela mesma, o número de vezes informado pela entrada *n*. Na análise de algoritmos, a base 2 é tida como a mais comum. Portanto, em geral, quando se fala de função expoente ou exponencial, subentende-se que se trata da função $f(n) = 2^n$ (GOODRICH; TAMASSIA, 2013).

2 Comparativo das taxas de crescimento

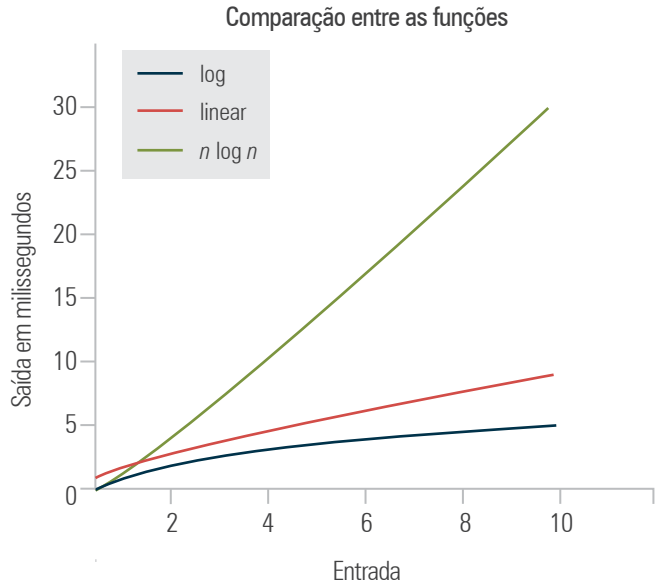
A funções podem ser organizadas dentro de uma tabela que demonstra a sequência hierárquica de uma função de acordo com o tempo de processamento, indo do menor tempo até o mais demorado entre elas. O quadro 1 apresenta a sequência hierárquica mencionada, sendo a coluna da extrema esquerda a função mais rápida e a da extrema direita, a mais demorada.

Quadro 1 – Tabela representando as sete funções para a análise de algoritmos, sendo da mais rápida até a mais demorada, da esquerda para a direita

CONSTANTE	LOGARÍTMICA	LINEAR	*N* LOG *N*	QUADRÁTICA	CÚBICA	EXPONENCIAL
1	$\log n$	n	$n \log n$	n^2	n^3	2^n

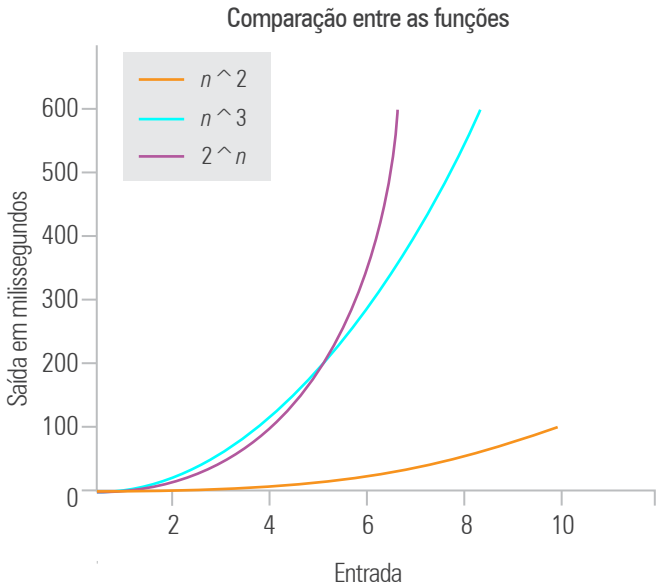
Essas taxas de crescimento podem ser utilizadas para definir as métricas de atuação, dependendo da aplicação. Por exemplo, quando se trata de estrutura de dados, existe um desejo de que as operações executadas sejam todas entre as funções constante e logarítmica. Já para grande parte dos algoritmos, o desejo é por um processamento linear ou $n \log n$. As funções polinomiais de grau 2 e 3 ainda são toleradas em casos muito específicos, mas não são desejadas, nem são alvo de qualquer desenvolvedor de algoritmos. Já a função exponencial é, computacionalmente, impraticável. Soluções exponenciais podem ser aceitas para entradas muito pequenas, mas, no geral, devem ser evitadas e, quando ocorrem, estudos de como otimizá-las devem ser conduzidos (GOODRICH; TAMASSIA, 2013). Os gráficos 1, 2 e 3 apresentam a comparação entre as taxas de crescimento. O gráfico 1 apresenta as funções logarítmica, linear e $n \log n$. A função constante foi omitida por sua representação ser uma linha constante na horizontal. O gráfico 2 tem as funções quadrática, cúbica e exponencial. O gráfico 3 apresenta uma comparação entre elas.

