Algoritmos
e seus Fundamentos

Sanderson L. Gonzaga de Oliveira

EDITORA
UFLA
UNIVERSIDADE FEDERAL DE LAVRAS

Algoritmos e seus fundamentos

Sanderson L. Gonzaga de Oliveira
Professor do Departamento de Ciência da Computação
da Universidade Federal de Lavras

Algoritmos e seus fundamentos

EDITORA UFLA
UNIVERSIDADE FEDERAL DE LAVRAS
Lavras - MG
2011

Impresso no Brasil – ISBN: 978-85-87672-97-9

UNIVERSIDADE FEDERAL DE LAVRAS
Reitor: Antônio Nazareno Guimarães Mendes
Vice-Reitor: José Roberto Soares Scolforo

Editora UFLA
Campus Universitário - Pavilhão 6 - Nave 2
Caixa Postal 3037 - 37200-000 - Lavras - MG
Tel.: (35) 3829-1532 - Fax: (35) 3829-1551
E-mail: editora@editora.ufla.br
Homepage: www.editora.ufla.br

Diretoria Executiva: Renato Paiva (Diretor)
Conselho Editorial: Renato Paiva (Presidente), Brígida de Souza, Flávio Meira Borém, Joelma Pereira e Luiz Antônio Augusto Gomes
Administração: Sebastião Gonçalves Filho
Secretaria: Mariana C. Alonso
Comercial/Financeiro: Douglas S. Pires, Glaucyane Paula A. Ramos e Quele P. de Gois
Revisão de português: Eveline de Oliveira
Referências Bibliográficas: Sanderson L. Gonzaga de Oliveira
Editoração Eletrônica: Sanderson L. Gonzaga de Oliveira
Capa: Fernanda C. Pereira com foto de Sanderson L. Gonzaga de Oliveira

Ficha Catalográfica Preparada pela Divisão de Processos Técnicos da Biblioteca Central da UFLA

Oliveira, Sanderson Lincohn Gonzaga de.
Algoritmos e seus fundamentos / Sanderson L. Gonzaga de Oliveira. - Lavras : UFLA, 2011.
420p. : il. ; 17,5 x 23 cm.

ISBN 978-85-87672-97-9

1. Algoritmos. 2. Dados - Estruturas (Ciência da computação). I. Universidade Federal de Lavras. II. Título.

CDD-005.1

Sumário

1 Introdução

De acordo com Knuth (1968, p. 1), a palavra *algoritmo* deriva do nome de Abu Ja'far Muhammad ibn M*ûsâ al-Khwârizmî*. Literalmente, o nome significa "Pai de Ja'far, Mohammed, filho de Moses, nativo de Khwarezm". Provavelmente, Al-Khw*â*rizm*î* nasceu entre 780 e 790 e faleceu entre 835 e 850 d.C. A palavra *algarismo* (ou dígito) também deriva do nome de al-Khw*â*rizm*î*.

O conhecimento humano está muito longe de compreender o modo de trabalho da mente humana, mas há razões para se acreditar que uma das maneiras pelas quais a mente humana realiza seus processos é executando algoritmos (DAVIS, 2004). Os algoritmos são profundamente relacionados a *números*, invenções da mente humana. Por sua vez, os números são relacionados a algum sistema de numeração. E o *sistema decimal*, conhecido também como Sistema de Numeração Indo-Arábico, é o sistema de numeração contemporâneo mais utilizado. O sistema decimal é uma forma compacta e posicional de codificar números e a aritmética pode ser realizada eficientemente seguindo-se passos elementares. Um sistema de numeração posicional baseia-se no princípio de que o valor do algarismo é determinado por sua posição na escrita dos números. Acredita-se que o formato atual do sistema decimal tenha sido inventado por matemáticos indianos, por volta do século V d.C. O sistema decimal revolucionou o raciocínio quantitativo (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2009, p. 2). Veja Ifrah (1997, p. 78-81) para uma descrição sobre as vantagens e os inconvenientes da base decimal.

Provavelmente em 825, Al-Khw*â*rizm*î* escreveu um livro, cujo título pode ser traduzido como *Sobre Cálculo com Números Indianos*, que era um tratado sobre o sistema decimal. Nesse tratado, Al-Khw*â*rizm*î* começou a popularizar de forma intensa as operações aritméticas básicas, inclusive extrair raiz quadrada, cálculos de π e o con-

ceito do zero (por exemplo, não constava o zero nos números romanos). Os métodos apresentados nesse tratado eram efetivos, isto é, determinísticos, finitos, precisos, não ambíguos, mecânicos, eficientes e corretos: eram *algoritmos* (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2009, p. 2).

No século XII, o tratado de Al-Khwârizmî foi traduzido para o latim com o título *Algoritmi de numero Indorum* e influenciou na introdução do sistema decimal no Ocidente. Este título significa *Algoritmi sobre a Arte Indiana de Contagem*, em que *Algoritmi* foi uma transcrição latina de Al-Khwârizmî. No entanto, a palavra gradualmente mudou de pronúncia e significado. Informalmente, o significado atual de algoritmo corresponde a *método de computar*, ou seja, grosso modo, é um conjunto de regras pré-estabelecidas para resolver um problema em um número finito de passos.

É importante citar que foi Leonardo de Pisa, ou Fibonacci, com o livro *Liber Abaci* (O Livro do Cálculo), de 1202, quem, principalmente, popularizou a aritmética, o sistema decimal, os números arábicos (como o sistema decimal foi chamado pelos europeus) e o uso do zero no Ocidente. Atualmente, Fibonacci é mais conhecido pela Série de Fibonacci. Como exemplos, ver Gersting (2004, p. 97-99) ou Dasgupta, Papadimitriou e Vazirani (2009, p. 2-6) para descrições detalhadas sobre essa série.

Após essa breve descrição histórica, a seguir descreve-se uma noção intuitiva de algoritmo. A noção intuitiva de *algoritmo* pode ser descrita como um método efetivo, iterativo, determinístico e finito para resolver um *problema computacional* bem definido p de instruções bem definidas, isto é, não ambíguas. Um algoritmo obtém algum valor, ou conjunto de valores, como entrada e computa (ou transforma para) algum valor, ou conjunto de valores, como saída. Informalmente, um problema é uma questão geral e relevante que merece ser respondida. A expressão do problema especifica, em termos gerais, as relações entre as entradas e as saídas desejadas. *Um algoritmo resolve um problema computacional se pode ser aplicado para qualquer instância do problema e garante produzir sempre uma solução para cada dada instância*. Isso é abordado em detalhes no capítulo 13 deste livro. Uma instância de um problema é obtida ao se especificar valores particulares para todos os parâmetros de entrada do problema ou uma entrada para um problema em particular. Uma instância satisfaz às restrições impostas na expressão do problema e é necessária para computar a solução do problema.

É importante ter noção apropriada da descrição anterior de algoritmo. *Efetivo* é usado no sentido de ser capaz de produzir um resultado pretendido, e hábil em realizar um propósito prático (isto é, não teórico), e ser completo no sentido de ser capaz de produzir um efeito (ou efeitos) decidido (positivo ou negativo), decisivo ou desejado para cada item no domínio do problema[1]. *Iterativo* é usado no sentido de que um algoritmo segue uma sequência (claramente ordenada) precisa de passos precisos rigorosamente definida. Note que a ordem da computação é essencial para o funcionamento do algoritmo. Um algoritmo é aplicado a todas as circunstâncias possíveis que podem ocorrer no problema a que se destina. Cada passo do algoritmo deve ser sistematicamente considerado. Ainda, os critérios para cada caso devem ser bem definidos e *computáveis*. A noção intuitiva do que é computável é abordada no capítulo 14 deste livro. Grosso modo, *determinístico* é usado no sentido de que sempre que determinada entrada for aplicada, o algoritmo produzirá sempre o mesmo resultado. *Finito* é usado no sentido de que o algoritmo deve, necessariamente, parar. Para a descrição da noção intuitiva de algoritmo, quatro adjetivos foram utilizados para enfatizar alguns conceitos, mas *efetivo* poderia sumarizá-los. Em geral, o objetivo é encontrar o algoritmo mais *eficiente* para resolver o problema. Isto é, encontrar o algoritmo mais eficiente em relação aos demais algoritmos que resolvem o mesmo problema. Nesse sentido, *eficiência* envolve todos os recursos computationais necessários para a sua execução. Mas, geralmente, o mais eficiente é o mais rápido (GAREY; JOHNSON, 1979, p. 5).

1.1 Revisão de matemática básica

Neste texto, supõe-se que o leitor esteja familiarizado com conteúdos elementares de matemática, como a notação de conjuntos, utilizada em diversas partes do texto. Por exemplo, veja Gersting (2004, p. 130-141) para descrição sobre a notação por conjuntos. Entretanto, para seguir adiante nos estudos com bom aproveitamento, há uma breve descrição de alguns conceitos matemáticos elementares. Além desta seção, outros conceitos básicos são apresentados no decorrer do texto. Preferiu-se abordar conceitos específicos quando citados.

[1] Descrição baseada em Webster's (2010).

1.1.1 Técnicas de demonstração de teoremas

Uma *conjectura* é uma afirmação (ou sentença) que se supõe verdadeira, geralmente baseada em alguma evidência ou intuição, mas ainda sem *prova* de que é válida para *todos* os casos a que se destina. Uma *prova* é uma demonstração precisa, sem controvérsias, que estabelece uma verdade matemática. Se for apresentado um só exemplo de que a conjectura é falsa, esse é chamado de *contraexemplo* e a conjectura é refutada.

Basicamente, há dois tipos de raciocínios para investigação científica. Apesar de distintos, ambos não se excluem e podem ser usados simultaneamente.

- No *raciocínio indutivo*, constrói-se uma conclusão baseada em experiência e observação. Nesse raciocínio, parte-se de casos particulares (ou de uma instância base) e verifica-se a passagem para instâncias superiores para provar casos gerais, até construir o problema geral. Ou seja, o raciocínio indutivo segue da observação direta de aspectos isolados da realidade. Após os aspectos serem entendidos e com o conhecimento adquirido, estabelecem-se princípios gerais que regem o assunto em estudo. Quando há uma conjectura, podem-se mostrar exemplos de que a conjectura é verdadeira. A cada novo exemplo mostrado ser verdade, fica-se mais confiante de que a conjectura é verdadeira. Esse é um exemplo do *raciocínio indutivo*. No entanto, na prática, somente os exemplos provados serão mostrados como verdadeiros.

- No *raciocínio dedutivo*, parte-se de casos gerais para provar os casos específicos. O raciocínio dedutivo estabelece um conjunto de pressupostos gerais. Esses pressupostos são lógicos e coerentes entre si e deles obtém-se um entendimento profundo do fato específico. Ao aplicar o *raciocínio dedutivo*, aplica-se uma técnica de demonstração para se construir um *teorema*.

Se for apresentada uma demonstração (ou prova) de que a conjectura é verdadeira, a conjectura passa a ser um *teorema*, que é considerado sempre verdade. Um teorema é uma afirmação de que existe uma demonstração matemática que prova que a conjectura é verdadeira. Se a conjectura for provada falsa, então não é considerada um teorema.

Há diversas técnicas de demonstração de teoremas, nas quais estão incluídas a prova direta, a prova por contraposição, a demonstração por casos, a demonstração por exaustão, a demonstração de existência e a demonstração de unicidade. Se a demonstração não é direta, então é chamada de demonstração indireta. Veja livros de Matemática Discreta para descrições detalhadas sobre demonstrações: o primeiro capítulo de Rosen (2007) é uma opção excelente.

Na prova por contradição (que é um tipo de redução ao absurdo), supõe-se que tanto a *hipótese* como a negação da conjectura são verdadeiras. Então, tenta-se obter algumas contradições a partir dessas suposições. Considere por *hipótese* uma suposição sobre algo do qual se obtém uma consequência.

1.1.2 Relações

Considere *tupla* como uma lista ordenada de elementos. Considere o Produto Cartesiano sobre n conjuntos $X_1, \cdots, X_n$ como $X_1 \times \cdots \times X_n = \{(x_1, \cdots, x_n) : x_i \in X_i, i = 1, 2, \cdots n\}$. O Produto Cartesiano é um conjunto de n-tuplas.

De maneira geral, uma relação descreve uma conexão entre objetos. Ou seja, uma *relação* é uma propriedade que atribui valores verdade a n-tuplas de elementos. Tipicamente, a propriedade descreve uma conexão possível entre os componentes de uma n-tupla. Por exemplo, veja Gersting (2004, p. 197-231) para descrição sobre relações.

A seguir, descrevem-se os tipos de relação. Considere uma relação $R \subseteq X \times Y$, $X_p \subseteq X$ a *pré-imagem* e $Y_i \subseteq Y$ a *imagem* da relação, que são definidas adiante. Se R é uma *relação funcional*, então

$$(\forall x \in X_p)(\forall y_1, y_2 \in Y_i)((xRy_1) \wedge (xRy_2) \rightarrow y_1 = y_2).$$

Uma relação funcional é chamada de *função parcial*. Se a pré-imagem (veja descrição a seguir) é igual ao domínio da relação, isto é,

$$(\forall x \in X)(\exists y \in Y)(xRy), \tag{1.1}$$

então, a relação é *total*. Ou seja, todos os elementos do domínio são *definidos*. Se a relação é funcional e total, então, ela é chamada de *função total* ou somente *função*. Para programadores de computador, esse entendimento é fácil: ao se passar um parâmetro específico para uma função (em uma linguagem de computador como C), esta não terá duas ou mais saídas diferentes. Ou seja, a função da linguagem de computador terá a mesma saída sempre que receber o mesmo parâmetro. Isso também é chamado de *determinismo* e ocorre em uma função de linguagem de computador, a menos que o algoritmo seja probabilístico, mas isso foge ao escopo deste texto.

Nota-se que, se f é uma função $f : X \to Y$ (diz-se que a função f *leva* do conjunto X para o conjunto Y), então $f \subseteq X \times Y$. O *domínio* (ou *domínio de definição*) é o conjunto de elementos de entrada de uma função. O *contradomínio* é o conjunto de elementos de chegada de uma função. Por exemplo, f tem domínio X e contradomínio Y.

Se o par ordenado $(x, y) \in f$, então: $y = f(x)$; y é a *imagem* de x sob f; x é a *imagem inversa* (ou pré-imagem) de y sob f; f *leva* x em y (GERSTING, 2004, p. 235). Ou seja, a *imagem* é o conjunto dos elementos de chegada que a função efetivamente possui. Seja X_p a pré-imagem de f. A imagem de um subconjunto $X_p \subseteq X$ sob f é o subconjunto $Y_i \subseteq Y$ definido por

$$Y_i = \{y \in Y : (\exists x \in X_p) | (y = f(x))\}. \tag{1.2}$$

Considere $Y_i \subseteq Y$ para descrever mais especificamente a *pré-imagem* (ou *imagem reversa* ou *imagem recíproca* ou *contraimagem*) $f^{-1}(Y_i) = \{x \in X : f(x) \in Y_i\}$ de uma função $f : X \to Y$. A *pré-imagem* de uma função f é o subconjunto $X_p \subseteq X$ de elementos para os quais a função é efetivamente definida, isto é, é o subconjunto de elementos em X que têm associação com elementos da imagem de f.

Por exemplo, considere $f : \mathbb{N} \to \mathbb{R}$ definida por $f(x) = \frac{1}{x^2}$. A pré-imagem é o conjunto $\mathbb{Z}^*$ e a imagem é o conjunto dos números reais positivos (contínuos) $\mathbb{R}^+$. Veja o capítulo 4 de Gersting (2004) para uma descrição detalhada de relações, funções e suas nomenclaturas.

Considere uma função $g : S \to T$. A função g é *injetora* se

$$(\forall s_1, s_2 \in S)(\forall t \in T)(g(s_1) = t \wedge g(s_2) = t) \to s_1 = s_2. \tag{1.3}$$

Em uma função injetora, dois elementos da pré-imagem não se relacionam com o mesmo elemento da imagem.

A função f é *sobrejetora* se a imagem é igual ao contradomínio da função, isto é,

$$(\forall t \in T)(\exists s \in S)(t = g(s)). \tag{1.4}$$

Isso significa que a função é definida *sobre* a totalidade dos elementos do contradomínio. Se uma função é injetora e sobrejetora, então, a função é *bijetora*: cada elemento do domínio é definido e cada elemento relaciona-se com um só elemento do contradomínio, que também tem todos os seus elementos definidos.

1.1.3 Funções e notações elementares

A seguir são descritas algumas funções matemáticas e notações. As descrições são baseadas, principalmente, no livro de Gersting (2004, p. 233-248).

1. *Funções chão $\lfloor x \rfloor$ e teto $\lceil x \rceil$.* A função chão $c : \mathbb{R} \to \mathbb{Z}$, dada por $\lfloor x \rfloor$=max$\{a \in \mathbb{Z} | a \leq x\}$, retorna o maior inteiro menor ou igual a x. A função teto $t : \mathbb{R} \to \mathbb{Z}$, dada por $\lceil x \rceil$=min$\{a \in \mathbb{Z} | a \geq x\}$, retorna o menor inteiro maior ou igual a x.

2. *Função polinomial p(n).* Dados $g \in \mathbb{N}$ e $a_i \in \mathbb{R}$, um polinômio em n de grau g é uma função $p(n)$ na forma $p(n) = \sum_{i=0}^{g} (a_i n^i)$, em que $a_g \neq 0$.

3. *Função exponencial $y = b^n \leftrightarrow \log_b y = n$,* em que $1 \neq b > 0$ é a base, n é o expoente e y é a potência. A função exponencial possui domínio $\mathbb{R}$ e contradomínio $\mathbb{R}^+$. Sejam $m, n \in \mathbb{R}$ e $(\forall b \in \mathbb{R}^+)$:

   $b^0 = 1$ (por convenção);

   $b^1 = b,\ b^{-1} = \frac{1}{b}$;

$(b^m)^n = (b^n)^m = b^{mn}$ (ao elevar uma potência a um expoente, multiplicam-se os expoentes);

$b^m b^n = b^{m+n}$ (ao multiplicar potências de mesma base, somam-se os expoentes);

$\frac{b^m}{b^n} = b^{m-n}$ (ao dividir potências de mesma base, subtraem-se os expoentes);

$0^0 = 1$, onde conveniente.

4. *Função logarítmica* y=$\log_b x$, em que b é a base do logaritmo, x é o logaritmando ou antilogaritmo e y é o logaritmo. Lembre-se que y=$\log_b x \leftrightarrow b^y = x$. Este texto utiliza a notação $n \cdot \log_b a = \log_b a \cdot n \neq \log_b(an)$. Considere $(\forall a, b, c \in \mathbb{R}^+)$, n qualquer e a base do logaritmo não é 1:

   - $\log_b 1 = 0$
     é verdade porque, traduzindo para a forma exponencial, obtém-se $b^0 = 1$;
   - $\log_b b = 1$
     porque $b = b^1$;
   - $\log_b b^n = n$
     porque, traduzindo para a forma exponencial, obtém-se $b^n = b^n$;
   - $b^{\log_b a} = a$
     porque $\log_b a = \log_b a \rightarrow a = b^{\log_b a}$; ou $\log_b a = x \rightarrow a = b^x \therefore b^{\log_b a} = a$. Também pode-se aplicar $\log_b$ em ambos os lados e obter-se $\log_b b^{\log_b a} = \log_b a \rightarrow \log_b a \cdot \cancel{\log_b b}^{1} = \log_b a$ (usando a propriedade a seguir);
   - $\log_b(a^n) = n \cdot \log_b a$
     significa que o logaritmo de uma potência é igual ao expoente da potência multiplicado pelo logaritmo da base da potência; isso é verdade porque $a^n = b^{\log_b a \cdot n}$, em que $\log_b \frac{1}{a} = \log_b a^{-1} = -\log_b a$ é um exemplo;
   - $\log_b(ac) = \log_b a + \log_b c$
     significa que o logaritmo de um produto é a soma dos logaritmos; isso é verdade porque
     $ac = b^{\log_b a} b^{\log_b c} = b^{(\log_b a + \log_b c)} \rightarrow \log_b(ac) = \log_b a + \log_b c$;

- $\log_b(\frac{a}{c}) = \log_b a - \log_b c$

  significa que o logaritmo de um quociente é a diferença dos logaritmos; isso é verdade porque $\frac{a}{c} = \frac{b^{\log_b a}}{b^{\log_b c}} = b^{(\log_b a - \log_b c)} \rightarrow \log_b(\frac{a}{c}) = \log_b a - \log_b c$;

- $\log_b a = \frac{\log_c a}{\log_c b}$

  essa fórmula para mudança de base é verdade porque $a = b^{\log_b a} \rightarrow \log_c a = \log_c b^{\log_b a} = \log_c b \cdot \log_b a \rightarrow \log_b a = \frac{\log_c a}{\log_c b}$; por exemplo, $\log_2 3 = \frac{\log_{10} 3}{\log_{10} 2}$;

- $\log_b a = \frac{1}{\log_a b}$

  isso é verdade porque $a = a^{\log_a b \cdot \log_b a} \rightarrow \log_a a = \log_a a^{\log_a b \cdot \log_b a} = \log_a b \cdot \log_b a \rightarrow \log_b a = \frac{1}{\log_a b}$; ou pela fórmula para mudança de base, pode-se usar o próprio logaritmando como a nova base;

- $a^{\log_b c} = c^{\log_b a}$

  significa que uma potência, em que o expoente é um logaritmo, é igual a se trocar a base da potência com o logaritmando que consta no expoente; isso é verdade porque $c = b^{\log_b a \cdot \log_a c} \rightarrow \log_b c = \log_a a^{\log_b c} = \log_b a \cdot \log_a c = \log_a c^{\log_b a} \rightarrow a^{\log_b c} = c^{\log_b a}$.

Quando aparece lg n, deve-se perceber qual a base do logaritmo no contexto. Muitas vezes, em Ciência da Computação, ocorre que $\lg n = \log_2 n$ porque são comuns algoritmos que dividem o número de itens da entrada pela metade a cada chamada recursiva. Mas geralmente, quando a base do logaritmo não está explícita, ela não importa. A característica de algoritmos dividirem o número de itens da entrada pela metada a cada passo do algoritmo é, às vezes, chamado de *balanceamento*.

Veja, por exemplo, Gersting (2004, p. 504), para mais detalhes sobre as propriedades da função logarítmica.

A notação deste texto utiliza $\log_b^n a = (\log_b a)^n$. A notação de produtórios $\prod$ é uma abreviação para escrever expressões de produtos de termos. Por exemplo, $\prod_{i=1}^{3} i = 1 \times 2 \times 3 = 6$. A notação de somatórios $\sum$ é uma abreviação para escrever expressões de somas de termos. Por exemplo, $\sum_{i=1}^{4} i = 1+2+3+4 = 10$.

5. *Função logarítmica iterativa* $\log_k{}^*n : \mathbb{N} \to \mathbb{N}$ é definida por $\log_k{}^*n = \min\{i \in \mathbb{N}: \log_k^{(i)} n \leq 0$, para $k \in \mathbb{N} - \{1\}\}$. Note que k é a base do logaritmo. A função $\log_k{}^*n$ é o número de vezes que a função $\log_k$ deve ser aplicada a n para se obter um valor menor ou igual a zero. Em geral, omite-se k por clareza e apresenta-se a função logarítmica iterativa como $\lg^*n$.

   Considere também uma família de funções $\mathrm{ex}_k(n) : \mathbb{N} \to \mathbb{N}$ definida por: $\mathrm{ex}_k(0)=0$ e se $n > 0$, então $\mathrm{ex}_k(n)=k^{\mathrm{ex}_k(n-1)}$. Na Tabela 1.1, observam-se alguns exemplos de retorno das funções $\log_2{}^*n$ e $\mathrm{ex}_2(n)$.

Tabela 1.1: Exemplos de $\log_2{}^*n$ e $\mathrm{ex}_2(n)$.

n	0	1	2	3	4	5	6	7	8
$\log_2{}^*n$	0	1	2	3	3	4	4	4	4
$\mathrm{ex}_2(n)$	0	1	2	4	16	2^{16}	$2^{2^{16}}$	$2^{2^{2^{16}}}$	$2^{2^{2^{65536}}}$
n	9	$\cdots$	16	17	$\cdots$	2^{16}	$2^{16}+1$	$\cdots$	$2^{2^{16}}$
$\log_2{}^*n$	4	$\cdots$	4	5	$\cdots$	5	6	$\cdots$	6

   Nota-se que a função $\mathrm{ex}_2(n)$ apresenta crescimento extremamente rápido e $\log_2{}^*n$ apresenta crescimento extremamente lento. Já que $2^{2^{16}}$ é um número um tanto grande, é provável que a maioria dos leitores raramente utilize $\lg^*n > 6$.

6. *Função fatorial* tem domínio em $\mathbb{N}$ e imagem em $\mathbb{N}^*$. Seja $n \in \mathbb{N}$,

$$n! = \begin{cases} 1 \text{ se } n = 0; \\ \prod_{i=1}^{n} i \text{ se } n > 0 \end{cases} . \tag{1.5}$$

   Acredita-se que a notação n! tenha sido introduzida por Christian Kramp, em 1808.

7. *Função módulo*. A função mod: $\mathbb{Z}^{+2} \to \mathbb{N}$ é definida como $\mathrm{mod}(x,t) = x - \lfloor \frac{x}{t} \rfloor t$.

Veja livros de Matemática Discreta para descrições detalhadas e outras provas sobre estas funções e notações. A obra de Gersting (2004) é uma excelente opção.

1.1.4 Princípio da indução matemática

Uma técnica de demonstração de teoremas muito utilizada na Ciência da Computação é o princípio da indução matemática, ou simplesmente *indução*. A indução é útil para provar asserções sobre a correção e a eficiência de algoritmos.

A indução consiste em inferir uma lei geral a partir de instâncias particulares. Entretanto, apesar de ser chamada de demonstração por *indução*, o método é dedutivo, pois não é baseado em evidências. A indução é baseada em *regras de inferências* a partir de premissas (ou hipóteses). As regras de inferências podem ser entendidas como modelos para a construção de *argumentos válidos*, em que *argumento* pode ser entendido como uma sequência de sentenças que terminam com uma conclusão, enquanto *válido* pode ser entendido que uma conclusão, ou a sentença final do argumento, deve seguir o valor-verdade hipótese do argumento (ROSEN, 2007, p. 63).

A indução matemática (ou simplesmente indução) tem sido utilizada desde tempos antigos. Manber (1989, p. 30) afirma que a descoberta da indução deve-se a Franciscus Maurolycus (1494-1575). Todavia, a primeira formulação explícita pode ter sido feita por Blaise Pascal (1623-1662), no *Traité du triangle arithmétique* entre 1653 e 1654 e publicado em 1665. Depois de ser utilizada por Jakob Bernoulli (1654-1705), a indução tornou-se conhecida por certa quantidade de pessoas.

Manber (1989, p. 113) afirma que a indução e as técnicas de provas matemáticas gerais em algoritmos tiveram origem nos fluxogramas de Goldstine e von Neumann (republicação em 1963). Ainda segundo Manber (1989, p. 113), a indução foi completamente desenvolvida no contexto de algoritmos por Floyd (1967).

Indução

Na *base da indução*, deve-se provar uma característica básica da conjectura. Em seguida, deve-se supor como verdadeira a *hipótese indutiva* que, geralmente, é a aplicação da conjectura. Finalmente, deve-se provar o *passo de indução*, que é um estágio adiante na conjectura.

Como o símbolo = é muito forte e no passo indutivo a igualdade é justamente o que se quer provar, então, deve-se utilizar $\stackrel{?}{=}$ em vez de =. No passo de indução, deve-se aplicar a hipótese indutiva para se alcançar o estágio seguinte. Isto é, deve-se tentar provar (muitas vezes, algebricamente) que o lado esquerdo é igual ao lado direito, ou vice-versa. Para a hipótese e passo indutivos, deve-se utilizar uma letra diferente da utilizada na conjectura.

Se a base é provada e prova-se como alcançar um passo adiante, então prova-se que se podem alcançar todos os casos. Em livros como de Gersting (2004, p. 76) e Rosen (2007, p. 263), os autores fazem a seguinte analogia. Considere subir uma escada infinita. Se é possível alcançar a base da escada e prova-se como é possível subir um degrau, então, prova-se que é possível subir todos os degraus. Por exemplo, seja T um teorema que tenha como parâmetro um número natural n. Para provar que T é válido para todos os valores de n a partir de uma constante c, provam-se duas condições: T é válido para $n = c$, que é o caso base e, para todo $n > c$, se T é válido para n - 1, então, T é válido para n. Provar a segunda condição é, geralmente, mais fácil que provar o teorema diretamente porque se pode usar a asserção de que T é válido para n-1. Esta afirmativa é chamada de hipótese ou passo indutivo. As condições implicam T válido para $n = c + 1$ o que, junto com a segunda condição, implica T também válido para $n = c + 2$, e assim por diante.

Fórmulas fechadas

Aplicam-se bem provas por indução ao se conjecturar *fórmulas fechadas* de somatórios. Uma fórmula fechada fornece o valor da função diretamente em termos do seu argumento. Comumente, as fórmulas fechadas contêm funções elementares "bem conhecidas". Exemplos de funções bem conhecidas são as constantes, uma variável, operações aritméticas elementares, raízes, os polinômios, quocientes de polinômios, logaritmos, exponenciais e funções trigonométricas.

Ainda, as fórmulas fechadas não contêm somatórios (lembre-se que seno(x), cosseno(x) e e^x são somatórios). Em Física e em engenharias, a terminologia usual é *solução analítica*, que é uma solução encontrada ao se avaliar funções e resolver equações por

meio, geralmente, de algebrismo. A seguir, são apresentados dois exemplos de fórmulas fechadas para somatórios, provadas por indução.

Exemplos de demonstrações por indução matemática

- *Série aritmética.*

  Há uma fórmula fechada para a série aritmética

$$\sum_{i=1}^{n} i = 1+2+\cdots+n = \frac{n(n+1)}{2}. \tag{1.6}$$

  Demonstração. A base da indução é $n = 1 : 1 = \frac{1(2)}{2}$. A hipótese indutiva é

$$1+2+\cdots+k = \frac{k(k+1)}{2}. \tag{1.7}$$

  O passo indutivo é

$$\overbrace{1+2+\cdots+k}^{\text{hipótese de indução}} +(k+1) \stackrel{?}{=} \frac{(k+1)(k+2)}{2}. \tag{1.8}$$

  Aplica-se a hipótese de indução em (1.8) e obtém-se

$$\frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{(k+1)(k+2)}{2}. \tag{1.9}$$

  Como o lado direito de (1.9) é igual ao lado direito de (1.8), completa-se a prova. □

  A equação (1.6) também pode ser provada como descrito a seguir.

  Demonstração. Ao se considerar A = 1 + 2 + $\cdots$ + (n-1) + n = n + (n-1) + $\cdots$ + 2 + 1 e A+A = 2A = n(n+1). □

- *Série geométrica (ou exponencial).*

  Seja $x \in \mathbb{R} - \{1\}$. Há uma fórmula fechada para a série geométrica

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}. \tag{1.10}$$

Demonstração. A base da indução é

$$n = 0 : 1 = \frac{x - 1}{x - 1}. \tag{1.11}$$

A hipótese indutiva é

$$1 + x + x^2 + \cdots x^k = \frac{x^{k+1} - 1}{x - 1}. \tag{1.12}$$

O passo indutivo é

$$\overbrace{1 + x + x^2 + \cdots + x^k}^{\text{hipótese de indução}} + x^{k+1} \stackrel{?}{=} \frac{x^{k+2} - 1}{x - 1}. \tag{1.13}$$

Aplica-se a hipótese de indução em (1.13) e obtém-se

$$\frac{x^{k+1} - 1}{x - 1} + x^{k+1}\frac{x - 1}{x - 1} = \frac{x^{k+1} - 1 + x^{k+2} - x^{k+1}}{x - 1} = \frac{x^{k+2} - 1}{x - 1}. \tag{1.14}$$

Como o lado direito de (1.14) é igual ao lado direito de (1.13), completa-se a prova. □

1.2 Estruturas de dados básicas

As estruturas de dados são fundamentais para os algoritmos. Em Ciência da Computação, manipula-se a informação, *o que* interessa. As informações são desmembradas em dados, armazenados na memória do sistema de computação, seja qual for o nível na hierarquia da memória do computador. Dessa forma, grosso modo, as estruturas de dados são *como* os dados são armazenados na memória do computador. Buscar estruturas de dados eficientes é um dos desafios de pesquisa na Ciência da Computação. Por

exemplo, há diferença de tempo de execução de algoritmos de busca, inserção e deleção para diferentes problemas, utilizando estruturas de dados específicas.

