Análise de Algoritmos e Estruturas de Dados

Carla Negri Lintzmayer
CMCC – Universidade Federal do ABC
carla.negri@ufabc.edu.br
Guilherme Oliveira Mota
IME – Universidade de São Paulo
mota@ime.usp.br

Versão: 28 de maio de 2021

Esta versão é um rascunho ainda em elaboração e não foi revisado.

Sumário

I Conhecimentos recomendados **1**

1 Revisão de conceitos importantes **5**
1.1 Potenciação 5
1.2 Logaritmos 6
1.3 Somatórios 9

II Princípios da análise de algoritmos **15**

2 Um problema simples **21**

3 Corretude de algoritmos iterativos **29**

4 Tempo de execução **37**
4.1 Análise de melhor caso, pior caso e caso médio 46
4.2 Bons algoritmos 49

5 Notação assintótica **51**
5.1 Notações O, Ω e Θ 52
5.2 Notações o e ω 60
5.3 Relações entre as notações assintóticas 61

6 Tempo com notação assintótica **63**
6.1 Exemplo completo de solução de problema 66

7 Recursividade **71**
7.1 Corretude de algoritmos recursivos 73
7.2 Fatorial de um número 74
7.3 Potência de um número 76
7.4 Busca binária 80
7.5 Algoritmos recursivos $\times$ algoritmos iterativos 83

8 Recorrências **87**
8.1 Método da substituição . . . 89
8.2 Método iterativo . . . 97
8.3 Método da árvore de recursão . . . 100
8.4 Método mestre . . . 103

III Estruturas de dados **113**

9 Estruturas lineares **119**
9.1 Vetor . . . 119
9.2 Lista encadeada . . . 120

10 Pilha e fila **125**
10.1 Pilha . . . 125
10.2 Fila . . . 127

11 Árvores **131**
11.1 Árvores binárias de busca . . . 132
11.2 Árvores balanceadas . . . 142

12 Fila de prioridades **143**
12.1 Heap binário . . . 144

13 Disjoint Set **159**
13.1 Union-Find . . . 159

14 Tabelas hash **165**

IV Algoritmos de ordenação **167**

15 Ordenação por inserção **173**
15.1 Insertion sort . . . 173
15.2 Shellsort . . . 177

16 Ordenação por intercalação **179**

17 Ordenação por seleção **189**
17.1 Selection sort . . . 189
17.2 Heapsort . . . 192

18 Ordenação por troca **199**
18.1 Quicksort . . . 199

19 Ordenação sem comparação **211**
19.1 Counting sort 212

V Técnicas de construção de algoritmos **215**

20 Divisão e conquista **221**
20.1 Multiplicação de inteiros 221

21 Algoritmos gulosos **229**
21.1 Escalonamento de tarefas compatíveis 229
21.2 Mochila fracionária 232
21.3 Compressão de dados 236

22 Programação dinâmica **243**
22.1 Sequência de Fibonacci 244
22.2 Corte de barras de ferro 247
22.3 Mochila inteira 253
22.4 Alinhamento de sequências 259

VI Algoritmos em grafos **263**

23 Conceitos essenciais em grafos **269**
23.1 Relação entre grafos e digrafos 271
23.2 Adjacências e incidências 272
23.3 Grafos e digrafos ponderados 274
23.4 Formas de representação 275
23.5 Pseudocódigos 277
23.6 Subgrafos 279
23.7 Passeios, trilhas, caminhos e ciclos 282
23.8 Conexidade 282
23.9 Distância entre vértices 283
23.10 Algumas classes importantes de grafos 284

24 Buscas em grafos **289**
24.1 Busca em largura 293
24.2 Busca em profundidade 299
24.3 Componentes conexas 304
24.4 Busca em digrafos 305
24.5 Outras aplicações dos algoritmos de busca 316

25 Árvores geradoras mínimas **317**
25.1 Algoritmo de Kruskal 321
25.2 Algoritmo de Prim 327

26 Trilhas Eulerianas **335**

27 Caminhos mínimos **341**
27.1 Única fonte 343
27.2 Todos os pares 356

VII Teoria da computação **367**

28 Redução entre problemas **373**
28.1 Redução entre problemas de otimização e decisão 377
28.2 Formalizando a redução 379
28.3 O que se ganha com redução? 383

29 Classes de complexidade **385**
29.1 Classe **NP**-completo 389
29.2 Exemplos de problemas **NP**-completos 390
29.3 Classe **NP**-difícil 393

30 Abordagens para lidar com problemas NP-difíceis **395**

Parte

I

Conhecimentos recomendados

Nesta parte

Antes de iniciar sua jornada no aprendizado de análise de algoritmos, existem alguns conteúdos com os quais é recomendado que você já esteja confortável ou que ao menos você já tenha sido introduzido a eles. Nesta parte faremos uma revisão desses conceitos, que em resumo são:

- conhecimentos de programação básica, em qualquer linguagem imperativa, com boas noções de algoritmos recursivos e familiaridade com estruturas de dados básicas, como vetores, listas, pilhas, filas e árvores;
- capacidade para reconhecer argumentos lógicos em uma prova matemática, principalmente por indução e por contradição;
- familiaridade com recursos matemáticos como somatório, potências, inequações e funções.

Capítulo 1

Revisão de conceitos importantes

1.1 Potenciação

Seja a um número real e x um número inteiro. A operação de potenciação é definida por

$$a^x = \begin{cases} a \times a \times a \times \cdots \times a \ (x \text{ vezes}) & \text{se } x > 0 \\ \dfrac{1}{a^x} & \text{se } x < 0 \\ 1 & \text{se } x = 0 \end{cases}$$

Por exemplo, $2^5 = 2 \times 2 \times 2 \times 2 \times 2 = 32$, $4^{-2} = \dfrac{1}{4^2} = \dfrac{1}{16}$ e $2357^0 = 1$.

Se a é um número real e $x = \frac{m}{n}$ é um número racional (portanto, m e n são inteiros), então a operação de potenciação é definida por

$$a^x = a^{\frac{m}{n}} = (a^{\frac{1}{n}})^m$$

em que

$$a^{\frac{1}{n}} = b \quad \Leftrightarrow \quad a = b^n$$

É interessante notar que

$$a^{\frac{1}{x}} \text{ também é escrito } \sqrt[x]{a}$$

Assim,

$$a^{\frac{m}{n}} = (a^m)^{\frac{1}{n}} = \sqrt[n]{a^m}$$

Abaixo listamos as propriedades mais comuns envolvendo manipulação de potências:

- $a^x a^y = a^{x+y}$
- $\dfrac{a^x}{a^y} = a^{x-y}$
- $(a^x)^y = a^{xy}$
- $(ab)^x = a^x b^x$
- $\left(\dfrac{a}{b}\right)^x = \dfrac{a^x}{b^x}$

1.2 Logaritmos

Sejam n e a números positivos, com $a \neq 1$.

Definição 1.1: *Logaritmo*

O *logaritmo de n na base a*, denotado $\log_a n$, é o valor x tal que

- x é o expoente a que a deve ser elevado para produzir n ($a^x = n$); ou
- x é a quantidade de vezes que a deve ser multiplicado por a para produzir n; ou
- x é a quantidade de vezes que n deve ser dividido por a para produzir 1.

Por definição,

$$\log_a n = b \text{ se e somente se } a^b = n\,.$$

No decorrer desse livro, vamos considerar que $\log n = \log_2 n$, omitindo a base.

Como exemplos, temos que o logaritmo de 32 na base 2 é 5 ($\log 32 = 5$) pois $2^5 = 32$ e o logaritmo de 81 na base 3 é 4 ($\log_3 81 = 4$) pois $3^4 = 81$.

Alguns logaritmos são fáceis de calcular:

- $\log 1024 = 10$, pois 1024 é potência de 2;
- $\log_{10} 100 = 2$, pois 100 é potência de 10;
- $\log_5 125 = 3$, pois 125 é potência de 5.

Outros precisam de calculadora:

- $\log 37 = 5,209453366$
- $\log_{10} 687 = 2,836956737$

- $\log_5 341 = 3,623552317$

Porém para os fins deste livro, não precisaremos utilizar calculadora. Podemos ter uma noção dos valores por meio de comparação com os logaritmos fáceis:

- $\log 37$ é algum valor entre 5 e 6, pois $\log 32 < \log 37 < \log 64$, $\log 32 = 5$ e $\log 64 = 6$;
- $\log_{10} 687$ é algum valor entre 2 e 3, pois $\log_{10} 100 < \log_{10} 687 < \log_{10} 1000$, $\log_{10} 100 = 2$ e $\log_{10} 1000 = 3$;
- $\log_5 341$ é algum valor entre 3 e 4, pois $\log_5 125 < \log_5 341 < \log_5 625$, $\log_5 125 = 3$ e $\log_5 625 = 4$.

Abaixo listamos as propriedades mais comuns envolvendo manipulação de logaritmos.

Fato 1.2

Dados números reais a, b, $c \geq 1$, as seguintes igualdades são válidas:

(i) $\log_a 1 = 0$

(ii) $\log_a a = 1$

(iii) $a^{\log_a b} = b$

(iv) $\log_c(ab) = \log_c a + \log_c b$

(v) $\log_c(a/b) = \log_c a - \log_c b$

(vi) $\log_c(a^b) = b \log_c a$

(vii) $\log_b a = \frac{\log_c a}{\log_c b}$

(viii) $\log_b a = \frac{1}{\log_a b}$

(ix) $a^{\log_c b} = b^{\log_c a}$

Demonstração. Por definição, temos que $\log_b a = x$ se e somente se $b^x = a$. No que segue vamos provar cada uma das identidades descritas no enunciado.

(i) $\log_a 1 = 0$. Segue diretamente da definição, uma vez que $a^0 = 1$.

(ii) $\log_a a = 1$. Segue diretamente da definição, uma vez que $a^1 = a$.

(iii) $a^{\log_a b} = b$. Segue diretamente da definição, uma vez que $a^x = b$ se e somente se $x = \log_a b$.

(iv) $\log_c(ab) = \log_c a + \log_c b$. Como a, b e c são positivos, existem números k e ℓ tais que $a = c^k$ e $b = c^\ell$. Assim, temos

$$\log_c(ab) = \log_c(c^k c^\ell) = \log_c\left(c^{k+\ell}\right) = k + \ell = \log_c a + \log_c b\,,$$

onde as duas últimas desigualdades seguem da definição de logaritmo.

(v) $\log_c(a/b) = \log_c a - \log_c b$. Como a, b e c são positivos, existem números k e ℓ tais que $a = c^k$ e $b = c^\ell$. Assim, temos

$$\log_c(a/b) = \log_c(c^k/c^\ell) = \log_c\left(c^{k-\ell}\right) = k - \ell = \log_c a - \log_c b\,.$$

(vi) $\log_c(a^b) = b\log_c a$. Como a, b e c são positivos, podemos escrever $a = c^k$ para algum número real k. Assim, temos

$$\log_c(a^b) = \log_c(c^{kb}) = kb = bk = b\log_c a\,.$$

(vii) $\log_b a = \frac{\log_c a}{\log_c b}$. Vamos mostrar que $\log_c a = (\log_b a)(\log_c b)$. Note que, pela identidade (iii), temos $\log_c a = \log_c\left(b^{\log_b a}\right)$. Assim, usando a identidade (vi), temos que $\log_c a = (\log_b a)(\log_c b)$.

(viii) $\log_b a = \frac{1}{\log_a b}$. Vamos somente usar (vii) e (ii):

$$\log_b a = \frac{\log_a a}{\log_a b} = \frac{1}{\log_a b}\,.$$

(ix) $a^{\log_c b} = b^{\log_c a}$. Esse fato segue das identidades (iii), (vii) e (viii). De fato,

$$\begin{aligned} a^{\log_c b} &= a^{(\log_a b)/(\log_a c)} \\ &= \left(a^{\log_a b}\right)^{1/(\log_a c)} \\ &= b^{1/(\log_a c)} \\ &= b^{\log_c a}\,. \end{aligned}$$

□

1.3 Somatórios

Sejam $a_1, a_2, \ldots, a_n$ números reais. O **somatório** desses números é denotado por

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + \ldots + a_n \,.$$

A expressão acima é lida como "a soma de termos a_i, com i variando de 1 até n". Chamamos i de variável.

Não é necessário que a variável do somatório comece de 1. Veja alguns exemplos abaixo:

- $\displaystyle\sum_{i=1}^{8} i = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$
- $\displaystyle\sum_{j=7}^{11} 4^j = 4^7 + 4^8 + 4^9 + 4^{10} + 4^{11}$
- $\displaystyle\sum_{k=0}^{4} (2k+1) = (2 \times 0 + 1) + (2 \times 1 + 1) + \cdots + (2 \times 4 + 1) = 1 + 3 + 5 + 7 + 9$
- $\displaystyle\sum_{i=4}^{n} i\, 3^i = (4 \times 3^4) + (5 \times 3^5) + \cdots + (n \times 3^n)$
- $\displaystyle\sum_{k=0}^{n} k^{k+1} = 0^1 + 1^2 + 2^3 + 3^4 + \cdots + n^{n+1}$

Outras notações comuns são:

- $\displaystyle\sum_{3 \leq k < 8} a_k = a_3 + a_4 + \cdots + a_7 = \sum_{k=3}^{7} a_k$
- $\displaystyle\sum_{x \in S} a_x$ é a soma de a_x para todo possível x de um conjunto S
  - Por exemplo, se $S = \{4, 8, 13\}$, então $\displaystyle\sum_{x \in S} x = 4 + 8 + 13$

Abaixo listamos as propriedades mais comuns envolvendo manipulação de somatórios.