Gráfico 1 – Apresentação gráfica das taxas de crescimento das funções logarítmica, linear e *n* log *n* para comparação visual



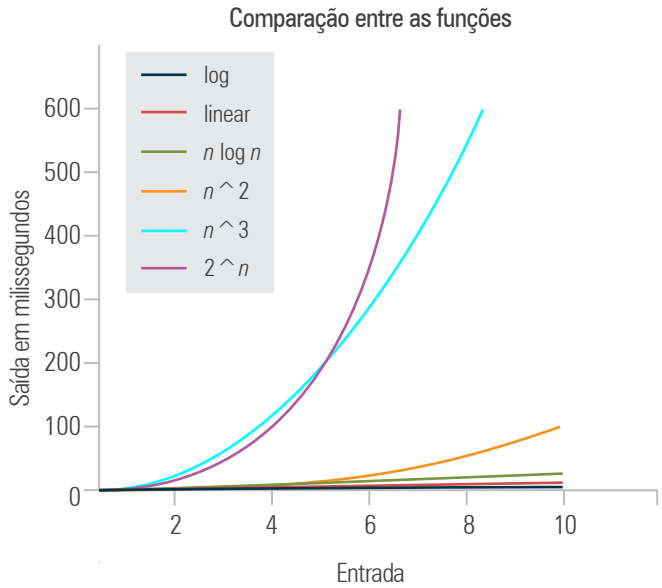
Veja no gráfico 1 como a taxa de crescimento logarítmica (linha azul) se mantém bem abaixo da função linear (linha vermelha). Para uma entrada igual a ou menor que 2, a função *n* log *n* (linha verde) também se mantém com uma taxa de crescimento menor que a linear. A partir de uma entrada *n* maior que 2 o crescimento já começa a aumentar. Para efeito comparativo, com uma entrada igual a 10, a função logarítmica possui uma saída de aproximadamente 2 milissegundos; a função linear atinge o processamento de 10 milissegundos; e a função *n* log *n* está com o processamento em torno de 35 milissegundos. Obviamente, milissegundo não é uma medida tão significativa com a qual se preocupar em relação ao desempenho do algoritmo. Mas, em um cenário escalável, o tempo de processamento começa a influenciar. A comparação continua com o gráfico 2 para as demais funções.

Gráfico 2 – Apresentação gráfica das taxas de crescimento das funções quadrática, cúbica e exponencial para comparação visual



A função quadrática (linha laranja) parece até uma boa escolha quando comparada com as funções cúbicas (linha azul) e exponencial (linha roxa). Porém, ao observar sua saída para a entrada de 10 elementos, ela levou 100 milissegundos para chegar ao resultado. Para uma melhor visualização do comportamento das funções, o gráfico ficou limitado à saída de 600 milissegundos, mas a função cúbica alcança o processamento de 512 milissegundos já com 8 elementos de entrada. A função exponencial com 10 elementos de entrada supera os 20.000 milissegundos de processamento. Note que, se um algoritmo for escalado para uma entrada de 1.000 registros, essas funções ficam impraticáveis. Por isso o desejo de sempre manter um algoritmo no máximo com a função $n \log n$. O gráfico 3 apresenta todas as funções para uma melhor visualização dessa justificativa.