Neste texto, supõe-se que o leitor conheça conceitos básicos de estruturas de dados elementares e tipos de dados como inteiros, *strings* e *arrays* (ou vetor) *n*-dimensionais. As estruturas de dados básicas foram descritas pela primeira vez em livro por Knuth (1968). Após essa publicação, foi publicado um grande número de livros que descrevem as estruturas de dados básicas. A seguir, o leitor tem uma lista grande de ótimos livros em português e em inglês, nos quais pode estudar detalhes sobre as estruturas de dados básicas. Aho, Hopcroft e Ullman (1974), Aho, Hopcroft e Ullman (1983), os capítulos 3 a 5 de Horowitz e Sahni (1984), o capítulo 4 de Manber (1989), os capítulos 1 e 4 de Wirth (1989), os capítulos 2 e 3 de Sedgewick (1990), os capítulos 5 e 6 de Aho e Ullman (1992), os capítulos 10 a 12 de Cormen et al. (2009), e o capítulo 3 de Ziviani (2011) são alguns dos livros nos quais o leitor pode obter detalhes sobre os *tipos abstratos de dados elementares*, como listas, filas, pilhas e árvores.

No decorrer do texto, representam-se posições de um vetor entre colchetes como notação para esse tipo abstrato de dados. Algumas estruturas de dados elementares são brevemente descritas a seguir.

1.2.1 Listas

Um *tipo abstrato de dados lista* é um conjunto finito de elementos pertinentes ao problema em questão. Geralmente, tais elementos são *registros*. Grosso modo, registros são tipos de dados definidos pelo programador e são compostos por um ou mais tipos de dados. Inclusive, podem ocorrer outros registros previamente definidos.

Uma lista é do tipo $lista \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n \rightarrow NULL$, em que e_i representa um elemento da lista e a flecha indica que um elemento *aponta* para outro. Em geral, além de dados pertinentes ao problema, um elemento e contém um tipo de dado chamado *ponteiro*. Os elementos permanecem na memória do sistema de computação. Da mesma forma, um ponteiro ocupa uma porção de memória e seu conteúdo é o endereço de outra posição de memória. Se válido, então, esse endereço é uma referência a outro elemento da lista neste contexto. Deve-se ter um ponteiro especial *lista* do tipo de e. O ponteiro

lista aponta para o primeiro elemento da lista. O último elemento da lista não aponta para um elemento válido, chamado *NULL*, em linguagens como C e C++, e NIL, em linguagem como Pascal. Isso significa que o conteúdo do ponteiro do último elemento da lista é, geralmente, composto por *bits* em que todos os dígitos são 0. Em geral, o endereço 0 da memória é reservado ao sistema operacional ou a algum processo que possibilita executá-lo. O mesmo ocorre em sistemas multitarefas, já que *NULL* indicaria a primeira posição (ou *offset*) de memória para início do processo. Claramente, acessar tais endereços de memória é inadequado e, geralmente, resultados inesperados ocorrem ao se alterar indiscriminadamente a memória de tais regiões.

Em geral, inicia-se a lista com o ponteiro *lista* apontando para *NULL*, ou $lista \rightarrow NULL$, que é uma lista vazia. Ao se inserir um elemento e_1 na lista, aloca-se memória para e_1, atribui-se o endereço de e_1 ao conteúdo de *lista* e atribui-se *NULL* ao ponteiro para o próximo elemento de e_1, resultando em $lista \rightarrow e_1 \rightarrow NULL$.

Podem-se inserir (se a memória já foi alocada) e eliminar elementos em (de) qualquer posição na lista. Por exemplo, ao inserir e_2 no final da lista, atribui-se o ponteiro para o próximo elemento de e_1 ao ponteiro para o próximo elemento de e_2 e, em seguida, atribui-se o endereço de e_2 ao ponteiro para o próximo elemento de e_1, resultando em $lista \rightarrow e_1 \rightarrow e_2 \rightarrow NULL$. Como outro exemplo, ao se inserir um elemento e_s entre os elementos e_1 e e_2, atribui-se o ponteiro para o próximo elemento de e_1 ao ponteiro para o próximo elemento de e_s e, em seguida, atribui-se o endereço de e_s ao ponteiro para o próximo elemento de e_1, resultando em $lista \rightarrow e_1 \rightarrow e_s \rightarrow e_2 \rightarrow NULL$. Note que a ordem de atribuições é importante ou algum ponteiro poderá "ficar perdido", no jargão de programadores em linguagem C. Encontrar ponteiros com endereços inválidos é, geralmente, uma tarefa árdua. Ao eliminar o elemento e_s, deve-se armazenar o endereço de e_s num ponteiro, por exemplo, e e fazer com que o ponteiro para o próximo elemento de e_1 aponte para o elemento apontado por e_s. Isso elimina e_s da lista. Após a utilização de e, é importante liberar a região de memória de e para o sistema operacional.

As listas podem ser percorridas em ambas as direções ao se inserir um ponteiro para o elemento anterior. Se isso ocorre, tem-se um lista duplamente encadeada do tipo $NULL \leftarrow e_1 \leftrightarrow e_2 \leftrightarrow \cdots \leftrightarrow e_n \rightarrow NULL$ e pode-se ter dois ponteiros que apontam o início da lista $inicio\_lista \rightarrow e_1$ e $fim\_lista \rightarrow e_n$.

1.2.2 Pilhas

É frequente o uso de *pilhas*, que é um tipo especial de lista. Imagine livros só possíveis de movimentar um de cada vez. Coloca-se um livro na mesa de estudo, o segundo em cima do primeiro e assim por diante. Isso cria uma pilha de livros, e não se pode tirar o livro encostado na mesa sem derrubá-la. Não se pode tirar o segundo livro mais acima na pilha sem antes tirar o que está no *topo* da pilha.

Essa é a ideia da estrutura de dados do tipo pilha. Essa é uma estrutura de dados do tipo *último que entra é o primeiro que sai* ou *primeiro que entra é o último que sai*. Com isso, as pilhas são geralmente codificadas em regiões contíguas de memória, ou seja, em *arrays*.

1.2.3 Filas

Outro tipo especial de lista é chamado de *fila*. Por exemplo, em uma fila simples de banco, o último indivíduo *i* que entra na fila terá que esperar que todos os que entraram antes dele sejam atendidos. Quem entra na fila depois de *i*, será atendido depois de *i*.

As filas podem ser implementadas de maneira semelhante às listas encadeadas, isto é, há uma variável apontadora que indica o primeiro elemento da fila que, por sua vez, aponta para o segundo, e assim por diante, até o último elemento da fila que, por sua vez, contém *NULL* no seu atributo *próximo*, se uma linguagem como C ou C++ for utilizada. Um elemento que entra na fila deve ser inserido na posição adequada. Quando o primeiro elemento *i* da fila é removido, a memória de *i* é liberada e os apontadores correspondentes são atualizados.

Como as pilhas, as filas também podem ser implementadas em regiões contíguas de memória, isto é, em *arrays*. Pode-se também implementar uma *fila circular* em que o último elemento aponta para o primeiro. Necessariamente, mantém-se em uma variável o endereço do elemento no início da fila.

Considere que a fila do banco é única e que idosos, gestantes e pessoas com crianças pequenas entram no início da fila ou após outras pessoas com essas características. Da mesma forma, elementos com prioridade alta podem entrar na estrutura de dados de

tipo fila em posições anteriores á de outros. Com isso, tem-se uma *fila de prioridades* que não é, geralmente, implementada por *array*. Há diversas maneiras interessantes para a implementação das filas de prioridades. Uma é a *heap*, descrita no capítulo 5. Como motivação para o estudo de fila de prioridades, no caso anterior, em que algumas pessoas entram na frente de outras na fila simples de um banco, um indivíduo *i* poderia nunca ser atendido. Isso é chamado de *starvation* (ou *bloqueio indefinido*), na Ciência da Computação. Esse problema ocorre quando a um processo ou transação são sempre negados os recursos necessários para que ele termine sua tarefa. O leitor aprende sobre *starvation* em algumas disciplinas da Ciência da Computação. No contexto de Sistemas Operacionais, especificamente em *escalonamento por prioridades*, pode-se implementar uma fila de prioridades com os processos prontos para serem executados. Entretanto, um processo *i* com baixa prioridade pode esperar indefinidamente para ser atendido, se processos com maior prioridade que *i* entram na fila por prioridade. Veja Silberchatz, Gagne e Galvin (2004, p. 129-130) ou Tanenbaum (2003b, p. 93-94), para detalhes. No contexto de banco de dados, *starvation* ocorre se uma transação não pode continuar por um período de tempo longo. Veja Elmasri e Navathe (2005, p. 426), para detalhes.

Há maneiras de contornar isso. Uma maneira simples é a prioridade de *i* ser em função do seu tempo na fila.

1.2.4 Árvores

Uma *árvore* é outra estrutura de dados interessante. As árvores, na natureza, têm suas raízes no chão e as folhas ficam para o alto. As árvores computacionais são um pouco diferentes. Uma árvore computacional tem uma só *raiz* e é, geralmente, representada no topo, como num organograma de uma empresa. Os *filhos* f_i da raiz são representados embaixo dela. Os filhos f_j dos nodos f_i são representados embaixo dos nodos f_i, e assim por diante. Os nodos f_i são chamados de *pais* dos nodos f_j. Os nodos que não têm filhos são chamados de *folhas* ou *nodos terminais*. Geralmente, representa-se a conexão entre nodos com segmentos de retas. O exemplo de uma árvore é visto na Figura 1.1.

Uma árvore é um *tipo abstrato de dados* que armazena hierarquicamente os elementos, ou seja, as árvores armazenam elementos em um relacionamento pai-filho. Um tipo

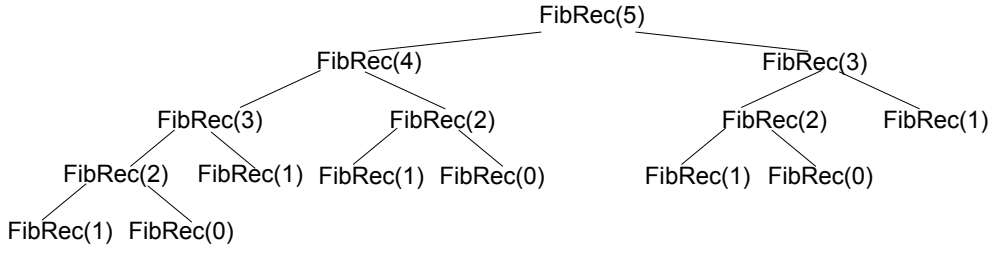

Figura 1.1: A Série de Fibonacci resolvida de forma recursiva é uma árvore.

de árvore especial é a *árvore binária*, em que cada nodo não-folha tem, no máximo, dois filhos. Um tipo de árvore binária especial é a *árvore binária de busca*, em que os nodos permanecem ordenados segundo seus conteúdos.

1.3 Corretude de algoritmos

Ao se desenvolver um algoritmo, deve-se verificar se ele realmente é efetivo: se produz um resultado pretendido; se realiza um propósito prático; se é completo no sentido de produzir um efeito (ou efeitos) decidido (positivo ou negativo), decisivo ou desejado para cada item no domínio do problema, como já descrito. Para isso, cada instância do conjunto de entradas E para o algoritmo deve ser descrita corretamente para que o algoritmo retorne uma *saída desejada*. Essas saídas desejadas compõem o conjunto de saída S.

Um algoritmo A é *parcialmente* correto em relação a $E \times S$ se $(\exists x \in E)$, então, A termina com resultado $s \in S$ *desejado* (ou $(\exists x \in E)|(s = A(x))$). Na corretude parcial, o intuito não é provar que o algoritmo termina, mas que, se termina, a saída é a desejada para uma determinada entrada x. Um algoritmo é *totalmente* correto se encontra a saída desejada, isto é, o algoritmo resolve corretamente o problema computacional para *qualquer* entrada corretamente descrita e o algoritmo *sempre termina*. Para a corretude total do algoritmo, basta verificar a corretude parcial do algoritmo e verificar se sempre termina. Na verdade, como exemplo o problema da parada (descrito no capítulo 14) que é indecidível, verificar se um algoritmo sempre termina pode ser complicado e neste texto aborda-se o assunto apenas superficialmente. O leitor poderá aprofundar-

se nesse assunto ao estudar que as provas de término de algoritmos são demonstrações matemáticas que têm um aspecto importante na Verificação Formal.

Um recurso bastante utilizado e simples para utilização em algoritmos também simples é o de *laços invariantes*. Um invariante é uma sequência de operações que fornece algo do tipo: se o predicado é verdadeiro antes de entrar na sequência, então o predicado é verdadeiro no final da sequência. A verificação de um laço invariante pode ser dividida nas três fases seguintes:

- inicialização, a corretude antes da primeira iteração do laço deve ser verificada; isso significa que se deve verificar a corretude após alguma instrução, como um comando de atribuição de valor a uma variável e exatamente antes do primeiro teste de satisfabilidade do laço;
- manutenção, a corretude entre as iterações do laço deve ser verificada, isto é, se cada iteração mantém o laço invariante;
- término, a corretude quando o laço termina deve ser verificada.

O algoritmo ingênuo a seguir encontra o máximo divisor comum entre dois números inteiros positivos.

Algoritmo 1.1: MDC.

Entrada: $x, y \in \mathbb{N}^*$;
Saída: $\max\{z \in \mathbb{N}^* : (\exists a, b \in \mathbb{N}^*)(x = az \wedge y = bz)\}$;

```
1 início
      // o MDC é, pelo menos, o menor entre x e y
2     t ← max(x,y), em que t ∈ ℕ*;
3     enquanto ( mod(x,t) ≠ 0 ∨ mod(y,t) ≠ 0 ) faça
4         t ← t − 1;
5     fim
6     retorna t;
7 fim.
```

Pode-se verificar esse algoritmo por laços invariantes, como descrito a seguir:

- inicialização, já que as entradas x e y são bem definidas e a função max retorna o maior entre ambos os argumentos, o algoritmo está correto antes de verificar a primeira condição da linha 3;
- manutenção, como a função mod das duas condições do laço é bem definida, o corpo do laço é simplesmente o decremento de uma variável inteira t e como o algoritmo se restringe aos números inteiros positivos, os argumentos x e y são números inteiros positivos e cada iteração mantém o laço invariante;
- término, o laço termina, pelo menos para t=1, já que as duas condições (da condição composta) inexoravelmente não serão satisfeitas; o decremento da variável t faz com que seja atribuído no mínimo 1 a t, pois isso fará com que ambas as condições não sejam satisfeitas; se ao final do laço, t é 1, então, x e y são números primos ou são 1.

Como a linha 6 simplesmente retorna o valor que convergiu em t e isso é bem definido, o algoritmo termina ao retornar o máximo divisor comum armazenado em t: o algoritmo é totalmente correto. A corretude de algoritmos e laços invariantes compõe a Lógica de Floyd-Hoare (FLOYD, 1967; HOARE, 1969). A verificação de corretude dos demais algoritmos deste texto fica como exercício para o leitor.

1.4 Exercícios

1. Prove que $\sum_{i=0}^{n} i^2 = \frac{(n^2+n)(2n+1)}{6}$.
2. Os Elementos de Euclides (aproximadamente 300 a.C.) tornaram-se a base de toda a Educação Matemática, não somente nos períodos Romano e Bizantino, mas também no século XX. Esse conjunto de 13 livros compõe a obra de maior sucesso na Matemática. Veja Boyer (1991, p. 100) e Wilson (2006, p. 278), para detalhes. Mostre a corretude do Algoritmo de Euclides abaixo.

Algoritmo 1.2: MDC_Er.

Entrada: $x, y \in \mathbb{N}^*$;
Saída: $\max\{z \in \mathbb{N}^* : (\exists a, b \in \mathbb{N}^*)(x = az \wedge y = bz)\}$;

```
1 início
2     se ( y = 0 ) então
3         t ← x, em que t ∈ ℕ*;
4     senão
5         t ← MDC_Er (y, mod(x,y));
6     fim
7     retorna t;
8 fim.
```

3. Mostre a corretude do Algoritmo de Euclides (EUCLIDES, aproximadamente 300 a.C.) abaixo.

Algoritmo 1.3: MDC_E.

Entrada: $x, y \in \mathbb{N}^*$;
Saída: $\max\{z \in \mathbb{N}^* : (\exists a, b \in \mathbb{N}^*)(x = az \wedge y = bz)\}$;

```
1 início
2     enquanto ( y ≠ 0 ) faça
3         t ← mod(x,y); x ← y; y ← t;
4     fim
5     retorna t;
6 fim.
```

1.5 Notas bibliográficas

É difícil escolher palavras para elogiar a obra grandiosa e a contribuição de Knuth (1968, 1981, 1998, 2001). Dentre os assuntos abordados, o autor esgota o assunto. É claro que não é trivial abordar assuntos complicados, abrangentes, escolher as palavras adequadas para ser sucinto e descrever apropriadamente todos os conceitos importantes. Knuth realiza sua obra com essas características. Com linguagem clara, sem excesso de tecnicidades, ele consegue abordar os assuntos com a profundidade adequada, e ainda

consegue ser divertido: ele incluiu comentários bem humorados. O estudo da Ciência da Computação é divertido com os livros de Knuth. Espero que o leitor tenha a oportunidade de estudar com as suas obras e divertir-se tanto quanto o responsável por estas notas. Como homenagem, estas notas iniciam-se com os ensinamentos eternizados por Knuth. Certamente, os volumes 5 (*Syntactical Algorithms*), 6 (*Theory of Languages*) e 7 (*Compilers*) deverão seguir a mesma qualidade dos anteriores.

2 *Paradigmas de Projeto de Algoritmos*

2.1 Introdução

Nos últimos setenta anos, cientistas da computação têm identificado diversas técnicas para o desenvolvimento de algoritmos eficientes para serem aplicados a problemas complexos. A seguir, descrevem-se algumas das técnicas básicas para a construção de algoritmos.

No texto a seguir, tempo de execução de algoritmo é comentado. A complexidade de algoritmos é descrita no capítulo 3. Por enquanto, entenda que a velocidade com que um algoritmo é executado em relação ao tamanho da entrada é relevante para se comparar diferentes algoritmos que resolvem o mesmo problema. Por exemplo, para entradas grandes, um algoritmo que tem seu tempo de execução expresso por uma função polinomial, isto é, expressa por um polinômio de determinado grau, é, em geral, mais rápido que um algoritmo que tem seu tempo de execução expresso por uma função exponencial.

2.2 Força bruta

Essa técnica também é chamada de *busca exaustiva*. Um algoritmo por força bruta enumera cada alternativa possível para a solução do problema e verifica se cada alternativa fornece uma solução ao problema. Um algoritmo por força bruta não termina enquanto não encontra uma solução para o problema ou não verifica sistematicamente todas as alternativas possíveis. A técnica força bruta é trivial, mas também é geral.

Por exemplo, para encontrar um item em uma tabela por força bruta, verificam-se, sequencialmente, todas as entradas da tabela. Às vezes, esse caso específico de força bruta é chamado de busca linear. Considere o problema de informar se um inteiro $y \in Y$, em que $n = |Y|$. Um algoritmo determinístico escrito em uma versão de Portugol é dado a seguir. A função recebe como entrada um inteiro y, um conjunto Y representado como um vetor computacional de inteiros e n. Se $y \in Y$, então, a função retorna a posição de y em Y. Se $y \notin Y$, então, a função retorna n+1.

Algoritmo 2.1: Pertence.

Entrada: inteiro y; conjunto de inteiros Y[ ]; inteiro n com o número de elementos de Y;
Saída: Saída: posição i de y em Y se $y \in Y$ ou n+1 se $y \notin Y$;

```
1 início
2     inteiro: i;
3     para ( i ← 1 até n ) faça
4         se ( y = Y[i] ) então
5             retorna i;
6         fim
7     fim
8     retorna i;
9 fim.
```

O custo de um algoritmo por força bruta é proporcional ao número de alternativas do problema. Em muitos problemas encontrados na prática, o número de alternativas cresce rapidamente em relação ao crescimento do tamanho da entrada do problema. Por causa disso, força bruta é, geralmente, utilizada quando o tamanho do problema é pequeno por causa do *overhead* de abordagens mais sofisticadas. Usa-se o termo *overhead* para se referir ao processamento necessário para se alcançar o objetivo, mas que não trata especificamente de dados úteis.

Força bruta também é utilizada quando a simplicidade de implementação é mais importante do que a velocidade para se responder ao problema. Por exemplo, força bruta pode ser adequada em aplicações críticas em que qualquer erro no algoritmo ou na sua implementação leva a consequências com prejuízos sérios.

Para outro exemplo, veja, em Mehlhorn e Sanders (2008, p. 246-247), o uso de força bruta no problema da mochila. Esse problema pode ser descrito como: dados n pares de inteiros (w_i, p_i) e inteiros M e P, decida se existe $I \subseteq \{1, \cdots, n\} | \sum_{i \in I} w_i \leq M \wedge \sum_{i \in I} p_i \geq P$.

2.3 Abordagem incremental

Em um algoritmo projetado pela abordagem incremental, um elemento por vez é inserido na solução. Um bom exemplo dessa abordagem é o algoritmo de Ordenação por Inserção. Esse algoritmo tem como entrada uma sequência de números $A[i] = a_1, a_2, \cdots, a_n$ e, como saída, uma permutação $a'_1, a'_2, \cdots, a'_n$ da sequência, tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$. Considera-se que o primeiro elemento, $A[1]$, está ordenado. Fazendo-se um laço de repetição na variável j de 2 a n, ordenado o *subarray* $A[1..j\text{-}1]$, insere-se o elemento $A[j]$ na sua própria posição, fornecendo o *subarray* ordenado $A[1..j]$. Veja Cormen et al. (2009, p. 29), para detalhes. Segundo Knuth (1998, p. 385), K. Zuse construiu um programa para Ordenação por Inserção direta em 1945.

Outro exemplo de utilização da abordagem incremental é em Geometria Computacional. O algoritmo de Green e Sibson (1977) constrói o Diagrama de Voronoi (1907) por inserção incremental de vértices. Um Diagrama de Voronoi é uma espécie de divisão do domínio computacional para ser utilizado em métodos computacionais e científicos.

2.4 Recursão

Um algoritmo recursivo é aquele que chama, direta ou indiretamente, a si próprio. Para resolver um problema por recursão, trata-se o problema como um todo e reduz-se o problema a subproblemas menores.

Há necessidade de que um algoritmo recursivo tenha um critério de parada (ou caso base ou ponto de parada), ou nunca pararia e não seria um algoritmo. Entretanto, mesmo que se projetasse um procedimento com recursão teoricamente infinita, o espaço de memória que possibilita as chamadas recursivas é necessariamente finito e o algoritmo pararia com erro. O algoritmo usaria toda a memória da pilha de métodos e o sistema

operacional daria um erro. Grosso modo, a pilha é uma estrutura de dados utilizada pelo sistema operacional para possibilitar a recursão, dentre outros processos.

Ao alcançar um caso base (ou trivial ou ponto de parada), um algoritmo recursivo inicia o processo de retorno, resolvendo cada subproblema no retorno de cada chamada recursiva. Se o valor de retorno da primeira chamada é o valor de retorno da última, esse processo é conhecido como *recursão de cauda*. Ou seja, uma maneira de se implementar um algoritmo recursivo é por recursão de cauda, em que o caso base retorna uma valor, e esse valor também é retornado por todos os demais níveis da recursão. As linguagens de programação e os compiladores implementam isso de maneira adequada e eficiente.

Técnicas recursivas são úteis quando um problema pode ser dividido em subproblemas com esforço de processamento razoável e a soma dos espaços utilizados pelos subproblemas resulta em uma quantidade pequena. Considere n o número de itens da entrada de um algoritmo recursivo. Em geral, se a soma dos tamanhos dos subproblemas é cn para alguma constante $c \in \mathbb{N}^*$, então, o algoritmo recursivo provavelmente terá complexidade polinomial. No entanto, se a divisão do problema de tamanho n resulta em n problemas de tamanho n-1, então, o algoritmo, provavelmente, terá crescimento do tempo de execução exponencial (AHO; HOPCROFT; ULLMAN, 1974, p. 67).

Note que qualquer algoritmo recursivo para o problema P deve resolver os mesmos subproblemas de P. Um algoritmo para o fatorial é dado a seguir.

Algoritmo 2.2: Fatorial.

Entrada: um inteiro n;
Saída: $n \times (n-1) \times (n-2) \times \cdots \times 1$;

```
1 início
      // condição de parada obrigatória num algoritmo recursivo
2     se ( n = 0 ) então
3         retorna 1;
4     fim
      // chama a si mesmo, o que o caracteriza como recursivo
5     retorna n * Fatorial (n-1);
6 fim.
```

Um algoritmo é sempre iterativo. Um algoritmo pode ser recursivo ou pode utilizar operações explícitas para resolver o problema, geralmente, ao utilizar um laço de repetição. Veja os capítulos 3 de Wirth (1989) e Tenenbaum, Langsam e Augenstein (1995), para descrição detalhada sobre algoritmos recursivos. Veja também Gersting (2004, p. 101-105), o capítulo 5 de Drozdek (2005) e o capítulo 12 de Pereira (2008), para se aprofundar em algoritmos recursivos.

2.5 Divisão e conquista

Neste paradigma, o algoritmo é projetado para *dividir* o problema em subproblemas menores. Os subproblemas menores são similares entre si e ao problema geral, ou seja, são do mesmo tipo ou relacionados. O problema é dividido até que se torne simples o suficiente para ser resolvido trivialmente. O algoritmo *conquista* os subproblemas ao fazer chamadas recursivas dos subproblemas. Em seguida, o algoritmo *combina* as soluções para construir a solução do problema original.

Este paradigma pode ser utilizado quando o problema pode ser dividido em subproblemas. Geralmente, os algoritmos desse paradigma são eficientes, isto é, com complexidade polinomial. Um algoritmo por divisão e conquista envolve três passos a cada nível de recursão: *i*) divida o problema em um certo número de subproblemas; *ii*) conquiste os subproblemas ao resolvê-los recursivamente, se os tamanhos dos subproblemas são pequenos, resolva-os de maneira trivial; *iii*) combine as soluções dos subproblemas em uma só solução para o problema original.

Apesar de o problema ser decomposto em apenas um subproblema, a Busca Binária pode ser considerada um exemplo de algoritmo divisão e conquista. Outro exemplo é o Algoritmo da Bisseção, que encontra zeros de funções. No entanto, nesses casos ou em outros em que o problema é decomposto em um só subproblema, pode ser considerado que utilizem simplesmente a recursão ou a *recursão de cauda*. Brassard e Bratley (1996) consideram que o paradigma divisão e conquista deveria ser utilizado somente quando cada problema pode gerar mais de um subproblema. Levitin (2002) propôs o termo *diminuição e conquista* para os algoritmos dessa classe. Esses algoritmos diminuição e conquista podem ser considerados casos especiais de algoritmos por divisão e conquista.

Algoritmos construídos pelo paradigma divisão e conquista dividem o problema em subproblemas independentes, resolvem os subproblemas recursivamente e, então, combinam as soluções para resolver o problema original. Dessa forma, um algoritmo construído pelo paradigma divisão e conquista pode realizar mais trabalho que o necessário, pois repetidamente resolve os mesmos subproblemas (CORMEN et al., 2009, p. 359). Isso pode ser utilizado, por exemplo, para comparar algoritmos por divisão e conquista com algoritmos construídos pelos conceitos da programação dinâmica, descrita na seção 2.10.

Cormen et al. (2009, p. 111) afirmam que o paradigma divisão e conquista para projeto de algoritmos já foi descrito por Karatsuba e Ofman (1963). Ainda afirmam que, segundo Heiderman, Johnson e Burrus (1984), C. F. Gauss apresentou o primeiro algorimo para Transformada Rápida de Fourier em 1805: a formulação de Gauss divide o problema em subproblemas cujas soluções são combinadas.

Um algoritmo divisão e conquista pode ser implementado não recursivamente ao armazenar explicitamente dados de subproblemas parciais em alguma estrutura de dados, como pilha, fila ou fila de prioridades. Entretanto, isso é bastante incomum e poderá ter o mesmo efeito de se utilizar a recursividade. A corretude de algoritmos por divisão e conquista é, geralmente, provada por indução. Considere a seguir um algoritmo divisão e conquista genérico com chamadas recursivas.

O Algoritmo D&C resolve o problema trivial se for menor ou igual a ε. Se o problema é maior que ε, então, haverá a chamadas recursivas, e o número de itens de cada chamada recursiva é similar. Por fim, o algoritmo D&C combina as soluções dos subproblemas a cada nível de recursão.

No capítulo 4, o Merge Sort é apresentado para exemplificar algoritmos por divisão e conquista. Outros exemplos de algoritmos por divisão e conquista incluem método de multiplicação rápido de Karatsuba, Quicksort e a Transformada Rápida de Fourier. Além desses, exemplos de utilização de divisão e conquista incluem multiplicação de polinômios, obter o k-ésimo maior elemento de um conjunto (problema da seleção) e problema do par mais próximo. Shamos e Hoey (1975) utilizaram divisão e conquista para manipular o Diagrama de Voronoi (1907). Guibas e Stolfi (1985) utilizaram divisão e conquista para construir a Triangulação de Delaunay (1934). A ideia desses autores

aproxima-se da proposta de Lee e Schachter (1980). Aho, Hopcroft e Ullman (1983), Terada (1991, p. 72-76) e Goodrich e Tamassia (2004, p. 274-277) mostram exemplos de algoritmos por divisão e conquista para a multiplicação de inteiros. O Algoritmo de Strassen utiliza divisão e conquista para a multiplicação de matrizes. Aho, Hopcroft e Ullman (1974) e Goodrich e Tamassia (2004, p. 274-277) mostram exemplos de algoritmo por divisão e conquista para a multiplicação de matrizes. Veja também Terada (1991, p. 43-48) e Ziviani (2011, p. 48-49), para exemplo de algoritmo por divisão e conquista para encontrar o menor e o maior inteiros de um conjunto.

Algoritmo 2.3: D&C.

Entrada: os dados estão entre i e j para serem combinados em a chamadas recursivas;
Saída: combinação dos dados entre i e j;

```
1  início
2      se ( j − i ≤ ε, em que ε é pequeno ) então
3          resolve (i, j);
4      senão
5          m_0 ← i − 1;
           // a chamadas recursivas
6          para ( k ← 1 até a-1 ) faça
7              m_k ← m_{k−1} + 1 + ⌊(j−i+1)/a⌋;
8              D&C (m_{k−1} + 1, m_k);
9          fim
           // a-ésima chamada recursiva, k = a
10         D&C (m_{k−1} + 1, j);
11         Combine (m_0 + 1, m_1, m_2, ··· , j);
12     fim
13 fim.
```

2.6 Balanceamento

Muitos algoritmos mantêm o balanceamento ao gerar subproblemas de tamanho (aproximadamente) igual a cada passo do algoritmo. Por exemplo, isso é frequente no paradigma divisão e conquista. É uma boa prática de projeto de algoritmos (e de estruturas de dados) gerar subproblemas de mesmo tamanho, mesmo em conjunto com qualquer outra técnica ou paradigma (AHO; HOPCROFT; ULLMAN, 1974, p. 65). Exemplo é o Merge Sort, que é mostrado no capítulo 4.

Como exemplo em uma estrutura de dados, considere uma árvore A com n elementos. Se A é balanceada, então, A apresenta limite na sua altura em um fator logarítmico em n. Se A não é balanceada, então, o ramo de maior altura pode degenerar para uma lista encadeada com fator linear em n. Note que uma função linear em n é assintóticamente maior que uma função logarítmica em n.

2.7 Algoritmos gulosos

Este paradigma é bastante utilizado em problemas de otimização. Segundo Cormen et al. (2009, p. 450), algoritmos gulosos surgiram em Otimização Combinatória com Edmonds (1971).

Os algoritmos projetados por este paradigma fazem uma escolha "gananciosa" a cada passo. Algoritmos gulosos constróem a solução a cada alternativa, sempre escolhendo a próxima que oferece o benefício mais evidente e imediato no estágio corrente. Isso significa que um algoritmo guloso *aproxima*-se da solução pela melhor decisão baseado no ótimo local corrente e na melhor decisão baseada nos passos anteriores, e nunca em passos futuros, característica que significa que ele *nunca reconsidera as escolhas já realizadas*. Algoritmos gulosos não verificam todas as alternativas de solução para o problema. Com isso, os algoritmos construídos por este paradigma nem sempre encontram soluções ótimas. Essas são as principais diferenças em relação à programação dinâmica. Um algoritmo guloso não é exaustivo e não fornece uma solução exata para muitos problemas. Entretanto, quando funciona, é, em geral, um algoritmo eficiente.

A principal diferença entre as abordagens gulosa e incremental é que a abordagem gulosa, geralmente, é empregada para problemas de otimização. Na abordagem incremental, geralmente, não há uma função objetivo a ser otimizada, ou seja, minimizada ou maximizada.

Os algoritmos de Prim e de Kruskal, apresentados no capítulo 10, e de Dijkstra, apresentado no capítulo 11, são exemplos de algoritmos gulosos. Veja Goodrich e Tamassia (2004, p. 264-267) para exemplos de algoritmos gulosos para o problema da mochila fracionária e escalonamento de tarefas, e o capítulo 16 de Cormen et al. (2009), para uma descrição detalhada sobre algoritmos gulosos.

2.8 *Backtracking*

Esta técnica também é chamada de tentativa e erro - veja Wirth (1989, p. 120) e Ziviani (2011, p. 44-45). *Backtracking* é, em geral, aplicado a problemas de decisão. Veja descrição de problemas de decisão no capítulo 13.