Fato 1.3

Sejam x e y números inteiros, com $x \leq y$, c um número real e a_i e b_i números quaisquer:

- $\displaystyle\sum_{i=x}^{y} ca_i = c\sum_{i=x}^{y} a_i$
- $\displaystyle\sum_{i=x}^{y} (a_i \pm b_i) = \sum_{i=x}^{y} a_i \pm \sum_{i=x}^{y} b_i$
- $\displaystyle\sum_{i=x}^{y} a_i = \sum_{i=x}^{z} a_i + \sum_{i=z+1}^{y} a_i$, com $x \leq z \leq y$

Vamos agora verificar como se obter fórmulas para algumas somas que aparecem com frequência.

Teorema 1.4

Seja c uma constante e x e y inteiros quaisquer, com $x \leq y$. Vale que

$$\sum_{i=x}^{y} c = (y - x + 1)c\,.$$

Demonstração. Note que $\sum_{i=x}^{y} c$ nada mais é do que a soma $c + c + \cdots + c$ com $y - x + 1$ parcelas. □

Teorema 1.5

Seja $n \geq 1$ um inteiro positivo qualquer. Vale que

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

Demonstração. Note que podemos escrever $n(n+1)$ como sendo $(1+n) + (2+(n-1)) + (3+(n-2)) + \cdots + ((n-1)+2) + (n+1)$, ou

$$n(n+1) = \sum_{i=1}^{n} (i + (n - i - 1))\,.$$

Pelas propriedades de somatório, temos

$$\begin{aligned} n(n+1) &= \sum_{i=1}^{n}(i+(n-i+1)) = \sum_{i=1}^{n} i + \sum_{i=1}^{n}(n-i+1) \\ &= \sum_{i=1}^{n} i + \sum_{i=1}^{n} i = 2\sum_{i=1}^{n} i\,, \end{aligned}$$

pois $\sum_{i=1}^{n}(n-i+1) = n+(n-1)+\cdots+2+1 = \sum_{i=1}^{n} i$. Assim, $n(n+1)/2 = \sum_{i=1}^{n} i$. $\square$

Outras somas importantes são as somas dos termos de *progressões aritméticas* e de *progressões geométricas*.

Definição 1.6: *Progressão aritmética*

Uma *progressão aritmética com razão* r é uma sequência de números, $(a_1, a_2, \ldots, a_n)$, que contém um termo inicial a_1 e todos os outros termos a_i, com $2 \leq i \leq n$, são definidos como $a_i = a_1 + (i-1)r$.

A soma dos termos dessa progressão é dada por

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n}(a_1 + (i-1)r)\,.$$

Definição 1.7: *Progressão geométrica*

Uma *progressão geométrica com razão* q é uma sequência de números, $(b_1, b_2, \ldots, b_n)$, que contém um termo inicial b_1 e todos os outros termos b_i, com $2 \leq i \leq n$, são definidos como $b_i = b_1 q^{i-1}$.

A soma dos termos dessa progressão é dada por

$$\sum_{i=1}^{n} b_i = \sum_{i=1}^{n}(b_1 q^{i-1})\,.$$

Teorema 1.8

A soma dos termos de uma progressão aritmética $(a_1, \ldots, a_n)$ com razão r é dada por $\dfrac{(a_1 + a_n)n}{2}$.

Demonstração. Considere a soma $\sum_{i=1}^{n}(a_1 + (i-1)r)$. Temos que

$$\begin{aligned}
\sum_{i=1}^{n}(a_1 + (i-1)r) &= a_1 n + r(1 + 2 + \cdots + (n-1)) \\
&= a_1 n + \frac{rn(n-1)}{2} \\
&= n\left(a_1 + \frac{r(n-1)}{2}\right) \\
&= n\left(\frac{a_1 + a_1 + r(n-1)}{2}\right) \\
&= \frac{n(a_1 + a_n)}{2},
\end{aligned}$$

onde na segunda igualdade utilizamos o Teorema 1.5. □

Teorema 1.9

A soma dos termos de uma progressão geométrica $(b_1, \ldots, b_n)$ com razão q é dada por $\dfrac{b_1(q^n - 1)}{q - 1}$.

Demonstração. Seja S a soma da progressão. Por definição, $S = b_1(1 + q + q^2 + \cdots + q^{n-2} + q^{n-1})$.

Note que ao multiplicar S por q, temos

$$qS = b_1(q + q^2 + q^3 + \cdots + q^{n-1} + q^n).$$

Portanto, subtraindo S de qS obtemos $(q-1)S = b_1(q^n - 1)$, de onde concluímos que

$$S = \frac{b_1(q^n - 1)}{q - 1}.$$

□

Corolário 1.10

Seja $n \geq 0$ um número inteiro e c uma constante qualquer. Vale que

$$\sum_{i=0}^{n} c^i = \frac{1 - c^{n+1}}{1 - c} = \frac{c^{n+1} - 1}{c - 1}.$$

Demonstração. Note que a sequência $(c^0, c^1, c^2, \ldots, c^n)$ é uma progressão geométrica cujo primeiro termo é $c^0 = 1$ e a razão é c. Assim, pelo Teorema 1.9, vale que

$$\sum_{i=0}^{n} c^i = \frac{1(c^{n+1} - 1)}{c - 1} = \frac{c^{n+1} - 1}{c - 1}\,.$$

Ao multiplicar esse resultado por $\frac{-1}{-1} = 1$, temos

$$\frac{c^{n+1} - 1}{c - 1}\frac{-1}{-1} = \frac{1 - c^{n+1}}{1 - c}\,.$$

$\square$

Parte

II

Princípios da análise de algoritmos

"Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to demonstrate that your solution method terminates and does so with the correct answer."

Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 2009.

Nesta parte

Imagine o problema de colocar um conjunto de fichas numeradas em ordem não-decrescente, ordenar um conjunto de cartas de baralho ou selecionar a cédula de maior valor em nossa carteira. Inconscientemente nós fazemos alguma sequência de passos de nossa preferência para resolvê-lo. Por exemplo, para colocar um conjunto de cartas de baralho em ordem não-decrescente há quem prefira olhar todas as cartas e encontrar a menor, depois verificar o restante das cartas e encontrar a próxima menor, e assim por diante. Outras pessoas preferem manter a pilha de cartas sobre a mesa e olhar uma por vez, colocando-a de forma ordenada com relação às cartas que já estão em sua mão. Existem diversas outras maneiras de fazer isso e cada uma delas é realizada por um procedimento que chamamos de *algoritmo*.

Formalmente, um *algoritmo* é uma sequência finita de passos descritos de forma não ambígua que corretamente resolvem um problema. Algoritmos estão presentes na vida das pessoas há muito tempo e são utilizados com frequência para tratar os mais diversos problemas e não apenas para ordenar um conjunto de itens. Por exemplo, também usamos algoritmos para descobrir qual o menor caminho entre dois locais, alocar disciplinas a professores e a salas de aula, controlar a informação de um estoque de mercadorias, etc. Ainda mais básico do que isso, você certamente aprendeu os algoritmos de soma, subtração, multiplicação e divisão de dois números inteiros quando ainda era criança e os utiliza até hoje, mas provavelmente não os chama com esse nome.

No decorrer desse livro, consideraremos que todo algoritmo recebe um conjunto de dados como entrada e devolve um conjunto de dados como saída. Cada problema tem uma descrição de entrada e saída específicos. Uma *instância* do problema é uma atribuição específica de valores válidos como entrada. Dizemos que um algoritmo *resolve* um problema, ou que ele está *correto*, se ele devolve uma solução correta para qualquer instância do problema.

A análise de algoritmos nos permite prever o comportamento ou desempenho de um algoritmo sem que seja necessário implementá-lo em um dispositivo específico. Isso é importante pois, em geral, não existe um único algoritmo que resolve um problema e, por isso, precisamos ter uma forma de comparar diferentes algoritmos para escolher o que melhor se adeque às

nossas necessidades. Além disso, existem várias formas de implementar um mesmo algoritmo (uma linguagem de programação específica ou a escolha de uma boa estrutura de dados pode fazer diferença), e a melhor delas pode ser escolhida antes que se perca tempo implementando todas elas. Estamos interessados, portanto, em entender os detalhes de como ele funciona, bem como em mostrar que, como esperado, o algoritmo funciona corretamente. Verificar se um algoritmo é eficiente é outro aspecto importantíssimo da análise de algoritmos.

Acontece que o comportamento e desempenho de um algoritmo envolve o uso de recursos computacionais como memória, largura de banda e, principalmente, *tempo*. Para descrever o uso desses recursos, levamos em consideração o *tamanho da entrada* e contamos a quantidade de *passos básicos* que são feitos pelo algoritmo. O tamanho da entrada depende muito do problema que está sendo estudado: em vários problemas, como o de ordenação descrito acima, o tamanho é dado pelo número de elementos na entrada; em outros, como o problema de somar dois números, o tamanho é dado pelo número total de bits necessários para representar esses números em notação binária. Com relação a passos básicos, consideraremos operações simples que podem ser feitas pelos processadores comuns atuais, como por exemplo somar, subtrair, multiplicar ou dividir dois números, atribuir um valor a uma variável, ou comparar dois números[1].

Nesta primeira parte do livro veremos um vocabulário básico necessário para projeto e análise de algoritmos em geral, explicando com mais detalhes os aspectos mencionados acima. Para isso, consideraremos um problema simples, de encontrar um certo valor em um dado conjunto de valores, e analisaremos alguns algoritmos que o resolvem. Para facilitar a discussão, vamos supor que esse conjunto de valores está armazenado em um vetor, a mais simples das estruturas de dados.

[1]Estamos falando aqui de números que possam ser representados por 32 ou 64 bits, que são facilmente manipulados por computadores.

Capítulo 2

Um problema simples

Vetores são estruturas de dados simples que armazenam um conjunto de objetos do mesmo tipo de forma contínua na memória. Essa forma de armazenamento permite que o acesso a um elemento do vetor possa ser feito de forma direta, através do *índice* do elemento. Um vetor A que armazena n elementos é representado por $A[1..n]$ ou $A = (a_1, a_2, \ldots, a_n)$ e $A[i] = a_i$ é o elemento que está armazenado na posição i, para todo $1 \leq i \leq n$[1]. Ademais, para quaisquer $1 \leq i < j \leq n$, denotamos por $A[i..j]$ o subvetor de A que contém os elementos $A[i], A[i+1], \ldots, A[j]$.

Problema 2.1: *Busca*

Dado um vetor $A[1..n]$ contendo n números reais e um número real k qualquer, descobrir se k está armazenado em A.

Por simplicidade, assumimos que todas as chaves em A são diferentes. Definimos o problema da busca sobre um vetor que contém apenas números reais, mas poderíamos facilmente supor que o vetor contém registros e assumir que a busca é feita sobre um campo específico dos registros que os diferenciam (por exemplo, se os registros armazenam informações sobre pessoas, pode haver um campo CPF, que é único para cada pessoa). Assim, é comum dizermos que k é uma *chave*.

O algoritmo mais simples para o Problema 2.1 é a *busca linear* e é descrito no Algoritmo 2.1. Ele percorre o vetor, examinando todos os seus elementos, um a um, até encontrar k ou até verificar todos os elementos de A e descobrir que k não está em A.

[1] Algumas linguagens de programação consideram vetores a partir do índice 0, e nós também o faremos em partes futuras do livro. Por enquanto, consideremos que o primeiro índice é o 1.

Algoritmo 2.1: BuscaLinear(A, n, k)

```
1 i = 1
2 enquanto i ≤ n faça
3     se A[i] == k então
4         devolve i
5     i = i + 1
6 devolve −1
```

No que segue, seja n a quantidade de elementos armazenados no vetor A (seu tamanho)[2]. O funcionamento do algoritmo BuscaLinear é bem simples. A variável i indica qual posição do vetor A estamos analisando e inicialmente fazemos $i = 1$. Incrementamos o valor de i em uma unidade sempre que as duas condições do laço **enquanto** forem satisfeitas, i.e., quando $A[i] \neq k$ e $i \leq n$. Assim, o laço enquanto apenas verifica se $A[i]$ é igual a k e se o vetor A já foi totalmente verificado. Caso k seja encontrado, o laço **enquanto** é encerrado e o algoritmo devolve o índice i tal que $A[i] = k$. Caso contrário, o algoritmo devolve -1, indicando que k não se encontra no vetor A. A Figura 2.1 apresenta dois exemplos de execução de BuscaLinear.