Gráfico 3 – Apresentação da comparação entre todas as funções para análise de algoritmos



No gráfico 3, é possível notar que a função logarítmica quase desaparece na escala com a função exponencial, mesmo limitada a 600 milissegundos. Com esse gráfico, nota-se o comportamento de um algoritmo de acordo com a sua implementação. Por isso, a construção de qualquer algoritmo deve ser planejada e pensada com calma.

Outro ponto importante é que, ao fazer uma análise de algoritmos, o intuito é determinar qual deles é o melhor. Como essa comparação geralmente é feita levando em consideração o tempo de processamento deles, a variável da taxa de crescimento da função passa a ser um fator de qualidade do algoritmo (GOODRICH; TAMASSIA, 2013).

3 Identificando a função do algoritmo

Como discutido nos tópicos anteriores, uma das maneiras de obter o tempo de processamento é realizando a medida ao executar o

algoritmo. Mas existem diversas afirmações sobre por que mensurar com base na execução não é uma boa ideia. Sendo assim, como encontrar a função que representa o algoritmo, de um modo que não dependa do equipamento e que seja a mais fidedigna possível? (CORMEN *et al*., 2009; NECAISE, 2010).

Para solucionar esse problema, é recomendável criar medidas mensuráveis e verificar que partes do algoritmo afetam diretamente o tempo de processamento das entradas. Assim, é possível identificar a ordem de magnitude daquele algoritmo, em outras palavras, a função real que representa as operações por ele realizadas (NECAISE, 2010).

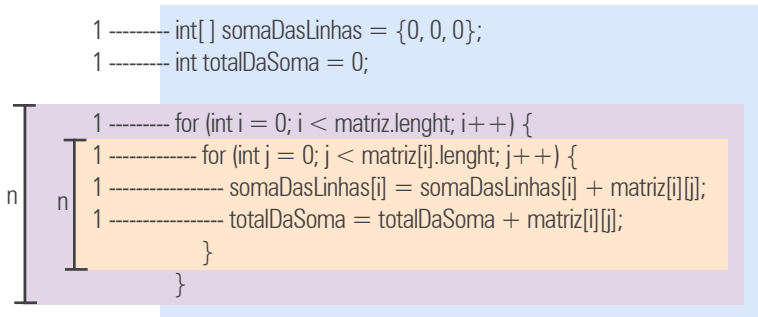
O primeiro passo é estipular o problema, implementar o algoritmo e, na sequência, fazer a sua análise, linha a linha. Como exemplo, é realizada a análise de uma algoritmo que implementa um somatório de uma matriz 3 por 5. Perceba que apesar de a análise ser feita em uma matriz com uma determinada dimensão, seu resultado deve ser genérico o suficiente para ser aplicado para quaisquer valores de entrada. A classe SomatorioMatriz é apresentada a seguir:

```
1.  public class SomatorioMatriz {
2.
3.      public static void main(String[] args) {
4.
5.              int[][] matriz = {
6.                              {1, 2, 3, 4, 5},
7.                              {6, 7, 8, 9, 10},
8.                              {11, 12, 13, 14, 15}
9.              };
10.
11.             int[] somaDasLinhas = {0, 0, 0};
12.             int totalDaSoma = 0;
13.
14.             for(int i = 0; i < matriz.length; i++) {
15.                     for(int j = 0; j < matriz[i].length;
j++) {
```

```
16.                         somaDasLinhas[i] =
somaDasLinhas[i] + matriz[i][j];
17.                         totalDaSoma = totalDaSoma +
matriz[i][j];
18.                 }
19.             }
20.
21.             System.out.printf("O soma total eh: %d\n",
totalDaSoma);
22.     }
23. }
```

Na classe SomatorioMatriz, o algoritmo está definido dentro do método *main*, realizando desde as declarações das variáveis auxiliares até o processamento do resultado final, entre as linhas 11 e 19. Da linha 5 até a 9, é declarada a matriz de entrada para o algoritmo. A figura 1 apresenta o trecho de código entre as linhas 11 e 19, com destaque para entender melhor o processo de análise do algoritmo.