Backtracking pode ser entendido como uma melhoria da técnica força bruta. Um algoritmo que utiliza a técnica *backtracking* consegue descartar grandes conjuntos de soluções inválidas sem a necessidade de examiná-las explicitamente. Com isso, quando aplicável, um algoritmo por *backtracking* fornece soluções mais rapidamente que um algoritmo por força bruta, por exemplo.

Conceitualmente, em algoritmos por *backtracking*, uma *árvore de busca* é examinada gradualmente. Neste contexto, essa árvore de busca ocorre implicitamente no algoritmo e contém dados com alguma relação de ordem. Note que chamadas recursivas podem, implicitamente, descrever uma árvore. Em *backtracking*, outras características dessa árvore de busca são:

- é composta pelos subproblemas decompostos do problema;
- é formada pelos possíveis caminhos para a solução do problema durante a execução de um algoritmo por *backtracking*;

- pode não ocorrer explicitamente no algoritmo e, geralmente, é fornecida por recursão;
- fornece os meios para se procurar exaustivamente a solução;
- os nós não folhas correspondem a soluções parciais e os nós terminais correspondem às alternativas de solução para o problema.

Percorrer a árvore da raiz em direção às folhas corresponde a obter soluções parciais no caminho da solução final. Com isso, se o problema não apresenta a característica de ter candidatos às soluções parciais, então o *backtracking* não pode ser empregado. Percorrer a árvore das folhas em direção à raiz corresponde a retroceder, ou *backtracking*, a alguma solução já obtida, a partir da qual seja viável prosseguir em direção a outras folhas.

Em muitos problemas, essa árvore cresce exponenciamente. Mas, o algoritmo por *backtracking* não prossegue num ramo de uma solução parcial ao verificar que não fornece uma solução válida. Tal melhoria economiza quantidade relevante de processamento que seria utilizado analisando-se soluções inválidas em um algoritmo construído por força bruta.

A eficácia de um algoritmo por *backtracking* depende de três importantes decisões de projeto (LEWIS; PAPADIMITRIOU, 2004, p. 330), que são:

- a maneira de escolher o próximo subproblema, ou seja, a partir de qual subproblema efetuar a ramificação;
- a maneira de refinar o subproblema escolhido em outros subproblemas simples;
- o tipo de teste a ser utilizado.

O algoritmo busca em profundidade, apresentado no capítulo 9, é um exemplo de utilização de *backtracking*. Veja Goodrich e Tamassia (2004, p. 618-621) para exemplos de algoritmos por *backtracking* para os problemas da satisfabilidade na forma normal conjuntiva e no problema da soma de subconjuntos. O problema da satisfabilidade tem como entrada uma expressão booleana E na forma normal conjuntiva e sua parentisação

com n proposições lógicas. Uma expressão booleana E contendo um produto de adições de variáveis booleanas é dita estar na forma normal conjuntiva. Pergunta-se se há um conjunto com n proposições lógicas (verdadeiro ou falso) que determine verdadeiro para a expressão booleana E. O problema da soma de subconjuntos consiste de verificar se existe um subconjunto de um conjunto de inteiros em que a soma dos elementos do subconjunto é zero. Esses dois problemas são tratados no capítulo 15. Ainda, Wirth (1976) e Ziviani (2011, p. 45-48) apresentam exemplo de algoritmo por *backtracking* para o passeio do cavalo no tabuleiro de xadrez.

Segundo Bitner e Reingold (1975), uma das mais antigas descrições de *backtracking* foi feita em matemática recreacional, por Lucas (1891). Bitner e Reingold (1975) afirmam que o termo *backtrack* foi primeiramente usado por LEHMER (1959) e que Walker (1960) foi o primeiro a descrever completamente a técnica. Alguns dos primeiros a utilizarem a técnica também foram Golomb e Baumert (1965).

2.9 *Branch-and-bound*

O *branch-and-bound* é uma variação de *backtracking* para problemas de otimização, isto é, que envolvem encontrar o mínimo (ou o máximo) de alguma função objetivo. O *branch-and-bound* é uma técnica geral para calcular limites sobre soluções parciais para limitar o número de soluções completas a serem examinadas. A técnica foi primeiramente proposta por Land e Doig (1960), para otimização combinatória.

Um algoritmo por esta técnica enumera sistematicamente todas as soluções viáveis. Entretanto, na procura pela melhor solução no espaço de soluções de um problema, enumerar explicitamente todos os candidatos à solução é impraticável para muitos problemas por causa do crescimento exponencial do número de soluções candidatas. Com isso, em algoritmos que utilizam a técnica *branch-and-bound*, diversos conjuntos de soluções inválidas não são examinados explicitamente porque limites são impostos na busca. O uso de limites para a função objetivo combinado com o valor da melhor solução corrente possibilita que o algoritmo procure partes do espaço de solução apenas de maneira implícita (CLAUSEN, 1999).

Os subespaços ainda não explorados são representados como nodos em uma árvore de busca gerada implícita e dinamicamente. A iteração de um algoritmo por *branch-and-bound* apresenta três componentes (CLAUSEN, 1999): seleção do nodo da árvore a processar; cálculo do limite; ramificação na árvore.

Veja Goodrich e Tamassia (2004, p. 622-626) para exemplos de algoritmos por *branch-and-bound* para o problema do caixeiro viajante. Esse problema é descrito no capítulo 15.

2.10 Introdução à programação dinâmica

A programação dinâmica é uma técnica para propósito geral, baseada na decomposição do problema, aplicada tipicamente a problemas de otimização. A expressão programação dinâmica foi usada originalmente na década de 1940, por R. Bellman, para descrever o processo de resolução de problemas em que é necessário encontrar uma sequência de decisões ótimas. Aproximadamente no ano de 1953, ele redefiniu o seu significado, o qual é utilizado atualmente. Aparentemente, a primeira publicação com esse significado é Bellman (1957), em que a programação dinâmica foi aplicada sistematicamente.

O termo programação refere-se ao tabelamento e não a código computacional. Eddy (2004) comenta que *dinâmica* foi escolhida pelo autor por ser um termo que impressionaria, e não porque descreve como o método é executado.

A programação dinâmica é útil quando um algoritmo por outra técnica apresenta complexidade exponencial. Quando a programação dinâmica pode ser aplicada, geralmente, o algoritmo tem tempo de execução assintótico menor que um algoritmo construído por força bruta, por exemplo.

Nesta técnica, busca-se resolver um problema complexo ao decompô-lo em subproblemas menores. Um algoritmo construído por programação dinâmica é aplicável quando os subproblemas são dependentes, ou seja, quando subproblemas compartilham subproblemas. Mais especificamente, se subproblemas podem ser aninhados recursivamente dentro de problemas maiores, então, a programação dinâmica pode ser utilizada.

Nesses casos, há uma relação direta entre os valores dos subproblemas em determinada sequência de decisões que leva ao valor de um problema maior que tais subproblemas.

A programação dinâmica é uma técnica exaustiva e garante encontrar a solução. A cada estágio, as decisões são baseadas em todas as decisões realizadas em estágios anteriores e pode reconsiderar o caminho de estágios prévios para encontrar a solução ótima do problema global.

Considere calcular um subconjunto S de um conjunto com n elementos. O subconjunto S deve satisfazer certas condições e é denominado solução viável para o problema. Associado a cada solução viável, tem-se um valor $f(S)$, em que f é chamada de função objetivo, cujo significado é dado na definição do problema. Entre todas as soluções viáveis, deseja-se aquela que tem valor $f(S)$ mínimo ou máximo. Essa solução é denominada *solução ótima*.

Algoritmos por programação dinâmica podem ser utilizados quando a solução de um problema pode ser obtida por meio de uma sequência de decisões. Para alguns problemas, uma sequência ótima de decisões, que resulta numa solução ótima, pode ser determinada, sendo as decisões feitas uma de cada vez, sem que decisões errôneas sejam feitas.

A solução ótima é baseada no *princípio da otimalidade*: em uma sequência ótima de decisões, cada subsequência deve ser ótima. Para alguns algoritmos, não é fácil encontrar uma sequência de decisões que resulte numa sequência ótima. Então, um algoritmo que utiliza a técnica programação dinâmica executa todas as sequências de decisões possíveis e escolhe a melhor. Isso significa que um algoritmo por programação dinâmica resolve todos os subproblemas gerados pela decomposição do problema global. Parte-se de subproblemas pequenos para os grandes. Um algoritmo construído por programação dinâmica resolve cada subproblema somente uma vez, e sua principal característica é que ele armazena os resultados dos subproblemas em uma *tabela*. Isso evita o trabalho de recomputar a resposta toda vez que for necessário utilizar o resultado do subsubproblema em um subproblema maior que ele (CORMEN et al., 2009, p. 359). Isso porque, em uma sequência de decisões, subproblemas pequenos podem ter soluções ótimas em determinada direção. No entanto, ao resolver subproblemas grandes, uma outra sequência de decisões pode resultar em uma solução melhor. Com isso,

calcular todas as soluções de subproblemas e armazená-las em uma tabela resulta em redução de trabalho porque os subproblemas pequenos não são recalculados e podem-se encontrar soluções ótimas para subproblemas grandes. Esse processo é realizado até que se encontre a solução ótima do problema global, que é obtida ao se encontrar a sequência ótima de decisões dos subproblemas que gera a solução ótima do problema global. O armazenamento em uma tabela reflete a principal característica e a vantagem da técnica: como descrito, quando um subproblema pequeno é resolvido e armazenado, não precisa mais ser recalculado; o resultado é utilizado diretamente por subproblemas maiores que ele.

É importante enfatizar essa característica dos algoritmos desenvolvidos por programação dinâmica: as soluções ótimas para subproblemas são obtidas e armazenadas para serem usadas na solução de subproblemas grandes, evitando-se cálculos redundantes. Claramente, a resolução dos subproblemas grandes utiliza os resultados dos pequenos. Ao final de um algoritmo construído pelos conceitos da programação dinâmica, resolve-se o problema ao combinar as soluções dos subproblemas na sequência que gera a solução ótima.

A técnica é aplicável a problemas que apresentam as propriedades de (GOODRICH; TAMASSIA, 2004):

- *subestruturas ótimas* - ocorrem quando um problema pode ser dividido recursivamente, por exemplo. Mais especificamente, subestrutura ótima significa que a solução para o problema de otimização pode ser obtida pela combinação de soluções ótimas para seus subproblemas. Consequentemente, o primeiro passo é verificar se o problema apresenta tal subestrutura ótima. Tais subestruturas ótimas são, geralmente, descritas por recursão. Por exemplo, dado um grafo G(V,A), o caminho mínimo p de um vértice u para o vértice v apresenta subestrutura ótima: obtenha qualquer vértice intermediário w no caminho mínimo p. Se p é verdadeiramente o caminho mínimo, então, o caminho p_1 de u para w e o de p_2 de w para v são, de fato, os caminhos mínimos entre os vértices correspondentes. Consequentemente, pode-se facilmente formular a solução ao se encontrar recursivamente os caminhos mínimos. É isso que faz o algoritmo de Bellman-Ford, que é descrito no capítulo 11;

- *decomposição de subproblemas* - é preciso existir uma forma de decompor o problema de otimização geral em subproblemas similares e menores que o problema original. Ainda, deve existir uma maneira simples de atribuir e acessar os resultados dos subproblemas usando apenas índices da tabela;

- *sobreposição de subproblemas* - se o problema pode ser dividido em subproblemas que são reutilizados múltiplas vezes, então, um problema é dito ter *sobreposição de subproblemas*. Isso também pode estar relacionado com a recursão. Em um problema em que não há sobreposição de seus subproblemas, ou seja os subproblemas decompostos são independentes, o armazenamento dos resultados não contribui e a programação dinâmica não é uma técnica adequada para ser aplicada.

Uma implementação ingênua para se obter a Série de Fibonacci pode ser:

Algoritmo 2.4: FibRec.

Entrada: inteiro n;
Saída: o n-ésimo termo da Série de Fibonacci;

```
1 início
2 |   se ( n = 0 ∨ n = 1 ) então
3 |   |   retorna n;
4 |   fim
5 |   retorna FibRec (n-1) + FibRec (n-2);
6 fim.
```

A avaliação de FibRec (n) depende de FibRec (n-1) + FibRec (n-2). Veja um exemplo com FibRec (5) na Figura 2.1.

No exemplo a seguir, mostra-se um algoritmo por programação dinâmica. Este algoritmo considera duas variáveis globais:

- $ind \leftarrow 1$;
- // inicializa vetor *mem* com os dois primeiros elementos da Série de Fibonacci
- vetor $mem[0] \leftarrow 0$, $mem[1] \leftarrow 1$;

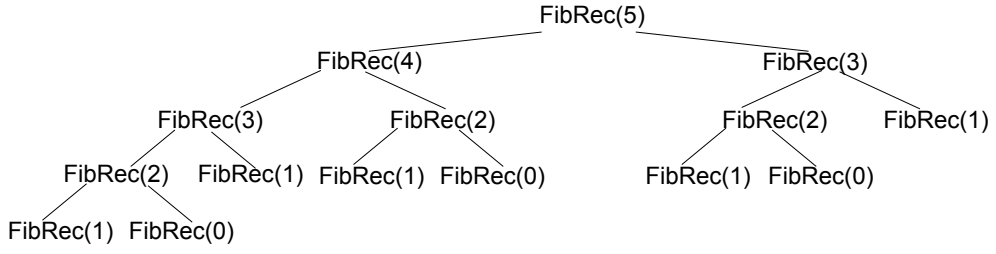

Figura 2.1: Série de Fibonacci de forma recursiva.

Um exemplo na Figura 2.2 mostra que o número de chamadas da versão por programação dinâmica é drasticamente reduzido em relação à versão simplemente recursiva. Mais especificamente, o tempo de execução da versão simplesmente recursiva é exponencial e a versão por programação dinâmica é linear em n.

A versão anterior é um exemplo de uso de recursidade com programação dinâmica. Uma versão não recursiva, mais simples que a anterior e com laço de repetição, é apresentada a seguir, em que as variáveis globais e suas inicializações são as mesmas do algoritmo anterior.

Exemplos de algoritmos que utilizam programação dinâmica são Floyd-Warshal e Belmman-Ford, para caminho mínimo em grafos, este de um vértice para os demais, aquele de todos os vértices para todos os demais. Esses algoritmos são apresentados no capítulo 11. Goodrich e Tamassia (2004, p. 278-285) mostram exemplos de algoritmos por programação dinâmica para produtos de matrizes encadeadas e no Problema da Mochila.

Veja o capítulo 15 de Cormen et al. (2009), para uma descrição detalhada sobre programação dinâmica.

Algoritmo 2.5: FibPDRec.

Entrada: inteiro n;
Saída: o n-ésimo termo da Série de Fibonacci;

```
1  início
       // até o índice ind, os valores da Série de Fibonacci
          estão armazenados no vetor mem
2      se ( n = ind + 1 ) então
           // se é o próximo elemento, então armazena na próxima
              posição do vetor mem
3          mem[ind+1] ← mem[ind] + mem[ind-1];
4          ind ← ind + 1;
5      senão se ( n > ind ) então
6          mem[n] ← FibPDRec (n-2) + FibPDRec (n-1);
7          ind ← n;
8      fim
9      retorna mem[n];
10 fim.
```

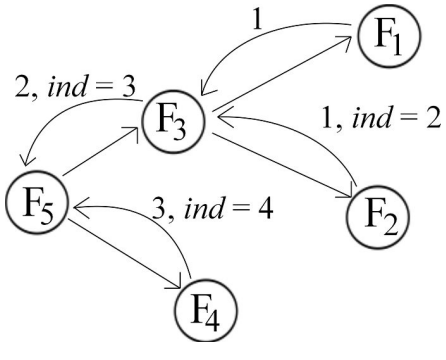

Figura 2.2: Série de Fibonacci por programação dinâmica.

Algoritmo 2.6: FibPDnRec.

Entrada: inteiro n;
Saída: o n-ésimo termo da Série de Fibonacci;

```
1 início
2     enquanto ( n > ind ) faça
3         mem[ind+1] ← mem[ind] + mem[ind-1];
4         ind ← ind + 1;
5     fim
6     retorna mem[n];
7 fim.
```

2.11 Exercícios

1. Escreva em pseudocódigo um algoritmo pelo paradigma algoritmo guloso para o problema do caixeiro viajante.

2. Descreva em detalhes a técnica *branch-and-bound* e descreva pelo menos um exemplo.

3. Descreva em detalhes a técnica *branch-and-bound* distinguindo-a de *backtracking* e de força bruta. Escreva algoritmos em pseudocódio que diferenciem as abordagens.

4. Escreva em pseudocódigo um algoritmo que utilize programação dinâmica e recursividade para resolver a Série de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, $\cdots$.

5. Na abordagem ....., o próximo elemento é inserido em sua devida posição na solução final.

6. Um algoritmo construído pela técnica ..... enumera sistematicamente todas as alternativas possíveis para a solução e verifica se cada alternativa satisfaz o problema.

7. ..... é uma técnica geral de calcular limites sobre soluções parciais para limitar o número de soluções completas a serem examinadas. ..... é uma variação de .....

8. Em um algoritmo desenvolvido pela técnica ....., um número grande de soluções inviáveis pode ser descartado sem ser explicitamente enumerado. ..... é uma árvore de busca exaustiva, cujos nós correspondem a soluções parciais. Percorrer a árvore da raiz em direção às folhas corresponde a obter soluções parciais no caminho da solução ..... Percorrer a árvore das folhas em direção à raiz corresponde a ..... a alguma solução parcial geral obtida anteriormente, a partir da qual seja viável prosseguir em direção a outras folhas.

9. A técnica geral de calcular limites sobre soluções parciais para limitar o número de soluções completas a serem examinadas é chamada de ..... Essa técnica é uma variação de ..... para problemas de otimização.

10. Algoritmos construídos pelo paradigma ..... dividem o problema em subproblemas independentes, resolve os subproblemas recursivamente e, então, combinam as soluções para resolver o problema original. Dessa forma, um algoritmo construído pelo paradigma ..... pode realizar mais trabalho que o necessário, pois, repetidamente, resolve subproblemas em comum.

11. Um algoritmo construído pelos conceitos da ..... resolve problemas ao combinar soluções de subproblemas. Ou seja, calcula a solução para todos os subproblemas, partindo de subproblemas menores para os maiores. Armazena os resultados dos subproblemas menores em uma ..... A resolução dos subproblemas maiores utiliza os resultados dos menores. Um algoritmo construído pela técnica ..... é aplicável quando os subproblemas não são independentes, ou seja, quando subproblemas compartilham subproblemas. Um algoritmo construído pela técnica ..... resolve cada subproblema somente uma vez e então armazena a resposta em uma .....

Isso evita o trabalho de recomputar a resposta toda vez que um subproblema é encontrado.

12. Algoritmos ..... constroem a solução a cada alternativa, sempre escolhendo a próxima que oferece o benefício mais evidente e imediato. Os algoritmos construídos por este paradigma nem sempre encontram soluções ótimas. Este paradigma é bastante utilizado em problemas de otimização. Algoritmos ..... podem ser aplicados a vários problemas. Estes problemas são do tipo: calcular um subconjunto S de um conjunto com n elementos. S deve satisfazer a certas condições e é denominado solução viável para o problema. Associado a cada solução viável tem-se um valor $f(S)$, em que f é uma função chamada função objetivo, cujo significado é dado na definição do problema. Entre todas as soluções viáveis, deseja-se aquela que tem valor $f(S)$ mínimo ou máximo, e esta é denominada solução ótima.

13. Os algoritmos abaixo são construídos pelo(a) paradigma/técnica:

    Prim: .....

    Kruskal: .....

    Dijkstra: .....

    Bellman-Ford: .....

    Floyd-Warshall: .....

    Ordenação por inserção: .....

    Merge-sort: .....

    Quick-sort: .....

    Busca linear (encontrar um item em uma tabela, verificando sequencialmente todas as entradas): .....

14. Descreva em detalhes o paradigma de projeto de algoritmos: algoritmos gulosos.

15. Descreva em detalhes programação dinâmica.

16. Escreva em pseudocódigo um algoritmo que utilize programação dinâmica e recursividade para resolver a Série de Fibonacci (0, 1, 1, 2, 3, 5, 8, 13, 21, ...).

17. Explique o paradigma divisão e conquista em detalhes.

18. Escolha a alternativa correta.

    I) Um algoritmo por *backtracking* enumera sistematicamente todos as alternativas possíveis para a solução e verifica se cada alternativa satisfaz o problema.

    II) *Branch-and-bound* é uma árvore de *backtracking*, cujos nós correspondem a soluções parciais.

    III) Em um algoritmo construído por *backtracking*, percorrer a árvore da raiz em direção às folhas corresponde a obter soluções parciais no caminho da solução final. Percorrer a árvore das folhas em direção à raiz corresponde a retroceder a alguma solução parcial geral obtida anteriormente, a partir da qual seja viável prosseguir em direção a outras folhas.

    IV) *Branch-and-bound* é uma técnica geral de calcular limites sobre soluções parciais para limitar o número de soluções completas a serem examinadas. Esta técnica é uma variação de *backtracking* para problemas de otimização, isto é, que envolvem encontrar o mínimo (ou o máximo) de alguma função objetivo.

    a) I: verdadeira; II: verdadeira; III: verdadeira; IV: falsa.

    b) I: verdadeira; II: verdadeira; III: falsa; IV: verdadeira.

    c) I: falsa; II: falsa; III: verdadeira; IV: verdadeira.

    d) I: verdadeira; II: falsa; III: verdadeira; IV: falsa.

    e) Todas são falsas.

    f) Todas são verdadeiras.

19. Escolha a alternativa correta.

    I) Algoritmos construídos por força bruta dividem o problema em subproblemas independentes, resolvem os subproblemas recursivamente e, então, combinam as soluções para resolver o problema original.

    II) Um algoritmo construído por divisão e conquista pode realizar mais trabalho que o necessário, pois, repetidamente, resolve subproblemas em comum.

    III) Algoritmos construídos por divisão e conquista resolvem problemas ao combinar soluções de subproblemas.

    IV) Algoritmos construídos por programação dinâmica resolvem problemas ao combinar soluções de subproblemas.

    a) I: verdadeira; II: falsa; III: falsa; IV: verdadeira.

    b) I: falsa; II: verdadeira; III: falsa; IV: verdadeira.

    c) I: falsa; II: verdadeira; III: verdadeira; IV: verdadeira.

    d) I: falsa; II: verdadeira; III: verdadeira; IV: falsa.

    e) Todas são falsas.

    f) Todas são verdadeiras.

20. Escolha a alternativa correta.

    I) Um algoritmo guloso sempre reconsidera a escolha realizada. Essa é a principal diferença em relação à programação dinâmica.

    II) Na técnica força bruta, o próximo elemento é inserido em sua devida posição na solução final.

    III) Em um algoritmo desenvolvido por força bruta, um número grande de soluções inviáeis pode ser descartado sem ser explicitamente enumerado.

    IV) O nome força bruta é porque se trata de uma técnica robusta que sempre fornece algoritmos eficientes.

    a) I: verdadeira; II: falsa; III: falsa; IV: falsa.

    b) I: falsa; II: verdadeira; III: falsa; IV: falsa.

    c) I: falsa; II: falsa; III: verdadeira; IV: falsa.

d) I: falsa; II: falsa; III: falsa; IV: verdadeira.

e) I: falsa; II: verdadeira; III: falsa; IV: verdadeira.

f) Todas são falsas.

21. Escolha a alternativa correta. I) Um algoritmo construído pelos conceitos da programação dinâmica resolve problemas ao combinar soluções de subproblemas. Ou seja, calcula a solução para todos os subproblemas, partindo de subproblemas maiores para os menores. Armazena os resultados dos subproblemas menores em uma tabela. A resolução dos subproblemas menores utiliza os resultados dos maiores.

    II) Um algoritmo construído por programação dinâmica é aplicável quando os subproblemas são independentes, ou seja, quando subproblemas compartilham subproblemas. Um algoritmo construído por programação dinâmica resolve cada subproblema somente uma vez e, então, armazena a resposta em uma tabela. No entanto, isso não evita o trabalho de recomputar a resposta toda vez em que é necessária a solução de um subproblema.

    III) Numa sequência de decisões em um algoritmo construído por programação dinâmica, subproblemas menores podem ter soluções ótimas em determinada direção. No entanto, ao resolver subproblemas maiores, outra sequência de decisões pode resultar em uma solução melhor.

    IV) Em um algoritmo construído por programação dinâmica, calcular todas as soluções de subproblemas e armazená-las em uma tabela resulta em redução de trabalho porque os subproblemas menores não são recalculados e podem-se encontrar soluções ótimas para subproblemas maiores. Esse processo é realizado até que se obtenha a solução ótima do problema original.

    a) I: verdadeira; II: verdadeira; III: falsa; IV: falsa.

    b) I: verdadeira; II: verdadeira; III: verdadeira; IV: falsa.

    c) I: falsa; II: verdadeira; III: falsa; IV: verdadeira.

    d) I: falsa; II: falsa; III: verdadeira; IV: verdadeira.

    e) I: falsa; II: falsa; III: verdadeira; IV: falsa.

    f) Todas são falsas.

22. Escolha a alternativa correta.

    I) Algoritmos gulosos constroem a solução a cada alternativa, sempre escolhendo a próxima alternativa que oferece o benefício mais evidente e imediato.

    II) Um algoritmo guloso sempre encontra a solução ótima.

    III) O paradigma guloso não é utilizado em problemas de otimização.

    IV) Algoritmos gulosos podem ser aplicados a vários problemas. Estes problemas são do tipo: calcular um subconjunto S de um conjunto com n elementos. S deve satisfazer certas condições e é denominado solução viável para o problema.

    a) I: verdadeira; II: falsa; III: falsa; IV: falsa.

    b) I: verdadeira; II: falsa; III:falsa; IV: verdadeira.

    c) I: falsa; II: verdadeira; III: falsa; IV: falsa.

    d) I: falsa; II: falsa; III: verdadeira; IV: falsa.

    e) Todas são verdadeiras.

    f) Todas são falsas.

23. Escolha a alternativa correta.

    I) Em um algoritmo guloso, associado a cada solução viável tem-se um valor $f(\mathrm{S})$, em que f é chamada de função objetivo, cujo significado é dado na definição do problema. Entre todas as soluções viáveis, deseja-se aquela que tem valor $f(\mathrm{S})$ mínimo ou máximo, e esta é denominada solução ótima.

    II) Um algoritmo guloso aproxima-se da solução pela melhor decisão possível, baseado no ótimo local corrente.

    III) O paradigma guloso é um esquema exaustivo.

    IV) Os algoritmos de Prim e Kruskal são construídos pelo paradigma guloso. Ambos encontram o caminho mínimo entre dois vértices de um grafo.

    a) I: verdadeira; II: verdadeira; III: falsa; IV: falsa.

    b) I: verdadeira; II: verdadeira; III: verdadeira; IV: falsa.

    c) I: verdadeira; II: verdadeira; III: verdadeira; IV: falsa.

    d) I: falsa; II: falsa; III: verdadeira; IV: verdadeira.

e) Todas são verdadeiras.

f) Todas são falsas.

24. Escolha a alternativa correta.

    I) Um exemplo de algoritmo guloso para encontrar o caminho mínimo entre dois vértices de um grafo escolhe a aresta que parece mais promissora em qualquer instante e pode reconsiderar essa decisão, dependendo do que acontecer mais tarde.

    II) Em um algoritmo guloso, uma função de seleção fornece o valor da solução encontrada. Em um algoritmo guloso, outra função é a função objetivo que aparece explicitamente no algoritmo.

    III) Em um algoritmo guloso, a função objetivo indica, a qualquer momento, quais dos candidatos restantes, que não foram nem escolhidos nem rejeitados, são os mais promissores.

    IV) Em um algoritmo guloso, a função de seleção nunca é relacionada à função objetivo.

    a) I: verdadeira; II: falsa; III: falsa; IV: falsa.

    b) I: falsa; II: verdadeira; III: falsa; IV: falsa.

    c) I: falsa; II: falsa; III: verdadeira; IV: falsa.

    d) I: falsa; II: falsa; III: falsa; IV: verdadeira.

    e) I: falsa; II: verdadeira; III: falsa; IV: verdadeira.

    f) Todas são falsas.

25. Complete o algoritmo por *backtracking* de Manber (1989, p. 358). Este algoritmo resolve o problema de verificar se os vértices de um grafo podem ser colorido com três cores, sabendo-se que vértices adjacentes não podem ser coloridos com a mesma cor.

Algoritmo 2.7: 3-cores.

Entrada: grafo *G=(V,E)* não orientado; um conjunto de vértices *U* com suas cores, sendo *U* inicialmente vazio;

Saída: uma atribuição de uma das três cores para cada vértice de *G*;

```
1  início
2      se ( ..... ) então
3          imprima "coloração completa";
4      senão
5          obtenha um vértice v ∉ U;
6          para ( c ← 1 até 3 ) faça
7              se ( nenhum vizinho de v tem cor C ) então
8                  adicione v a U com cor C;
9                  .....;
10             fim
11         fim
12     fim
13 fim.
```

26. Quais linhas do algoritmo anterior devem ser alteradas para construir um algoritmo por *branch-and-bound* para resolver o problema de determinar o número mínimo de cores para colorir os vértices de um grafo, sabendo-se que vértices adjacentes não podem ser coloridos com a mesma cor?

2.12 Notas bibliográficas

Os livros citados de cada seção apresentam diversos exemplos de cada técnica. Nas obras de Goodrich e Tamassia (2004) e Cormen et al. (2009) são encontradas excelentes descrições sobre as principais técnicas para a construção de algoritmos. Ziviani (2011) é uma das melhores obras sobre esses temas, escrita diretamente em português.

3 *Complexidade de tempo*

3.1 Introdução

A análise de complexidade de um algoritmo consiste em mensurar seu tempo de execução e espaço, para que se possa comparar a eficiência de algoritmos diferentes para o mesmo problema. Pode-se considerar a análise de complexidade de tempo como a verificação do tempo que o algoritmo demanda para resolver a tarefa, e a análise de complexidade de espaço, como a verificação da quantidade de memória que o algoritmo demanda para resolver o problema.

Deve-se fixar um modelo de computação para a análise de complexidade de um algoritmo e também, a noção de tamanho da entrada do problema. Ainda, o que expressa o tempo de execução deve ser bem entendido. Esses assuntos são abordados após alguns comentários iniciais e, em seguida, o fato de que o tempo de execução de algoritmos é, geralmente, expresso por funções com termos bastante conhecidos e estes devem ser estudados.

Também deve ser entendido o que significa o crescimento de funções para que a análise assintótica seja realizada. Depois, os tempos assintóticos no pior, no melhor e no caso médio são abordados. Por fim, a análise amortizada é brevemente descrita.

3.2 Comentários iniciais

Na graduação, na disciplina de Arquitetura de Computadores, o professor nos passou a tarefa de criarmos um sistema operacional multitarefa em uma linguagem *assembly*

para um processador específico. A cada nova tarefa, o professor dava pontos para o programa mais rápido e para o menor programa. Para considerar o programa mais rápido, somávamos o tempo de execução de cada mnemômico multiplicado pelo número de vezes que executava. Para considerar o espaço de memória ocupado pelo programa, verificávamos o tamanho dos mnemômicos, os tamanhos dos operandos e da memória ocupada pelas estruturas de dados que utilizavam.

Havia equipes que se desafiavam: duas ou três vezes por semana, comunicavam que tinham conseguido um programa com tempo menor ou que ocupasse menos espaço que o comentado anteriormente. Alguma equipe entrava no "jogo" e também desafiava-os, falando sobre o tempo de execução e a ocupação de espaço obtidos. A disputa era acirrada, instrução a instrução, mnemônico a mnemônico. Poderiam simplesmente não comentar os resultados e esperar que a outra equipe considerasse ter o programa mais rápido e menor, mas o(a)s jovens consideravam divertido comentar e ver a expressão de desânimo das outras equipes.

Em uma das últimas tarefas, um processo importante precisava ser feito, no qual o tempo de execução era essencial. Uma equipe, digamos A, desenvolveu um processo com doze das menores e mais rápidas instruções do *assembly* daquele processador: ganhariam os dois pontos da tarefa, mas avisaram os concorrentes antes do final da semana. No início da semana seguinte, os membros de outra equipe, digamos B, disseram que conseguiram também um programa com doze instruções com a mesma ocupação de espaço, mas que seria um programa mais rápido que o da equipe A. O pessoal da equipe A trabalhou mais e conseguiu um programa com onze instruções, mas menos rápido que o da equipe B. A equipe B ganhou o ponto de programa mais rápido. Depois, o professor disse que o programa da equipe B previamente preparava toda a memória principal com dados específicos e, por isso, conseguiu um programa mais rápido que os programas das demais equipes. Mas, a equipe A ganhou o ponto do programa com menor ocupação de memória.