Intuitivamente, é fácil acreditar que BuscaLinear resolve o problema da busca, isto é, que *para quaisquer* vetor A de números reais e número real k, o algoritmo irá devolver a posição de k em A caso ela exista, ou irá devolver -1 caso k não esteja em A. Mas como podemos ter certeza que o comportamento de BuscaLinear é sempre como esperamos que seja? Sempre que o algoritmo devolver -1, é verdade que k realmente não existe no vetor? No Capítulo 3 veremos uma forma de *provar* que algoritmos funcionam corretamente. Mas antes, vejamos outro problema de busca em vetores.

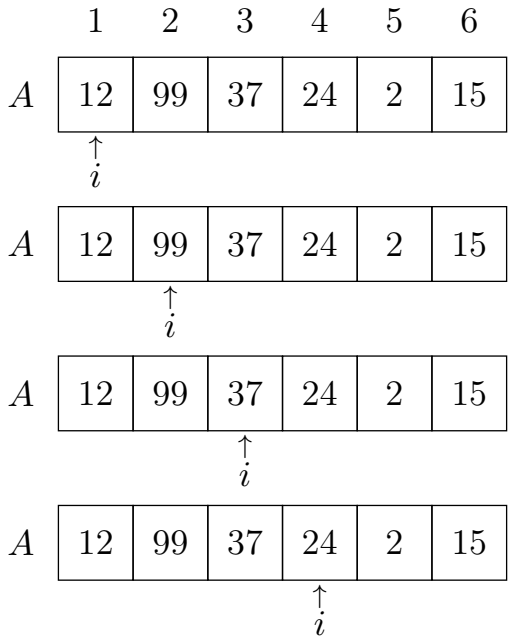
Problema 2.2: ***Busca em dados ordenados***

Dado um vetor $A[1..n]$ contendo n números reais em ordem não-decrescente, i.e., $A[i] \leq A[i+1]$ para todo $1 \leq i < n$, e um número real k qualquer, descobrir se k está armazenado em A.

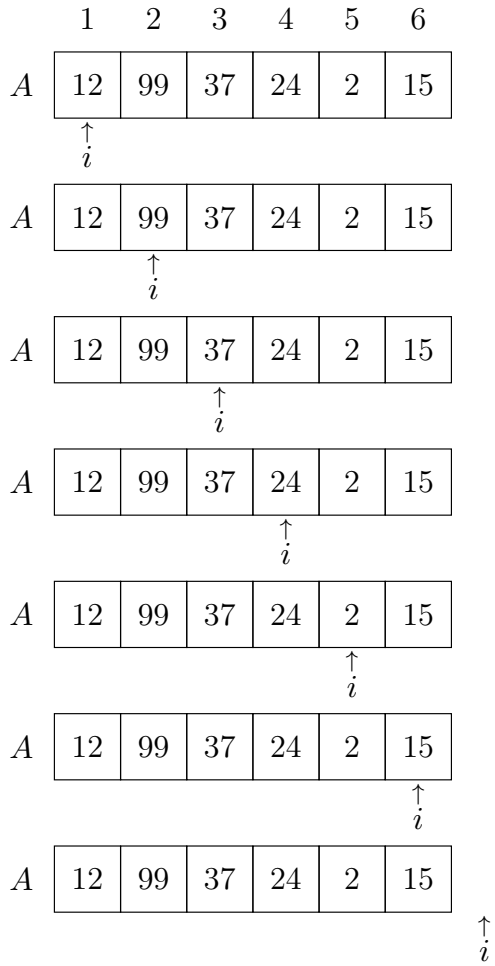
Utilizamos o termo *não-decrescente* em vez de *crescente*, pois pode ser que $A[i] = A[i+1]$, para algum i.

Também de forma intuitiva, é fácil acreditar que o algoritmo BuscaLinear resolve o

[2]Em outros pontos do livro, iremos diferenciar o *tamanho* de um vetor – quantidade de elementos armazenados – de sua *capacidade* – quantidade máxima de elementos que podem ser armazenados.



(a) Execução de BuscaLinear(A, 6, 24).



(b) Execução de BuscaLinear(A, 6, 10).

Figura 2.1: Exemplos de execução de BuscaLinear(A, n, k) (Algoritmo 2.1), com $A = (12, 99, 37, 24, 2, 15)$, $n = 6$ e diferentes valores de k.

problema da busca em dados ordenados. Mas, será que ele é o algoritmo mais *eficiente* para resolver esse problema? Afinal, o fato de o vetor estar em ordem nos dá mais informações. Por exemplo, para qualquer índice válido i, o subvetor $A[1..i-1]$ contém todos os elementos que são menores ou iguais ao elemento $A[i]$ e o subvetor $A[i+1..n]$ contém todos os elementos que são maiores ou iguais ao elemento $A[i]$[3].

Seja i um índice entre 1 e n. Se $k < A[i]$, pelo fato do vetor estar ordenado de forma não-decrescente, certamente não é possível que k esteja armazenado no subvetor $A[i..n]$. Com isso, podemos ter um terceiro critério de parada para a busca linear. Essa nova ideia está formalizada no Algoritmo 2.2. A Figura 2.2 apresenta dois exemplos de execução de BuscaLinearEmOrdem.

Algoritmo 2.2: BuscaLinearEmOrdem(A, n, k)

```
1 i = 1
2 enquanto i ≤ n e k ≥ A[i] faça
3     se A[i] == k então
4         devolve i
5     i = i + 1
6 devolve −1
```

No caso do Problema 2.2, existe ainda um terceiro procedimento, chamado de *busca binária*, que também consegue realizar a busca por uma chave k em um vetor ordenado A com n posições. Na discussão a seguir, por simplicidade, assumimos que n é múltiplo de 2.

A busca binária se aproveita da informação extra de que o vetor está ordenado comparando inicialmente k com o elemento mais ao centro do vetor, $A[n/2]$. Ao realizar essa comparação, existem apenas três possibilidades: $k = A[n/2]$, $k < A[n/2]$ ou $k > A[n/2]$. Se $k = A[n/2]$, então encontramos a chave e a busca está encerrada com sucesso. Caso contrário, se $k < A[n/2]$, então temos a certeza de que, se k estiver em A, então k estará na primeira metade de A, i.e., k pode estar somente em $A[1..n/2-1]$ (isso segue do fato de A estar ordenado). Caso $k > A[n/2]$, então sabemos que, se k estiver em A, então k pode estar somente no vetor $A[n/2+1..n]$. Note que se $k \neq A[n/2]$, então essa estratégia elimina metade do espaço de busca, isto é, antes o elemento k tinha possibilidade de estar em qualquer uma das n posições do vetor e agora basta procurá-lo em no máximo $n/2$ posições.

Agora suponha que, das três possibilidades, temos que $k < A[n/2]$. Note que podemos verificar se k está em $A[1..n/2-1]$ utilizando a mesma estratégia, i.e., comparamos k com o

[3]Note como essa frase é verdadeira mesmo quando $i = 1$, caso em que $A[1..i-1]$ é um vetor vazio, ou quando $i = n$, caso em que $A[i+1..n]$ é vazio.

	1	2	3	4	5	6	7	8	9	10
A	2	8	12	15	24	32	37	45	87	99
	$\uparrow i$									
A	2	8	12	15	24	32	37	45	87	99
		$\uparrow i$								
A	2	8	12	15	24	32	37	45	87	99
			$\uparrow i$							
A	2	8	12	15	24	32	37	45	87	99
				$\uparrow i$						
A	2	8	12	15	24	32	37	45	87	99
					$\uparrow i$					

(a) Execução de BuscaLinearEmOrdem(A, 10, 24).

	1	2	3	4	5	6	7	8	9	10
A	2	8	12	15	24	32	37	45	87	99
	$\uparrow i$									
A	2	8	12	15	24	32	37	45	87	99
		$\uparrow i$								
A	2	8	12	15	24	32	37	45	87	99
			$\uparrow i$							

(b) Execução de BuscaLinearEmOrdem(A, 10, 11).

Figura 2.2: Exemplos de execução de BuscaLinearEmOrdem(A, n, k) (Algoritmo 2.2) com $A = (2, 8, 12, 15, 24, 32, 37, 45, 87, 99)$, $n = 10$ e vários valores de k.

valor que está na metade do vetor $A[1..n/2-1]$, a chave em $A[n/4]$, para poder encontrá-lo ou então verificar se ele estará na primeira ou na segunda metade desse subvetor, dependendo do resultado da comparação. Esse procedimento se repete até que a chave k seja encontrada ou até chegarmos em um subvetor vazio. Estratégias como essa são chamadas de estratégias *recursivas* e são apresentadas no Capítulo 7.

O Algoritmo 2.3 formaliza a ideia da busca binária, que recebe um vetor $A[1..n]$ ordenado de modo não-decrescente e um valor k a ser buscado. Ele devolve a posição em que k está armazenado, se k estiver em A, e devolve -1, caso contrário. As variáveis esq e dir armazenam, respectivamente, as posições inicial e final do espaço de busca, isto é, o elemento k está sendo procurado no subvetor $A[esq..dir]$. Assim, inicialmente $esq = 1$ e $dir = n$. A Figura 2.3 apresenta exemplos de execução de BuscaBinaria.

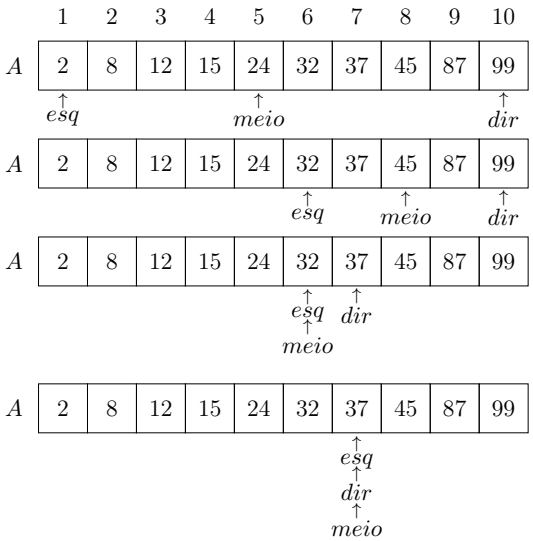
Seja $meio$ uma variável que vai armazenar o índice mais ao centro do vetor $A[esq..dir]$, que é um vetor com $dir - esq + 1$ posições. Se $dir - esq + 1$ é ímpar, então os vetores $A[esq..meio-1]$ e $A[meio+1..dir]$ têm exatamente o mesmo tamanho, i.e., $meio - esq = dir - meio$, o que implica $meio = (esq + dir)/2$. Caso $dir - esq + 1$ seja par, então os tamanhos dos vetores $A[esq..meio-1]$ e $A[meio+1..dir]$ diferem de uma unidade. Digamos que $A[esq..meio-1]$ seja o menor desses vetores, i.e., $meio - esq = dir - meio - 1$, o que implica $meio = (esq + dir - 1)/2 = \lfloor (esq + dir)/2 \rfloor$. Portanto, não importa a paridade do tamanho do vetor $A[esq..dir]$, podemos sempre encontrar o índice do elemento mais ao centro

no vetor fazendo $meio = \lfloor (esq + dir)/2 \rfloor$.

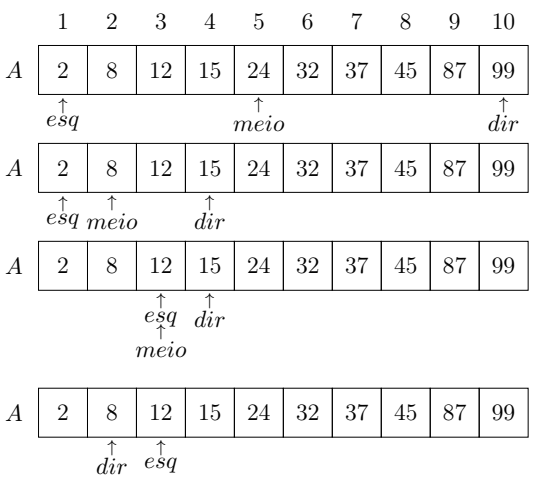
Algoritmo 2.3: BUSCABINARIA(A, n, k)

```
esq = 1
dir = n
enquanto esq ≤ dir faça
    meio = ⌊(esq + dir)/2⌋
    se A[meio] == k então
        devolve meio
    senão se k > A[meio] então
        esq = meio + 1
    senão
        dir = meio − 1
devolve −1
```

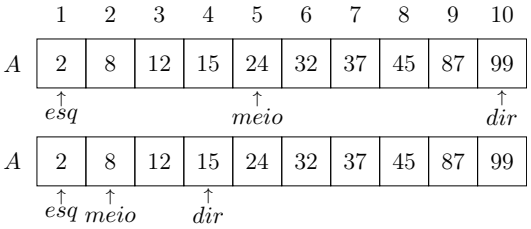
Em um primeiro momento, não é tão intuitivo o fato de que a busca binária resolve corretamente o problema da busca em dados ordenados. Será que para qualquer vetor A recebido, contanto que esteja ordenado de forma não-decrescente, e para qualquer valor k, a busca binária corretamente encontra k se ele está armazenado em A? É possível que o algoritmo devolva -1 mesmo se o elemento k estiver armazenado em A? Ademais, se esses algoritmos funcionarem corretamente, então temos três algoritmos diferentes que resolvem o mesmo problema! Qual deles é o mais eficiente?