Figura 1 – Destaque do valor de cada operação e bloco de instrução na análise do algoritmo da somatória de matrizes



Na figura 1, é apresentada a análise do algoritmo para efetuar a soma entre todos os elementos de uma matriz. Observe, na parte azul, que existe a declaração de duas variáveis, independentemente do espaço de memória que vão ocupar, cada uma dessas operações é executada em um tempo constante 1. Na sequência, temos o primeiro *for* (bloco roxo).

A linha do *for*, especificamente, leva um tempo constante 1 para ser executada, porém essa linha é executada *n* vezes, sendo a quantidade de linhas da matriz. Dentro desse *for*, existe um outro *for* (bloco laranja), que também tem o mesmo padrão de execução. Contudo, o segundo *for* está aninhado ao primeiro, portanto ele executa *n* vezes multiplicadas por *n*. Essa análise é feita pois, para cada iteração do *for* externo, ele executará seu bloco *n* vezes.

Por fim, dentro do segundo *for*, existem duas operações sendo executadas a um tempo constante 1 cada. Juntando as operações de cada bloco e linha, temos uma função n^2 que executa duas operações constantes. A equação dos blocos *for* é definida como $2n^2$. Somando o tempo constante das duas operações isoladas no bloco azul, a equação desse algoritmo fica:

$$2n^2 + 2$$

Agora, pode-se dizer que o algoritmo de somatório de matrizes tem uma função $f(n) = 2n^2 + 2$. Por uma questão de simplificação, é dito, em termos gerais, que esse é um algoritmo com função n^2. Omitem-se as constantes, com o intuito de simplificar a análise. Apenas em algoritmos de mesma grandeza que esse tipo de detalhe é levado em consideração, para que uma classificação mais precisa seja determinada (CORMEN *et al*., 2009; NECAISE, 2010).

4 Notação Big-O

Em geral, como visto ao longo do capítulo, a classificação do algoritmo é realizada com base na sua ordem de magnitude, ou seja, a função que o representa. A preferência pela ordem de magnitude como medida de classificação ocorre porque ela pode ser utilizada tanto para o tempo de processamento como para a ocupação de espaço na memória (DEITEL; DEITEL, 2008).

A análise feita para atribuir a ordem de magnitude do algoritmo é chamada de análise assintótica. Para facilitar a classificação de um algoritmo em um determinado ranking, cunhou-se um termo específico e bem comum em qualquer literatura: notação Big-O. Em análise de algoritmos, devemos ler a notação Big-O como "em ordem de" (HARDY; LITTLEWOOD, 1914; KNUTH, 1976). A função *f*(*n*) determina a taxa de crescimento de um algoritmo baseando-se no aumento do número de elementos que compõem a entrada *n* (CORMEN *et al*., 2009; NECAISE, 2010).

A complexidade de um algoritmo é determinada com base na função de crescimento. Então, após a conclusão da análise, é dito que o algoritmo executa suas operações na ordem de *f*(*n*). A notação formal para essa expressão é dada por O(*f*(*n*)) (NECAISE, 2010).

Quando uma análise assintótica é realizada em um algoritmo, o objetivo principal é encontrar a função que determine o limite superior do tempo de processamento do algoritmo. O limite superior também é conhecido como o cenário de pior caso de um algoritmo. Pense que, se o algoritmo tiver um bom desempenho dentro do pior cenário possível, ele irá consegui processar as informações dos cenários menos complexos de uma maneira muito mais tranquila (CORMEN *et al*., 2009; NECAISE, 2010).