Professores utilizam exemplos como esse para que os alunos façam algoritmos rápidos e com baixa ocupação de espaço. Mas contar o tempo de execução e a quantidade de memória utilizada pode não ser fácil se uma linguagem de alto nível for utilizada. Ainda, seria necessário um conhecimento profundo dos mnemônicos do processador e

tecnicidades que levariam muito tempo para serem resolvidas. Talvez, o tempo para isso fosse relevante em relação a desenvolver todo um projeto e, mesmo assim, todo o estudo de tempo e espaço somente serviria para o sistema de computação específico. Se o programa fosse construído em outro sistema de computação, o estudo deveria ser refeito. Ainda, para qualquer alteração no próprio sistema de computação particular, como, por exemplo, trocar a memória principal por uma mais veloz, o tempo de execução deveria ser reestudado.

Então, de maneira geral, a análise de complexidade fornece uma maneira de mensurar o tempo e o espaço de algoritmos que resolvem o mesmo problema para se poder compará-los, sem haver a necessidade de implementá-los e realizarem-se simulações experimentais. Embora, em casos complicados, essa segunda opção seja frequentemente utilizada, a análise de complexidade é uma maneira elegante e interessante para a tarefa de comparar algoritmos que resolvem o mesmo problema.

Com a análise de um algoritmo, podem-se verificar os pontos críticos ou que demandam mais atenção para que possam ser melhorados (*assintoticamente*). Esse termo, *assintótico*, é explicado adiante. Em especial, como, atualmente, memória é menos caro que o tempo necessário para execução, então, a análise de complexidade de tempo é considerada mais relevante que a análise de complexidade de espaço.

3.3 Modelo de computação

Apesar de comprovado que os modelos de computação razoáveis conhecidos são equivalentes (veja o capítulo 14), para a análise de complexidade de algoritmos é necessário fixar-se em um modelo de computação. Um modelo de computação é a definição de um conjunto de operações utilizadas na computação e seus custos e é empregado para mensurar a complexidade de tempo e de espaço de um algoritmo. Um modelo de computação é uma *abstração* matemática de uma máquina. Ou seja, essas abstrações não existem na prática e não podem existir fisicamente, devido às suas características intrínsecas.

As complexidades dos algoritmos deste capítulo baseiam-se no modelo de computação hipotético chamado máquina de acesso aleatório, *randon accesss machine*,

ou apenas RAM (SHEPHERDSON; STURGIS, 1963). Não se deve confundir RAM com memória de acesso aleatório. O termo aleatório refere-se à capacidade da unidade central de processamento (CPU) de acessar uma posição arbitrária de memória em apenas uma *operação primitiva*. Entenda por operações primitivas as instruções mais básicas do computador. Os mnemônicos da linguagem *assembly* da máquina representam e permitem codificar essas instruções. As operações primitivas são abordadas adiante. O modelo RAM fornece uma abstração do padrão da arquitetura mostrada por von Neumann (republicação em 1963). Mais especificamente, uma RAM pode ser considerada um modelo simplificado dos computadores reais.

Uma RAM consiste de uma CPU conectada a uma memória, que é onde os dados são depositados e que é composta por um número infinito de registradores (ou localizações de memória). Pode-se armazenar uma só palavra em cada localização da memória. Se o conteúdo dessa localização de memória é válido, então, esse conteúdo representa um tipo básico da linguagem utilizada. O tipo básico é o inteiro e o tipo *float* pode também ser considerado por clareza. Lembre-se que um caracter é um inteiro de oito *bits*.

O modelo RAM limita o número de *bits* a serem armazenados em uma localização de memória. Haveria consequências indesejadas se essa suposição não fosse considerada. Se isso não fosse suposto, poder-se-ia construir algoritmos no modelo RAM que seriam implementados em computadores existentes somente de uma maneira muito ineficiente. Com isso, haveria um colapso das classes de complexidade P-*Time* e P-*SPACE*. A definição da classe P-*Time* e uma introdução à classe P-*SPACE* o leitor encontra no capítulo 15, deste texto.

As operações primitivas de uma RAM manipulam os registradores e a memória e realizam operações aritméticas. Mais especificamente, uma operação primitiva é uma instrução de baixo nível com tempo de execução constante. Essas instruções são executadas sequencialmente, isto é, não há concorrência. Numa RAM, há instruções comumente encontradas em computadores reais.

Para a análise dos algoritmos a seguir, considere uma RAM simplificada com determinadas operações primitivas. Estão incluídas entre essas operações primitivas: operações aritméticas (*add, subtract, multiply, divide, remainder, floor, ceiling*), movi-

mento de dados (*load, store, copy*), controle (*jump* condicional e incondicional, chamada e retorno de sub-rotina) e endereçamento direto e indireto para indexação de vetores. Veja detalhes em Cormen et al. (2009, p. 23-24). Cada instrução exige uma quantidade de tempo específica e fixa. Utiliza-se neste texto o *critério de custo uniforme*, em que cada instrução exige uma unidade de tempo e cada localização de memória representa uma unidade de espaço:

- cada valor é um item primitivo de dados;
- a memória ocupada por uma variável é o número de entradas no vetor que a representa;
- a memória utilizada por uma RAM é o total de memória ocupada pelos dados;
- o tempo de execução de um programa em uma RAM é o número de instruções executadas.

Um programa de uma RAM não pode modificar-se porque não permanece armazenado na memória da RAM. Em outro modelo de computação similar à RAM, uma máquina com programa armazenado de acesso aleatório ou RASP (*random access stored program machine*) (SHEPHERDSON; STURGIS, 1963), o programa permanece na memória da máquina e pode modificar-se. Os modelos de computação RAM e RASP são quase idênticos. Numa máquina RASP, o endereçamento indireto não existe porque não é necessário, mas pode ser simulado. Uma introdução a RAMs, suas equivalências a máquinas de Turing e exemplos podem ser encontradas no primeiro capítulo de Aho, Hopcroft e Ullman (1974).

3.4 Tamanho e formato da entrada do algoritmo

O tempo que um algoritmo demora para resolver um problema depende do tamanho de sua entrada. Por exemplo, em geral, um algoritmo de ordenação demora mais para ordenar um conjunto grande de números do que um conjunto pequeno. Em geral,

o tempo de execução de um algoritmo cresce proporcionalmente com o tamanho da entrada e com isso, ele é, geralmente, expresso por uma função que depende do tamanho da entrada do algoritmo.

A noção de tamanho de entrada depende do problema que o algoritmo resolve. Para muitos problemas, a medida adequada é o número de itens da entrada. Novamente, um exemplo frequentemente utilizado é a quantidade de itens n de uma sequência a ser ordenada. A quantidade de itens n é o parâmetro de uma função que expressa o tempo de execução do algoritmo de ordenação. Para outros problemas, a medida apropriada do tamanho da entrada é o número de *bits* necessários para representar a entrada em notação binária ordinária. Um exemplo clássico dessa forma de representação da entrada é a multiplicação de dois inteiros. Outras vezes, é mais apropriado descrever o tamanho da entrada com dois números em vez de um. Nesse caso, o exemplo clássico é um problema que tem como entrada um grafo. Algoritmos em grafos são descritos nos capítulos de 8 a 12, deste texto. Se a entrada de um algoritmo é um grafo, o tamanho da entrada é descrita adequadamente pelo número de vértices e o número de arestas na instância do grafo. É importate lembrar que se *deve indicar a medida do tamanho da entrada que está sendo usada em cada problema sendo estudado*. A descrição anterior é baseada em Cormen et al. (2009, p. 25), obra que deve ser consultada para detalhes sobre o tamanho da entrada de algoritmos.

A *forma* como os dados estão na entrada também é relevante no tempo de execução de algoritmos. Isso porque há algoritmos que dependem de como a entrada é disposta. Exemplo clássico é que há algoritmos de ordenação que demoram diferentes tempos para ordenar (ou afirmar que a sequência está ordenada) duas sequências de mesmo tamanho. Isso depende de quão ordenada a sequência de entrada já está (CORMEN et al., 2009, p. 24), o que será abordado adiante.

3.5 Tempo de execução

Para verificar o tempo exato de execução de determinado programa, poder-se-ia contar o número de vezes que ocorre cada operação primitiva, multiplicado pelo tempo em *clocks* da operação primitiva da máquina. O somatório dessas multiplicações forneceria

o tempo exato do programa, mas seria uma tarefa extremamente árdua, principalmente em linguagens de alto nível. Ainda, essa análise seria dependente da máquina.

Para tornar essa tarefa viável, conta-se o número de vezes que cada linha do *algoritmo* é executada. Cada linha (exceto comentários, início, etc) do algoritmo é composta por um conjunto de uma ou mais operações primitivas que têm seus determinados tempos de execução. O tempo de execução de uma operação primitiva é, geralmente, diferente em sistemas computacionais distintos e é necessário que a descrição do tempo de execução do algoritmo seja tão independente de máquina quanto possível. Com isso, são diferentes os tempos de execução de linhas distintas porque, geralmente, apresentam operações primitivas desiguais. Ocorre que, independente do número de operações primitivas de uma determinada linha e seu tempo de execução, uma execução dessa linha terá tempo constante em qualquer sistema de computação em que esse algoritmo venha a ser implementado.

Com isso, em vez de determinar o tempo de execução exato de cada algoritmo através de cada operação primitiva, conta-se o número de vezes que cada linha do algoritmo será executada. Geralmente, a linha que é a mais executada determina o termo mais relevante da função que expressa o tempo de execução do algoritmo. Mais especificamente, um conjunto de linhas do algoritmo pode ser considerado uma *iteração* do algoritmo. O tempo de execução do algoritmo é "dominado" pelo número de iterações da linha (ou conjunto de operações primitivas) mais executada desse conjunto.

Certamente, o número de vezes que determinadas linhas são executadas é dependente da entrada do algoritmo. Como um primeiro exemplo, será vista a análise de um algoritmo que mostra o somatório de 0 a $n \in \mathbb{N}$.

Nos algoritmos deste texto, as linhas de início (linha 1) e fim (linha 11 do algoritmo Somat) são numeradas. Considere uma linha de início apenas como um rótulo para a primeira instrução do algoritmo. Com isso, pode-se não considerar o tempo de execução de uma linha de início. A linha fim representa a instrução de retorno do algoritmo. O tempo de execução dessa linha pode ser considerada como $\Theta(1)$ em todos os algoritmos. Com isso, linhas de início e fim não serão consideradas nas análises deste texto.

Algoritmo 3.1: Somat.

Entrada: $n \in \mathbb{N}$;
Saída: $s = \sum_{i=0}^{n} i$;

```
1  início
2      se ( n < 0 ) então
3          retorna -1;
4      fim
5      inteiro s ← 0;
6      enquanto ( n > 0 ) faça
7          s ← s + n;
8          n ← n − 1;
9      fim
10     retorna s;
11 fim.
```

No algoritmo anterior, cada uma das linhas 2, 5 e 10 é executada uma só vez e essas linhas são independentes do tamanho da entrada n. A linha 4 é um rótulo para o salto (*jump*) da linha 2, se codificado em linguagem *assembly*. Com isso, linhas com *fim-se*, como a linha 4, não têm tempo de execução e podem ser desconsideradas na análise. Por exemplo, livros, como o de Cormen et al. (2009), nem mostram essas linhas em seus algoritmos. Teoricamente, a condição da linha 2 é desnecessária, mas foi incluída para exemplificação. A linha 3 só seria executada se a condição da linha 2 fosse satisfeita, e mesmo que isso ocorresse, a linha 3 seria executada uma só vez. Já o número de vezes que a linha 6 é executada depende do tamanho da entrada n. A linha 6 é executada $(n+1) \in \mathbb{N}^*$ vezes. Em linguagem *assembly*, na linha 9 haveria um salto incondicional para a linha 6. Esse salto é realizado enquanto a condição da linha 6 é satisfeita. Com isso, o número de vezes que são executadas linhas f_l que, como a linha 9, representam o final de laços de repetição, é sempre uma vez a menos que o número de vezes que são executadas linhas i_l que, como a linha 6, contêm a condição do laço de repetição. Como descrito, o tempo para executar uma vez cada linha é constante. Com isso, o tempo das n vezes em que linhas f_l, como a linha 9, são executadas é assintoticamente (descrito adiante) irrelevante em relação às n+1 vezes em que linhas i_l, como a linha

6, são executadas. Por exemplo, obras como a de Cormen et al. (2009) também não mostram linhas de final de laço de repetição em seus algoritmos. Por fim, o número de vezes em que cada das linhas 7 e 8 é executada também é uma vez a menos que o número de vezes que a linha 6 é executada.

O número de vezes em que as linhas de Somat são executadas é mostrado na Tabela 3.1. Claramente, se a linha 3 é executada, então, as linhas 5 a 10 não são.

Tabela 3.1: Número de execuções das linhas de Somat.

linha	número de vezes executada
2	1
3	0 ou 1
4	não se aplica
5	1
6	n+1
7	n
8	n
9	n
10	1

Considere c_i o tempo de execução da linha i e *T(n)* o tempo de execução de Somat. Em Somat, se $n < 0$, então, seria retornado um código de erro. Se $n = 0$, então, este é o melhor caso para o tempo de execução do algoritmo. A linha 3 não é executada e $c_2 + c_5 + c_6 + c_{10}$ representa o tempo de execução do algoritmo. Se $n > 0$, então, a função que expressa o tempo de execução de Somat no pior caso é linear em n, isto é,

$$T(n) = c_1 + c_4 + (n+1)c_5 + (c_6 + c_7 + c_8)n + c_9. \tag{3.1}$$

A equação (3.1) pode ser escrita como

$$T(n) = an + b = O(n), \tag{3.2}$$

em que $a = c_5 + c_6 + c_7 + c_8$ e $b = c_1 + c_4 + c_5 + c_9 = \Theta(1)$.

O tempo de execução de Somat no caso médio é equivalente ao pior caso. O caso médio será visto adiante. Outro aspecto importante é que linhas de pseudocódigos com descrições genéricas devem ser estudadas separadamente, ou deve-se considerá-las como chamadas para sub-rotina, e analisa-se separadamente o tempo de execução de tal "sub-rotina" (CORMEN et al., 2009, p. 25).

Este texto limita-se a introduzir a análise de complexidade de tempo de algoritmos. No entanto, o mesmo pode ser definido para a análise de complexidade de espaço, ao se substituir o *tempo* utilizado para executar uma instrução pelo *espaço* ocupado de cada localização de memória. Para se aprofundar na análise de complexidade de espaço, veja Aho, Hopcroft e Ullman (1974, p. 12), a seção 17.2 de Sudkamp (2006) e o capítulo 8 de Sipser (2007), como exemplos de livros que abordam esse tema.

3.6 Crescimento de funções e notação assintótica

Cormen et al. (2009, p. 43) explicam que a ordem de crescimento de funções do tempo de execução dos algoritmos fornece uma caracterização da eficiência do algoritmo e também permite comparar o desempenho relativo de algoritmos distintos que resolvem o mesmo problema. Como descrito, apesar de, por vezes, ser possível determinar o tempo de execução exato de um algoritmo, a precisão pode não justificar a problemática do cálculo. Para entradas bastante grandes, as constantes multiplicativas e termos de baixa ordem de um tempo de execução exato são dominados pelos efeitos do tamanho da entrada no termo de mais alta ordem da função. Ao se estudar tamanhos de entradas bastante grandes para se conhecer somente a ordem de crescimento relevante, estuda-se a eficiência *assintótica* de algoritmos: como o tempo de execução de um algoritmo aumenta com o tamanho da entrada *no limite*. Note que não é descrito *no infinito*, pois um computador é uma máquina finita, mas essa noção está no contexto. Geralmente, um algoritmo que é assintoticamente mais eficiente será a melhor escolha, exceto para entradas muito pequenas (CORMEN et al., 2009, p. 43).

Expressar uma igualdade é extremamente forte. Por exemplo, em $x = 3{,}14159265$, afirma-se que x é exatamente igual ao número real descrito. Já afirmar que $x = \pi$ é algo diferente, pois não é possível representar π exatamente num computador, pelo fato de

ele ser uma máquina finita. O que se fez, nesse exemplo, foi obter-se uma aproximação, mesmo que o número de dígitos utilizados na mantissa seja muito grande.

Matematicamente, não se pode afirmar que um objeto, por exemplo, uma cadeira, é igual a outra cadeira. No máximo, as cadeiras são similares ou pertencem à mesma classe de objetos. Todavia, não são exatamente iguais porque alguma característica, por menor que seja (em que nome, código e número são exemplos), faz com que sejam distintas. Por outro lado, se há objetos matemáticos que são considerados os mesmos, então, afirma-se que são *equivalentes*. Com isso, pode-se ter a noção do que significa uma aproximação em matemática: $f(n) \cong g(n) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$.

Uma das primeiras dificuldades ao se aprender programação pode ser entender o incremento de uma variável, por exemplo, $x = x + 1$. Após aprender que o conteúdo da variável, que é armazenado em memória principal, é copiado para outras posições de memória, por lá incrementado e depois o resultado é atribuído à posição de memória correspondente à variável x, é que se pode entender por que se faz tal abuso em linguagens, como Fortran, Basic, C, C++, Java, bem como o ':=' do Pascal e Visual Basic. A seguir são formalizadas as notações assintóticas utilizadas na análise de complexidade de algoritmos.

3.6.1 Notação *O*

A notação *O* ("O" grande, *big-oh* ou Omicron) fornece um *limite superior de crescimento assintótico de uma função* em relação a um fator constante. Geralmente, ela é utilizada para expressar o pior caso da execução de um algoritmo. Para uma função dada *g(n)*, denota-se por *O(g(n))* o conjunto de funções

$$O(g(n)) = \{f(n) : (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}^*) \mid (\forall n \geq n_0)\ 0 < f(n) \leq cg(n)\}. \tag{3.3}$$

Essa definição significa que uma função $f(n) \in O(g(n))$ se há constante positiva c, tal que *f(n)* é igual ou inferior a *cg(n)*, *para n suficientemente grande*. Uma representação da notação *O* pode ser observada na Figura 3.1a.

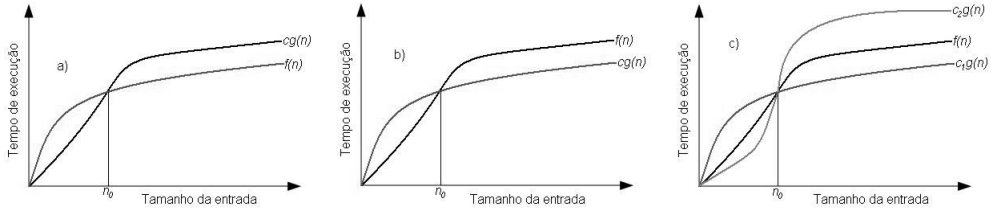

Figura 3.1: Representação das notações a) O, b) Ω e c) Θ.

Note que, na definição, $f(n) > 0$. Isso porque, mesmo que n = 0, o algoritmo terá algum tempo de execução ao realizar algum processamento. Neste caso, possivelmente, o tempo de execução é expresso por $O(1)$.

A notação assintótica apresenta diversas peculiaridades. É comum encontrar o seguinte abuso de linguagem: usa-se *f(n) = O(g(n))* para expressar $f(n) \in O(g(n))$: diz-se que *f(n)* é da ordem de *g(n)*. Na notação *f(n) = O(g(n))*, pode ser estranho igualar uma função a um conjunto de funções, mas "é prática comum de milhares de matemáticos por tantos anos", nas palavras de Knuth (1976, p. 20). Depois dessa carta, a quantidade de pesquisadores que passou a utilizar essa notação aumentou. Mesmo assim, não use algo como *O(g(n)) = f(n)*.

Como um exemplo do uso da notação O, para verificar que $2^n = O(4^n)$, deve-se verificar $2^n \leq c \cdot 4^n$. Basta escolher, por exemplo, c = 1 e n_0=1 e tem-se $2^n \leq 4^n$, para todo $n \geq n_0$. Note que não é necessário encontrar todas as constantes c e n_0, mas *alguma* constante c e *alguma* constante n_0 que satisfaçam à inequação.

Uma notação comum é $\log_b n = O(\lg n)$ porque $n = a^{\log_a n}$ e, aplicando-se $\log_c$ em ambos os lados, obtém-se $\log_c n = log_c a^{\log_a n} = \overbrace{\log_c a}^{\text{constante}} \cdot \log_a n = O(\lg n)$, em que a e c são constantes. Também pode-se demonstrar isso como $\log_b n = \frac{\log_a n}{\log_a b} = O(\lg n)$, em que a, b e c são constantes.

Note que, para uma constante c, é verdade que $c \cdot f(n) = O(O(f(n))) = O(f(n))$. Note também que $O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$, mas $\overbrace{O(1) + O(1) + \cdots + O(1)}^{n \text{ vezes}} = n \cdot O(1) = O(n \cdot 1) = O(n)$. Também ocorre que

$f(n)O(g(n)) = O(f(n))O(g(n)) = O(f(n)g(n))$. Ainda ocorre que $O(\lg n) \nsubseteq O(\sqrt{n}) \nsubseteq O(n) \nsubseteq O(n \lg n) \nsubseteq O(n^2) \nsubseteq O(n^3) \nsubseteq O(2^n) \nsubseteq O(n!)$. Lembre-se que $O(.)$ é um conjunto. Então, ocorre que $O(\lg n) \nsubseteq O(\sqrt{n}) \nsubseteq O(n) \nsubseteq O(n \lg n) \nsubseteq O(n^2) \nsubseteq O(n^3) \nsubseteq O(2^n) \nsubseteq O(n!)$. Também é interessante estudar este exemplo: $O(\lg n)^{-1} \nsubseteq O(\lg^{-1} n)$, que fica como exercício.

Por último, o limite assintótico superior fornecido pela notação O pode não ser assintoticamente estrito. O limite $4n^3 \in O(n^3)$ é assintoticamente estrito, mas, o limite $n^2 \in O(n^3)$ não é assintoticamente estrito.

3.6.2 Notação Ω

A notação Ω fornece um *limite inferior de crescimento assintótico de uma função* em relação a um fator constante. Geralmente, ela é utilizada para expressar o melhor caso da execução de um algoritmo. Para uma função dada *g(n)*, denota-se por $\Omega(g(n))$ o conjunto de funções

$$\Omega(g(n)) = \{f(n) : (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}^*) \mid (\forall n \geq n_0)\ 0 < cg(n) \leq f(n)\}. \tag{3.4}$$

Essa definição significa que uma função $f(n) \in \Omega(g(n))$ se há constante positiva c, tal que *f(n)* é igual ou superior a *cg(n)*, *para n suficientemente grande*. Uma representação da notação Ω pode ser observada na Figura 3.1b.

Note que, na definição, $cg(n) > 0$. Isso porque, mesmo que $n = 0$, o algoritmo terá algum tempo de execução ao realizar algum processamento. Nesse caso, possivelmente, o tempo de execução é expresso por $\Omega(1)$.

É comum encontrar o seguinte abuso de linguagem: usa-se $f(n) = \Omega(g(n))$ para expressar $f(n) \in \Omega(g(n))$. Por exemplo, para verificar que $4^n = \Omega(2^n)$, deve-se verificar $4^n \geq c \cdot 2^n$. Basta escolher, por exemplo, $c = 1$ e n_0=1 e tem-se $4^n \geq 2^n$, para todo $n \geq n_0$. Note que não é necessário encontrar todas as constantes c e n_0, mas *alguma* constante c e *alguma* constante n_0 que satisfaçam à inequação.

Note que $f(n) \in \Omega(g(n)) \leftrightarrow g(n) \in O(f(n))$, que é uma simetria transposta. Um exemplo para ser estudado é $\Omega(\lg n)^{-1} \subset O(\lg^{-1} n)$, que é correto, apesar de ser uma notação "esquisita" (GRAHAM; KNUTH; PATASHNIK, 1995, p. 328).

Por último, o limite assintótico inferior fornecido pela notação Ω pode não ser assintoticamente estrito. O limite $4n^3 \in \Omega(n^3)$ é assintoticamente estrito, mas, o limite $4n^3 \in \Omega(n^2)$ não é assintoticamente estrito.

3.6.3 Notação Θ

A notação Θ fornece um limite estrito, isto é, *limites superior e inferior*, para o *crescimento assintótico de uma função*. Para uma função dada *g(n)*, denota-se por $\Theta(g(n))$ o conjunto de funções

$$\Theta(g(n)) = \{f(n) : (\exists c_1, c_2 \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}^*) \mid (\forall n \geq n_0)\ 0 < c_1 g(n) \leq f(n) \leq c_2 g(n)\}. \tag{3.5}$$

Essa definição significa que uma função $f(n) \in \Theta(g(n))$ se há constantes positivas c_1 e c_2, tal que *f(n)* esteja entre $c_1 g(n)$ e $c_2 g(n)$, *para n suficientemente grande*. Note que

$$f(n) \in \Theta(g(n)) \leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n)). \tag{3.6}$$

Perceba que na definição, $c_1 g(n) > 0$. Isso porque, mesmo que $n = 0$, o algoritmo terá algum tempo de execução ao realizar algum processamento. Nesse caso, possivelmente, o tempo de execução é expresso por $\Theta(1)$.

Note ainda que $\Theta(g(n))$ está contido em *O(g(n))*. É comum encontrar o seguinte abuso de linguagem: usa-se $f(n) = \Theta(g(n))$ para expressar $f(n) \in \Theta(g(n))$. Por exemplo, para verificar que $2n^2 - 1 = \Theta(n^2)$, devem-se escolher constantes c_1 e c_2, tal que $c_1 n^2 \leq 2n^2 - 1 \leq c_2 n^2$. Pode-se escolher $n_0 = 1$ e obtêm-se $c_1 \leq 1$ e $c_2 \geq 1$ para verificar que $2n^2 - 1 = \Theta(n^2)$. Note que é suficiente escolher *alguma* para cada das constantes n_0, c_1 e c_2, para satisfazer à inequação.

Outro abuso de linguagem bastante utilizado é expressar como $\Theta(n^0)$, $\Theta(1)$ ou $O(1)$ a linha de um algoritmo com tempo de execução constante. Isso porque toda constante é um polinômio de grau zero. Mais especificamente, $\Theta(1)$ e $O(1)$ expressam uma função constante em relação a alguma variável. Uma representação da notação Θ pode ser observada na Figura 3.1c.

3.6.4 Notação *o*

Usa-se a notação *o* para expressar um limite superior que não é assintoticamente estrito. Para uma função dada *g(n)*, denota-se por *o(g(n))* o conjunto de funções

$$o(g(n)) = \{f(n) : (\forall c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}^*) \mid (\forall n \geq n_0)\ 0 < f(n) < cg(n)\}. \tag{3.7}$$

Se $f(n) = o(g(n))$, então, $f(n)$ é *assintoticamente menor* que $g(n)$. Por exemplo, $n^2 \in o(n^3)$, mas $4n^3 \notin o(n^3)$. A principal diferença entre as notações *o* e *O* é que: se $f(n) \in O(g(n))$, então, o limite $0 \leq f(n) \leq cg(n)$ é verificado para *alguma* constante $c > 0$; se $f(n) \in o(g(n))$, então, o limite $0 \leq f(n) \leq cg(n)$ é verificado para *toda* constante $c > 0$. Intuitivamente, na notação *o*, a função *f(n)* torna-se insignificante em relação a *g(n)*:

$$f(n) \in o(g(n)) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0. \tag{3.8}$$

A notação $f(n) \prec g(n) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ foi introduzida por Bois-Reymond (1871), conforme Knuth (1976, p. 21).

3.6.5 Notação ω

Usa-se a notação ω para expressar um limite inferior que não é assintoticamente estrito. Para uma função dada *g(n)*, denota-se por $\omega(g(n))$ o conjunto de funções

$$\omega(g(n)) = \{f(n) : (\forall c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}^*) \mid (\forall n \geq n_0)\ 0 < cg(n) < f(n)\}. \tag{3.9}$$

Se $f(n) = \omega(g(n))$, então, $f(n)$ é *assintoticamente maior* que $g(n)$. Por exemplo, $4n^3 \in \omega(n^2)$, mas $4n^2 \notin \omega(n^2)$.

A principal diferença entre as notações ω e Ω é dada a seguir. Se $f(n) \in \Omega(g(n))$, então, o limite $0 \leq cg(n) \leq f(n)$ é verificado para *alguma* constante $c > 0$. Se $f(n) \in \omega(g(n))$, então, o limite $0 \leq cg(n) \leq f(n)$ é verificado para *toda* constante $c > 0$:

$$f(n) \in \omega(g(n)) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty. \tag{3.10}$$

Se o limite existe, então, *f(n)* torna-se arbitrariamente grande em relação a $g(n)$, quando $n \to \infty$. Note que $f(n) \in \omega(g(n)) \leftrightarrow g(n) \in o(f(n))$, que é uma simetria transposta.

3.6.6 Algumas considerações

Deve-se ter a noção de quando e como utilizar a notação assintótica. Com a notação assintótica, enfatiza-se o que é relevante na expressão que denota o tempo de execução de um algoritmo e ignora-se o que não é importante.

Note que, ao analisar exatamente o tempo de execução de um algoritmo, pode-se obter uma equação do tipo $T(n) = 4n^3 + 5n^2 + 3n + 7n + 2$, em que *T(n)* expressa o tempo de execução exato. No entanto, toda vez que se quisesse expressar o tempo de execução do algoritmo, seria desnecessário utilizar todos os termos do polinômio, já que $\lim_{n\to\infty} \frac{5n^2+3n+7n+2}{4n^3} = 0$. Afirma-se que $g(n) = 4n^3$ *domina* ou é assintoticamente maior que $f(n) = 5n^2+3n+7n+2$. Nota-se que $g_1(n) = 5n^2$ domina $f_1(n) = 3n+7n+2$ e assim por diante. Então, pode-se afirmar que o tempo de execução do algoritmo é dominado pelo termo de maior grau, que é $4n^3$. Ainda no exemplo, a constante multiplicativa 4 (ou qualquer outra constante multiplicativa) não é relevante para n suficientemente grande. Dessa forma, pode-se afirmar que $T(n) = O(n^3)$.

Deve-se ter a noção de que funções que aparecem com frequência são assintoticamente maiores ou menores entre si. Como exemplos, veja uma sequência de funções ordenadas a partir das assintoticamente menores para as maiores: $\lg n$, $\sqrt{n}$, n, $n \lg n$, n^2, n^3, 2^n, $n!$, n^n.

Ainda, deve-se ter a noção de quais termos de funções são assintoticamente maiores, menores ou de mesmo tamanho para se obter o termo de maior grau da expressão e ignorar-se os demais. Por exemplo, na equação $T(n) = n! + 2^n + n^9$, o termo 2^n domina o tempo n^9 para n suficientemente grande. Por sua vez, o termo $n!$ domina o termo 2^n para n suficientemente grande. Note que essa relação é transitiva: se *a(n)* domina *b(n)* e *b(n)* domina *d(n)* para n suficientemente grande, então, *a(n)* domina *d(n)* para n suficientemente grande. Dessa forma, no exemplo, o termo $n!$ domina o termo n^9 para n suficientemente grande. Lembre-se que uma relação R transitiva significa que $(\forall x, y, z)(xRy \wedge yRz) \rightarrow (xRz)$. O capítulo 9 da obra de Graham, Knuth e Patashnik (1995) é uma fonte excelente para a descrição detalhada sobre comportamento assintótico.

3.7 Exemplos de análise de complexidade: algoritmos para avaliação de polinômios

O problema da avaliação de um polinômio de grau n pode ser descrito como, dado x, calcular $p_n(x) = \sum_{i=0}^{n} (a_i x^i)$, em que $a_n \neq 0$. Veja Knuth (1981, p. 467-496) para descrição detalhada sobre avaliação de polinômios. A seguir, são apresentadas análises de algoritmos para avaliação dessas funções[1].

3.7.1 Algoritmo ingênuo

Um programador iniciante poderia construir o algoritmo ingênuo a seguir. A linha 2 é executada uma vez. A linha 3 é executada n+1 vezes. A linha 4 é executada n vezes por causa do laço de repetição da linha 3. No entanto, a cada iteração, o número de execuções da linha 4 é diferente. Mais precisamente, por causa da exponenciação x^i, o custo de execução da linha 4 é i. Isso significa que na primeira iteração, a linha 4 é executada uma vez. Por sua vez, na segunda iteração, a linha 4 é executada duas vezes,

[1] As análises desta subseção foram apresentadas na forma de *slides* guardados como notas das aulas proferidas pelo professor C. R. Ribeiro da Universidade Federal Fluminense e, gentilmente, cedidos pelo professor H. A. X. da Costa da Universidade Federal de Lavras.

e assim por diante. Isso ocorre até que $i = n$. Nessa iteração, a linha 4 é executada n vezes. O número de execuções da linha 4 é dado por $\sum_{i=1}^{n} i = \frac{n^2+n}{2}$. A linha 5 é executada n vezes. A linha 6 é executada uma vez. Considerando $T_1(n)$ como a função que expressa o tempo de execução desse algoritmo e c_i o tempo de execução da linha i, obtém-se a equação 3.11.

Algoritmo 3.2: PolinômioIng.