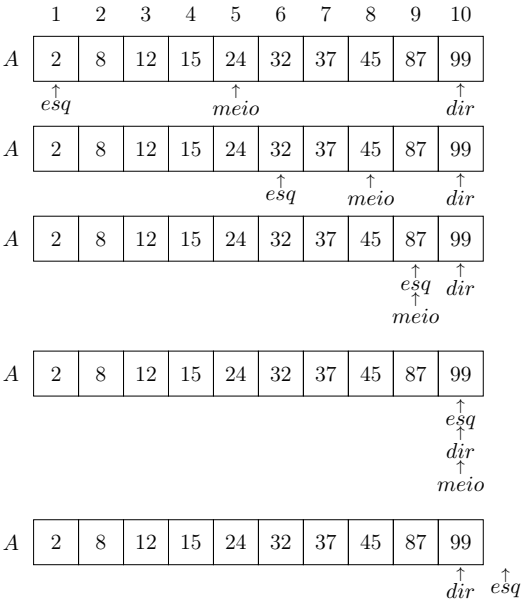
(a) Execução de BuscaBinaria(A, 10, 37).



(c) Execução de BuscaBinaria(A, 10, 11).



(b) Execução de BuscaBinaria(A, 10, 8).



(d) Execução de BuscaBinaria(A, 10, 103).

Figura 2.3: Exemplos de execução de BuscaBinaria(A, n, k) (Algoritmo 2.3) com $A = (2, 8, 12, 15, 24, 32, 37, 45, 87, 99)$, $n = 10$ e diversos valores de k.

Capítulo 3

Corretude de algoritmos iterativos

Ao criar um algoritmo para resolver um determinado problema, esperamos que ele *sempre* dê a resposta correta, qualquer que seja a instância de entrada recebida[1]. Mostrar que o algoritmo está incorreto, portanto, é algo simples: basta encontrar uma instância de entrada específica para a qual o algoritmo devolva uma resposta errada. Mas como mostrar o algoritmo está correto? A seguir veremos uma maneira possível de responder a essa pergunta quando temos algoritmos iterativos (isto é, algoritmos que não são recursivos). De forma geral, mostraremos que o algoritmo possui certas propriedades e que elas continuam verdadeiras após cada iteração de um determinado laço (**para** ou **enquanto**). Ao final do algoritmo essas propriedades devem fornecer uma prova de que o algoritmo foi executado corretamente.

Considere o Algoritmo 3.1, que promete resolver o problema de calcular a soma dos valores armazenados em um vetor A de tamanho n, isto é, promete calcular $\sum_{i=1}^{n} A[i]$.

Algoritmo 3.1: Somatorio(A, n)

```
1 soma = 0
2 para i = 1 até n, incrementando faça
3     soma = soma + A[i]
4 devolve soma
```

Veja que o laço **para** do algoritmo começa fazendo $i = 1$ e, a cada iteração, incrementa o valor de i em uma unidade. Assim, na primeira iteração acessamos $A[1]$, na segunda acessamos $A[2]$ e na terceira acessamos $A[3]$. Quer dizer, logo antes da quarta iteração começar,

[1] Já consideramos que sempre temos uma instância válida para o problema.

já acessamos os valores em $A[1..3]$. Nessa iteração acessaremos apenas o elemento $A[4]$, e mesmo assim podemos dizer que antes da quinta iteração começar, os elementos em $A[1..4]$ já foram acessados. Seguindo com esse raciocínio, é possível perceber que qualquer que seja o valor de i, antes da i-ésima iteração começar os elementos em $A[1..i-1]$ já foram acessados. Nessa iteração acessaremos $A[i]$, fazendo com que os elementos em $A[1..i]$ estejam acessados antes da próxima iteração começar (a $(i+1)$-ésima iteração). Dizemos então que a proposição

$$\text{“antes da } i\text{-ésima iteração começar, os elementos de } A[1..i-1] \text{ já foram acessados”} \tag{3.1}$$

é uma *invariante de laço*. Essa proposição apresenta uma propriedade do algoritmo que *não varia* durante todas as iterações do laço.

E antes da primeira iteração, é verdade que os elementos do subvetor $A[1..0]$ já foram acessados? Esse é um caso especial em que a proposição (3.1) é verdadeira por *vacuidade*. O subvetor $A[1..0]$ nem existe (ou, é vazio), então de fato seus elementos já foram acessados[2].

Note que verificar se a frase vale para quando $i = 1$, depois para quando $i = 2$, depois para quando $i = 3$, etc., não é algo viável. Primeiro porque se o vetor armazena 1000 elementos essa tarefa se torna extremamente tediosa e longa. Mas, pior do que isso, não sabemos o valor de n, de forma que essa tarefa é na verdade impossível! Acontece que não precisamos verificar *explicitamente* que a frase vale para cada valor de i. Basta verificar que ela vale para o primeiro valor e que, dado que ela vale até um certo valor i, então ela valerá para o próximo. Esse é, de forma bem simplista, o *princípio da indução*.

Definição 3.1: *Invariante de laço*

É uma proposição $P(\cdot)$ sobre variáveis $a_1, a_2, \ldots, a_k$ que são importantes para o laço tal que:

(i) se $z_1, z_2, \ldots, z_k$ são os valores de $a_1, a_2, \ldots, a_k$, respectivamente, antes do laço começar, então $P(z_1, z_2, \ldots, z_k)$ é verdadeira;

(ii) se $x_1, x_2, \ldots, x_k$ são os valores de $a_1, a_2, \ldots, a_k$, respectivamente, no início de uma iteração qualquer e $y_1, y_2, \ldots, y_k$ são os valores das mesmas variáveis no início da iteração seguinte, então

se $P(x_1, x_2, \ldots, x_k)$ é verdadeira, então $P(y_1, y_2, \ldots, y_k)$ também é verdadeira.

[2] A proposição “Todos os unicórnios vomitam arcos-íris” é verdadeira por vacuidade, uma vez que o conjunto de unicórnios é vazio.

Uma vez que a proposição "antes da i-ésima iteração começar, os elementos de $A[1..i-1]$ já foram acessados" é de fato uma invariante para o algoritmo Somatorio (ela satisfaz os itens (i) e (ii) da definição acima), então a proposição

$$\text{"antes da } (n+1)\text{-ésima iteração começar, os elementos de } A[1..n] \text{ já foram acessados"} \tag{3.2}$$

também é verdadeira, pois como não há uma $(n+1)$-ésima iteração (o laço não executa e termina quando $i = n+1$), podemos concluir que no momento em que $i = n+1$, o algoritmo acabou de executar a n-ésima iteração do laço. Sabemos então, por (3.2), que todos os elementos do vetor A foram acessados. Apesar de verdadeira, essa informação não é nem um pouco útil para nos ajudar a entender o funcionamento do algoritmo. De fato, muitos algoritmos que recebem um vetor acabam acessando todos os seus elementos, então isso não caracteriza de forma alguma o algoritmo Somatorio.

Para ser útil, uma invariante de laço precisa permitir que após a última iteração do laço possamos concluir que o algoritmo funciona corretamente. Ela precisa, portanto, ter alguma relação com o problema que estamos querendo resolver. No caso do algoritmo Somatorio, precisamos que, ao fim do laço **para**, o valor $\sum_{i=1}^{n} A[i]$ seja calculado. Assim, a invariante a seguir é bem mais útil.

Invariante: *Laço para* – Somatorio

$$P(x) = \text{"Antes da iteração em que } i = x \text{ começar, temos } soma = \sum_{j=1}^{x-1} A[j]\text{."}$$

A frase acima é realmente uma invariante? Quer dizer, o fato de ela aparecer em um quadro de destaque não deveria te convencer disso. Vamos *provar* que ela é uma invariante, mostrando que os itens (i) e (ii) da Definição 3.1 são válidos.

Em primeiro lugar, vamos mostrar que ela vale antes do laço começar. Antes da primeira iteração, também temos $soma = 0$ e $i = 1$, então na verdade queremos verificar se $P(1)$ vale. Como $\sum_{j=1}^{i-1} A[j] = \sum_{j=1}^{0} A[j]$ não está somando nada (o índice inicial do somatório é maior do que o final), então podemos dizer, por vacuidade, que $soma$ é de fato igual a esse valor. Então o item (i) está satisfeito.

Já o item (ii) nos pede para mostrar que a frase continuará verdadeira antes da próxima iteração começar, mas nos permite supor que a frase já era verdadeira no início da iteração. Por causa disso, basta considerar o que a iteração atual irá fazer. Em outras palavras, supondo que $P(i')$ valha para algum valor i', precisamos mostrar que $P(i'+1)$ vale também

(pois se i' é o valor de i em uma iteração qualquer, então $i' + 1$ será seu valor na próxima iteração).

Considere então que o laço começou e seja i' o valor da variável i na iteração atual. Seja $soma'$ o valor da variável *soma* no início desta iteração. Note que o que o algoritmo faz na iteração atual é somar à variável *soma* o valor $A[i']$. Assim, ao fim da iteração atual e, por consequência, antes do início da próxima, o valor da variável *soma* é $soma' + A[i']$. Mas como estamos supondo que $P(i')$ é verdadeira, então sabemos que $soma' = \sum_{j=1}^{i'-1} A[j]$. Logo, o valor na variável *soma* antes da próxima iteração começar é $(\sum_{j=1}^{i'-1} A[j]) + A[i']$, que é justamente $\sum_{j=1}^{i'} A[j]$. Como na próxima iteração o valor da variável i será $i' + 1$, $P(i' + 1)$ é verdadeira e o item (ii) também foi satisfeito.

Agora, sabendo que a frase de fato é uma invariante, ela nos diz que $P(n+1)$ vale, ou seja, ao fim do laço, quando $i = n + 1$, temos $soma = \sum_{j=1}^{i-1} A[j] = \sum_{j=1}^{n} A[j]$, que é justamente o que precisamos para mostrar que o algoritmo está correto.

Analisaremos agora o Algoritmo 3.2, que promete resolver outro problema simples. Ele recebe um vetor $A[1..n]$ e deve devolver o produtório de seus elementos, i.e., $\prod_{i=1}^{n} A[i]$.

Algoritmo 3.2: PRODUTORIO(A, n)

```
1 produto = 1
2 para i = 1 até n, incrementando faça
3     produto = produto · A[i]
4 devolve produto
```

Como podemos definir uma invariante de laço que nos ajude a mostrar a corretude de PRODUTORIO(A, n)? Veja que a cada iteração do laço **para** nós ganhamos mais informação. Precisamos entender como essa informação ajuda a obter a saída desejada do algoritmo. No caso de PRODUTORIO, conseguimos perceber que ao fim da i-ésima iteração (imediatamente antes de iniciar a $(i + 1)$-ésima iteração) temos o produtório dos elementos de $A[1..i]$. Isso é interessante, pois podemos usar esse fato para ajudar no cálculo do produtório dos elementos de $A[1..n]$. De fato, a cada iteração caminhamos um passo no sentido de calcular o produtório desejado. Assim, a seguinte invariante parece uma boa opção para mostrar que PRODUTORIO funciona.

Invariante: *Laço para* – PRODUTORIO

$P(x)$ = "Antes da iteração em que $i = x$ começar, a variável *produto* contém o

produtório dos elementos em $A[1..x-1]$, isto é, temos $produto = \prod_{j=1}^{x-1} A[j]$."

Antes do laço começar, $i = 1$, e veja que $P(1)$ é trivialmente válida, de modo que o item (i) da definição de invariante de laço é válido. Para verificar o item (ii), suponha que $P(i')$ vale, isto é, antes da iteração em que o valor de i é i' começar, a variável *produto* tem valor igual a $\prod_{j=1}^{i'-1} A[j]$. Dentro da iteração atual do laço **para**, em que $i = i'$, vamos fazer

$$produto = produto \cdot A[i'] = \left(\prod_{j=1}^{i'-1} A[j]\right) \cdot A[i'] = \prod_{j=1}^{i'} A[j]\,,$$

o que mostra que $P(i'+1)$ é verdadeira. Como $i'+1$ é o valor de i na iteração posterior à iteração em que $i = i'$, acabamos de confirmar a validade do item (ii).

Note que na última vez que a linha 2 do algoritmo é executada temos $i = n+1$. Assim, o algoritmo não executa a linha 3, e devolve *produto*. Como P é uma invariante, $P(n+1)$ vale e temos que $produto = \prod_{i=1}^{n} A[i]$, que é de fato o resultado desejado. Portanto, o algoritmo funciona corretamente.

Vamos agora analisar o algoritmo BuscaLinear, que promete resolver o problema da busca em vetores visto no Capítulo 2. Perceba que a frase "antes da i-ésima iteração começar, os elementos do subvetor $A[1..i-1]$ já foram acessados" também é uma invariante de laço para esse algoritmo. Acontece que, novamente, ela só nos diz que os elementos foram acessados. No caso da busca linear, precisamos mostrar que se a chave buscada k está no vetor, então o algoritmo devolve um índice i entre 1 e n correspondente à posição em que a chave se encontra. Por outro lado, se k não está no vetor, o algoritmo deve devolver -1.