A notação Big-O representa justamente o limite superior, baseando-se no grau mais alto encontrado para *n* (DEITEL; DEITEL, 2008). Dois algoritmos podem ter a mesma taxa de crescimento baseando-se na função *f*(*n*), porém, em uma análise mais detalhada, descobre-se que existe um valor constante que irá determinar uma classificação melhor ou pior. Essa constante apresentada como fator determinante para definir qual o melhor algoritmo é chamada de constante da proporcionalidade (NECAISE, 2010).

Para auxiliar o processo de análise, algumas considerações devem ser mencionadas. Considere que todas as operações (linhas de código) entram na avaliação de um algoritmo, seja uma simples condição lógica ou operações aritméticas. Nesse ponto, assume-se que quaisquer operações básicas ou declarações levam o mesmo tempo de processamento.

Em um nível mais abstrato, segundo a terminologia, ele passa a ser chamado de tempo constante (DEITEL; DEITEL, 2008; NECAISE, 2010).

Os passos do algoritmo que possuem o tempo constante, geralmente, são omitidos, porque são representados através da declaração da constante da proporcionalidade inserida dentro da definição da equação. A partir desse momento, a classificação de um algoritmo é realizada pelo termo dominante da função. Esse termo é tão grande e com valores de entradas *n* tão altos que os demais termos da função podem ser ignorados. Por exemplo, com a equação $2n^2 + 15n + 500$, o termo n^2 torna-se dominante na equação inteira, fazendo com que os demais termos tenham uma relevância muito baixa e sejam insignificantes ao resultado final (CORMEN *et al.*, 2009; NECAISE, 2010). Daí em diante, toda a classificação dos algoritmos é realizada através da notação Big-O (NECAISE, 2010).

PARA PENSAR

Comumente, operações realizadas com a classe String possuem o tempo de processamento de acordo com o seu tamanho. Contudo, durante a análise, assume-se que a String possui um valor constante na operação.

5 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Qual função representa o algoritmo abaixo?

```
// n é informado pelo usuário
n = ?
cont = 0
i = 1
while(i <= n) {
    print i
```

```
    i = i + 1
    cont = cont + 1
}
print cont
```

2. Qual função representa o algoritmo abaixo?

```
// n é informado pelo usuário
n = ?
cont = 0
i = 1
while(i <= n) {
    print i
    i = i * 2
    cont = cont + 1
}
print cont
```

3. Qual função representa o algoritmo abaixo?

```
// n é informado pelo usuário
i = 1
j = 1
while(i <= n) {
    i = i * 2
}

while(j <= n) {
     j = j + 1
}
```

Considerações finais

Neste capítulo, foram apresentadas as diversas maneiras pelas quais um algoritmo pode ser avaliado e classificado. Além disso, apresentamos as sete funções mais comuns para análise de algoritmos, as quais devem ser aplicadas no dia a dia. Cada função apresentada na análise dos algoritmos é utilizada na notação Big-O para saber qual algoritmo é melhor para cada situação.

Variáveis como consumo de bateria, ciclo de processamento, velocidade e armazenamento podem ser fatores de classificação do algoritmo, utilizando a notação Big-O. E foram apresentadas também quais são as funções em cada tipo de aplicação de algoritmos e estrutura de dados.

Referências

CORMEN, Thomas H. *et al*. **Introduction to algorithms**. Cambridge: MIT Press, 2009.

DEITEL, Paul J.; DEITEL, Harvey M. **Java**: como programar. São Paulo: Pearson, 2008.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman Editora, 2013.

HARDY, G. H.; LITTLEWOOD, J. E. Some problems of Diophantine approximation. **Acta Mathematica**, v. 37, n. 1, 1914.

KNUTH, Donald. Big Omicron and big Omega and big Theta. **SIGACT News**, apr./june 1976.

NECAISE, Rance D. **Data structures and algorithms using Python**. Hoboken: John Wiley & Sons, Inc., 2010.

XAVIER, Gley F. C. **Lógica de programação**. São Paulo: Editora Senac São Paulo, 2014.