Entrada: $x \in \mathbb{R}, n \in \mathbb{N}$ e os n+1 coeficientes do polinômio em um *array* A;
Saída: $p_n(x) = \sum_{i=0}^{n} (A_i x^i)$;

```
1 início
2     p_n ← A_0;
3     para ( i ← 1 até n ) faça
4         p_n ← p_n + A_i * x^i;
5     fim
6     retorna p_n;
7 fim.
```

$$T_1(n) = c_2 + c_3(n+1) + c_4\frac{n^2+n}{2} + c_5 n + c_6 = \Theta(1) + \Theta(n) + \Theta(n^2) + \Theta(n) + \Theta(1) = \Theta(n^2). \tag{3.11}$$

3.7.2 Algoritmo linear

Para o tempo de execução assintótico do algoritmo anterior ser melhorado, a linha 4 deve ser repensada. Um segundo algoritmo para a avaliação de um polinômio de grau n é descrito a seguir.

As linhas 2, 3 e 8 são executadas uma vez cada. A linha 4 é executada n+1 vezes. As linhas 5, 6 e 7 são executadas n vezes cada. Considerando c_i o tempo de execução da linha i e $T_2(n)$ como a função que expressa o tempo de execução desse algoritmo, obtém-se

$$T_2(n) = c_2 + c_3 + c_8 + c_4(n+1) + (c_5 + c_6 + c_7)n = (c_4 + c_5 + c_6 + c_7)n + c_2 + c_3 + c_4 + c_8$$

como o tempo de execução exato desse algoritmo, que pode ser expresso em notação assintótica como $\Theta(n)$. Nota-se que existem constantes $c_l \leq c_4 + c_5 + c_6 + c_7 \leq c_g$ para $n_0 \geq 1$.

Algoritmo 3.3: Polinômio.

Entrada: $x \in \mathbb{R}, n \in \mathbb{N}$ e os n+1 coeficientes do polinômio em um *array* A;
Saída: $p_n(x) = \sum_{i=0}^{n} (A_i x^i)$;

```
1 início
2     p_n ← A_0;
3     y ← x;
4     para ( i ← 1 até n ) faça
5         p_n ← p_n + A_i * y;
6         y ← y * x;
7     fim
8     retorna p_n;
9 fim.
```

3.7.3 Algoritmo de Horner

Considere $(\forall k \in \mathbb{N}^*)\ p_k = a_{n-k} + p_{k-1}x$ e $p_0(x) = a_n$, ou seja,

$$\begin{aligned}
&p_0(x) = a_n \\
&p_1(x) = a_{n-1} + a_n x \\
&p_2(x) = a_{n-2} + (a_{n-1} + a_n x)x = a_{n-2} + a_{n-1}x + a_n x^2 \\
&p_3(x) = a_{n-3} + x(a_{n-2} + a_{n-1}x + a_n x^2) = a_{n-3} + a_{n-2}x + a_{n-1}x^2 + a_n x^3) \\
&\ldots \\
&p_n(x) = \sum_{i=0}^{n} (a_i x^i)
\end{aligned}$$

é o Algoritmo de Horner para avaliação eficiente de polinômios na forma monomial. Um monômio é a forma mais simples de uma expressão algébrica. Um monômio contém apenas o produto de constantes e variáveis. Ou seja, um monômio é um polinômio com um só termo. Uma base monomial é um modo de descrever unicamente um polinômio ao utilizar uma combinação linear de monômios. Por exemplo, veja Lang (1974) para detalhes. O Algoritmo de Horner é descrito a seguir.

Algoritmo 3.4: Horner.

Entrada: $x \in \mathbb{R}, n \in \mathbb{N}$ e os n+1 coeficientes do polinômio em um *array* A;
Saída: $p_n(x) = \sum_{i=0}^{n} (A_i x^i)$;

```
1 início
2 |   p_n ← A_n;
3 |   para ( i ← n − 1 até 0, passo -1 ) faça
4 |   |   p_n ← A_i + x ∗ p_n;
5 |   fim
6 |   retorna p_n;
7 fim.
```

As linha 2 e 6 são executadas uma vez cada. A linha 3 é executada n+1 vezes. As linhas 4 e 5 são executadas n vezes cada. Considerando c_i o tempo de execução da linha i e $T_3(n)$ como a função que expressa o tempo de execução desse algoritmo, obtém-se

$$T_3(n) = c_2 + c_6 + c_3(n+1) + (c_4 + c_5)n = (c_3 + c_4 + c_5)n + c_2 + c_3 + c_6 \qquad (3.12)$$

como o tempo de execução exato desse algoritmo, que pode ser expresso em notação assintótica como $\Theta(n)$. Nota-se que existem constantes $c_l \leq c_3 + c_4 + c_5 \leq c_g$ para $n_0 \geq 1$.

O termo $c_3 n$ é associado com n decrementos da variável do laço e com n condições do laço de repetição. O tempo de $c_3 n$ de $T_3(n)$ é similar ao tempo de $c_4 n$ de $T_2(n)$. O termo $c_4 n$ de $T_3(n)$ é associado com n multiplicações, adições e atribuições. O tempo de $c_4 n$ de $T_3(n)$ é similar ao tempo de $c_5 n$ de $T_2(n)$. Os tempos de execução dos termos $c_5 n$

de $T_3(n)$ e c_7n de $T_2(n)$ são similares, já que ambos representam n saltos incondicionais. Em $T_2(n)$ ainda há o tempo de execução do termo c_6n associado com n multiplicações e atribuições.

Ambos, $T_2(n)$ e $T_3(n)$, contêm uma atribuição inicial associada com a constante c_2, uma atribuição de variável do laço e uma condição do laço de repetição associadas com c_3 em $T_2(n)$ e c_4 em $T_3(n)$, e as constantes c_6 em $T_3(n)$ e c_8 em $T_2(n)$ são associadas ao retorno do procedimento. No entanto, $T_2(n)$ ainda contém a constante c_3 associada com uma atribuição. Nota-se que o Algoritmo de Horner apresenta menos operações primitivas em relação ao algoritmo linear, mas as operações adicionais realizadas neste são *assintoticamente* irrelevantes em relação às operações realizadas em $T_3(n)$.

O detalhamento da diferença do tempo de execução exato desses dois algoritmos não é difícil porque são pequenos. Mas, imagine a quantidade de esforço para comparar dois algoritmos grandes. Já que ambos os algoritmos apresentam tempo de execução da ordem de *O(n)*, nota-se como a notação assintótica facilita a análise ao ignorar termos de pequena relevância e constantes multiplicativas na função que expressa o tempo de execução de um algoritmo.

Apesar de o algoritmo ser conhecido por Horner (1819), o método já era conhecido há séculos. Veja Temple (1986, p. 142) e Berggren (1990) para descrições detalhadas.

3.8 Algoritmo para teste de primalidade

O algoritmo a seguir verifica a primalidade de um número. Se a condição da linha 2, com custo $O(1)$, é satisfeita, então, o algoritmo termina. As linhas 3 e 8 são executadas uma vez cada e têm custo $O(1)$. Por causa da condição $i * i \leq n$ do laço de repetição na linha 4 e o incremento da variável do laço na linha 6, o custo de execução da linha 4 é $\lfloor \sqrt{n} \rfloor$. Claramente, o custo de execução da linha 4 domina assintoticamente o custo de execução das linhas internas do laço de repetição. Dessa forma, o custo de execução desse algoritmo é $O(\sqrt{n})$.

Algoritmo 3.5: Primalidade.

Entrada: $n \in \mathbb{N}^*$;
Saída: booleano com a primalidade de n;

```
1 início
2     se ( n < 2 ) então retorna falso;
3     i ← 2;
4     enquanto ( i * i ≤ n ) faça
5         se ( mod(n,i) = 0 ) então retorna falso;
6         i ← i + 1;
7     fim
8     retorna verdadeiro;
9 fim.
```

3.9 Análise de complexidade de dois algoritmos de ordenação

Aplica-se o raciocínio dedutivo na análise de um algoritmo para determinar o limite assintótico do seu tempo de execução. Estuda-se uma versão do pseudocódigo do algoritmo, determina-se quais seriam as situações de pior, melhor e médio casos para esse algoritmo e técnicas matemáticas são utilizadas para se determinar o tempo de execução assintótico do algoritmo.

A complexidade de pior caso de um algoritmo é dada pela situação em que o algoritmo terá o maior número de iterações. A complexidade de melhor caso de um algoritmo é dada pela situação em que o algoritmo terá o menor número de iterações. A complexidade do caso médio do algoritmo envolve alguma probabilidade no número de iterações do algoritmo.

A seguir, serão mostradas as análises de complexidade de dois dos mais simples algoritmos de ordenação: ordenação por seleção e ordenação por bolha. Isso porque ordenação é considerada o problema algorítmico mais fundamental. Nas análises a seguir, cada linha k tem tempo de execução expresso pela constante c_k.

3.9.1 Ordenação por seleção

O algoritmo de ordenação por seleção a seguir, no passo *i*, encontra o menor entre os itens *i* e *n* de um *array* com *n* itens e coloca-o na posição *i*.

Algoritmo 3.6: OrdenaçãoSeleção.

Entrada: *array a*[ ] com *n* elementos;
Saída: *array a*[ ] ordenado de forma crescente;

```
1  início
2      para ( i ← 1 até n-1 ) faça
3          min ← i;
4          para ( j ← i + 1 até n ) faça
5              se ( a[j] < a[min] ) então
6                  min ← j;
7              fim
8          fim
9          temp ← a[min];
10         a[min] ← a[i];
11         a[i] ← temp;
12     fim
13 fim.
```

Para analisar um algoritmo, deve-se conhecê-lo a fundo e verificar em quais situações o algoritmo terá: mais iterações, o pior caso de execução; menos iterações, o melhor caso de execução; o caso médio de iterações. No algoritmo ordenação por seleção, o pior, o melhor e o caso médio são os mesmos porque, necessariamente, o algoritmo encontra o menor item a cada iteração *i*. Note que a troca nas linhas 9 a 11 é realizada mesmo que *min* seja igual à *i*. Incluir uma condição para essa troca não afeta o tempo de execução assintótico do algoritmo. A linha 2 do algoritmo é executada *n* vezes. Note que a variável *i* vai até *n* para, então, ser maior que *n*-1 e a condição do laço de repetição não ser satisfeita. As linhas 3, 9, 10 e 11 são executadas *n*-1 vezes cada. Como descrito, em geral, uma linha interna a um laço de repetição é executada uma vez menos que a linha da condição do laço de repetição.

Na primeira execução do laço para interno, na linha 4 do algoritmo, a variável j varia de 2 a n+1. Com isso, a linha 4 é executada n vezes, para encontrar o menor elemento. Na segunda execução do laço para interno, na linha 4 do algoritmo, a variável j varia de 3 a n+1. Com isso, a linha 4 do algoritmo é executada n-1 vezes, para encontrar o segundo menor elemento. Na terceira execução do laço para interno, na linha 4 do algoritmo, a variável j varia de 4 a n+1. Com isso, a linha 4 do algoritmo é executada n-2 vezes para encontrar o terceiro menor elemento, e assim por diante. Isso é realizado até que a variável i contenha n-1. Nesse passo, a linha 4 é executada duas vezes. Estas execuções podem ser expressas pelo somatório

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1. \tag{3.13}$$

A linha 5 é executada uma vez a menos que a linha 4, a cada iteração do laço de repetição interno. No primeiro passo, a linha 5 é executada n-1 vezes. No segundo passo, a linha 5 é executada n-2 vezes. No terceiro passo, a linha 5 é executada n-3 vezes e assim por diante, até que é executada uma só vez, quando i conterá n-1. Estas execuções podem ser expressas pelo somatório

$$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}. \tag{3.14}$$

Como descrito, na prática, a linha 7 "inexiste" em código de máquina. Esta linha é um rótulo para salto da linha 5 se a condição da linha 5 não for satisfeita. Esse é um rótulo para a instrução da linha seguinte, a linha 8. A linha 8, por sua vez, é um salto incondicional para a linha 4 e é executada uma vez a menos que o número de execuções da linha 3, a cada iteração. Com isso, o número de vezes que o algoritmo executa a linha 8 é o mesmo número de vezes que executa a linha 5, isto é, o número de execuções da linha 8 também é expresso pelo somatório (3.14). Similarmente, o número de vezes que a linha 12, um salto incondicional para a linha 2, é executada é igual ao número de vezes que cada uma das linhas 3, 9, 10 e 11 é executada, isto é, n-1 vezes.

Considere *T(n)* a função que expressa o tempo de execução do algoritmo OrdenaçãoSeleção e que o tempo de executar uma vez a linha i é c_i. Somando-se o número de execuções de cada linha, resulta em

$$T(n) = \overbrace{c_2 n}^{\text{linha 2}} + \overbrace{(c_3 + c_9 + c_{10} + c_{11} + c_{12})(n-1)}^{\text{linhas 3, 9, 10, 11 e 12}} + \overbrace{c_4(\frac{n^2+n}{2} - 1)}^{\text{linha 4}} + \overbrace{(c_5 + c_8)\frac{n^2-n}{2}}^{\text{linhas 5 e 8}}. \tag{3.15}$$

Resta analisar o número de vezes que a linha 6 é executada. A análise é dada a seguir.

1. Pior caso. Deve-se analisar o pior caso deste algoritmo quando a linha 6 é executada em todas as vezes que a linha 5 é executada. Já que o algoritmo ordena de forma crescente, o pior caso deste algoritmo se dá quando a entrada está quase ordenada. Isto é, o pior caso deste algoritmo ocorre quando os elementos estão ordenados de forma crescente, a menos do menor elemento, que se encontra na última posição. Verificar por que o pior caso ocorre com os dados nesse formato fica como exercício para o leitor. Com os dados nesse formato, o número de vezes que a linha 6 é executada é o mesmo número de vezes que a linha 5 é executada, pois a condição da linha 5 é satisfeita todas as vezes. Com isso, o tempo de execução é

$$T(n) = \overbrace{c_2 n}^{\text{linha 2}} + \overbrace{(c_3 + c_9 + c_{10} + c_{11} + c_{12})(n-1)}^{\text{linhas 3, 9, 10, 11 e 12}} + \overbrace{c_4(\frac{n^2+n}{2} - 1)}^{\text{linha 4}} + \overbrace{(c_5 + c_6 + c_8)\frac{n^2-n}{2}}^{\text{linhas 5, 6 e 8}}. \tag{3.16}$$

   Essa função fornece uma medida exata do tempo de execução no pior caso para o algoritmo. Essa equação pode ser expressa como

$$T(n) = an^2 + bn + c, \tag{3.17}$$

em que $a = \frac{c_4+d}{2}$, $b = c_2 + e + \frac{c_4-d}{2}$, $c = -e - c_4$, $d = c_5 + c_6 + c_8$ e $e = c_3 + c_9 + c_{10} + c_{11} + c_{12}$. Aplicando-se notação assintótica, obtém-se

$$T(n) = O(n^2) + O(n) + O(1) = O(n^2). \tag{3.18}$$

2. Melhor caso. O melhor caso deste algoritmo é quando a entrada está em ordem crescente. Com isso, a linha 6 não é executada, pois a condição da linha 5 nunca é satisfeita. Dessa forma, o tempo de execução exato deste algoritmo no melhor caso é dado pela equação (3.16) e pode ser reescrita como

$$T(n) = an^2 + bn + c, \tag{3.19}$$

em que $a = \frac{c_4+d}{2}$, $b = c_2 + e + \frac{c_4-d}{2}$, $c = -e - c_4$, $d = c_5 + c_8$ e $e = c_3 + c_9 + c_{10} + c_{11} + c_{12}$.

Essa função fornece uma medida exata do tempo de execução no melhor caso para o algoritmo. Aplicando-se notação assintótica, obtém-se

$$T(n) = \Omega(n^2) + \Omega(n) + \Omega(1) = \Omega(n^2). \tag{3.20}$$

3. Caso médio. O caso médio deste algoritmo ocorre quando na metade das vezes a condição da linha 5 é satisfeita. Com isso, a linha 6 é executada a metade das vezes que a linha 5 é executada. Adicionando-se esse termo à equação (3.16), obtém-se

$$T(n) = \overbrace{c_2 n}^{\text{linha 2}} + \overbrace{(c_3 + c_9 + c_{10} + c_{11} + c_{12})(n-1)}^{\text{linhas 3, 9, 10, 11 e 12}} + \overbrace{c_4(\frac{n^2+n}{2} - 1)}^{\text{linha 4}} + \overbrace{(c_5 + c_8)(\frac{n^2-n}{2})}^{\text{linhas 5 e 8}} + \overbrace{c_6 \frac{n^2-n}{4}}^{\text{linha 6}}. \tag{3.21}$$

A função do lado direito da equação (3.21) fornece uma medida exata do tempo de execução no caso médio para o algoritmo e pode ser reescrita como

$$T(n) = an^2 + bn + c, \tag{3.22}$$

em que $a = \frac{c_4+d}{2}$, $b = c_2 + e + \frac{c_4-d}{2}$, $c = -e - c_4$, $d = c_5 + c_8 + \frac{c_6}{2}$ e $e = c_3 + c_9 + c_{10} + c_{11} + c_{12}$.

Aplicando-se notação assintótica, obtém-se o mesmo termo do lado direito de da equação (3.18). Note que é mais relevante expressar o caso médio pela notação O do que pela notação Ω.

Como este algoritmo apresenta a mesma função para os limites superior e inferior de crescimento assintótico, o tempo de execução assintótico pode ser expresso por $\Theta(n^2)$. Ainda, o algoritmo realiza $\Theta(n)$ trocas e o custo de ocupação de memória é linear no tamanho da entrada.

3.9.2 Ordenação por bolha

O algoritmo de ordenação por bolha, a seguir, ordena a sequência a ao trocar cada item na posição i com o item da posição i+1 se $a[i] > a[i+1]$.

Considere a análise do tempo de execução desse algoritmo. A linha 2 do algoritmo é executada uma vez. A linha 3 do algoritmo é executada n vezes. A linha 4 do algoritmo é executada n-1 vezes.

A linha 16 é executa n-1 vezes. As linhas 12, 14 e 15 não são consideradas na análise. O número de vezes que as demais linhas são executadas depende do caso e isso é descrito a seguir.

Algoritmo 3.7: OrdenaçãoBolha.

```
   Entrada: array a[ ] com n elementos;
   Saída: array a[ ] ordenado de forma crescente;
 1 início
 2     alterado ← verdadeiro;
 3     para ( i ← 1 até n-1 ) faça
 4         se ( alterado = verdadeiro ) então
 5             alterado ← falso;
 6             para ( j ← 1 até n-i ) faça
 7                 se ( a[j] > a[j+1] ) então
 8                     temp ← a[j+1];
 9                     a[j+1] ← a[j];
10                     a[j] ← temp;
11                     alterado ← verdadeiro;
12                 fim
13             fim
14             // se (alterado = falso) então retorna;
15         fim
16     fim
17 fim.
```

1. Pior caso. O pior caso deste algoritmo se dá quando a entrada está em ordem decrescente, já que o algoritmo ordena de forma crescente. Com isso, o número de vezes que a linha 5 é executada é o mesmo número de vezes que a linha 4 é executada, pois a condição da linha 4 é satisfeita todas a vezes: em todos os passos haverá troca de itens e a variável *alterado* será estabelecida como *verdadeiro*. Com isso, a linha 5 é executada n-1 vezes.

   No primeiro passo, a linha 6 é executada n vezes. No segundo passo, a linha 6 é executada n-1 vezes. No terceiro passo, a linha 6 é executada n-2 vezes, e assim por diante, até que $i = n$-1. Neste passo, a linha 6 é executada duas vezes. Isso resulta em

$$n+(n-1)+(n-2)+\cdots+2=\sum_{i=2}^{n} j=\frac{n(n+1)}{2}-1=\frac{n^2}{2}+\frac{n}{2}-1. \quad (3.23)$$

No primeiro passo, a linha 7 é executada n-1 vezes. No segundo passo, a linha 7 é executada n-2 vezes. No terceiro passo, a linha 7 é executada n-3 vezes, e assim por diante, até que $i = n$-1. Neste passo, a linha 7 é executada uma vez. Isso resulta em

$$(n-1)+(n-2)+\cdots+1=\sum_{i=1}^{n-1} j=\frac{n^2}{2}-\frac{n}{2}. \quad (3.24)$$

As linhas 8 a 11, e 13 são executadas, cada uma, o mesmo número de vezes que a linha 7. Com isso, considerando-se que o custo da linha i é c_i, obtém-se

$$T(n)=\overbrace{c_2}^{\textit{linha } 2}+\overbrace{c_3 n}^{\textit{linhas } 3}+\overbrace{(c_4+c_5+c_{16})(n-1)}^{\textit{linhas } 4,5,16}+\overbrace{c_6(\frac{n^2}{2}+\frac{n}{2}-1)}^{\textit{linha } 6}+$$
$$\overbrace{(c_7+c_8+c_9+c_{10}+c_{11}+c_{13})(\frac{n^2}{2}-\frac{n}{2})}^{\textit{linhas } 7,8,9,10,11,13}.$$

Essa função fornece uma medida exata do tempo de execução no pior caso para o algoritmo. Essa função pode ser expressa como $T(n) = an^2 + bn + c$, em que $a=\frac{c_6+c_7+c_8+c_9+c_{10}+c_{11}+c_{13}}{2}$, $b = c_3+\frac{c_6}{2}-a+f$, $f = c_4+c_5+c_{16}$ e $c = c_2-c_6-f$. Aplicando-se notação assintótica, obtém-se $T(n) = O(n^2)+O(n)+O(1) = O(n^2)$.

2. Melhor caso. O melhor caso deste algoritmo ocorre quando a entrada está em ordem crescente, já que o algoritmo ordena nesta ordem. Com isso, a condição da linha 7 nunca é satisfeita. Por causa disso, a linha 5 é executada uma só vez. A linha 6 é executada n vezes: também só é executada no primeiro passo, quando i = 1. Com isso, as linhas 7 e 13 são executadas n-1 vezes. Como a condição da linha 7 nunca é satisfeita, as linhas 8 a 11 não são executadas. Com isso, obtém-se

$$T(n) = \overbrace{c_2 + c_5}^{\text{linhas } 2,5} + \overbrace{(c_3 + c_6)n}^{\text{linhas } 3,6} + \overbrace{(c_4 + c_7 + c_{13} + c_{16})(n-1)}^{\text{linhas } 4,7,13,16} \tag{3.25}$$

Essa função fornece uma medida exata do tempo de execução no melhor caso para o algoritmo. Essa função pode ser expressa como $T(n) = an + b$, em que $a = c_3 + c_6 + c$, $c = c_4 + c_7 + c_{13} + c_{16}$ e $b = c_2 + c_5 - c$. Aplicando-se notação assintótica, obtém-se $T(n) = \Omega(n) + \Omega(1) = \Omega(n)$. Note que incluir (ou descomentar) a linha 14 do algoritmo não melhora sua complexidade assintótica.

3. Caso médio. O caso médio deste algoritmo é quando na metade das vezes a condição da linha 7 é satisfeita. Com isso, as linhas 8 a 11 são executadas a metade das vezes que no pior caso. Com isso, obtém-se

$$T(n) = \overbrace{c_2}^{\text{linha } 2} + \overbrace{c_3 n}^{\text{linhas } 3} + \overbrace{(c_4 + c_5 + c_{16})(n-1)}^{\text{linhas } 4,5,16} + \overbrace{c_6(\frac{n^2}{2} + \frac{n}{2} - 1)}^{\text{linha } 6} + \overbrace{(c_8 + c_9 + c_{10} + c_{11})(\frac{n^2}{4} - \frac{n}{4})}^{\text{linhas } 8,9,10,11} + \overbrace{(c_7 + c_{13})(\frac{n^2}{2} - \frac{n}{2})}^{\text{linhas } 7,13}.$$

Essa função fornece uma medida exata do tempo de execução no pior caso para o algoritmo. Essa função pode ser expressa como $T(n) = an^2 + bn + c$, em que $a = \frac{c_6 + c_7 + c_{13}}{2} + \frac{c_8 + c_9 + c_{10} + c_{11}}{4}$, $b = c_3 + \frac{c_6 - c_7 - c_{13}}{2} + f$, $f = c_4 + c_5 + c_{16}$ e $c = c_2 - c_6 - f$. Aplicando-se notação assintótica, obtém-se $T(n) = O(n^2) + O(n) + O(1) = O(n^2)$.

Note que ambos, pior e caso médio, são representados como $O(n^2)$, enquanto o melhor caso é representado por $\Omega(n)$. Como o melhor e o pior casos não são expressos pela mesma função, *não* se pode representar o tempo de execução com a notação assintótica no caso estrito. O número de trocas do algoritmo é proporcional ao custo assintótico do tempo de execução e o custo de ocupação de memória é linear no tamanho da entrada.

3.10 Análise amortizada

Este texto faz uma muito breve introdução à análise de complexidade. Devem-se estudar as referências para abordagens aprofundadas. Na comparação de alguns algoritmos no decorrer destas notas, descreve-se a análise *amortizada* do algoritmo.

A seguir descreve-se uma noção dessa técnica. Novamente, deve-se seguir as referências para estudos aprofundados.

Na técnica anterior de análise de complexidade de algoritmos, determina-se o custo da sequência de passos ao se determinar o custo de cada passo. A análise amortizada é uma técnica para analisar algoritmos que realizam uma sequência similar de passos. O termo *amortizado* faz alusão à contabilidade, no sentido de que passos não caros compensam os passos caros.

A análise amortizada pode ser utilizada para fornecer um limite do custo de uma sequência de passos inteira. Mais especificamente, a análise amortizada pode ser usada para mostrar que o custo médio de uma sequência de operações é pequena em relação a uma determinada operação na sequência, mesmo que esta seja muito cara. Em uma sequência de passos similares, pode não ocorrer que todos os passos executem no pior tempo. Alguns passos podem ser caros, mas outros podem não ser. Isso significa que, na análise amortizada, é feita a *média* de todas as operações realizadas para se determinar o tempo exigido para executar uma sequência de operações.

A análise amortizada difere da análise do caso médio. Não há probabilidade envolvida na análise amortizada e ela garante o desempenho médio de cada operação no pior caso. Apesar de não haver qualquer probabilidade envolvida, uma análise amortizada fornece o tempo de execução *esperado* de uma sequência.

Uma das técnicas mais comuns de análise amortizada é a *agregada*. Na análise agregada, determina-se um limite superior $T(n)$ para o custo total de uma sequência de n operações. No pior caso, o custo médio ou custo amortizado por operação é $\frac{T(n)}{n}$. Considera-se o custo médio como o custo amortizado para cada operação. Isso ocorre mesmo quando há vários tipos de operações na sequência. Resulta disso que todas as operações têm o mesmo custo amortizado. Em duas outras técnicas comuns de análise

amortizada, o método da contabilidade e o método potencial, podem-se atribuir custos amortizados diferentes para tipos diferentes de operações.

Cormen et al. (2009) dedicaram o capítulo 17 de sua obra à análise amortizada. Em especial, a obra de Cormen et al. (2009) é uma referência excelente para se aprofundar em muitos outros fundamentos da Ciência da Computação. Tarjan (1983, 1985) também são excelentes para se aprofundar em análise amortizada.

3.11 Considerações finais

Em especial, algoritmos com tempo de execução polinomial terão custo de armazenamento também polinomial. Um algoritmo com tempo de execução polinomial não pode acessar todos os elementos de uma estrutura de dados com complexidade exponencial. Por outro lado, um algoritmo pode ter custo de armazenamento polinomial, mas ter tempo de execução exponencial. Isso ocorre porque o algoritmo pode realizar diversas operações em determinadas tarefas ou pode acessar a mesma posição de memória diversas vezes.

A técnica para análise de complexidade apresentada é eficiente, mas, como descrevem Goodrich e Tamassia (2004, p. 55), apresentam ao menos três limitações:

- a análise assintótica nem sempre esclarece os fatores constantes implícitos na notação assintótica. Considere que o algoritmo a_1 é assintoticamente menor que o algoritmo a_2 para o problema A. Considere que: a constante c_g é associada a a_1; a constante c_p é associada a a_2; $c_p << c_g$. A notação assintótica fornece poucas informações sobre se se deve escolher o algoritmo assintoticamente lento a_2, mas que tem associado a ele uma constante pequena c_p, em vez de se escolher o algoritmo rápido a_1, mas que tem associado a ele uma constante grande c_g;
- geralmente, analisa-se o pior caso e esse pode não ser representativo para um dado problema;
- quando o algoritmo é complicado, pode-se também ser muito complicado limitar exatamente seu tempo de execução.

Nesses casos, simulações devem ser empregadas para se comparar exatamente o desempenho dos dois algoritmos para o problema A. Veja a obra de Goodrich e Tamassia (2004, p. 55-57) para detalhes sobre técnicas e princípios para se realizar simulações experimentais de algoritmos.

3.12 Exercícios

1. A notação ..... fornece um limite ..... para a função que expressa o tempo de execução ..... de um algoritmo. Para uma função $g(n)$ dada, denota-se por .....($g(n)$) o conjunto de funções .....($g(n)$) = $\{f(n)$: existem constantes positivas c e n_0, tal que $0 < cg(n) \leq f(n)$ $(\forall n \geq n_0)\}$.

2. A notação ..... fornece um limite ....., isto é, ... e ....., para a função que expressa o tempo de execução ..... de um algoritmo. Para uma função $g(n)$ dada, denota-se por .....($g(n)$) o conjunto de funções .....($g(n)$) = $\{f(n)$: existem constantes positivas c_1, c_2 e n_0, tal que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ $(\forall n \geq n_0)\}$.

3. Considere as oito afirmações abaixo e escolha a alternativa correta, sendo que max($f(n)$,$g(n)$) retorna a maior das duas funções $f(n)$ ou $g(n)$.

   I. $\log_2 n = O(\lg n)$;

   II. $f(n) = O(f(n))$;

   III. $O(f(n)) + O(f(n)) = O(f(n))$;

   IV. $O(O(f(n))) = O(f(n))$;

   V. $O(f(n)) + O(g(n)) = O(max(f(n),g(n)))$;

   VI. $O(f(n))O(g(n)) = O(f(n)g(n))$;

   VII. $f(n)O(g(n)) = O(f(n)g(n))$;

   VIII. $f(n) = \Omega(g(n)) \leftrightarrow g(n) = O(f(n))$.

   a) I. Verdadeira; II. Verdadeira; III. Verdadeira; IV. Falsa; V. Verdadeira; VI. Falsa; VII. Falsa; VIII. Verdadeira.

b) I. Verdadeira; II. Falsa; III. Falsa; IV. Verdadeira; V. Falsa; VI. Verdadeira; VII. Verdadeira; VIII. Falsa.

c) I. Falsa; II. Verdadeira; III. Verdadeira; IV. Falsa; V. Falsa; VI. Falsa; VII. Falsa; VIII. Falsa;

d) Todas são falsas.

e) Todas são verdadeiras.

4. Pela definição da notação assintótica O, forneça constantes c e n_0 válidas para a expressão $4^n = O(8^n)$.

5. Se $f(n) = \Omega(g(n))$, então, $g(n)$ = ...($f(n)$).

6. Sejam duas funções $f(n)$ e $g(n)$. Obtém-se $f(n) =$ .....$(g(n)) \leftrightarrow f(n) =$ .....$(g(n)) \wedge f(n) = \Omega(g(n))$.

3.13 Notas bibliográficas

Os modelos de computação RAM e RASP foram definidos para análise de algoritmos por Shepherdson e Sturgis (1963). Ao estudar Máquina de Turing de várias fitas, Hartmanis e Hartmanis e Stearns (1965) deram atenção especial ao problema de como seu tempo de execução é comparado com o modelo de uma fita.

A análise de complexidade de espaço foi primeiramente estudada por Hartmanis, LEWIS II e Stearns (1965) e LEWIS II, Stearns e Hartmanis (1965). Considera-se que o trabalho desses autores iniciou a complexidade computacional. Todavia, Knuth (1968) foi o primeiro a divulgar o estudo sistemático do tempo de execução de programas. Mais especificamente, Knuth (1968) divulgou a análise assintótica de algoritmos como uma maneira de comparar o desempenho relativo de algoritmos e a notação O. Aho, Hopcroft e Ullman (1974) também contribuíram para popularizar a análise assintótica de algoritmos.

Segundo Knuth (1976, p. 19), a notação O foi, provavelmente, introduzida por Bachmann (1894) para análise assintótica e foi popularizada por Landau (1909) e outros. Veja Graham, Knuth e Patashnik (1995, p. 323), para detalhes. Ainda segundo Knuth

(1976, p. 19), a notação Ω foi introduzida por Hardy e Littlewood (1914) com um significado um pouco diferente do fornecido por Knuth (1976, p. 19). A definição atual de Ω, e apresentada neste texto, foi introduzida por Knuth (1976, p. 19).

A notação Θ foi introduzida por Knuth (1976, p. 19) e, segundo o autor, sugerida independentemente por R. E. Tarjan e M. Paterson. A definição dessas notações em termos de conjuntos foi também foi introduzida nessa carta e, segundo o autor, sugerida por R. Rivest.

A análise agregada foi utilizada por Aho, Hopcroft e Ullman (1974). Cormen et al. (2009, p. 478) atribuem o termo *amortizado* a D. D. Sleator e R. E. Tarjan.