Teorema 3.2

Seja $A[1..n]$ um vetor de números reais e seja k um número real. O algoritmo BuscaLinear(A, n, k) devolve i tal que $A[i] = k$ se k está em A, ou devolve -1 caso contrário.

Demonstração. Considere primeiro o caso em que k está no vetor A, onde chamamos de ℓ o índice de k em A, i.e., $A[\ell] = k$. Note que certamente a ℓ-ésima iteração do laço é executada. De fato, caso isso não acontecesse, significaria que o algoritmo executou a linha 4 em uma j-ésima iteração, onde $j < \ell$. Assim, isso implica que $A[j] = k$, um absurdo, pois sabemos que $A[\ell] = k$ (estamos assumindo que todas as chaves são diferentes). Portanto, podemos assumir que a ℓ-ésima iteração do laço foi executada. Nesse caso, a linha 3 vai verificar que

$A[\ell] = k$ e devolve ℓ, como queríamos mostrar.

De agora em diante assuma que k *não* está no vetor A. Vamos mostrar que a seguinte proposição é uma invariante de laço para esse algoritmo.

Invariante: *Laço enquanto* – BuscaLinear

$P(x) =$ "Antes da iteração em que $i = x$ começar, o vetor $A[1..x-1]$ não contém k."

Note que P satisfaz o item (i) da definição de invariante (Definição 3.1), pois antes da primeira iteração temos $i = 1$, e nesse caso $P(1)$ trata do vetor $A[1..0]$, que é vazio, e, logo, não pode conter k. Para verificar o item (ii), considere o momento em que o algoritmo vai iniciar a iteração em que o valor da variável i é i'. Assuma que $P(i')$ é válida, isto é, que neste momento o vetor $A[1..i'-1]$ não contém k. Como k não está no vetor, temos $A[i'] \neq k$, de forma que a linha 4 não será executada e a iteração atual irá terminar normalmente. O fato de $A[i'] \neq k$ juntamente com o fato de que $k \notin A[1..i'-1]$, implica que $k \notin A[1..i']$. Assim, $P(i'+1)$ é verdadeira e concluímos que a frase acima é de fato uma invariante, pois $i'+1$ é o valor de i na iteração seguinte.

Precisamos agora utilizar a invariante para concluir que o algoritmo funciona corretamente, i.e., no caso em que estamos considerando, onde k não está em A, o algoritmo deve devolver -1. Note que o algoritmo irá executar todas as n iterações do laço, pois caso contrário ele teria devolvido algum índice $i \leq n$ na linha 4. Por isso, sabemos que chegamos em $i = n+1$. Pela invariante de laço, temos que $k \notin A[1..i-1]$, i.e., $k \notin A[1..n]$. Na última linha o algoritmo devolve -1, que era o desejado no caso em que k não está em A. □

Perceba que na prova anterior não fizemos nenhuma suposição sobre os dados contidos em A ou sobre o valor de k. Portanto, não resta dúvidas de que o algoritmo funciona corretamente para qualquer instância.

À primeira vista, todo o processo que fizemos para mostrar que os algoritmos Somatorio, Produtorio e BuscaLinear funcionam corretamente pode parecer excessivamente complicado. Porém, essa impressão vem do fato desses algoritmos serem muito simples (assim, a análise de algo simples parece ser desnecessariamente longa). Veja o próximo exemplo.

Considere o seguinte problema.

Problema 3.3: *Conversão de base*

Dado um inteiro n escrito em base decimal, encontrar sua representação binária.

Por exemplo, se $n = 2$ queremos encontrar 1, se $n = 10$ queremos encontrar 1010, e

se $n = 327$ queremos encontrar 101000111. Esse é um problema clássico que aparece nos primeiros cursos de programação. O Algoritmo 3.3, também clássico, promete resolvê-lo.

Algoritmo 3.3: ConverteBase(n)

```
seja B[1..n] um vetor
i = 1
t = n
enquanto t > 0 faça
    B[i] = resto da divisão de t por 2
    i = i + 1
    t = ⌊t/2⌋
devolve B
```

O funcionamento desse algoritmo é bem simples. Para evitar confusão, manteremos o valor da variável n e por isso apresentamos uma nova variável t, que inicialmente é igual a n. Um vetor B será utilizado de tal forma que $B[1..j]$ deverá representar o inteiro $\sum_{k=1}^{j} 2^{k-1}B[k]$. Note que a cada iteração, o algoritmo coloca um valor (0 ou 1) na i-ésima posição de B. Esse valor é o resto da divisão por 2 do valor atual em t e em seguida t é atualizado para aproximadamente metade do valor atual. Se r é a quantidade de iterações do laço, resta provar que $B[1..r]$ terá a representação binária de n.

Mostraremos agora que a seguinte frase é uma invariante para esse algoritmo.

Invariante: *Laço enquanto* – ConverteBase

$P(x) =$ "Antes da iteração em que $i = x$ começar, o vetor $B[1..x-1]$ representa um inteiro m tal que $n = m + t2^{x-1}$."

Se essa é realmente uma invariante de ConverteBase, então $P(r+1)$ nos diz que $B[1..r]$ representa um inteiro m tal que $n = m$, ou seja, de fato ela é útil para provar a corretude do algoritmo.

Primeiro precisamos provar que $P(1)$ vale. Antes da primeira iteração temos $t = n$ e, de fato, $B[1..i-1] = B[1..0]$ é vazio, representa $m = 0$ e $n = 0 + n2^0 = n$. Assim, o primeiro item da definição de invariante está satisfeito.

Agora precisamos mostrar que se vale $P(i')$, então valerá $P(i'+1)$, para qualquer iteração em que $i = i'$. Supondo que $P(i')$ vale, então sabemos que $B[1..i'-1]$ representa um inteiro m' tal que $n = m' + t'2^{i'-1}$, onde t' é o valor de t no início dessa iteração. Então $m' = \sum_{j=1}^{i'-1} 2^{j-1}B[j]$. Veja que precisamos mostrar que $B[1..i']$ vai representar um inteiro z

tal que $n = z + \lfloor t'/2 \rfloor 2^{i'}$.

Nessa iteração, fazemos $B[i']$ receber o resto da divisão de t' por 2 e t receber $\lfloor t'/2 \rfloor$. Agora, $B[1..i']$ representa o inteiro $m' + 2^{i'-1}B[i']$. Temos dois casos. Se t' é par, então $B[i'] = 0$ e $B[1..i']$ ainda representa m'. Como $n = m' + t'2^{i'-1} = m' + (t'/2)2^{i'}$, temos que $P(i'+1)$ vale. Se t' é ímpar, então $B[i'] = 1$ e $B[1..i']$ representa $m' + 2^{i'-1}$. Como $n = m' + t'2^{i'-1} = m' + (t'+1-1)2^{i'-1} = (m' + 2^{i'-1}) + ((t'-1)/2)2^{i'} = (m' + 2^{i'-1}) + \lfloor t'/2 \rfloor 2^{i'}$, temos que $P(i'-1)$ vale.

Veremos ainda outros casos onde a corretude de um dado algoritmo não é tão clara, de modo que a necessidade de se utilizar invariantes de laço é evidente.

Novamente, perceba que mostrar que uma invariante se mantém durante a execução de um algoritmo nada mais é que uma prova por indução na quantidade de iterações de um dado laço. Um bom exercício é provar a corretude de BuscaBinaria utilizando invariante de laço. A frase "antes da iteração em que $esq = x$ e $dir = y$ começar, $A[1..x-1]$ e $A[y+1..n]$ não contêm k" pode ser útil.

Capítulo 4

Tempo de execução

Uma vez que implementamos um algoritmo, é simples verificar seu tempo de execução. Basta fornecer uma entrada a ele, e calcular o tempo entre o início e o fim de sua execução. Mas veja que vários fatores afetam esse tempo. Ele será mais rápido quando executado em um computador mais potente do que em um menos potente. Também será mais rápido se a instância de entrada for pequena, ao contrário de instâncias grandes. O sistema operacional utilizado, a linguagem de programação utilizada, a velocidade do processador, o modo como o algoritmo foi implementado ou a estrutura de dados utilizada influenciam diretamente o tempo de execução de um algoritmo. Assim, queremos um conceito de tempo que seja independente de detalhes da instância de entrada, da plataforma utilizada e que possa ser de alguma forma quantificado concretamente. O motivo para isso é que quando temos vários algoritmos para o mesmo problema, gostaríamos de poder escolher um dentre eles sem que seja necessário gastar todo o tempo de implementação, correção de erros e testes. Afinal, um algoritmo pode estar correto e sua implementação não, porque algum erro não relacionado com o problema pode ser inserido nesse processo. Esses detalhes de implementação não são nosso foco, que é resolver problemas.

Mesmo assim, é importante assumir algum meio de implementação, que possa representar os processadores reais. Consideraremos um modelo de computação que possui algumas *operações primitivas* ou *passos básicos* que podem ser realizados rapidamente sobre número pequenos[1]: operações aritméticas (soma, subtração, multiplicação, divisão, resto, piso, teto), operações relacionais (maior, menor, igual), operações lógicas (conjunção, disjunção, negação), movimentação em variáveis simples (atribuição, cópia) e operações de controle (condi-

[1] Um número x pode ser armazenado em $\log x$ bits. Em geral, estamos falando de 32 ou 64 bits.

cional, chamada a função[2], retorno).

Definição 4.1: *Tempo de execução*

O *tempo de execução* de um algoritmo é dado pela quantidade de passos básicos executados por ele sobre uma certa instância de entrada.

Em geral, o tempo de execução de um algoritmo cresce junto com a quantidade de dados passados na entrada. Portanto, escrevemos o tempo de execução como uma função T sobre o *tamanho da entrada*, de forma que $T(n)$ é a quantidade de passos básicos realizados pelo algoritmo quando recebe uma entrada de tamanho n. O tamanho da entrada é um fator que independe de detalhes de implementação e, por isso, o tempo de execução definido dessa forma nos possibilita obter uma boa estimativa do quão rápido um algoritmo é. Em geral, ele reflete justamente "o que pode crescer". Por exemplo, no problema de busca por um elemento em um vetor, em geral o que cresce e torna o problema mais difícil de ser resolvido é a quantidade de elementos armazenados no vetor. Nesses casos, o tamanho do vetor é considerado o tamanho da entrada. Por outro lado, quando falamos do problema de multiplicar dois números inteiros, o que cresce e torna o problema difícil é a quantidade de dígitos em cada número. Nesses casos, a quantidade de bits necessários para representar os números é o tamanho da entrada.

Como exemplo concreto, considere novamente o Algoritmo 3.1, Somatorio, que, dado um vetor A com n elementos, devolve $\sum_{i=1}^{n} A[i]$. Para esse problema de somar os números de um vetor, o tamanho da entrada é n, que é a quantidade de elementos que são recebidos (tamanho do vetor). Os passos básicos existentes nesse algoritmo são: quatro atribuições de valores a variáveis ($soma = 0$, $i = 1$, $soma = soma + A[i]$, $i = i + 1$ — do laço **para**), uma operação lógica ($i \leq n$ — do laço **para**), duas operações aritméticas sobre números pequenos ($i + 1$ — do laço **para**, $soma + A[i]$), uma operação de acesso a vetor ($A[i]$) e uma operação de retorno. Uma vez que esse algoritmo é implementado, cada operação dessas leva um certo tempo para ser executada. Um tempo muito pequeno, certamente, mas algum tempo. Vamos considerar que cada uma delas leva t unidades de tempo, em média, para ser executada (algumas podem levar mais tempo do que outras, mas podemos imaginar que essa diferença é pequena demais para ser levada em conta). Assim, uma única execução do corpo do laço **para** leva tempo $3t$ (uma atribuição, um acesso a vetor e uma soma). Como o corpo do laço é executado apenas quando i tem um valor entre 1 e n e i é incrementado de 1 a cada iteração, temos que o corpo do laço executa n vezes, levando tempo total $3tn$. Pelo mesmo motivo, n também é a quantidade de vezes que a operação de incremento de i é feita. O teste do laço **para**, por outro lado, é executado $n + 1$ vezes (uma para cada valor

[2] A chamada à função é executada rapidamente, mas a função em si pode demorar bem mais.

válido de i e uma para quando $i = n + 1$ e o teste $i \leq n$ falha). Resumindo, o tempo $T(n)$ de execução de SOMATORIO é

$$T(n) = \underbrace{t}_{soma=0} + \underbrace{t}_{i=1} + \underbrace{t(n+1)}_{i \leq n} + \underbrace{2tn}_{i=i+1} + \underbrace{3tn}_{soma=soma+A[i]} + \underbrace{t}_{\textbf{devolve } soma} = 6tn + 4t\,.$$

Considere agora o problema de encontrar a soma dos elementos pares de um vetor. Ele é resolvido pelo Algoritmo 4.1, que é bem pouco diferente do algoritmo SOMATORIO.

Algoritmo 4.1: SOMATORIOPAR(A, n)

```
1 soma = 0
2 para i = 1 até n, incrementando faça
3     se A[i] é par então
4         soma = soma + A[i]
5 devolve soma
```

Agora temos um teste extra sendo feito, para verificar se o elemento em $A[i]$ é par e, apenas se for, considerá-lo na soma. Assim, a linha 3 sempre é executada enquanto que a linha 4 nem sempre é. A quantidade de vezes que ela será executada irá depender do conteúdo de A, mas com certeza será algo entre 0 e n execuções (0 se A não tiver nenhum elemento par, 1 se tiver um único, e assim por diante, sendo que A pode ter no máximo n elementos pares — todos eles). Se A não tem nenhum número par, então o tempo de execução é

$$\underbrace{t}_{soma=0} + \underbrace{t}_{i=1} + \underbrace{t(n+1)}_{i \leq n} + \underbrace{2tn}_{i=i+1} + \underbrace{3tn}_{A[i]\ par} + \underbrace{t}_{\textbf{devolve } soma} = 6tn + 4t\,.$$

Se todos os números de A são pares, então o tempo de execução é

$$\underbrace{t}_{soma=0} + \underbrace{t}_{i=1} + \underbrace{t(n+1)}_{i \leq n} + \underbrace{2tn}_{i=i+1} + \underbrace{3tn}_{A[i]\ par} + \underbrace{3tn}_{soma=soma+A[i]} + \underbrace{t}_{\textbf{devolve } soma} = 9tn + 4t\,.$$

Consideramos que a operação de teste para verificar se $A[i]$ é par contém 3 operações: acesso ao vetor, divisão por 2 e comparação do resto da divisão com 0. Não estamos considerando as operações de desvio de instrução (se o teste é falso, a instrução seguinte não pode ser executada e o fluxo do algoritmo é desviado). Assim, se $T(n)$ é o tempo de execução de SOMATORIOPAR sobre qualquer vetor de tamanho n, certamente vale que

$$6tn + 4t \leq T(n) \leq 9tn + 4t\,.$$

Uma vez que conseguimos descrever o tempo de execução dos algoritmos como funções sobre o tamanho da entrada, podemos começar a comparar um algoritmo com outros que resolvem o mesmo problema (e, assim, recebem as mesmas instâncias de entrada) por meio da *ordem de crescimento* dessas funções. Por exemplo, a função $f(x) = x$ cresce mais devagar do que $g(x) = x^2$ e mais rápido do que $h(x) = \log x$. É importante, portanto, entender quais funções crescem mais rápido e conseguir comparar duas funções para saber qual delas é mais rápida do que a outra. As funções c, $\log n$, n^c, $n^c \log n$, 2^n e $n!$, onde c é uma constante, são as mais comuns em análise de algoritmos. Veja as Figuras 4.1 e 4.2.

Para entender melhor como fazer essas comparações, vamos analisar os algoritmos BuscaLinear, BuscaLinearEmOrdem e BuscaBinaria vistos no Capítulo 2. Lembre-se que BuscaLinear resolve o problema da busca em vetores (Problema 2.1) e todos os três, BuscaLinear, BuscaLinearEmOrdem e BuscaBinaria, resolvem o problema da busca em vetores já ordenados (Problema 2.2).

Vamos novamente assumir que um passo básico leva tempo t para ser executado. Por comodidade, repetimos o algoritmo BuscaLinear no Algoritmo 4.2.

Algoritmo 4.2: BuscaLinear(A, n, x)

```
1 i = 1
2 enquanto i ≤ n faça
3     se A[i] == x então
4         devolve i
5     i = i + 1
6 devolve −1
```

Considere inicialmente que o elemento x está no vetor $A[1..n]$. Assim, denote por p_x sua posição em A, isto é, a posição tal que $A[p_x] = x$. Note que a linha 1 é executada somente uma vez, assim como a linha 4 (dado que o algoritmo encerra quando devolve um valor). Já o teste do laço **enquanto** na linha 2 é executado p_x vezes, a linha 3 é executada p_x vezes e a linha 5 é executada $p_x - 1$ vezes. Assim, o tempo de execução total $T_{BLE}(n)$ de BuscaLinear(A, n, x) quando x está em A é

$$T_{BLE}(n) = t + tp_x + 2tp_x + 2t(p_x - 1) + t = 5tp_x\,. \tag{4.1}$$

O tempo de execução, portanto, depende de onde x se encontra no vetor A. Se x está na última posição de A, então $T_{BLE}(n) = 5tn$. Se x está na primeira posição de A, então temos $T_{BLE}(n) = 5t$.

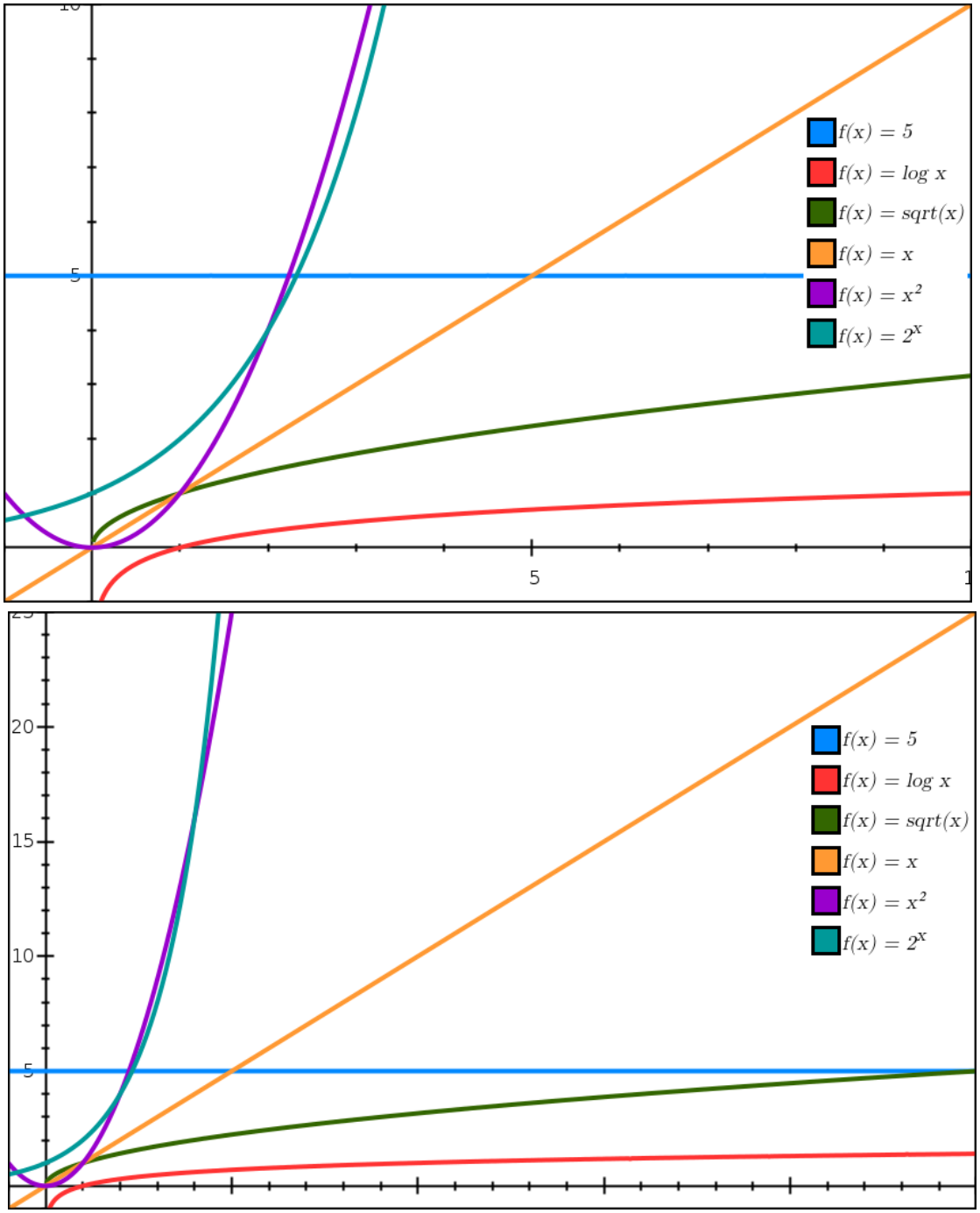


Figura 4.1: Comportamento das funções mais comuns em análise de algoritmos. Observe que na primeira figura, pode parecer que 2^x é menor do que x^2 ou que 5 é maior do que $\sqrt{x}$.

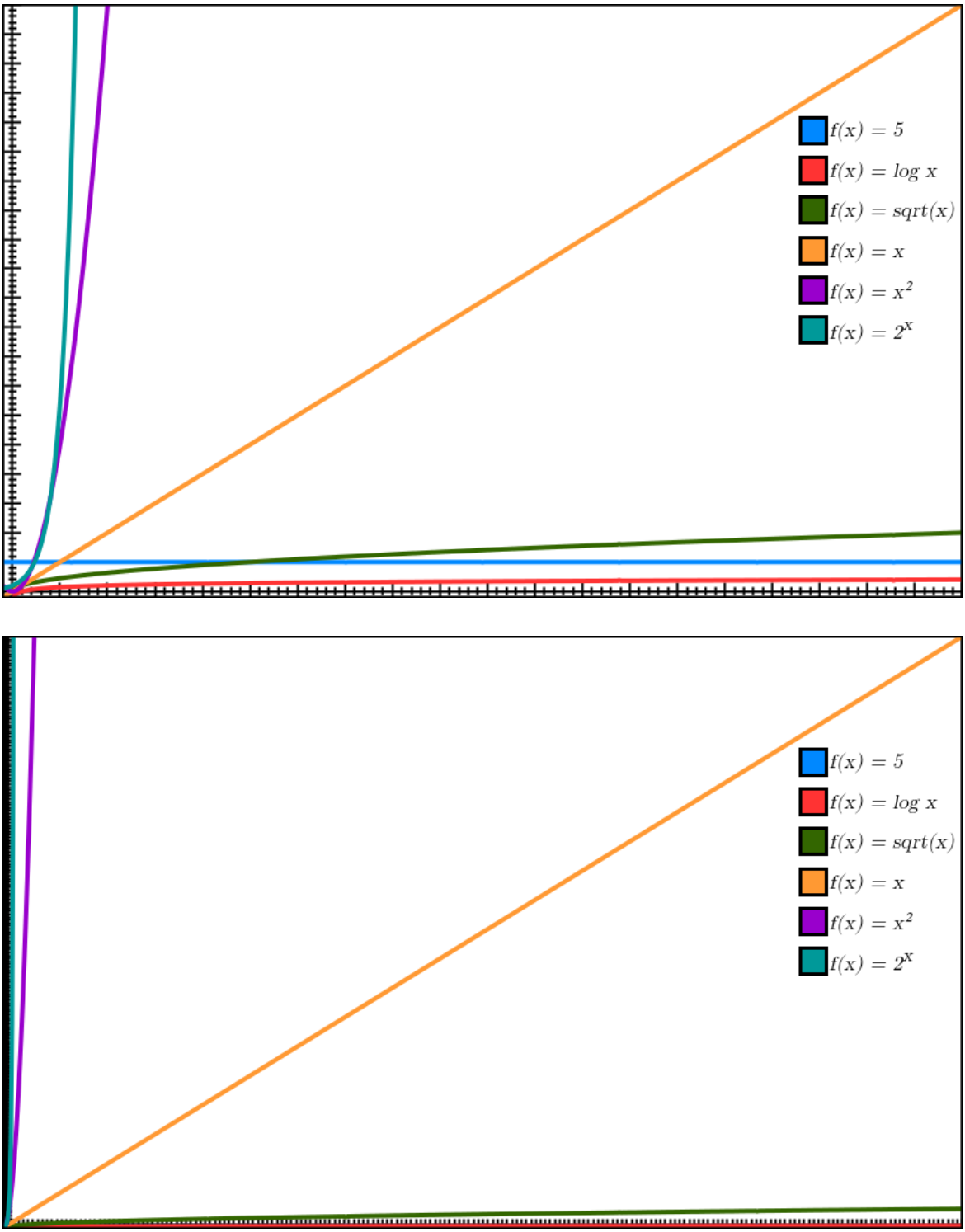


Figura 4.2: Comportamento das funções mais comuns em análise de algoritmos conforme o valor de x cresce. Note como 2^x é claramente a maior das funções observadas enquanto que 5 desaparece completamente.

Agora considere que x não está no vetor $A[1..n]$. Nesse caso, temos que a linha 1 é executada somente uma vez, assim como a linha 6. O teste do laço **enquanto** na linha 2 é executado $n+1$ vezes, a linha 3 é executada n vezes e a linha 5 é executada n vezes. Assim, o tempo de execução total $T_{BLN}(n)$ de BuscaLinear(A, n, x) quando x não está em A é

$$T_{BLN}(n) = t + t(n+1) + 2tn + 2tn + t = 5tn + 3t\,. \tag{4.2}$$

Note que como $1 \leq p_x \leq n$, temos que $5t \leq T_{BLE}(n) \leq 5tn$. E como $5tn \leq 5tn + 3t$, podemos ainda dizer que para o tempo de execução $T_{BL}(n)$ de BuscaLinear(A, n, x) vale que

$$5t \leq T_{BL}(n) \leq 5tn + 3t\,. \tag{4.3}$$

Perceba que toda análise feita acima também vale se o vetor A estiver ordenado. Vamos agora analisar o algoritmo BuscaLinearEmOrdem. Lembre-se que nele assumimos que o vetor está ordenado de modo não-decrescente. Por comodidade, o repetimos no Algoritmo 4.3.