A abordagem deste capítulo baseia-se nos capítulos 2 e 3 de Cormen et al. (2009). O estudo desse livro é fortemente recomendado a todo estudante e profissional da Ciência da Computação. Um dos motivos do grande sucesso desse livro pode ser o equilíbrio entre didática e rigor matemático na apresentação dos conteúdos. A obra de Cormen et al. (2009) aborda os principais conceitos básicos de Ciência da Computação de forma razoavelmente aprofundada. Um grande número de profissionais tem se formado nos últimos 20 anos (a primeira edição é de 1991), em todo o mundo, com os ensinamentos deste livro. Pode ser que estas sejam razões para que o livro de Cormen et al. (2009) seja uma das obras mais referenciadas na Ciência da Computação. Para o estudante que não deseja misturar o inglês técnico desse livro com temas, por vezes, não triviais, pode estudar com a tradução para o português da segunda edição: Cormen et al. (2002).

15 NP-Completude

15.1 Introdução

Os *problemas* são motivações na Ciência. Geralmente, em Ciência, considera-se *problema* uma questão geral e relevante a ser respondida, geralmente processando vários parâmetros (ou *variáveis livres*), cujos valores não são especificados na definição do problema. Em oposição a uma variável livre, uma *variável ligada* ocorre em duas situações: é uma variável que identifica o que o quantificador quantifica ou a variável está dentro do escopo de um quantificador universal ($\forall$) ou existencial ($\exists$). Um problema é descrito como uma descrição geral de todos seus parâmetros e uma expressão de quais propriedades a resposta, ou *solução*, exige.

15.1.1 Problemas computacionais

A teoria sobre NP-Completude trata de problemas computacionais, que podem ser divididos em três categorias:

i) os problemas que são resolvidos por algoritmos polinomias $O(n^c)$ para uma constante $c \in \mathbb{N}$, em função do tamanho da entrada;

ii) os problemas que, aparentemente, não possuem limite inferior com complexidade intrinsecamente polinomial, ou seja, não se conhece algoritmo que resolva o problema em tempo polinomial e ainda não foi apresentada uma demonstração de que não se pode ter um algoritmo com limite inferior não polinomial de crescimento assintótico da função que representa o tempo de execução do algoritmo;

iii) os problemas que são resolvidos por algoritmos, demonstradamente, com limite inferior não polinomial de crescimento assintótico da função que representa seu tempo de execução.

Os problemas da terceira categoria exigem, intrinsecamente, para a sua solução, um número de operações proporcional a uma função não polinomial (por exemplo, exponencial) no tamanho da instância do problema. Isso porque geram um número não polinomial (por exemplo, exponencial) de subproblemas. Exemplos de problemas que exigem, para a sua solução, um número exponencial de operações incluem a geração de todas as árvores geradoras de um grafo, de todos os circuitos de um grafo e de todas as cliques (subgrafos completos) de um grafo (TERADA, 1991, p. 231). Esses problemas são considerados *intratáveis*. Por outro lado, diz-se que os problemas da primeira categoria são *bem resolvidos*. É, principalmente, sobre a segunda categoria de problemas que trata a teoria sobre NP-Completude.

Há problemas para os quais não se conhece solução em tempo polinomial e a solução conhecida também não é expressa por uma função exponencial. No entanto, em geral, nessa teoria, quando se refere a um algoritmo com tempo de execução expresso por uma função exponencial, por clareza, a intenção é referir-se que a função que expressa o tempo de execução do algoritmo não é polinomial.

15.1.2 Algumas palavras sobre problemas tratáveis e intratáveis

Autores deste assunto costumam chamar um problema de "intratável" se é tão "difícil" que nenhum algoritmo em tempo polinomial pode resolvê-lo. Veja Aho, Hopcroft e Ullman (1974) Garey e Johnson (1979, p. 8) e Cormen et al. (2009, p. 1048), para detalhes. A noção de *difícil* nesta área é relacionada com a complexidade de tempo de um algoritmo para resolver o problema.

Um problema "intratável" tem solução algorítmica, mas não por um algoritmo que tenha tempo de execução polinomial em função do tamanho da entrada. Muitos algoritmos com tempo de execução expressos por funções exponenciais são meramente variações de busca exaustiva. Já os algoritmos com tempo de execução polinomial geralmente são obtidos por algum ganho que, por sua vez, é obtido por meio da descoberta

de características especiais e/ou de um entendimento profundo (ou uma *boa ideia*) da estrutura do problema (GAREY; JOHNSON, 1979, p. 8). Nesse sentido, autores, como Manber (1989, p. 341) e Rosen (2007, p. 197,836), referem-se a problemas *tratáveis* como sendo aqueles resolvidos por algoritmos com tempo de execução polinomial.

15.1.3 Esquema de codificação

Essa noção de intratabilidade é independente do *esquema de codificação*. Por exemplo, considere um problema que tenha como entrada um grafo e a representação computacional desse grafo. O tamanho da entrada é diferente quando um grafo é representado por uma matriz de adjacências, uma matriz de incidências ou por listas de adjacências. Considere o grafo G(V,A) da Figura 15.1, já mostrado no capítulo 8. Na Tabela 15.1, mostra-se uma representação do grafo da Figura 15.1 por uma matriz adjacências. O grafo G(V,A) também pode ser representado por uma lista de adjacências: $1 \rightarrow 2 \rightarrow 3$; $2 \rightarrow$; $3 \rightarrow 1$ e ainda pode ser representado pela matriz de incidências mostrada na Tabela 15.2, em que os vértices estão nas linhas e as arestas, nas colunas. Cada representação forneceu um espaço de armazenamento diferente para o mesmo grafo, mas o tamanho das diferenças é, no máximo, polinomial. Então, qualquer algoritmo que tenha tempo de execução polinomial sob um desses esquemas de codificação também terá tempo de execução polinomial sob todos os outros esquemas de codificação.

A codificação da instância de um problema deve ser concisa e não deve ser adicionada de informações ou símbolos desnecessários. Os números que ocorrem na instância devem ser representados em binário, decimal, octal ou em outra base fixa que não seja 1 (GAREY; JOHNSON, 1979, p. 10). Por exemplo, um inteiro *i* não pode ser representado com *i* dígitos 1.

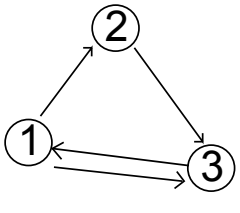

Figura 15.1: Grafo G(V,A) direcionado e não ponderado.

Tabela 15.1: Adjacências.

vértices de V	1	2	3
1	0	1	1
2	0	0	1
3	1	0	0

Tabela 15.2: Matriz de incidências de G(V,A).

vértices de V	(1,2)	(1,3)	(2,3)	(3,1)
1	1	1	0	-1
2	-1	0	1	0
3	0	-1	-1	1

15.1.4 Modelos de computação

Sugere-se reler a descrição sobre modelos de computação da seção 3.3. Para o texto que segue, considere modelos de computação razoáveis, como a máquina de Turing, a máquina de acesso aleatório - RAM, descrita por von Neumann (reimpresso em 1982) e a máquina de programa armazenado de acesso aleatório, RASP, que pode simular qualquer outra RAM.

Esses modelos de computação são equivalentes em relação ao tempo de execução polinomial de um algoritmo. Há um limite polinomial na quantidade de trabalho que pode ser feita em uma unidade simples de tempo. Um modelo de computação com a capacidade de realizar arbitrariamente muitas operações em paralelo não é considerado *razoável* e, de fato, nenhum computador tem essa capacidade (GAREY; JOHNSON, 1979, p. 11).

15.1.5 Complexidade e intratabilidade

O termo *complexidade* é relacionado, em outros contextos, à noção de *complicado*. Entretanto, complexidade computacional refere-se ao esforço computacional que o algoritmo requer para solucionar o problema em função do tamanho da entrada. A forma mais comum é a análise do tempo de execução do algoritmo, ou sua complexidade de

tempo. Isso porque a ocupação de espaço também é estudada. Mas, como, ultimamente, memória tornou-se relativamente de menor custo em relação a outros componentes de um sistema de computação, o esforço de tempo tem sido mais estudado em determinadas áreas do que a ocupação de memória. Em geral, algoritmos não são *complicados*, pois, se há solução para o problema, alguém pode encontrar uma solução algorítmica para ele.

Geralmente, um problema não é considerado *bem resolvido* se não há um algoritmo polinomial que o resolva, embora haja excessões. Exemplos clássicos são os problemas resolvidos pelo método simplex (DANTZIG, 1951) para programação linear, o qual apresenta complexidade de tempo exponencial - veja Klee e Minty (1972) e Zadeh (1973). Todavia, obtém-se, com esse método, tempo de execução rápido na prática. Com isso, os problemas resolvidos por esse método são considerados bem resolvidos. Ainda assim, mesmo para problemas resolvidos pelo método simplex, pesquisadores continuam a procurar algoritmos polinomiais para resolver tais problemas (GAREY; JOHNSON, 1979, p. 8).

Como o exemplo do simplex é incomum, na teoria de NP-Completude, os *algoritmos* com tempo de execução expressos por algum polinômio no tamanho da entrada são chamados de *eficientes*. Essa associação ocorre porque polinômios formam uma classe de funções

$$C_P = \{p(n) : (\forall n \in \mathbb{R}^*)(\forall g \in \mathbb{N})(\forall a_i \in \mathbb{R},\, i = 0, 1, \cdots, g-1)(a_g \in \mathbb{R}^*)\ p(n) = \sum_{i=0}^{g} (a_i n^i)\}$$

de crescimento moderado em relação ao tamanho da entrada n e são fechados sob operações de multiplicação por uma constante $cp(n) \in C_P$, adição $p(n) + q(n) \in C_P$, multiplicação $p(n) \cdot q(n) \in C_P$ e composição de funções $p(q(n)) \in C_P$, em que $p(n)$ e $q(n) \in C_P$. Um conjunto *fechado* sob determinada operação é a propriedade em que o resultado da operação sempre existe e pertence ao próprio conjunto. Essas propriedades garantem o fechamento da classe de algoritmos eficientes sob composições naturais de algoritmos. Isso significa que se podem combinar ou compor algoritmos em tempo polinomial, para se obter novos algoritmos de tempo polinomial.

Pela tese de Cobham-Edmonds, se o modelo de compução é *razoável*, então, a classe de algoritmos em tempo polinomial é independente do modelo de computação específico. A tese de Cobham-Edmonds afirma que as complexidades de tempo de quaisquer dois modelos de computação *razoáveis e gerais* são polinomialmente relacionados: um problema tem complexidade de tempo n em algum modelo de computação *razoável e geral* se, e somente se, esse problema tem complexidade $O(n^c)$ em um modelo de máquina de Turing de uma fita, para alguma constante c (GOLDREICH, 2008, p. 33). A máquina de Turing padrão é descrita no capítulo 13, para formalização das classes básicas dessa teoria.

Como já mencionado, é chamado de *intratável* o *problema* que, de tão *difícil*, não seja possível propor um algoritmo polinomial para resolvê-lo. Note que *intratável* está associado ao problema resolvido por algoritmos em que o tempo de execução *explode* quando o tamanho da entrada aumenta ou a problemas indecidíveis. Problemas indecidíveis são comentados adiante.

Na Tabela 15.3, mostra-se uma das razões pelas quais algoritmos polinomiais são, geralmente, mais desejáveis que algoritmos exponenciais. Mais especificamente, mostram-se as diferenças de taxas de crescimento entre várias complexidades típicas, em que as funções expressam o tempo de execução e, para algumas, houve aproximação. Considera-se uma máquina que executa cada entrada em 1 nanosegundos (ns), isto é, ciclo de tempo para uma frequência de 1GHz, ou 1×10^9 *hertz* e anos com 365 dias. Dificilmente, um algoritmo eficiente (com complexidade de tempo polinomial) apresenta tempo de execução expresso por um polinômio da ordem de n^{10}. Seria algo como 10 laços de repetição aninhados e cada laço consideraria todos os n elementos da entrada. Mesmo assim, considere esse exemplo. Aparentemente, um polinônio como n^{10} pode ser não melhor que um algoritmo exponencial por força bruta que tenha complexidade da ordem de 2^n. No entanto, à medida que o tamanho da entrada fica grande, uma função exponencial apresenta crescimento mais rápido que uma função polinomial.

Garey e Johnson (1979, p. 11) apresentam duas causas de intratabilidade:

- o problema é tão difícil que uma quantidade de tempo exponencial é necessária para descobrir a solução;

Tabela 15.3: Comparação entre funções com complexidade de tempo polinomial e exponencial.

Complexidade	Tamanho da entrada n (ns onde não indicado)		
função	8	16	64
$\lg n$	3	4	6
n	8	16	64
$n \cdot \lg n$	24	64	384
n^2	64	256	4096
n^3	512	4096	262144
n^4	4096	65536	$1{,}6777216 \times 10^{-2}$ segundo
n^{10}	1,073741824 segundo	$\approx$18 minutos	36,558901085 dias
2^n	256	65536	584,942417355 anos
$n!$	40320	$\approx$6 horas	$4{,}02355823 \times 10^{69}$ milênios
n^n	$1{,}6777216 \times 10^{-2}$ segundo	$\approx$585 anos	$1{,}24942942 \times 10^{96}$ milênios

- a própria solução é tão extensa que não pode ser descrita por uma expressão com tamanho limitado por uma função polinomial do tamanho da entrada.

Considere o problema do caixeiro viajante: dado um conjunto de cidades (ou vértices) e as distâncias entre elas, deseja-se saber o passeio de custo mínimo por todas as cidades. A segunda causa de intrabilidade ocorre, por exemplo, na variação do problema do caixeiro viajante que inclui um número k como parâmetro que pergunta por todos os percursos com tamanho menor ou igual a k. Devem-se perceber, em sua definição, problemas como esse. Eles podem não ser realísticos porque pergunta-se por mais informação do que se poderia tratar. Dessa forma, a atenção, geralmente, é focada na primeira causa de intratabilidade: somente problemas que têm solução com tamanho limitado por uma função polinomial do tamanho da entrada são, geralmente, considerados (GAREY; JOHNSON, 1979, p. 11).

Os primeiros resultados sobre intratabilidade são os estudos clássicos de Turing (1936) sobre indecibilidade. Turing demonstrou que há problemas que, de tão difíceis, são indecidíveis, no sentido de que nenhum algoritmo pode resolvê-los. Turing provou que é impossível especificar um algoritmo que, dado um programa de computador arbitrário p e *qualquer* entrada arbitrária s para esse programa, possa decidir se p sempre parará quando receber a entrada s. Os problemas provavelmente intratáveis ou são inde-

cidíveis ou são *não-deterministicamente* intratáveis (GAREY; JOHNSON, 1979, p. 12). Entenda por determinismo a característica de um modelo de computador que, quando recebe entradas idênticas, sempre produz o mesmo resultado. Por outro lado, um modelo de computador não-determinístico tem a habilidade de realizar um número não limitado de sequências computacionais independentes em paralelo. Para uma dada entrada, o resultado pode ser qualquer uma das muitas possíveis soluções para o problema. Garey e Johnson (1979, p. 12, tradução nossa) fornecem a seguinte explicação sobre problemas provavelmente intratáveis:

> ... muitos dos problemas aparentemente intratáveis encontrados na prática são decidíveis e podem ser resolvidos em tempo (de execução) polinomial com a ajuda de um computador não-determinístico. Assim, nenhuma das técnicas de prova desenvolvidas até hoje é poderosa o bastante para verificar a aparente intratabilidade desses problemas.

15.1.6 Guia do capítulo

Antes de apresentar as fomalizações das classes básicas da NP-Completude, são necessários conceitos básicos de teoria da computação, que são: problemas de decisão, símbolos, *strings*, alfabetos, linguagens e associar as linguagens aos problemas de decisão. Em especial, a teoria sobre NP-Completude é aplicada a problemas de decisão. Para compreender adequadamente o conteúdo deste capítulo, é fortemente recomendada a leitura do capítulo 13, no qual são encontradas as definições de *string*, alfabeto e problema de decisão.

No texto a seguir, a máquina de Turing é utilizada para definir a classe P de todas as linguagens reconhecidas deterministicamente em tempo polinomial. Esse modelo é aumentado com a capacidade de se adivinhar um testemunho para o problema de decisão, sendo utilizado para definir a classe NP de todas as linguagens reconhecíveis não-deterministicamente em tempo polinomial. Em seguida, as classe NP-Completo e NP-Difícil são descritas. Finalmente, alguns exemplos de provas de NP-Completude são apresentadas. A abordagem seguinte é conforme o segundo capítulo de Garey e Johnson (1979).

15.2 Classe P

Em uma descrição informal, P é a classe de problemas que podem ser resolvidos em tempo polinomial. A natureza fundamental da distinção entre algoritmos polinomiais e exponenciais foi primeiramente discutida e introduzida por Cobham (1964). Essa classe também foi introduzida independentemente por Edmonds (1965). A definição formal da classe P, em termos de máquinas de Turing determinísticas (MTD), a seguir, é fornecida por Garey e Johnson (1979, p. 27).

P-*Time* = $\{L$: há uma MTD M com tempo polinomial em que $L = L_M\}$.

P abrevia a descrição da classe de linguagens que são reconhecidas por uma MTD M em tempo polinomial. A classe de *strings* que codificam instâncias com esquema de codificação s de um problema Π e a resposta é 1, é a partição de Σ^* em uma classe de *strings*: é a linguagem associada com Π e s ($L[\Pi, s]$). A MTD aceita $x \in L$, x é uma entrada que responde *sim* ao problema. P é a classe de linguagens que podem ser reconhecidas em tempo polinomial por uma MTD. A MTD reconhece $L \subset \{0,1\}^*$.

Cormen et al. (2009, p. 1059) fornecem a seguinte definição da classe P, em termos de algoritmos.

P = $\{L \subset \{0,1\}^*$: há um algoritmo que verifica se L é decidível em tempo polinomial$\}$.

De fato, P é também a classe de linguagens que podem ser reconhecidas em tempo polinomial. Perceba que as definições exprimem a mesma noção.

P = $\{L$: L é reconhecida por um algoritmo em tempo polinomial$\}$.

Um problema de decisão $\Pi \in$ P sob o esquema de codificação s, se $L[\Pi, s] \in P$; um problema de decisão $\Pi \in$ P, se há uma MTD em tempo polinomial que resolve Π sob o esquema de codificação s. Em geral, omite-se a especificação de um esquema de codificação razoável e simplesmente diz-se $\Pi \in$ P (GAREY; JOHNSON, 1979, p. 27).

15.3 Classe NP

Define-se a classe NP em termos de algoritmos não-determinísticos. Recomenda-se estudar o capítulo 13 deste livro antes de continuar neste texto. Para mostrar que um determinado problema pertence à classe NP, pode-se apresentar um algoritmo não-determinístico que execute em tempo polinomial para resolvê-lo. Então, existe um verificador para o problema que executa em tempo polinomial. Com isso, pode-se descrever NP como a classe de problemas de decisão em que a resposta é sim ou não, que podem ser *verificados* em tempo polinominal. Dessa forma, a classe NP pode ser definida, informalmente, como a classe de todos os problemas de decisão que, sob esquemas de codificação razoáveis, podem ser *resolvidos* por algoritmos não-determinísticos em tempo polinomial.

Considere, novamente, o problema do caixeiro viajante, que pergunta se, dados um conjunto de cidades (ou vértices de um grafo), as distâncias entre elas e um limite k, existe um passeio por todas as cidades com custo k ou menor. Não há um algoritmo com tempo de execução polinomial conhecido que resolva esse problema. Entretanto, suponha que alguém apresente uma instância específica que responda *sim* para esse problema. Sendo céticos, pode-se exigir que se *prove* que tal instância responde *sim* ao problema. Deve-se avaliar a verdade ou a falsidade meramente ao se verificar o custo da instância e compará-lo com k. Deve-se especificar esse procedimento de verificação como um algoritmo que tem complexidade de tempo polinomial no tamanho da instância (GAREY; JOHNSON, 1979, p. 27-28).

Um algoritmo que resolva um problema da classe NP em tempo polinomial não é conhecido nem foi provado que há um limite inferior não-polinomial para qualquer um dos problemas dessa classe. NP é um acrônimo para problemas polinomiais não-determinísticos (*problem solved by a nondeterministic polynomial-time algorithm*).

Como todo problema da classe P é verificado em tempo polinomial, então, $\mathrm{P} \subseteq \mathrm{NP}$, mas não se sabe se $\mathrm{P} \subset \mathrm{NP}$. A classe NP foi sugerida por Edmonds (1965), que conjecturou que $\mathrm{P} \neq \mathrm{NP}$.

15.3.1 Verificar polinomialmente se a resposta de um problema de decisão é *sim*

Nos problemas pertencentes à classe NP, há uma instância x pertencente ao problema Π que é verificável pelo algoritmo A em tempo polinomial. Para classificar um problema Π como pertencente à classe NP, mostra-se uma instância $x \in \Pi$ que é verificável por A em tempo polinomial.

Geralmente, é mais complicado resolver um problema na sua forma gereralizada que simplesmente verificar uma solução apresentada. Por exemplo, considere o problema da soma de elementos de um subconjunto. Suponha um conjunto de inteiros, por exemplo -8, -5, -4, -1, 3, 9 e deseja-se saber se a soma de alguns desses inteiros resulta em zero.

Para se responder a esse problema de decisão, é necessário verificar todos os subconjuntos possíveis para se determinar se a soma dos números de algum dos subconjuntos é zero. Um algoritmo para *resolver* esse problema de decisão terá crescimento exponencial do número de subconjuntos em relação ao número de inteiros da entrada. No entanto, ao ser fornecido um subconjunto específico t, chamado de *certificado* ou *testemunho*, pode-se facilmente *verificar* se a soma dos elementos de t é zero. Se a soma dos elementos de t é realmente zero, então, t é a *prova* ou testemunho do fato de que a resposta é *sim* para o problema de decisão.

No exemplo dado, a resposta ao problema de decisão é *sim*, pois -5, -4, 9 corresponde à soma (-5) + (-4) + (9) = 0. Dessa forma, o testemunho é $t = \{-5, -4, 9\}$. Perceba que o problema de decisão não foi *resolvido*. Somente *verificou-se* se uma determinada instância responde afirmativamente ao problema de decisão.

Um algoritmo A que verifica se a soma de t é zero é chamado de *verificador*. Note, pelo exemplo, que o testemunho t pode ser uma parte da instância x. Assim, um problema pertence à classe NP se, e somente se, existe um verificador para o problema que execute em tempo polinomial. No caso do problema da soma de elementos de um subconjunto, A necessita somente de tempo polinomial.

Pode-se definir um verificador para uma linguagem L como um algoritmo A de dois argumentos: um dos argumentos é uma *string* de entrada x e o outro argumento é uma

string binária t chamada de certificado. Dessa forma, $A(x,t) = e_S \in E$, em que E é o conjunto de estados de uma MTD e e_S é o estado de parada de aceitação. A linguagem verificada pelo algoritmo A é dada a seguir, ou seja, A aceita (x,t).

$L = \{x \in \Sigma^*\text{: } (\exists t \in \Sigma^*) | A(x,t) = e_S \in E\}$.

Um algoritmo A verifica uma linguagem L se, para qualquer *string* $x \in L$, há um certificado t de que A pode ser usado para provar que $x \in L$. Para qualquer *string* $x \notin L$, não deve haver um certificado que prove que $x \in L$ (CORMEN et al., 2009, p. 1063).

Mensura-se o tempo de um verificador apenas em termos do comprimento da entrada x. Ou seja, um *verificador em tempo polinomial* executa em tempo polinomial no tamanho de x. Claramente, uma linguagem L é polinomialmente verificável se tem um verificador em tempo polinomial. Garey e Johnson (1979, p. 28, tradução nossa) fornecem a seguinte explicação sobre *verificabilidade* em tempo polinomial e sua relação com a classe NP:

> É esta noção de "verificabilidade" em tempo polinomial que a classe NP pretende isolar. Note que verificabilidade em tempo polinomial não implica solucionalidade em tempo polinomial. Ao dizer que alguém pode verificar uma resposta "sim" de uma instância do caixeiro viajante em tempo polinomial, nós não contamos o tempo que alguém teria que utilizar na busca possivelmente muito exponencial de passeios para uma das formas desejadas. Nós meramente afirmamos que, dado qualquer passeio para uma instância I, nós podemos verificar, em tempo polinomial, se o passeio *prova* ou não que a resposta para I é *sim*.

15.3.2 Definição da classe NP

Seja uma MTND M composta pelo estágio de "adivinhação" M_1 e pelo estágio de verificação M_2. Considere uma constante $c \in \mathbb{N}$. Se existe testemunho t para a *string* de entrada x, então, t é fornecida por $M_1(x)$.

NP-*Time* = $\{L \subset \{0,1\}^* : (\forall x \in L)(\exists t \in \{0,1\}^*) \mid ((t = M_1(x) \wedge |t| = O|x|^c) \wedge M_2(x,t) = 1) \rightarrow L_M = L\}$.

Note que $L_M = \{x \in \Sigma* : M \textit{ aceita } x\}$. Ocorre, ainda, que $(\forall x \notin L)(\forall t \in \{0,1\}^*)(t = M_1(x)) \rightarrow (M_2(x,t) = 0)$. A definição formal anterior corresponde à definição informal: NP é a classe de linguagens reconhecidas em tempo polinomial por uma MTND. Um problema de decisão $\Pi \in$ NP sob o esquema de codificação s se a linguagem $L[\Pi, s] \in$ NP. $L \in$ NP se, e somente se, existe um certificado t com tamanho máximo polinomial em função da *string* de entrada x no qual há uma MTND M composta pelos estágios M_1 e M_2 que aceita a entrada x e o testemunho t. Ainda, se não há testemunho t para x, então, M rejeita a entrada x e o testemunho t: M_2 aceita x e t se $x \in L$ e rejeita x e t, se $x \notin L$. Se $x \notin L$, então, não há testemunho de que $x \in L$ e o estágio pode escrever um só símbolo $a \in \Sigma$ como o testemunho t e passar para o estágio de verificação que computa $\Delta(e_0, a) = (e_N, d, \leftarrow \text{ ou } \rightarrow)$ e d é algum símbolo de Σ. Dessa forma, *M verifica* a linguagem L em tempo polinomial: NP é a classe de linguagens verificadas em tempo polinomial.

Como em P, diz-se livremente que $\Pi \in$ NP sem fornecer um esquema de codificação específico, já que *algum* esquema de codificação para Π fornecerá uma linguagem que está em NP. Essa definição está de acordo com: NP é a classe de problemas de decisão "resolvidos" por algoritmos não-determinísticos em tempo polinomial. Apesar da definição das classes P e NP, em termos de máquinas de Turing, poderia ser utilizado qualquer um de vários modelos de computação (GAREY; JOHNSON, 1979, p. 32).

As sequências possíveis de movimentos que M pode realizar na entrada x podem ser estruturadas em uma árvore de decisões. Cada caminho da raiz a uma folha representa uma sequência de movimentos possíveis. Se σ é a sequência de movimentos mais curta que termina com M aceitando x, então, tão logo M realize $|\sigma|$ movimentos, M para e aceita a entrada x. O tempo para processar x é o tamanho de σ.

Geralmente, é conveniente pensar em M somente em adivinhar os movimentos na sequência σ e verificar que σ de fato termina em aceitar a entrada. No entanto, uma máquina determinística não pode, normalmente, adivinhar uma sequência de movimentos antes do processamento. Uma simulação determinística de M exigiria traçar a árvore de todas as sequências possíveis de movimentos de x, em alguma ordem, até encontrar uma sequência mais curta que termine em aceitar a entrada. Se nenhuma sequência de movimentos leva a aceitar a entrada, então, uma simulação determinística de M pode-

ria executar para sempre, a menos que haja algum limite no tamanho da sequência de aceitação mais curta estabelecida *a priori*.

É natural ter a suspeita de que MTNDs são capazes de realizar tarefas que não podem ser realizadas por qualquer máquina determinística com complexidades de tempo e espaço iguais. No entanto, MTNDs são equivalentes a MTDs, que são casos especiais de MTNDs. É possível simular MTNDs com MTDs.

É uma questão em aberto se há linguagens que são aceitas por uma MTND de tempo fixo ou complexidade de espaço, mas não por uma MTD com as mesmas complexidades. Ninguém ainda foi hábil em provar que há uma linguagem em NP que não está em P. Se P $\subset$ NP, então, a quantidade de problemas em NP-P é significativa e importante: todos os problemas em P podem ser resolvidos com algoritmos determinísticos em tempo polinomial, enquanto todos os problemas em NP-P são intratáveis (GAREY; JOHNSON, 1979, p. 34). Esta é uma forma de descrever o problema P$\stackrel{?}{=}$NP.

15.3.3 Classe coNP

O complemento de uma linguagem (conjunto ou classe) é a linguagem constituída de todas as cadeias que não constam nela. Se existe um algoritmo não-determinístico que *aceita* uma entrada $x \in L$ em tempo polinomial, então, $L \in$ NP. Se existe um algoritmo não-determinístico que *rejeita* uma entrada $x \in L$ em tempo polinomial, então, $L \in$ coNP. Diz-se que uma linguagem é coTuring-reconhecível se for o complemento de uma linguagem Turing-reconhecível. coNP é a classe de problemas de decisão que têm certificados que levam a se obter resposta *não*, também chamados de contraexemplos. Em particular, uma linguagem é decidível se, e somente se, ela é Turing-reconhecível e coTuring-reconhecível. Para detalhes sobre esse assunto, veja uma prova em Sipser (2007, p. 191), por exemplo.

Ocorre também que P $\subset$ coNP. Se P = NP, então, coNP = NP. Isso porque qualquer algoritmo em tempo polinomial que determina se $x \in L$ pode determinar se $x \notin L$. Se for definida a classe coP de maneira análoga à coNP, então, coP, necessariamente, é igual a P. Mas, isso não é verdade para algoritmos não-determinísticos. Um algoritmo não-determinístico que determina se $x \in L$ pode não ser usado para determinar se $x \notin L$

(HEDMAN, 2006). É uma questão em aberto se todos os problemas em NP também têm certificados com respostas *não* e, assim, estão em coNP. Muitos pesquisadores acreditam que P $\neq$ NP $\neq$ coNP. Outra forma de perceber essa questão é verificar se P = NP $\cap$ coNP.

15.4 Reduções polinomiais

As noções de redução polinomial e problemas relacionados são básicas para a compreensão das classes NP-Difícil e NP-Completo. A principal técnica usada para demonstrar que dois problemas são relacionados é a *redução* em tempo polinomial de um para outro. É esta a técnica, chamada de *redução de Karp*, apresentada nesta seção. Para reduções mais gerais, ou seja, reduções de Cook (ou de Turing ou de Levin), veja referências como Goldreich (2008, p. 59) e o capítulo 6 de Sipser (2007).

Uma *redução* converte um problema em outro, de forma que uma solução para o segundo problema possa ser combinada com o algoritmo da redução e, com isso, serem usados para resolver o primeiro problema. Uma redução pode ser realizada ao se construir uma transformação que mapeia qualquer instância do primeiro problema para uma instância equivalente do segundo problema. Esse mapeamento pode ser de muitos para um. Isso significa que a função de transformação não é, necessariamente, injetora, bem como não é, necessarimante, sobrejetora. Tal transformação fornece os meios de converter qualquer algoritmo que resolva o segundo problema no algoritmo correspondente para resolver o primeiro problema. Sipser (2007, p. 198) exemplifica que o problema de se resolver um sistema de equações lineares se reduz ao problema de se inverter uma matriz.

Para se ter uma noção formal de redutibilidade, precisa-se da definição de *função computável*. Veja o capítulo 14 deste livro, caso não esteja familiarizado com essa expressão. Note que, para ser função, a relação deve ser funcional e total. Sipser (2007, p. 217) fornece a seguinte definição de função computável:

Definição. Uma função $f : \Sigma^* \rightarrow \Sigma^*$ é uma *função computável* se alguma MT, sobre toda entrada x, para com exatamente $f(x)$ sobre a fita de saída.

Uma linguagem L_1 é transformável polinomialmente para uma linguagem L_2 se há uma MTD polinomial que converta cada *string* $x_1 \in L_1$ para a *string* $x_2 \in L_2$. Uma transformação polinomial de uma linguagem $L_1 \subset \Sigma_1{}^* $ para uma linguagem $L_2 \subset \Sigma_2{}^*$ é uma *função computável* $f : \Sigma_1{}^* \rightarrow \Sigma_2{}^*$ que satisfaz duas condições:

i) há uma MTD em *tempo polinomial* que computa f;

ii) $(\forall x \in \Sigma_1^*)\ x \in L_1 \leftrightarrow f(x) \in L_2$.

Se há uma transformação polinomial de L_1 para L_2, escreve-se $L_1 \propto L_2$, conforme Karp (1972), ou $L_1 \leq_p L_2$, conforme Cormen et al. (2009, p. 1067). Com isso, pode-se ter a seguinte definição, em que a função f é chamada de *redução* de L_1 para L_2:

Definição. A linguagem L_1 é *redutível por mapeamento* à linguagem L_2, escrito $L_1 \propto L_2$, se existe uma função computável $f : \Sigma_1{}^* \rightarrow \Sigma_2{}^*$, em que $(\forall x \in L_1 \leftrightarrow f(x) \in L_2)$.