Algoritmo 4.3: BuscaLinearEmOrdem(A, n, x)

```
1 i = 1
2 enquanto i ≤ n e x ≥ A[i] faça
3     se A[i] == x então
4         devolve i
5     i = i + 1
6 devolve −1
```

A diferença entre BuscaLinear e BuscaLinearEmOrdem é de um teste extra que é feito no laço **enquanto**. Para analisar esse algoritmo, vamos novamente assumir inicialmente que x está em A. Assim, denote também por p_x sua posição em A, isto é, a posição tal que $A[p_x] = x$. Perceba que os testes de parada do laço **enquanto** e o teste da linha 3 são executados p_x vezes cada enquanto que o incremento da variável i na linha 5 é executado $p_x - 1$ vezes. Assim, o tempo de execução total $T_{BOE}(n)$ de BuscaLinearEmOrdem(A, n, x) quando x está em A é

$$T_{BOE}(n) = t + 4tp_x + 2tp_x + 2t(p_x - 1) + t = 8tp_x\,. \tag{4.4}$$

Esse tempo depende, portanto, de onde x se encontra em A. Se x está na primeira posição, então $T_{BOE}(n) = 8t$, e se está na última posição, então $T_{BOE}(n) = 8tn$.

Agora suponha que x não está no vetor A. Vamos definir a_x como sendo a posição do

maior elemento contido em A que é estritamente menor do que x, onde fazemos $a_x = 0$ se x for menor do que todos os elementos em A. Isso significa que, como A está em ordem não-decrescente, se x estivesse presente em A, então ele deveria estar na posição $a_x + 1$. Com essa nomenclatura, podemos perceber que os testes de parada do laço **enquanto** é executado $a_x + 1$ vezes, enquanto o teste da linha 3 e o incremento de i são executados a_x vezes cada. Assim, o tempo total de execução $T_{BON}(n)$ de BuscaLinearEmOrdem(A, n, x) quando x não está em A é

$$T_{BON}(n) = t + 4t(a_x + 1) + 2ta_x + 2ta_x + t = 8ta_x + 6t\,. \tag{4.5}$$

Agora, o tempo de execução depende de onde x deveria se encontrar em A.

Note que como $1 \leq p_x \leq n$, temos que $8t \leq T_{BOE}(n) \leq 8tn$. Como $0 \leq a_x \leq n$, temos que $6t \leq T_{BON}(n) \leq 8tn + 6t$. Assim, podemos dizer que para o tempo de execução $T_{BO}(n)$ de BuscaLinearEmOrdem(A, n, x) vale que

$$6t \leq T_{BO}(n) \leq 8tn + 6t\,. \tag{4.6}$$

Para a busca binária, vamos fazer uma análise semelhante. Novamente por comodidade, repetimos o algoritmo BuscaBinaria no Algoritmo 4.4.

Algoritmo 4.4: BuscaBinaria(A, n, x)

```
1  esq = 1
2  dir = n
3  enquanto esq ≤ dir faça
4      meio = ⌊(esq + dir)/2⌋
5      se A[meio] == x então
6          devolve meio
7      senão se x > A[meio] então
8          esq = meio + 1
9      senão
10         dir = meio − 1
11 devolve −1
```

Inicialmente assuma que x está em A e denote por r_x a quantidade de vezes que o teste do laço **enquanto** na linha 3 é executado (note que isso depende de onde x está em A). As linhas 1 e 2 são executadas uma vez cada, assim como a linhas 6. A linha 4 é executada r_x

vezes (e ela tem 4 operações básicas), a linha 5 é executada r_x vezes (mas note que o teste só será verdadeiro uma única vez), a linha 7 é executada $r_x - 1$ vezes, e as linhas 8 e 10 são executadas um total de no máximo $r_x - 1$ vezes (quando o teste da linha 5 falha, certamente uma das duas é executada). Assim, o tempo de execução $T_{BBE}(n)$ de BuscaBinaria(A, n, x) quando x está em A é

$$T_{BBE}(n) = 2t + tr_x + 4tr_x + 2tr_x + 2t(r_x - 1) + 2t(r_x - 1) + t = 11tr_x - t\,. \tag{4.7}$$

Assim como nos dois algoritmos de busca linear anteriores, o tempo de execução depende de onde x se encontra no vetor A. Note que o algoritmo de busca binária sempre descarta metade do vetor que está sendo considerado, diminuindo o tamanho do vetor analisado pela metade, até que encontre x. Como sempre metade do vetor é descartado, o algoritmo analisa, nessa ordem, vetores de tamanho n, $n/2$, $n/2^2$, ..., $n/2^i$, onde o último vetor analisado *pode chegar a ter tamanho* 1, caso em que $n/2^i = 1$, o que implica $i = \log n$. Assim, o teste do laço **enquanto** é executado entre 1 e $\log n$ vezes quando x está em A, de modo que temos $1 \leq r_x \leq \log n$. Com isso, vale que $10t \leq T_{BBE}(n) \leq 11t \log n - t$.

Agora considere que x não está em A. Conforme mencionado acima, o algoritmo irá sempre descartar metade do vetor que está sendo considerado, diminuindo o tamanho do vetor pela metade até descobrir que x não está em A. Nesse caso, o teste do laço **enquanto** é realizado $\log n + 1$ vezes, de modo que o tempo de execução $T_{BBN}(n)$ de BuscaBinaria(A, n, x) quando x não está em A é

$$T_{BBN}(n) = 2t + t(\log n + 1) + 4t \log n + 2t \log n + 2t \log n + 2t \log n + t = 11t \log n + 4t\,. \tag{4.8}$$

Assim, podemos dizer que para o tempo de execução $T_{BB}(n)$ de BuscaBinaria(A, n, x) vale que

$$10t \leq T_{BB}(n) \leq 11t \log n + 4t\,. \tag{4.9}$$

Agora temos o tempo de execução de três algoritmos que resolvem o Problema 2.2, da busca por um elemento x em um vetor em ordem não-decrescente A, resumidos a seguir:

	BuscaLinear	BuscaLinearEmOrdem	BuscaBinaria
$x \in A$	$5tp_x$	$8tp_x$	$11tr_x - t$
$x = A[1]$	$5t$	$8t$	$11t \log n - t$
$x = A[n/2]$	$\frac{5}{2}tn$	$4tn$	$10t$
$x = A[n]$	$5tn$	$8tn$	$11t \log n - t$
$x \notin A$	$5tn + 3t$	$8ta_x + 6t$	$11t \log n + 4t$

4.1 Análise de melhor caso, pior caso e caso médio

Perceba que, na análise de tempo que fizemos para os algoritmos de busca linear e binária, mesmo considerando instâncias de um mesmo tamanho n, o tempo de execução dependia de *qual* instância era dada (x está ou não em A, está em qual posição de A).

Definição 4.2: *Tempo de melhor caso*

O tempo de execução de *melhor caso* de um algoritmo é o tempo de execução de uma instância que executa de forma mais rápida, dentre todas as instâncias possíveis de um dado tamanho n.

No caso da BuscaLinear, é necessário executar a inicialização da variável i, o teste do laço **enquanto** e o teste do comando condicional *obrigatoriamente*. Agora, se esse teste ($A[i] == x$) for verdadeiro, a busca termina sua execução com o comando de retorno. Esse certamente é o menor tempo de execução desse algoritmo. E note que isso acontece quando $x = A[1]$. Dizemos então que "o melhor caso de BuscaLinear ocorre quando x está na primeira posição do vetor". Como o tempo de execução de BuscaLinear quando x está em A é $T_{BLE}(n) = 5tp_x$ (veja (4.1)), onde p_x é a posição de x em A, temos que, no melhor caso, o tempo de execução $T_{BL}(n)$ da busca linear é $T_{BL}(n) = 5t$.

No caso da BuscaLinearEmOrdem, também é necessário executar a inicialização da variável i e os testes do laço **enquanto** obrigatoriamente. Acontece que esse teste pode falhar se $x < A[1]$, fazendo o algoritmo retornar -1. Esse certamente é o menor tempo de execução desse algoritmo, que ocorre quando x não está em A e é menor do que todos os elementos já armazenados. Como o tempo de execução de BuscaLinearEmOrdem quando x não está em A é dado por $T_{BON}(n) = 8ta_x + 6t$, onde a_x é a posição do maior elemento que é menor do que x, então no melhor caso o tempo de execução $T_{BO}(n)$ da busca linear em ordem é $T_{BO}(n) = 6t$ ($a_x = 0$).

Já no caso da BuscaBinaria, a inicialização das variáveis, o primeiro teste do laço **enquanto**, o cálculo do valor *meio* e o teste se $A[meio] == x$ devem ser obrigatoriamente realizados. Caso esse último teste dê verdadeiro, o algoritmo para, e esse é o mínimo de operações que ele irá realizar. Isso só ocorre se x estiver exatamente na posição da metade do vetor A, i.e., $A\big[\lfloor (n-1)/2 \rfloor\big] = x$. Assim, como o tempo de execução de BuscaBinaria quando x está em A é dado por $T_{BBE}(n) = 11tr_x - t$, onde r_x é o número de vezes que o teste do laço é executado, temos que no melhor caso da busca binária o tempo de execução $T_{BB}(n)$ é $T_{BB}(n) = 10t$.

O tempo de execução de melhor caso de um algoritmo nos dá a garantia de que, qualquer

que seja a instância de entrada recebida, pelo menos tal tempo será necessário. Assim, como o tempo de execução $T_{BL}(n)$ de BuscaLinear no melhor caso é $T_{BL}(n) = 5t$, podemos dizer que o tempo de execução de BuscaLinear é $T_{BL}(n) \geq 5t$. Perceba aqui a diferença no uso da igualdade e da desigualdade de acordo com as palavras que as precedem.

Geralmente, no entanto, estamos interessados no tempo de execução de *pior caso*.

Definição 4.3: *Tempo de pior caso*

O tempo de execução de *pior caso* de um algoritmo é o maior tempo de execução do algoritmo dentre todas as instâncias de entrada possíveis de um dado tamanho n.

Perceba que o pior caso de BuscaLinear e de BuscaLinearEmOrdem ocorre quando o laço **enquanto** executa o máximo de vezes que puder executar. Para isso acontecer na BuscaLinear, o teste do condicional precisa falhar sempre, isto é, $A[i] \neq x$ sempre, de forma que o laço termina apenas porque i ficou maior do que n (a condição do laço falhou). Isso acontece quando o elemento x a ser buscado não se encontra no vetor A. No caso da BuscaLinearEmOrdem, o teste do condicional também precisa falhar sempre, bem como o segundo teste de condição de parada do próprio laço precisa ser verdadeiro sempre. Assim, o laço termina apenas quando $i > n$, nos garantindo ainda que x é maior do que qualquer elemento armazenado em A, de forma que $a_x = n$. Assim, o tempo de execução do pior caso da BuscaLinear é $T_{BL}(n) = 5tn + 3t$ e o tempo de execução do pior caso da BuscaLinearEmOrdem é $T_{BO}(n) = 8tn + 6t$.

No caso de BuscaBinaria, o pior caso também ocorre quando o laço **enquanto** executa o máximo de vezes que puder e o maior número de linhas internas ao corpo do laço são executadas a cada iteração. Isso significa que os dois primeiros testes do corpo do laço falham sempre, o que significa que $x < A[meio]$ sempre e a variável *dir* vai ser atualizada em toda iteração. Com isso, o vetor vai ser subdividido na metade a cada iteração, até que $esq > dir$ e o laço termine. Isso significa que o elemento x também não está no vetor A e que o laço executou $\log n + 1$ vezes. Com isso, o tempo de pior caso da busca binária é $T_{BB}(n) = 11t \log n + 4t$.

A análise de pior caso é muito importante pois limita superiormente o tempo de execução para qualquer instância, garantindo que o algoritmo nunca vai demorar mais do que esse limite. Por exemplo, com as análises feitas acima, podemos dizer que o tempo de execução $T_{BL}(n)$ de BuscaLinear no pior caso é $T_{BL}(n) = 5tn + 3t$, o que nos permite dizer que o tempo de execução de BuscaLinear é $T_{BL}(n) \leq 5tn + 3t$. Outra razão para a análise de pior caso ser considerada é que, para alguns algoritmos, o pior caso (ou algum caso próximo do pior) ocorre com muita frequência.