Como descrito, um problema de decisão pode ser visto com um problema de reconhecimento de linguagem. Sejam D_1 e D_2 os conjuntos de todas as entradas possíveis para dois problemas de decisão. Considere, ainda, L_1 e L_2 como duas linguagens dos espaços de entrada dos problemas de decisão D_1 e D_2, respectivamente. Escreve-se $L_1 \propto L_2$ se há um algoritmo polinomial que converte cada entrada $u_1 \in D_1$ para outra entrada $u_2 \in D_2$, tal que $u_1 \in L_1$ se, e somente se, $u_2 \in L_2$. O algoritmo deve ser polinomial no tamanho da entrada u_1. Supõe-se que a noção de tamanho é bem definida no espaço das entradas D_1 e D_2. Em particular, o tamanho de u_2 é também polinomial no tamanho de u_1.

Dessa forma, há um algoritmo que converte um problema no outro. Ao se ter um algoritmo para L_2, então, podem-se compor os dois algoritmos para produzir um algoritmo para L_1. Sejam AC o algoritmo de conversão e AL_2 o algoritmo para L_2. Dada uma entrada arbitrária $u_1 \in D_1$, pode-se usar AC para converter u_1 para uma entrada $u_2 \in D_2$. Pode-se, então, usar AL_2 para determinar se u_2 pertence à L_2. Isso dirá se u_1 pertence à L_1 (MANBER, 1989, p. 343).

15.4.1 Características das reduções polinomiais

A operação $\propto$ é assimétrica. O fato de $L_1 \propto L_2$ não implica que $L_2 \propto L_1$. Essa assimetria é decorrente do fato de que a definição de reducibilidade polinomial exige que *qualquer* entrada de L_1 possa ser convertida para uma entrada equivalente de L_2, mas não vice-versa. É possível, e em muitos casos provável, que as entradas de L_2 envolvidas na redução sejam somente uma pequena fração de todas as entradas possíveis de L_2. Assim, se $L_1 \propto L_2$, então, L_2 é considerado um problema *pelo menos tão difícil* quanto L_1.

A operação de reducibilidade polinomial é transitiva: $L_1 \propto L_2 \wedge L_2 \propto L_3 \rightarrow L_1 \propto L_3$. Podem-se compor os dois algoritmos de conversão para formar um algoritmo de conversão de L_1 para L_3. Uma entrada $u_1 \in L_1$ será primeiro convertida para uma entrada $u_2 \in L_2$ e, então, para uma entrada $u_3 \in L_3$. Usar reduções polinomiais e uma composição de duas funções polinomiais ainda é uma função polinomial. O resultado é um algoritmo de conversão polinomial. Manber (1989, p. 343) observa que esse é um dos motivos de se usar polinômios nessa teoria.

Duas linguagens L_1 e L_2 (ou dois problemas de decisão Π_1 e Π_2) são *polinomialmente equivalentes*, ou simplesmente equivalentes, se $((L_1 \propto L_2) \wedge (L_2 \propto L_1))$. Particularmente, todos os problemas não-triviais tratáveis são equivalentes porque todos têm algoritmos polinomiais que os resolvem (MANBER, 1989, p. 343).

15.4.2 Aplicação das reduções polinomiais nesta teoria

A principal utilização da técnica de redução polinomial é que: $(L_1 \propto L_2 \wedge L_2 \in \mathrm{P} \rightarrow L_1 \in \mathrm{P}) \vee (L_1 \propto L_2 \wedge L_1 \notin \mathrm{P} \rightarrow L_2 \notin \mathrm{P})$. Isso significa que, se $L_1 \propto L_2$ e há um algoritmo polinomial para L_2, então, há um algoritmo polinomial para L_1. Se $L_1 \propto L_2$ e não há um algoritmo polinomial para L_1, então, não há um algoritmo polinomial para L_2.

Por outro lado, ao se afirmar que não existe um algoritmo polinomial para L_2, nada se pode afirmar sobre L_1. Ainda, ao se afirmar que existe um algoritmo em tempo polinomial para L_1, nada se pode afirmar sobre L_2.

Para o leitor entender bem este conceito, pode-se fazer a seguinte analogia. Considere que Lucas (L_1) é um corredor mais veloz que Samuel (L_2) e as seguintes afirmações:

- ao se verificar que Lucas não corre 100 metros em 10 segundos, implica que Samuel também não corre 100 metros em 10 segundos;
- ao se verificar que Samuel corre 100 metros em 10 segundos, implica que Lucas também corre 100 metros em 10 segundos;
- ao se verificar que Samuel não corre 100 metros em 10 segundos, nada se pode afirmar sobre Lucas;
- ao se verificar que Lucas corre 100 metros em 10 segundos, nada se pode afirmar sobre Samuel.

15.5 Teorema de Cook-Levin

Cook (1971) formulou a seguinte questão: "existe algum problema em NP tal que se ele for mostrado estar em P então, implica que P = NP?". Cook procurou um problema em NP de tal forma que se existir algoritmo polinomial determinístico para resolvê-lo, então, todos os problemas em NP poderiam ser resolvidos em tempo polinomial. Esse é o problema da satisfabilidade (SAT), que tem como entrada uma expressão booleana E na forma normal conjuntiva e sua parentisação com n proposições lógicas. Uma expressão booleana E contendo um produto (ou conjunção lóliga) de adições (ou disjunções lóligas) de variáveis booleanas é dita estar na forma normal conjuntiva. Pergunta-se se há um conjunto com n proposições lógicas (verdadeiro ou falso) que determine verdadeiro para a expressão booleana E. Como o problema da satisfabilidade possui 2^n atribuições possíves, não se sabe se pode ou não ser desenvolvido um algoritmo polinomial que mostre uma instância para o problema, sem se verificar as 2^n possibilidades. Cook enfatizou quatro itens nessa teoria (GAREY; JOHNSON, 1979, p. 13-14):

1. o significado de *reducibilidade em tempo polinomial*: reduções em que há uma transformação que pode ser executada por um algoritmo em tempo polinomial. Se

há uma redução em tempo polinomial de um problema Π em outro problema Π', isso assegura que qualquer algoritmo com tempo de execução polinomial para Π' pode ser convertido em um algoritmo com tempo de execução polinomial correspondente para Π. A transformação executada por um algoritmo em tempo polinomial de Π para Π' significa que o problema Π' é pelo menos tão difícil quanto Π. Por outro lado, se for provado que não há algoritmo em tempo polinomial para Π, implica que não há algoritmo em tempo polinomial para Π'. Turing usou uma redução de uma linguagem L_1 para uma linguagem L_2 por uma máquina em tempo polinomial que poderia realizar consultas a L_1. Essas reduções são, atualmente, chamadas de reduções de Cook (ou de Levin ou de Turing): grosso modo, dado um *oráculo* para o problema Π, seria possível reconhecer polinomialmente qualquer linguagem de NP;

2. Cook focou a atenção na classe NP de problemas de decisão que podem ser resolvidos em tempo de execução *polinomial* por um computador *não-determinístico*. Muitos dos problemas aparentemente intratáveis encontrados na prática, quando expressos como problemas de decisão, pertencem a essa classe;

3. como mencionado (o mais importante nessa publicação), ele procurou um problema na classe NP que, se fosse resolvido em tempo polinomial, implicaria que *todos* os problemas da classe NP poderiam ser resolvidos em tempo polinomial. Isso significaria que esse problema seria, pelo menos, tão difícil quanto *todos* os problemas da classe NP: SAT pertence à classe NP-Difícil (veja definição de NP-Difícil na seção sobre essa classe). Mais especificamente, Cook provou que o problema da satisfabilidade tem a propriedade de que todos os problemas pertencentes à classe NP podem ser reduzidos para ele. Se SAT puder ser resolvido por um algoritmo com tempo de execução polinomial, então, todo problema pertencente à classe NP poderá ser resolvido com tempo de execução polinomial. Se for provado que qualquer problema pertencente à classe NP é intratável, então, SAT também é. Cook também mostrou uma redução polinomial de SAT para uma variação, o problema 3SAT, que será descrito adiante. Isto é, 3SAT é, pelo menos, tão difícil quanto SAT; logo, 3SAT também pertence à classe NP-Difícil;

4. está implícito no artigo de Cook que CLIQUE compartilharia a mesma propriedade de SAT, isto é, que CLIQUE é um dos membros mais difíceis de ser resolvido na classe NP. CLIQUE: "há um subgrafo completo com k vértices em um dado grafo?". Essa classe de problemas, que consiste, em determinado sentido, dos problemas mais difíceis da classe NP, foi nomeada de classe NP-Completo; se são problemas NP e NP-Difícil, então, são problemas *completos* em NP (KARP, 1972).

Teorema de Cook-Levin. SAT $\in$ NP-Completo.

A essência da publicação de Cook (1971) é que SAT $\in$ NP-Completo $\rightarrow$ ((SAT $\in$ P $\leftrightarrow$ P = NP) $\vee$ (SAT $\notin$ P $\leftrightarrow$ P $\neq$ NP)). A prova de que SAT $\in$ NP-Completo mostra como construir a expressão booleana E a partir do algoritmo A e a entrada x. E pode ser longa, mas pode ser construída em tempo polinomial no tamanho de x. O problema SAT pertence à classe NP porque podem-se estabelecer 0s e 1s para as variáveis nas quais satisfaçam E em tempo polinomial. Isto é, SAT $\in$ NP, já que um algoritmo não-determinístico para ele necessita somente adivinhar uma atribuição verdadeira para as dadas variáveis e verificar se a atribuição satisfaz a todas as cláusulas na coleção dada. Isso pode ser realizado por um algoritmo não-determinístico em tempo polinomial no tamanho de E (ZIVIANI, 2011, p. 415).

Deve-se mostrar que, $\forall L \in$ NP, $L \propto$ SAT (veja seção 15.4). Reverte-se ao nível de linguagem, em que SAT é representado por uma linguagem $L_{SAT} = L$[SAT,s] para algum esquema de codificação razoável s. A prova usa a máquina de Turing não-determinística, capaz de resolver em tempo polinomial qualquer problema em NP. Assim, uma correspondência é estabelecida entre todo problema em NP e alguma instância do problema da satisfabilidade.

As linguagens em NP são bastante diversas entre si e há muitas delas. Então, não se pode esperar que se apresente uma transformação para cada uma delas. No entanto, cada uma das linguagens em NP pode ser descrita formalmente, simplesmente, ao se fornecer uma MTND em tempo polinomial que as reconhece. Isso permite que se trabalhe com uma MTND em tempo polinomial genérico e também derivar uma transformação genérica da linguagem que reconhece cada linguagem em NP para L_{SAT}.

A ideia principal da prova de que SAT $\in$ NP-Difícil é que a máquina de Turing, mesmo uma não-determinística, e todas as suas operações em E podem ser descritas por uma expressão booleana. Entenda por *descritas* a expressão que será satisfazível se, e somente se, a MTND reconhecer E. Isso não é fácil de ser feito e tal expressão torna-se bastante grande e complicada, ainda que seu tamanho não seja maior que um polinômio no número de passos que a MTND realiza.

Demonstrações completas de que SAT $\in$ NP-Completo podem ser encontradas em Aho, Hopcroft e Ullman (1974, p. 378-384) e Garey e Johnson (1979, p. 39-44), inclusive com descrições completas de máquinas de Turing determinísticas e não-determinísticas. Outros excelentes livros em que a prova completa pode ser encontrada são os de Sudkamp (2006, p. 499-500) e Sipser (2007, p. 293-299). Cormen et al. (2009, p. 1070-1077) provam que CIRCUIT-SAT $\in$ NP-Completo. O CIRCUIT-SAT é o problema que recebe como entrada um circuito booleano com um só nodo de saída e pergunta se existe uma atribuição de valores satisfatória para as entradas do circuito, isto é, de forma que a saída seja 1.

Há alguns anos, essa prova passou a ser chamada de Teorema de Cook-Levin. Levin (1973) provou, independentemente, que seis problemas de busca pertencem a NP-Completo. Levin chamou esses problemas de *universais* (ou NP-Completos), no sentido que, ao se resolvê-los, permite-se resolver qualquer outro problema *razoável* (em NP). Trakhtenbrot (1973) menciona que os resultados foram citados em palestras e submetidos à publicação em 1971. A abordagem de Levin é um pouco diferente da abordagem de Cook e Karp, pois Levin considerou problemas de busca. Problemas de busca exigem encontrar soluções, em vez de simplesmente determinar a existência.

15.6 Classe NP-Difícil

Com a discussão anterior, pode-se definir a seguinte classe de funções.

NP-Difícil = $\{\Pi : (\forall \Pi' \in \text{NP})\ \Pi' \propto \Pi\}$.

NP-Difícil é a classe de problemas, pelo menos, tão difíceis quanto qualquer outro da classe NP. Lembre-se que a noção de *difícil* neste contexto está relacionada com a

complexidade de tempo de um algoritmo resolver o problema. Como descrito, Cook (1971) mostrou que o problema da satisfabilidade pertence à classe NP-Difícil.

Os problemas NP-Difíceis não se restringem a problemas de decisão. Isso significa que os problemas pertencentes à classe NP-Difícil não pertencem, necessariamente, à classe NP. Por exemplo, um problema de otimização é, pelo menos, mais difícil que sua versão como problema de decisão.

Ainda, mesmo que o problema NP-Difícil seja um problema de decisão, não necessariamente pertence à classe NP. Então, em ($\forall \Pi$' $\in$ NP) $\Pi' \propto \Pi \rightarrow \Pi \in$ NP-Difícil, ocorre que $\Pi \in$ NP $\vee$ $\Pi \notin$ NP. Isso significa que o problema $\Pi \in$ NP-Difícil, se todo problema pertencente à classe NP pode ser reduzido polinomialmente para Π, mas Π pode não pertencer a NP. O problema da parada é o exemplo clássico de problema de decisão que pertence à classe NP-Difícil, mas não pertence à classe NP, pois o problema da parada é indecidível. Veja descrição do problema da parada no capítulo 13 deste livro. Com isso, a tese de Church-Turing pode ser enunciada da seguinte forma: se um problema de decisão tem solução, então, existe uma MTD que *decide* o problema de decisão para quaisquer instâncias. Como o problema da parada é indecidível, não pertence à classe NP, pois os problemas pertencentes à classe NP são decidíveis. Mesmo que viesse a ser provado que P = NP, não garantiria que todos os problemas em NP-Difícil pudessem ser resolvidos em tempo polinomial.

Também se pode mostrar que um problema de decisão Π percente à classe NP-Difícil da seguinte forma: ($\exists \Pi' \in$ NP-Difícil) ($\Pi' \propto \Pi$) $\rightarrow$ ($\Pi \in$ NP-Difícil). Isso significa que se Π é pelo menos mais difícil que outro problema NP-Difícil, então, $\Pi \in$ NP-Difícil.

Garey e Johnson (1979, p. 13) ensinam que, antes do trabalho de Cook (1971), vários outros mostraram alguns problemas como sendo polinomialmente relacionados ou que um problema é pelo menos tão difícil quanto outro. Mas, os autores não perceberam a extensão da classe. Exemplos incluem Dantzig, Blattner e Rao (1967), que relacionaram o problema do caixeiro viajante ao problema do caminho mínimo com pesos negativos, e Divertti e Grasselli (1968) que mostraram a relação entre o problema da cobertura de vértices com o problema do conjunto de arestas de retorno (AHO; HOPCROFT; ULLMAN, 1974).

15.7 Classe NP-Completo

Com as discussões anteriores, pode-se definir a seguinte classe de funções.

NP-Completo = $\{\Pi \in \text{NP}: (\forall \Pi' \in \text{NP})\ \Pi' \propto \Pi\}$.

Essa definição significa que $(\Pi \in \text{NP}) \wedge (\Pi \in \text{NP-Difícil}) \rightarrow \Pi \in \text{NP-Completo}$. Informalmente, os problemas da classe NP-Completo pertencem à classe NP e também à classe NP-Difícil: cada problema da classe NP-Completo é, pelo menos, tão difícil quanto qualquer outro problema pertencente à classe NP.

A Figura 15.2, é possível observar que NP-Completo = NP $\cap$ NP-Difícil. A classe NP-Completo é a interseção das classes NP e NP-Difícil. Claramente, NP-Completo $\subseteq$ NP e NP-Completo $\subseteq$ NP-Difícil.

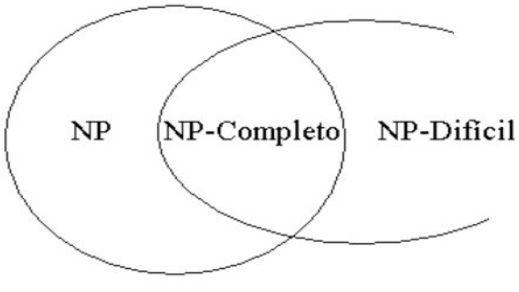

Figura 15.2: Relação entre as classes NP, NP-Difícil e NP-Completo.

Os problemas da classe NP-Completo são *completos* dentro da classe NP, isto é, são NP e também NP-Difícil, como descrito. Os problemas da classe NP-Completo são, em determinado sentido, os mais difíceis na classe NP. Nesse sentido, os problemas da classe P são os menos difíceis da mesma classe.

Em termos de linguagens, em $(\forall L, L' \in \text{NP})\ (L' \propto L) \rightarrow (L \in \text{NP-Completo})$, significa que, se uma linguagem $L \in$ NP-Completo, então, $(L \in \text{NP})((\forall L' \in \text{NP})\ (L' \propto L))$. Uma linguagem $L \in$ NP-Completo se $L \in$ NP e, para *todas* as linguagens $L' \in$ NP, ocorre $L' \propto L$.

Como descrito, o primeiro problema que se demonstrou que pode ser reduzido polinomialmente de todos os problemas da classe NP foi SAT. Dessa forma, Cook (1971) incluiu SAT nas classes NP-Difícil e NP-Completo. Os demais foram reduzidos polinomialmente a partir de SAT ou de algum outro problema que fora reduzido polinomialmente de SAT: lembre-se que a operação de redução polinomial é transitiva. Isso significa que a classe NP-Completo contém *problemas de decisão* que podem ser reduzidos polinomialmente de outros problemas pertencentes à classe NP-Completo, como descrito a seguir.

A publicação de Cook (1971) não teria tamanha relevância na Ciência da Computação e matemática se não fosse a publicação de Karp (1972), que mostrou que uma coleção de versões em problemas de decisão de problemas combinatórios conhecidos, inclusive o problema do caixeiro viajante, são, pelo menos, tão difíceis quanto SAT. Mais especifamente, Karp provou que 21 problemas pertencem à classe NP-Completo; eles são pelo menos tão difíceis quanto SAT. Ainda, Karp provou que três problemas em teoria dos autômatos e linguagens podem ser reduzidos de todo problema em NP-Completo. No entanto, havia dúvidas se estariam em NP. Com esses resultados, Karp mostrou como um problema combinatório Π_1 pode ser polinomialmente reduzido de outro problema combinatório $\Pi_2 \in$ NP-Completo. Isso significa que, se for apresentado um algoritmo com complexidade polinomial para Π_1, então, Π_2 tem um algoritmo com complexidade polinomial para resolvê-lo, o que implicaria P = NP. Se for desmonstrado que Π_2 tem limite inferior (de crescimento da função que expressa sua complexidade de tempo) não-polinomial, implica que o mesmo ocorre para o problema Π_1. Consequentemente, isso implicaria P $\neq$ NP.

Em busca de motivar a leitura de Karp (1972), considere a explicação a seguir. Para provar que $\Pi \in$ NP-Completo, mostra-se que $\Pi \in$ NP e de algum problema $\Pi' \in$ NP-Completo, computa-se $\Pi' \propto \Pi$. Isso significa que o problema de decisão Π_1 é *completo* dentro da classe NP se Π_1 é reduzido polinomialmente de $\Pi_2 \in$ NP-Completo. Em termos de linguagens, mostra-se $(L_1, L_2 \in \text{NP})(L_1 \in \text{NP-Completo})(L_1 \propto L_2) \rightarrow (L_2 \in$ NP-Completo).

Desde a publicação de Karp (1972), uma grande variedade de problemas foi provada ter dificuldade equivalente a esses problemas, os quais são comumente encontrados em

áreas diversas das ciências. Exemplos estão incluídos na teoria dos números e construção de compiladores. Também há exemplos nas engenharias e na otimização combinatória. Ainda há exemplos na indústria, como no planejamento de tarefas.

Se qualquer problema na classe NP-Completo puder ser resolvido em tempo polinomial, então, todos os problemas em NP poderão ser resolvidos em tempo polinomial. Isso implicaria P=NP. Se qualquer problema em NP é intratável, então, todos os problemas em NP-Completo e em NP-Difícil também são. Isso implicaria P≠NP. Claramente, se fosse conhecido que $\Pi \in$ NP e, posteriormente, fosse provado que $\Pi \in$ NP-P, então, implicaria que P≠NP. O inverso também é claramente verdadeiro.

15.8 Relações entre as classes NP, NP-Difícil e NP-Completo

Foi descrito que um problema Π pode pertencer à NP-Difícil, mas Π pode não pertencer à NP. Por exemplo, Π não é, necessariamente, um problema de decisão ou pode ser indecidível. Por exemplo, o problema da parada é indecidível. Não há um algoritmo que resolva o problema da parada.

Ziviani (2011, p. 415) mostra uma redução polinomial do problema da satisfabilidade para o problema da parada. Essa é outra confirmação de que o problema da parada pertence à classe NP-Difícil. É pelo menos tão difícil quanto qualquer outro problema da classe NP. O problema da parada não pertence à classe NP porque todos os problemas pertencentes a esta classe são decidíveis. Consequentemente, o problema da parada não pertence à classe NP-Completo.

Outro exemplo de problema pertencente à classe NP-Difícil é o problema do caixeiro viajante, que pergunta pela rota de custo *mínimo* num grafo ponderado com n vértices. Esta é a *versão de otimização* de um problema de decisão que pode ser descrito em perguntar se há uma rota de custo igual ou menor a k num grafo ponderado com n vértices, que pertence à classe NP-Completo. Também pode ocorrer que um problema de decisão decidível esteja em NP, mas não em NP-Completo, quando ainda não se mostrou uma redução polinomial para ele a partir de algum problema que já esteja em NP-Difícil. Particularmente, Ladner (1975) mostrou que há problemas que não pertencem a P nem a NP-Completo, se P $\neq$ NP.

Há autores, como, por exemplo, Hopcroft, Ullman e Motwani (2003, p. 456), que aceitam chamar de intratável também os problemas NP-Difíceis. Exemplos incluem os problemas indecidíveis. Entretanto, é melhor prestar atenção ao problema a que se refere para considerá-lo intratável. Chamar um problema de otimização de intratável porque é *provável* que não tenha um algoritmo em tempo polinomial é considerar realmente que P$\neq$NP, o que ainda não foi formalmente provado.

Considere um problema de otimização $\Pi_d \in$ NP-Difícil $\wedge\ \Pi_d \notin$ NP-Completo. De fato, se alguém provar que P=NP, nada se diz sobre esses problemas NP-Difíceis como Π_d. Por outro lado, os problemas indecidíveis são realmente intratáveis e há aqueles pertencentes à classe NP-Difícil, como o problema da parada. Pode-se afirmar que há problemas indecidíveis que são NP-Difíceis e intratáveis e há problemas de otimização que são NP-Difíceis, mas não é completamente adequado tratá-los como intratáveis.

Outra classe importante é a P-*SPACE*. Esta classe é composta por problemas de decisão que podem ser resolvidos por máquinas de Turing com espaço limitado por um polinômio. Sabe-se que P $\subseteq$ NP $\subseteq$ P-*SPACE* e ocorre que P = P-*SPACE* $\rightarrow$ P = NP, mas P = NP $\not\rightarrow$ P = P-*SPACE*. Veja Hartmanis e Hartmanis e Simon (1974), o capítulo 11 de Hopcroft, Ullman e Motwani (2003) ou o capítulo 17 de Sudkamp (2006), para detalhes da classe P-*SPACE*.

As classes apresentadas neste texto são as básicas dessa teoria. Para se aprofundar nesse assunto, veja, como exemplos, o capítulo 11 de Hopcroft, Ullman e Motwani (2003) e o capítulo 17 de Sudkamp (2006).

15.9 P$\stackrel{?}{=}$NP

A questão se os problemas pertencentes à classe NP-Completo são intratáveis ou não é considerada, desde a década de 1970, um dos mais importantes problemas em aberto da Matemática e da Ciência da Computação. Apesar da conjectura de que os problemas pertencentes à classe NP-Completo são intratáveis, pouco progresso foi feito para estabelecer uma prova ou contraprova. Ou seja, o problema P$\stackrel{?}{=}$NP é, geralmente, considerado não resolvido. Muitos amadores e alguns pesquisadores profissionais têm procurado solução para o problema. Fortnow e Homer (2003) e Fortnow (2009) forne-

cem revisões sobre as técnicas já utilizadas para mostrar que P≠NP. Woeginger (2011) mantém uma lista dessas tentativas.

15.9.1 Tentativa de solução em 2010

Este problema é tão importante que propostas de soluções fogem aos padrões de publicações de resultados científicos. Geralmente, um artigo científico é desenvolvido por pesquisadores profissionais ou professores universitários. Para ser publicado, o artigo deve ser revisado por especialistas na área do artigo. O autor (ou autores) determina o veículo científico em que seu artigo melhor se enquadra. Geralmente, os jornais científicos têm um escopo bem delimitado dos assuntos dos trabalhos que publicam. Note que uma ciência como a Computação tem dezenas, talvez centenas, de disciplinas em que a pesquisa é intensa. O autor submete o trabalho para o periódico científico escolhido e o editor determina se o artigo pode ser revisado por seu corpo de revisores. Esses revisores permanecem anônimos, no processo de revisão e depois. Esses revisores são especialistas na área e, geralmente, são *muito* ocupados. Por isso, um artigo pode levar de três meses a três anos para ser revisado. O artigo pode ser recusado ou aceito. Se aceito, pode haver pequenas alterações que devem ser realizadas. Outras vezes, os revisores pedem (na verdade, exigem) grandes correções ou mais pesquisa para que o artigo possa ser publicado. Muitas vezes, um estudante de graduação não imagina o nível de exigência necessário para descrever um resultado científico.

No caso de alguém provar que P=NP, provavelmente não enviaria para ser publicado. Já se alguém provar que P≠NP, então, provavelmente, também não enviaria para esse processo de revisão por pares, que pode demorar anos. Foi o que ocorreu em agosto de 2010. Deolalikar (2010) enviou uma suposta prova de que P≠NP por *e-mail* para alguns dos mais respeitados pesquisadores em teoria da computação (ainda bem que a tentativa foi P≠NP, pois, assim, "tudo" continua como está e disciplinas como algoritmos aproximativos e otimização combinatória são relevantes e compensam os esforços das últimas décadas.) Alguns desses pesquisadores que receberam o artigo de Deolalikar reenviaram o artigo para outros e setores da imprensa noticiaram precipitadamente que o problema havia sido resolvido. Ainda, alguns desses pesquisadores comentaram o artigo em seus *blogs*. Essa notíca foi muito comentada pela internet, nos meios acadêmicos e

pela imprensa. Distinto dos processos costumeiros de revisão por pares, o artigo foi revisado publicamente. Como exemplos, veja Nielsen (2010) e a descrição de Rehmeyer (2010). Immerman (2010) apontou erros sérios no artigo. Mesmo com as correções enviadas por Deolalikar dias depois, Clarkson (2010) também apontou erros no artigo.

15.9.2 Um dos problemas do milênio

P$\stackrel{?}{=}$NP é conhecido como um dos problemas do milênio. O Clay Mathematics Institute (2011) é uma fundação particular dedicada a fomentar e disseminar o conhecimento matemático. Esse instituto elegeu o problema P$\stackrel{?}{=}$NP como um dos sete problemas em aberto mais importantes na Matemática e oferece US$1M para quem resolvê-lo formalmente. Na verdade, agora são 6 porque, em 2006, o instituto anunciou o prêmio a G. Perelman, pela resolução da conjectura de poincaré. Alguns poderiam oferecer um tanto a mais, pois, se alguém descobrir que P=NP, é possível que obtenha muito mais de outras formas, em que quebrar cifras de chave pública baseadas em fatores de número primos é somente um exemplo. Mesmo sem uma prova de que NP-Completude implica intratabilidade, o conhecimento de que o problema é NP-Completo pelo menos sugere que um maior avanço da tecnologia será necessário para resolvê-lo com um algoritmo em tempo de execução polinomial.

Baker, Gill e Solovay (1975) discutiram a *relativização* desta questão a oráculos computáveis. Os autores mostraram que, dependendo do oráculo usado, a proposição relativizada poderia ser verdadeira ou falsa. Com isso, quando relativizado a um oráculo, nenhum método de prova poderia, possivelmente, resolver a questão (DAVIS, 2004).

No primeiro Teorema da Incompletude, Gödel (1931) provou que existe uma sentença *G* que não pode ser provada em um sistema formal poderoso o suficiente, como por exemplo, o que inclui a aritmética. De maneira simples, isso pode ser descrito como: se *G* pode ser provada, então, *G* é falsa e o sistema é, portanto, inconsistente; se *G* não pode ser provada, então, *G* é verdadeira e o sistema é, portanto, incompleto. Conclui-se, desse teorema, que, em existindo uma conjectura matemática resistente à prova, essa conjectura pode ser um exemplo de problema insolúvel. Como aplicação desse teorema, o problema P$\stackrel{?}{=}$NP pode ser insolúvel, como sugerido pelo professor J. Q. Uchôa da Universidade Federal de Lavras, em comunicação pessoal.

15.10 Provas de NP-Completude

A seguir, são dadas duas provas de NP-Completude. A primeira mostra SAT $\propto$ 3SAT e a segunda mostra 3SAT $\propto$ CLIQUE.

15.10.1 3SAT

O problema 3SAT é uma simplificação do problema SAT original. Uma instância de 3SAT é uma expressão booleana em que cada cláusula contém exatamente três variáveis. Dada uma expressão booleana na forma normal conjuntiva, tal que cada cláusula contém exatamente três variáveis, determine se a expressão é satisfazível. A prova a seguir, que SAT é reduzível polinomialmente a 3SAT, pode ser encontrada também em Manber (1989, p. 350-351).

Teorema. *3SAT $\in$ NP-Completo.*

Demonstração. 3SAT $\in$ NP, pois podem-se atribuir valores booleanos para as variáveis booleanas para que tornem a expressão booleana verdadeira. Pode-se verificar que as atribuições satisfazem a expressão booleana em tempo polinomial.

Lema. SAT $\propto$ 3SAT.

Prova. Seja E um instância de SAT. Substitui-se cada cláusula de E por várias cláusulas, cada uma contendo exatamente três variáveis. Seja C = $(x_1 \wedge x_2 \wedge ... \wedge x_k)$ uma cláusula arbitrária de E, tal que $k \geq 4$. Escreve-se cada variável em sua forma *positiva*, isto é, não se usa $\neg x_i$ somente para clareza da apresentação. Mostra-se como substituir C por várias cláusulas, cada uma com três variáveis. A ideia é introduzir k-3 variáveis novas y_1, $y_2 \cdots$, y_{k-3} que transformem a cláusula C em uma formulação 3SAT, sem afetar sua satisfabilidade. Usam-se novas (e diferentes) variáveis para cada cláusula. C é transformada em

C' = $(x_1 \wedge x_2 \wedge y_1) \vee (x_3 \wedge \neg y_1 \wedge y_2) \vee (x_4 \wedge \neg y_2 \wedge y_3) \vee (x_5 \wedge \neg y_3 \wedge y_4) \vee \cdots \vee (x_{k-1} \wedge x_k \wedge \neg y_{k-3})$.

Afirma-se que C' é satisfazível se, e somente se, C é satisfazível. Se C é satisfazível, então, um dos x_i deve ser estabelecido como 1 (ou verdadeiro). Nesse caso, estabelecem-se valores de $y_j = 1$ em C', para todo $j < i-1$, e $y_l = 0$ para os demais, tal que todas as cláusulas em C' são também satisfeitas. Por exemplo, se $x_3 = 1$, então, estabelece-se $y_1 = 1$ (que resolve a primeira cláusula), $y_2 = 0$ (que resolve a segunda, cláusula já que $x_3 = 1$) e os demais $y_j = 0$ (porque em todas ocorre $\neg y_j$). Da mesma forma, se C' é satisfazível, então, afirma-se que pelo menos um dos x_i deve ser 1. De fato, se todos x_i=0, então, a expressão se torna $(y_1) \vee (\neg y_1 \wedge y_2) \vee (\neg y_2 \wedge y_3) \vee \cdots \vee (\neg y_{k-3})$, que é insatisfazível.

Veja um exemplo para *k=4*. Então, C = $(x_1 \wedge x_2 \wedge x_3 \wedge x_4)$ e C' = $(x_1 \wedge x_2 \wedge y_1) \vee (x_3 \wedge x_4 \wedge \neg y_1)$. Se $x_1 = x_2 = x_3 = x_4 = 0$, então, C' = $(y_1) \vee (\neg y_1)$, que é insatisfazível. Se $x_1 = 1$ ou $x_2 = 1$ em C, então, $y_1 = 0$, o que resulta que C' é satisfazível. Se $x_1 = x_2 = 0$ e $x_3 = 1$ ou $x_4 = 1$ em C, então, $y_1 = 1$, o que resulta que C' é satisfazível.

Com essa redução, substitui-se qualquer cláusula que tenha mais do que três variáveis por várias cláusulas, cada uma com exatamente três variáveis. Resta transformar cláusulas com uma ou duas variáveis. Se C tem somente duas variáveis, isto é, C = $(x_1 \wedge x_2)$, então, C' = $(x_1 \wedge x_2 \wedge y) \vee (x_1 \wedge x_2 \wedge \neg y)$, em que y é uma variável nova. Se C tem somente uma variável, isto é, C = x, então, C' = $(x \wedge y \wedge z) \vee (x \wedge \neg y \wedge z) \vee (x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge \neg z)$, em que y e z são variáveis novas. Reduziu-se uma instância geral de SAT para uma instância de 3SAT, tal que uma instância é satisfazível se, e somente se, a outra também é. É fácil verificar que a redução pode ser realizada em tempo polinomial. □

15.10.2 Clique

Uma clique em um grafo não-direcionado G=(V,E) é um subconjunto V' $\subset$ V de vértices, em que cada par é conectado por uma aresta em E. Em outras palavras, uma clique é um subgrafo completo de G. O tamanho da clique é o número de vértices que ele contém, isto é —V'—. O problema da clique é um problema de otimização. Deseja-se encontrar a clique de tamanho máximo em um grafo. Tratando-se de problemas de decisão, pergunta-se se a clique de um dado tamanho k existe no grafo. Uma definição formal é:

CLIQUE = (G,k) : G é um grafo com uma clique de tamanho k.

A prova a seguir, que 3SAT é reduzível polinomialmente ao problema da clique, pode ser encontrada também em Cormen et al. (2009, p. 1086-1089).

Teorema. CLIQUE $\in$ NP-Completo.

Demonstração. Para mostrar que CLIQUE $\in$ NP, para um dado grafo G=(V,E), usa-se o conjunto V' contido em V de vértices na clique como um certificado de G. A verificação se V' é uma clique pode ser realizada em tempo polinomial ao verificar, para cada par $u, v \in$ V', se a aresta $(u,v) \in$ E.

Lema 4. *3SAT* $\propto$ *CLIQUE.*

A prova mostra que o problema da clique é NP-Difícil. O algoritmo de redução começa com uma instância de 3SAT. Seja $\phi = C_1 \vee C_2 \vee \cdots \vee C_k$ *uma expressão booleana na forma normal conjuntiva com três variáveis booleanas em cada cláusula. Para* r = *1, 2,* $\cdots$, k, *cada cláusula* C_r *tem exatamente três literais distintos* l_1^r, l_2^r *e* l_3^r. *Deve-se construir um grafo G tal que* ϕ *é satisfazível se, e somente se, G tem uma clique de tamanho* k.

O grafo G=(V,E) é construído como a seguir. Para cada cláusula $C_r = (l_1^r \wedge l_2^r \wedge l_3^r)$ *em* ϕ, *coloca-se uma tripla de vértices* v_1^r, v_2^r *e* v_3^r *em V. Coloca-se uma aresta entre dois vértices* v_i^r *e* v_j^r *se ambos atendem a: 1)* v_i^r *e* v_j^s *estão em triplas diferentes, isto é* $r \neq s$ *e 2) seus literais correspondentes são consistentes, isto é,* l_i^r *não é a negação de* l_i^s.

O grafo da Figura 15.3 pode ser computado facilmente a partir de ϕ *em tempo polinomial. Veja o exemplo:* $\phi = C_1 \vee C_2 \vee C_3$, *em que* $C_1 = x_1 \wedge x_2 \wedge \neg x_3$, $C_2 = x_1 \wedge \neg x_2 \wedge x_3$ *e* $C_3 = \neg x_1 \wedge \neg x_2 \wedge \neg x_3$. *Ao estabelecer* $x_1 = 1$, $x_2 = 0$, $x_3 = 0$ *ou 1, satisfaz* ϕ, *pois satisfaz* C_1 *com* $x_1 = 1$, *satisfaz* C_2 *com* $x_1 = 1$ *ou* $x_2 = 0$, *e satisfaz* C_3 *com* $x_2 = 0$.

Deve-se mostrar que esta transformação de ϕ *em G é uma redução. Supõe-se que* ϕ *tem uma atribuição satisfazível. Então, cada cláusula* C_r *contém, pelo menos, um literal* l_i^r *que é estabelecido como 1, e cada literal corresponde ao vértice* v_i^r. *Obtendo-se um desses literais valorados como* verdadeiro *de cada cláusula, fornece-se um conjunto V' de k vértices.*

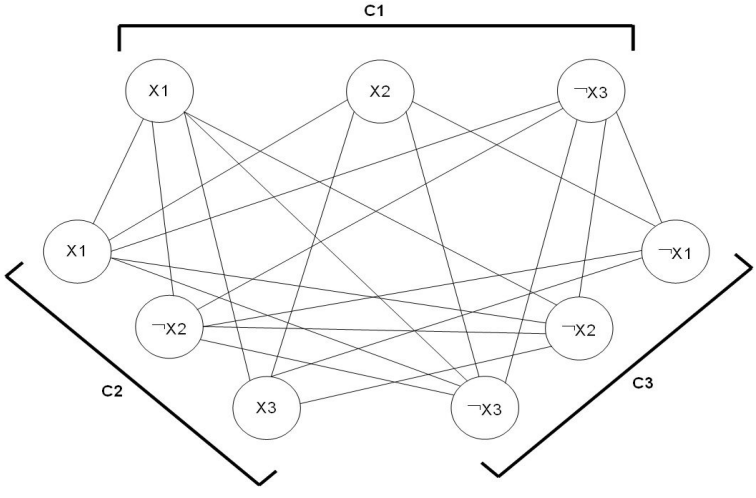

Figura 15.3: Redução de 3SAT para CLIQUE.

Afirma-se que V' é uma clique. Para cada dois vértices v_i^r, $v_j^s \in V'$, *em que* $r \neq s$, *ambos correspondendo a literais* l_i^r *e* l_i^s, *são mapeados para 1 pela atribuição de satisfabilidade dada, e assim os literais não podem ser complementos. Assim, pela construção de G, a aresta* $(v_i^r, v_j^s) \in E$.

De maneira reversa, suponha que G tem uma clique V' de tamanho k. *Nenhuma aresta em G conecta vértices na mesma tripla, e então, V' contém exatamente um vértice por tripla. Pode-se atribuir 1 para cada literal* l_i^r, *tal que* $v_i^r \in V'$ *sem risco de atribuir 1 tanto para um literal e seu complemento, já que G não contém aresta entre literais inconsistentes. Cada cláusula é satisfeita, e então,* ϕ *é satisfeita. Quaisquer variáveis que não correspondem a vértices na clique podem ser estabelecidas arbitrariamente.*

□

Observe que, na prova, reduziu-se uma instância arbitrária de 3SAT para uma instância de CLIQUE com uma estrutura particular. Poderia parecer que foi mostrado somente que CLIQUE é NP-Difícil nos grafos em que vértices são restritos a ocorrência de triplas e nos quais não há arestas entre vértices na mesma tripla. De fato, mostrou-se

que CLIQUE é NP-Difícil somente nesse caso restrito. Mas, isso é prova suficiente para mostrar que CLIQUE é NP-Difícil em grafos gerais. Isso porque, caso se tivesse um algoritmo polinomial que resolvesse CLIQUE em grafos gerais, ele resolveria CLIQUE em grafos restritos.

Observe que a redução utilizou uma instância de 3SAT, mas não a solução. Seria uma erro para a redução em tempo polinomial ter se baseado no conhecimento de que a fórmula ϕ é satisfazível, já que não se saberia como determinar essa informação em tempo polinomial.

15.11 Considerações finais

A essência desta discussão é investigar problemas equivalentes quando um algoritmo eficiente não pode ser encontrado. É por isso que o título deste capítulo, e dos capítulos dos livros dos principais autores nesta área, é NP-Completude. Quando há um problema para o qual não se conhece uma solução eficiente, tenta-se mostrar que o problema é equivalente a outros problemas que já se sabe que são difíceis. A classe NP-Completo contém centenas de problemas equivalentes. Pode-se considerar que há milhares de problemas na classe NP-Completo se variações do mesmo problema forem contados.

Uma das relevâncias dessa teoria é que, se um problema pertence à classe NP, principalmente à classe NP-Difícil e em especial à classe NP-Completo, pode ser mais compensador procurar um algoritmo aproximado e eficiente para o problema do que se esforçar para encontrar um algoritmo que resulte na solução exata em tempo polinomial, ou seja, eficiente.

Para se tornar um bom desenvolvedor de *software*, deve-se entender a NP-Completude. Ao se estabelecer um problema como NP-Completo, mostra-se o quão difícil o problema provavelmente é. Pode então, ser melhor desenvolver um algoritmo aproximativo ou resolver um caso especial por um algoritmo eficiente.

Há pesquisadores de uma linha de pesquisa chamada otimização combinatória, às vezes chamada de programação matemática, que trabalham em:

- casos especiais do problema - com alguma restrição na estrutura de entrada do problema, obtêm-se soluções rápidas para entradas pequenas, aceitam-se algoritmos exponenciais, etc;

- algoritmos aproximativos - busca-se não a solução exata, mas uma aproximação com boa qualidade e com esforço computacional reduzido;

- algoritmos estocásticos - algum tipo de aleatoriedade é introduzido para se obter tempo de execução médio rápido, por exemplo, o Método de Monte Carlo;

- parametrização - obtêm-se algoritmos eficientes se alguns parâmetros são fixados;

- heurísticas, meta-heurísticas, ou versões híbridas: um algoritmo que executa razoavelmente bem em muitos casos, mas não se tem prova de que é sempre eficiente e produz bons resultados. Como explicam Dasgupta, Papadimitriou e Vazirani (2009), heurísticas baseiam-se em engenhosidade, intuição, entendimento profundo do problema, experimentação meticulosa e conhecimentos oriundos de outras áreas da Ciência.

Como descreve Manber (1989, p. 342), essas soluções podem não ser ótimas e nem sempre funcionarem, mas são melhores que nada.

15.12 Exercícios

1. Em relação à NP-Completude, explique as classes abordadas.

2. Em relação à NP-Completude, complete:

   a) P é a classe de problemas que podem ser _____ em tempo polinomial.

   b) NP é a classe de problemas de _____, em que a resposta é sim ou não, que podem ser _____ em tempo polinomial.

   c) NP é a classe de problemas de _____ que podem ser _____ por algoritmos polinomiais não-determinísticos.

   d) NP-Difícil é a classe de problemas pelo menos _____ que qualquer outro problema pertencente à classe NP.

e) Definição. Um problema A pertence à classe NP-Completo se (1) _____ e (2) _____.

f) Um problema de _____ A é denominado NP-Completo quando: (1) _____ e (2) todo problema de _____ B $\in$ _____ pode ser _____ para A.

Nas questões seguintes, defina formal (quando possível) e detalhadamente NP-Completude. Forneça todas as definições, conceitos e características necessários para explicar esta teoria. Dê exemplos para cada um.

3. Explique problemas, problemas de decisão e de otimização, e instância de um problema.

4. Explique algoritmos, tempo de execução de algoritmos, tamanho da entrada de algoritmos.

5. Explique características da relação entre algoritmos e problemas.

6. A NP-Completude só trata problemas de decisão? Por quê? Sua resposta deve ser abrangente: outros tipos de problemas devem ser contemplados na resposta.

7. Defina formalmente uma linguagem associada a um problema de decisão.

8. Explique o problema da parada. Explique se o problema é decidível.

9. Explique em detalhes as classes P e NP, em termos de linguagens e máquinas de Turing.

10. Explique em detalhes a importância das reduções polinomiais no contexto da NP-Completude.

11. Cite e explique em detalhes o Teorema de Cook-Levin.

12. Defina formalmente as classes NP-Completo e NP-Difícil, as relações entre elas e com as classes P e NP.

13. Considere que A $\in$ NP, mas não se pode afirmar que A $\in$ NP-Difícil. Explique se isso pode ocorrer.

14. Considere que A $\in$ NP-Difícil, mas não se pode afirmar que A $\in$ NP. Explique se isso pode ocorrer.

15. Considere que SAT pertence à NP-Completo. Prove que SAT $\propto$ 3SAT.

16. Explique os problemas 3SAT e CLIQUE. Considere que 3SAT pertence à NP-Completo. Prove que CLIQUE pertence à NP-Completo.

17. Descreva a essência da teoria sobre NP-Completude e a relação com algoritmos aproximativos.

18. Explique três opções para se resolver um problema que pertence às classes NP-Completo ou NP-Difícil.

19. Escolha a alternativa correta.

    I) A Teoria sobre NP-Completude é aplicada a problemas de decisão. Problemas de decisão têm duas respostas possíveis: sim ou não.

    II) Um problema de decisão Π consiste simplesmente de um conjunto D_Π de instâncias e um subconjunto $S_\Pi \subset D_\Pi$ de instâncias "sim".

    III) Se uma função objetivo é relativamente fácil de ser avaliada, o problema de decisão pode ser mais difícil que o problema de otimização correspondente.

    IV) Se fosse possível encontrar um custo mínimo para o problema do caixeiro viajante em tempo polinomial, então, seria possível resolver um problema de decisão associado em tempo polinomial. Se for demonstrado que o problema de decisão associado ao problema do caixeiro viajante é NP-Completo (e de fato é), seria conhecido que o problema de otimização associado é pelo menos tão difícil.

    V) Apesar de a teoria sobre NP-Completude ser restrita a problemas de decisão, podem-se estender suas implicações para problemas de otimização.

    a) I: verdadeira; II: falsa; III: verdadeira; IV: falsa. V. falsa.

    b) I: falsa; II: falsa; III: falsa; IV: verdadeira. V. falsa.

    c) I: falsa; II: verdadeira; III: falsa; IV: falsa. V. verdadeira.

    d) I: verdadeira; II: verdadeira; III: falsa; IV: verdadeira. V. verdadeira

    e) Todas são verdadeiras.

20. Escolha a alternativa correta.

    I) A razão da teoria sobre NP-Completude se restringir a problemas de decisão é que eles têm uma contraparte natural em matemática: um objeto matemático preciso e adequado para o estudo em teoria da computação: linguagem. Para qualquer conjunto finito Σ de símbolos, denota-se por Σ^* o conjunto de todas as *strings* finitas de símbolos de Σ.

    II) Considere o alfabeto $\Sigma = \{0,1\}$. Então, Σ^* consiste da *string* vazia, e das *strings* 0, 1, 00, 01, 10, 11, 000, 001, e todas as outras *strings* finitas de 0s e 1s. Se L é um subconjunto de Σ^*, diz-se que L é uma linguagem sobre o alfabeto Σ. $\{01,001,111,1101010\}$ é uma linguagem sobre $\{0,1\}$.

    III) A correspondência entre problemas de decisão e linguagens é realizada pelo esquema de codificação utilizado para especificar as instâncias do problema quando se pretende computá-los. Um esquema de codificação e para um problema Π fornece um modo de descrever cada instância de Π por uma *string* de símbolos apropriada sobre algum alfabeto Σ fixo.

    IV) O problema Π e o esquema de codificação e para Π particiona Σ^* em três classes de *strings*, as que não são codificações de instâncias de Π; as que codificam instâncias de Π e a resposta é "não"; as que codificam instâncias de Π e a resposta é "sim".

    V) A classe de *strings* 3 é a linguagem associada com Π e e e estabelece: $L[\Pi, e] = \{x \in \Sigma^* : \Sigma$ é o alfabeto utilizado por e, e x é a codificação sob e de uma instância $I \in S_\Pi\}$.

    a) I: verdadeira; II: falsa; III: falsa; IV: falsa. V. verdadeira.

    b) I: verdadeira; II: verdadeira; III: falsa; IV: falsa. V. falsa.

    c) I: falsa; II: verdadeira; III: verdadeira; IV: verdadeira. V. verdadeira.

    d) I: falsa; II: falsa; III: verdadeira; IV: verdadeira. V. falsa.

    e) Todas são verdadeiras.

21. Escolha a alternativa correta.

    I) O tempo usado na computação de um programa M em uma máquina de Turing determinística com a entrada x é o número de passos que ocorrem na computação

até que o estado de parada é inserido. Para um programa M em uma máquina de Turing determinística que para para todas as entradas $x \in \Sigma^*$, sua função de complexidade de tempo $T_M : \mathbb{N} \to \mathbb{N}$ é dada por $T_M(n) = \max \{m : (\exists x \in \Sigma^*)$, onde $n = |x|$, tal que a computação de M com entrada x tem tempo $m\}$.

II) M é chamado de programa de máquina de Turing determinística com tempo polinomial, se $(\forall n \in \mathbb{N})\ T_M(n) = O(n^c)$, em que $c \in \mathbb{N}$ e $n = |x|$. Isso significa que há um polinômio p tal que $T_M(n) \leq p(n)$. Se M não para, então, a complexidade de tempo é indefinida.

III) P-*Time* = $\{L$: há um programa M de máquina de Turing determinística com tempo polinomial em que $L = L_M\}$.

IV) O programa M aceita $x \in L$, x é uma entrada que responde sim ao problema de decisão associado.

V) P é a classe de linguagens que podem ser reconhecidas em tempo polinomial por um programa para uma máquina de Turing determinística. O programa para uma máquina de Turing determinística reconhece $L \subset \{0,1\}^*$, ou seja, aceita $x \in L$.

a) I: verdadeira; II: verdadeira; III: falsa; IV: falsa. V. falsa.

b) I: verdadeira; II: verdadeira; III: verdadeira; IV: falsa. V. verdadeira.

c) I: falsa; II: verdadeira; III: verdadeira; IV: falsa. V. verdadeira.

d) I: falsa; II: falsa; III: verdadeira; IV: verdadeira. V. falsa.

e) Todas são verdadeiras.

22. Escolha a alternativa correta.

Pode-se definir NP em termos do que se chama de algoritmo não-determinístico. Há dois estágios em um algoritmo não-determinístico: o estágio do oráculo que adivinha a resposta sim e o estágio de verificação.

II) Dada uma instância I de um problema, o primeiro estágio meramente "adivinha" alguma estrutura S. Então, fornecem-se ambos, I e S, como entrada para o estágio de verificação. O estágio de verificação é computado de uma maneira determinística. O estágio de verificação ou para com a resposta sim ou para com

a resposta não ou nunca para. Os dois últimos casos não necessitam ser diferenciados.

III) Um algoritmo não-determinístico "resolve" um problema de decisão Π se as duas propriedades seguintes são comprovadas para todas as instâncias $I \in D_\Pi$: 1. Se $I \in Y_\Pi$, então, existe alguma estrutura S que, quando adivinhada para a entrada I, levará o estágio de verificação a responder sim para I e S; 2. Se $I \notin Y_\Pi$, então, não existe estrutura S que, quando adivinhada para a entrada I, levará o estágio de verificação a responder sim para I e S.

IV) Um algoritmo não-determinístico que resolve um problema de decisão Π é dito operar em "tempo polinomial" se há um polinômio p tal que, para toda instância $I \in Y_\Pi$, há alguma "adivinhação" S que leva o estágio de verificação determinístico a responder sim para I e S com tempo p no tamanho de I. Isso tem o efeito de impor um limite polinomial no tamanho da estrutura S adivinhada, já que somente uma quantidade de tempo com limite polinomial pode ser utilizada para examinar S.

V) A classe NP é definida informalmente como a classe de todos os problemas de decisão Π tal que, sob esquemas de codificação razoáveis, podem ser resolvidos por algoritmos não-determinísticos em tempo polinomial.

a) I: verdadeira; II: verdadeira; III: falsa; IV: falsa. V. falsa.

b) I: verdadeira; II: verdadeira; III: verdadeira; IV: falsa. V. verdadeira.

c) I: falsa; II: verdadeira; III: verdadeira; IV: falsa. V. verdadeira.

d) I: falsa; II: falsa; III: verdadeira; IV: verdadeira. V. falsa.

e) Todas são verdadeiras.

23. Escolha a alternativa correta.

I) Em uma máquina de Turing não determinística, a *string* testemunho adivinhada pelo estágio de adivinhação pode (e geralmente será) ser examinada durante o estágio de verificação. A computação termina quando e se o controle de estado finito entra em um dos dois estados de parada, ou q_S ou q_N, e é uma computação aceitável se para com estado q_S. Todas as demais computações, parando ou não, são classificadas juntas como computações não aceitáveis.

II) Qualquer programa de uma máquina de Turing não-determinística M terá um número grande de computações possíveis para uma dada *string* de entrada x, uma para cada *string* testemunho possível de Σ^*. Um programa M de uma máquina de Turing não-determinística aceita a *string* x se, pelo menos, uma das computações é uma computação aceitável. A linguagem reconhecida por M é $L_M = \{x \in \Sigma^* : M$ aceita $x\}$.

III) O tempo exigido por um programa M de uma máquina de Turing não-determinística para aceitar a *string* $x \in L_M$ é definido como o mínimo sobre todas as computações aceitáveis de M para x, do número de passos que ocorrem nos estágios de adivinhação e verificação até que o estado de parada q_S ocorra. A função de complexidade de tempo $T_M : \mathbb{N} \to \mathbb{N}$ para M é $T_M(n) = \max \{\{1\} \cup \{m : \exists x \in L_M$ com $n = |x|$, tal que o tempo para aceitar x por M é $m\}\}$.

IV) A função de complexidade de tempo para M depende somente do número de passos que ocorrem em computações aceitáveis. Por convenção, $T_M(n)$ é estabelecido como 1 quando nenhuma entrada de tamanho n é aceita por M. O programa M é um programa de máquina de Turing não-determinística com tempo polinomial, se existe um polinômio p tal que $(\forall n \geq 1)\ T_M(n) \leq p(n)$. Isso significa que $T_M(n) = O(n^c)$, em que $n \in \mathbb{N}$* e $c \in \mathbb{N}$.

V) NP-*Time* = $\{L$: há um programa M de máquina de Turing não-determinística em tempo polinomial pelo qual $L_M = L\}$.

a) I: verdadeira; II: verdadeira; III: falsa; IV: falsa. V. falsa.

b) I: verdadeira; II: verdadeira; III: verdadeira; IV: falsa. V. verdadeira.

c) I: falsa; II: verdadeira; III: verdadeira; IV: falsa. V. verdadeira.

d) I: falsa; II: falsa; III: verdadeira; IV: verdadeira. V. falsa.

e) Todas são verdadeiras.

24. Escolha a alternativa correta.

I) Uma redução polinomial de uma linguagem $L_1 \subset \Sigma_1^*$ para uma linguagem $L_2 \subset \Sigma_2^*$ é uma função $f : \Sigma_1^* \to \Sigma_2^*$ que satisfaz às condições: há um programa de máquina de Turing determinística em tempo polinomial que computa

f; $(\forall x \in \Sigma_1^*)\ x \in L_1 \leftrightarrow f(x) \in L_2$. Se há uma transformação polinomial de L_1 para L_2, então, escreve-se $L_1 \propto L_2$.

II) $(L_1 \propto L_2 \wedge L_2 \in \text{P}) \rightarrow (L_1 \notin \text{P})$.

III) $(L_1 \propto L_2 \wedge L_1 \notin \text{P}) \rightarrow (L_2 \in \text{P})$.

IV) $(L_1 \propto L_2 \wedge L_2 \propto L3) \rightarrow (L_1 \propto L3)$.

V) Se $L_1 \propto L_2 \wedge L_2 \propto L_1$, então, pode-se afirmar que L_1 e L_2 (associados a dois problemas de decisão Π_1 e Π_2, respectivamente) são polinomialmente equivalentes.

a) I: verdadeira; II: verdadeira; III: falsa; IV: falsa. V. falsa.

b) I: verdadeira; II: falsa; III: falsa; IV: verdadeira. V. verdadeira.

c) I: falsa; II: verdadeira; III: verdadeira; IV: falsa. V. verdadeira.

d) I: falsa; II: falsa; III: verdadeira; IV: verdadeira. V. falsa.

e) Todas são verdadeiras.

25. Escolha a alternativa correta.

I) Considere a linguagem L. $((L \in \text{NP})\ (\exists L' \in \text{NP})\ (L' \propto L)) \rightarrow (L \in$ NP-Completo).

II) Um problema de decisão $\Pi \in$ NP-Completo se $\Pi \in$ NP e existe uma redução polinomial $\Pi' \propto \Pi$ em que $\Pi' \in$ NP.

III) Pode-se afirmar que se algum problema na classe NP puder ser resolvido em tempo polinomial, então, existe pelo menos um problema em NP-Completo que pode ser resolvido em tempo polinomial.

IV) Pode-se afirmar que se algum problema em NP é resolvido em tempo polinomial, então, todos os problemas em NP-Completo também são resolvidos em tempo polinomial.

V) Considere que $\Pi \in$ NP: (P $\neq$ NP)$\rightarrow$($\Pi \in$ NP-P). Isso significa que $(\Pi \in \text{P}) \leftrightarrow$ (P = NP).

a) I: verdadeira; II: verdadeira; III: falsa; IV: falsa. V. falsa.

b) I: verdadeira; II: falsa; III: falsa; IV: verdadeira. V. verdadeira.

c) I: falsa; II: falsa; III: falsa; IV: falsa. V. verdadeira.

d) Todas são falsas.

e) Todas são verdadeiras.

26. Escolha a alternativa correta.

I) $(L_1, L_2 \in \text{NP})$ $(L_1 \in \text{NP-Completo})$ $(L_1 \propto L_2) \rightarrow (L_2 \notin \text{NP-Completo})$.

II) Para provar que $\Pi \in$ NP-Completo mostra-se que: 1. $\Pi \in$ NP; 2. $(\exists \Pi' \in$ NP-Completo$)$ $\Pi' \propto \Pi$.

III) NP-Completo é a classe composta por todos os problemas que podem ser "resolvidos" em tempo polinomial por uma máquina de Turing não-determinística.

IV) NP-Completo é a classe composta por todos os problemas que podem ser verificados em tempo polinomial por uma máquina de Turing determinística.

V) $(L_1, L_2 \in \text{NP})$ $(L_1 \in \text{NP-Completo})$ $(L_1 \propto L_2) \rightarrow (L_2 \notin \text{NP-Difícil})$.

a) I: falsa; II: verdadeira; III: falsa; IV: falsa. V. falsa.

b) I: verdadeira; II: falsa; III: falsa; IV: verdadeira. V. verdadeira.

c) I: falsa; II: falsa; III: falsa; IV: falsa. V. verdadeira.

d) Todas são falsas.

e) Todas são verdadeiras.

27. Escolha a alternativa correta.

I) Considere que SAT é o problema da satisfabilidade. SAT $\in$ NP, já que um algoritmo não-determinístico para ele necessita somente adivinhar uma atribuição verdadeira para as dadas variáveis e verificar se a atribuição satisfaz todas as cláusulas na coleção dada. Isso pode ser realizado por um algoritmo não-determinístico em tempo polinomial.

II) Deve-se mostrar que $(\forall L \in \text{NP})$ $L \propto$ SAT. Reverte-se ao nível de linguagem, onde SAT é representado por uma linguagem $L_{\text{SAT}} = L\{\text{SAT},e\}$, para algum esquema de codificação razoável e.

III) SAT $\in$ NP-COMPLETO. (SAT $\in$ P) $\rightarrow$ (P $\neq$ NP).

IV) SAT $\in$ NP-COMPLETO. (SAT $\notin$ P) $\rightarrow$ (P = NP).

V) SAT $\in$ NP-COMPLETO. (SAT $\in$ NP-Difícil) $\rightarrow$ (SAT $\in$ P).

a) I: verdadeira; II: verdadeira; III: falsa; IV: falsa. V. falsa.

b) I: verdadeira; II: falsa; III: falsa; IV: verdadeira. V. verdadeira.

c) I: falsa; II: falsa; III: falsa; IV: falsa. V. verdadeira.

d) Todas são falsas.

e) Todas são verdadeiras.

15.13 Notas bibliográficas

Segundo Goldreich (2008, p. 43), o ponto de partida de Cobham (1964) foi o desejo de apresentar filosoficamente o conceito de algoritmos eficientes. O ponto de partida de Edmonds (1965) foi o desejo de articular por que certos algoritmos são eficientes na prática. O ponto de partida de Cook (1971) parece ter sido seu interesse em procedimentos para provar teoremas para cálculo proposicional; ao tentar provar evidência da dificuldade de decidir se uma dada fórmula é uma tautologia, ele identificou NP como a classe que contém *aparentemente muitos problemas difíceis*.

A obra de Garey e Johnson (1979) é a referência mais clássica em NP-Completude e é um dos livros mais citados na Ciência da Computação. O primeiro capítulo é uma introdução agradável à NP-Completude. O segundo capítulo explica NP-Completude de forma didática, clara e também objetiva. Os autores apresentam os principais conceitos para se entender a NP-Completude. Os demais capítulos formam uma enciclopédia dos problemas relacionados à teoria. Cormen et al. (2009) no capítulo 34 e Ziviani (2011), na seção 9.1, também apresentam abordagens bastante didáticas sobre a NP-Completude. Como descrito, todos os assuntos descritos por essas obras são feitas de forma didática.

Talvez a Computação tenha "começado a nascer" há milênios, passando por gregos, Al-Khwârizmî, algoritmos, pela Lógica, por Frege, Peano, Cantor, pela formalização da Matemática, Russel e muitos outros. Com certeza, houve com Hilbert um grande impulso no que viria a ser chamado de Computação. Esse impulso foi dado também pelos alunos de Hilbert, com a sua busca em colocar a matemática em fundamentos sólidos, com seus problemas, com seu programa, o *entscheidungsproblem*, o maior problema da

lógica matemática, em 1928, segundo Hilbert & Ackerman e a lógica de predicados de primeira ordem, as perguntas se é a matemática decidível, completa e consistente. Essas últimas, em conjunto com o Principia Mathematica, de Whitehead e Russel (1927), foram as motivações para os Teoremas da Incompletude de Gödel (1931), que mostraram que um sistema formal é poderoso o bastante para incluir a aritmética ou é completo ou é consistente e que um sistema formal não pode mostrar sua própria consistência, frustrando Hilbert (REID; WEYL, 1970). Cinco anos depois, houve a prova de indecibilidade de Church (1936a) (o caso geral do *Entscheidungsproblem* da Lógica de Predicados de Primeira Ordem é insolúvel) e a consequente demonstração de Turing (1936). Conforme Petzold (2008, p. 49), com Hilbert: "nós devemos saber, nós saberemos" e com Gödel, Church e Turing: incompletude e indecibilidade. As obras de Fonseca Filho (2007) e Petzold (2008) apresentam formidavelmente esse trecho (e os demais) da História da Computação, descrevendo detalhes dos conceitos relacionados e os cientistas que trabalharam nos assuntos.

Aprender algoritmos adequadamente é essencial para que um profissional se destaque em informática e este texto apresenta elementos para que o leitor adquira esse conhecimento. O objetivo principal é apresentar conceitos fundamentais de algoritmos e suas estruturas de dados. O texto é abrangente e aborda os principais algoritmos conhecidos, que formam o núcleo da Ciência da Computação.

Os assuntos abordados são apaixonantes e desafiadores, capazes de motivar uma carreira profissional. A abordagem é um equilíbrio entre rigor matemático e didática nas apresentações e há ênfase na compreensão do conceito matemático de cada algoritmo.
Este texto atende a uma demanda por livros com linguagem matemática precisa, sem formalismo e abstração excessivos, em língua portuguesa, nas disciplinas nas quais foi concebido. Para facilitar a implementação, o pseudocódigo utilizado pode ser aplicado em qualquer linguagem de programação utilizada atualmente.

Para o professor, é útil para ajudá-lo a apresentar os assuntos consolidados, mas atuais, na Ciência da Computação. Todos os capítulos contêm notas bibliográficas para os leitores que desejarem obter informações aprofundadas sobre os tópicos e há dezenas de exercícios. A Editora UFLA tem a convicção de que entrega a professores, alunos e profissionais do nosso país um livro de altíssima qualidade que contribuirá muito para o progresso no ensino das disciplinas a que se relaciona.

Aplicações: Livro-texto de uma terceira e uma quarta disciplinas de algoritmos e estruturas de dados em cursos de Ciência da Computação, Sistemas de Informação, Engenharia da Computação, Matemática Computacional, Engenharia de Controle e Automação, Licenciatura em Computação, cursos tecnólogos de computação e outras áreas técnico-científicas, como engenharias, geociências, matemática e física. Também pode ser utilizado como leitura complementar da disciplina Teoria da Computação e como livro-texto em cursos de nivelamento em disciplinas computacionais de pós-graduação em Ciência da Computação, Modelagem Computacional, Matemática Aplicada, Física e engenharias.

ISBN 978-85-87692-97-9

www.editora.ufla.br