O tempo de execução no melhor e no pior caso são *limitantes* para o tempo de execução do algoritmo sobre qualquer instância. Em geral, eles são calculados pensando-se em instâncias específicas, que forçam o algoritmo a executar o mínimo e o máximo de comandos possíveis. No caso da BuscaLinear, sabemos das análises anteriores que qualquer que seja o vetor A e o elemento x dados ao algoritmo, seu tempo de execução $T_{BL}(n)$ será tal que $5t \leq T_{BL}(n) \leq 5tn + 3t$. Veja que a diferença entre esses dois valores é da ordem de n, isto é, quando $n = 2$ o tempo de execução pode ser algo entre $5t$ e $13t$, mas quando $n = 50$ o tempo pode ser qualquer coisa entre $5t$ e $253t$. Assim, quanto maior o valor de n, mais distantes serão esses limitantes. Por isso, em alguns casos pode ser interessante ter uma previsão um pouco mais justa sobre o tempo de execução, podendo-se considerar o tempo no caso médio.

Definição 4.4: *Tempo do caso médio*

O tempo de execução do *caso médio* de um algoritmo é a média do tempo de execução dentre todas as instâncias de entrada possíveis de um dado tamanho n.

Por exemplo, para os algoritmos de busca, assuma por simplicidade que x está em A. Agora considere que quaisquer uma das $n!$ permutações dos n elementos de A têm a mesma chance de ser passada como o vetor de entrada. Note que, nesse caso, cada número tem a mesma probabilidade de estar em quaisquer das n posições do vetor. Assim, em média, a posição p_x de x em A é dada por $(1 + 2 + \cdots + n)/n = (n + 1)/2$. Logo, o tempo médio de execução da busca linear é dado por $T_{BL}(n) = 5tp_x = (5tn + 5t)/2$.

O tempo de execução de caso médio da busca binária envolve calcular a média de r_x dentre todas as ordenações possíveis do vetor, onde, lembre-se, r_x é a quantidade de vezes que o teste do laço principal é executado. Calcular precisamente essa média não é difícil, mas vamos evitar essa tecnicalidade nesse momento, apenas mencionando que, no caso médio, o tempo de execução da busca binária é dado por $d \log n$, para alguma constante d (um número que não é uma função de n).

Muitas vezes o tempo de execução no caso médio é quase tão ruim quanto no pior caso. No caso das buscas, vimos que a busca linear tem tempo de execução $5tn + 3t$ no pior caso, e $(5tn+5t)/2$ no caso médio, sendo ambos uma expressão da forma $an+b$, para constantes a e b, isto é, funções lineares em n. Assim, ambos possuem tempo de execução *linear* no tamanho da entrada. Mas é necessário deixar claro que esse nem sempre é o caso. Por exemplo, seja n o tamanho de um vetor que desejamos ordenar. Existe um algoritmo de ordenação chamado *Quicksort* que tem tempo de execução de pior caso quadrático em n (i.e., da forma $an^2 + bn + c$, para constantes a, b e c), mas em média o tempo gasto é da ordem de $n \log n$, que é muito menor que uma função quadrática em n, conforme n cresce. Embora o tempo de

execução de pior caso do *Quicksort* seja pior do que de outros algoritmos de ordenação (e.g., *Mergesort*, *Heapsort*), ele é comumente utilizado, dado que seu pior caso raramente ocorre. Por fim, vale mencionar que nem sempre é simples descrever o que seria uma "entrada média" para um algoritmo, e análises de caso médio são geralmente mais complicadas do que análises de pior caso.

4.2 Bons algoritmos

Ao projetar algoritmos, estamos sempre em busca daqueles que são *eficientes*.

Definição 4.5: *Algoritmo eficiente*

Um algoritmo é *eficiente* se seu tempo de execução *no pior caso* puder ser descrito por uma função *polinomial* no tamanho da entrada.

Mas e quando temos vários algoritmos eficientes que resolvem um mesmo problema, qual deles escolher? A resposta para essas perguntas é *depende*. Depende se temos informação sobre a entrada, sobre os dados que estão sendo passados, ou mesmo da aplicação em que usaremos esses algoritmos. Quando não se sabe nada, respondemos a pergunta escolhendo pelo algoritmo *mais* eficiente, que é o que tem melhor tempo de execução no pior caso.

Assim, por exemplo, dos três algoritmos que temos para resolver o problema da busca em vetor ordenado, o mais eficiente é a BuscaBinaria, que tem uma função logarítmica de tempo no pior caso, ao contrário da BuscaLinear e da BuscaLinearEmOrdem, cujo pior caso tem tempo linear no tamanho da entrada.

É importante ter em mente que outras informações devem ser levadas em consideração sempre que possível. Quando o tamanho da entrada é muito pequeno, por exemplo, um algoritmo cuja ordem de crescimento do tempo de pior caso é menor do que a de outro não necessariamente é a melhor escolha (quando $n < 6$, por exemplo, a BuscaLinear executa menos operações do que a BuscaBinaria).

Capítulo 5

Notação assintótica

Uma abstração que ajuda bastante na análise do tempo de execução de algoritmos é o estudo da *taxa de crescimento* de funções. Esse estudo nos permite comparar tempo de execução de algoritmos independentemente da plataforma utilizada, da linguagem, etc[1].

Se um algoritmo leva tempo $f(n) = an^2 + bn + c$ para ser executado, onde a, b e c são constantes e n é o tamanho da entrada, então o termo que realmente importa para grandes valores de n é an^2. Ademais, as constantes também podem ser desconsideradas, de modo que o tempo de execução nesse caso seria "da ordem de n^2". Por exemplo, para $n = 1000$ e $a = b = c = 2$, temos $an^2 + bn + c = 2000000 + 2000 + 2 = 2002002$ e $n^2 = 1000000$.

Estamos interessados no que acontece com $f(n)$ quando n tende a infinito, o que chamamos de *análise assintótica* de $f(n)$. Da mesma forma, ao comparar duas funções $f(n)$ e $g(n)$, estaremos considerando o que acontece quando n cresce indefinidamente, de forma que a ordem de crescimento das duas funções será suficiente para realizarmos a comparação.

As notações assintóticas nada mais são do que formalismos que indicam limitantes para funções e que escondem alguns detalhes. Isso é necessário porque para valores pequenos de n os comportamentos das funções podem ainda variar. Por exemplo, considere $f(n) = 475n+34$ e $g(n) = n^2 + 3$. Claramente, não é verdade que $f(n) \leq g(n)$ sempre, porque, por exemplo, $f(4) = 1934$ e $g(4) = 19$. Porém, a partir do momento que n for maior do que 476, teremos $f(n) \leq g(n)$. Também é fácil dizer $f(n) \leq 475g(n) = 475n^2 + 1425$. Essas constantes que multiplicam as funções e os valores iniciais de n ficam escondidos na notação assintótica.

[1] Observe como o valor t que utilizamos para calcular os tempos de execução das buscas em vetor no capítulo anterior polui as expressões e atrapalham as nossas análises.

5.1 Notações O, Ω e Θ

As notações O e Ω ajudam a limitar funções superiormente e inferiormente, respectivamente.

Definição 5.1: ***Notações O e Ω***

Seja n um inteiro positivo e sejam $f(n)$ e $g(n)$ funções positivas. Dizemos que

- $f(n) = O(g(n))$ se existem constantes positivas C e n_0 tais que $f(n) \leq Cg(n)$ para todo $n \geq n_0$;
- $f(n) = \Omega(g(n))$ se existem constantes positivas c e n_0 tais que $cg(n) \leq f(n)$ para todo $n \geq n_0$.

Em outras palavras, $f(n) = O(g(n))$ quando, para todo n suficientemente grande (maior que um n_0), temos $f(n)$ é limitada superiormente por $Cg(n)$. Dizemos que $f(n)$ é no máximo *da ordem* de $g(n)$. Por outro lado, $f(n) = \Omega(g(n))$ quando, para todo n suficientemente grande (maior que um n_0), $f(n)$ é limitada inferiormente por $cg(n)$. Veja a Figura 5.1.

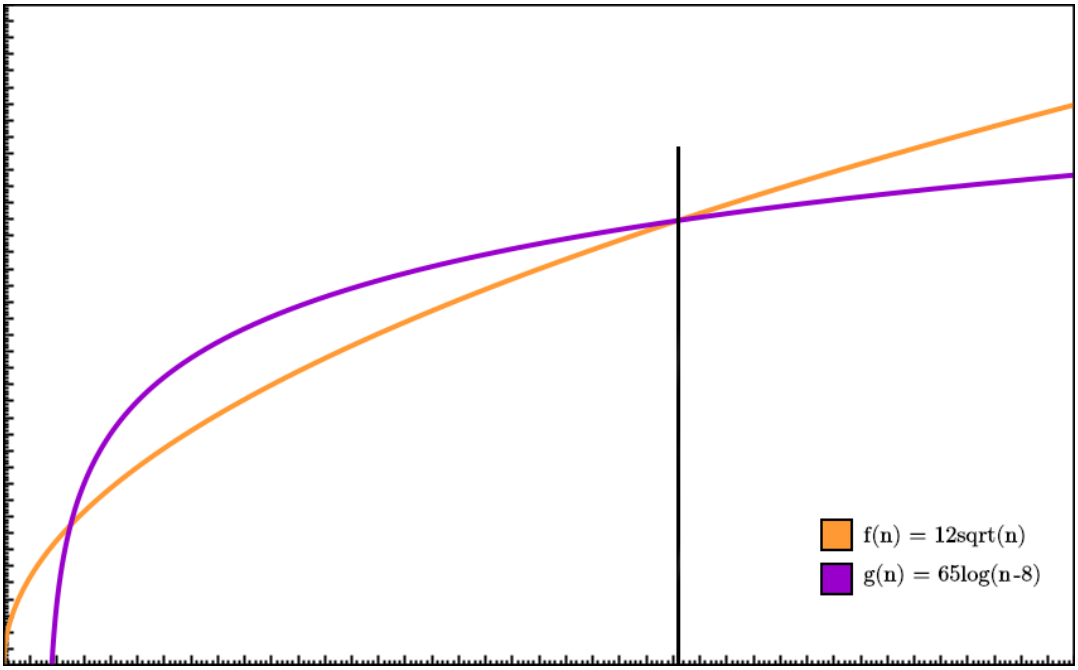


Figura 5.1: A função $g(n) = 65\log(n-8)$ é $O(f(n)))$, e isso pode ser visualizado no gráfico, pois a partir de um certo ponto (a barra vertical) temos $f(n)$ sempre acima de $g(n)$. Também veja que $f(n) = 12\sqrt{n}$ é $\Omega(g(n))$, pois a partir de um certo ponto (no caso, o mesmo) temos $g(n)$ sempre abaixo de $f(n)$.

Considere as funções $f(n) = 12\sqrt{n}$ e $g(n) = 65\log(n-8)$ da Figura 5.1. Intuitivamente, já sabemos que funções logarítmicas tendem a crescer mais devagar do que funções polinomiais (a figura nos mostra isso, porém nem sempre temos o apoio de ferramentas visuais). Por isso, vamos tentar primeiro mostrar que $g(n)$ é $O(f(n))$. Veja que

$$g(n) = 65\log(n-8) \leq 65\log(n) \leq 65\sqrt{n} = 65\frac{12}{12}\sqrt{n} = \frac{65}{12}f(n)\,,$$

onde só podemos usar o fato que $\log n \leq \sqrt{n}$ se $n \geq 16$. Assim, tomando $C = \frac{65}{12}$ e $n_0 = 16$ temos, por definição, que $g(n)$ é $O(f(n))$. Note, na figura, como a barra que indica o momento em que $f(n)$ fica sempre acima de $g(n)$ não está sobre o valor 16. Realmente, não existem constantes únicas C e n_0 com as quais podemos mostrar que $g(n)$ é $O(g(n))$. Veja que o resultado anterior diretamente nos dá que $f(n) \geq \frac{1}{C}g(n)$ sempre que $n \geq n_0$, e tomando $c = \frac{1}{C}$ e o mesmo n_0, temos, por definição, que $f(n)$ é $\Omega(g(n))$. De fato, esse último argumento pode ser generalizado para mostrar que sempre que uma função $f_1(n)$ for $O(f_2(n))$, teremos que $f_2(n)$ é $\Omega(f_1(n))$.

Por outro lado, perceba que $g(n)$ não é $\Omega(f(n))$ (pela figura, vemos que $g(n)$ fica abaixo e não o contrário). Suponha que isso fosse verdade. Então deveriam haver constantes c e n_0 tais que $65\log(n-8) \geq c \cdot 12\sqrt{n}$ sempre que $n \geq n_0$. Se a expressão anterior vale, também vale que $c \leq \frac{65\log(n-8)}{12\sqrt{n}}$. Acontece que $\frac{65\log(n-8)}{12\sqrt{n}}$ tende a 0 conforme n cresce indefinidamente, o que significa que é impossível que c, positiva, sempre seja menor do que isso. Com essa contradição, provamos que $g(n)$ não é $\Omega(f(n))$. Com um raciocínio parecido podemos mostrar que $f(n)$ não é $O(g(n))$.

A notação Θ ajuda a limitar funções de forma justa. Dadas funções $f(n)$ e $g(n)$, se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$, então dizemos que $f(n) = \Theta(g(n))$. Veja a Figura 5.2.

Definição 5.2: *Notação* Θ

Seja n um inteiro positivo e sejam $f(n)$ e $g(n)$ funções positivas. Dizemos que $f(n) = \Theta(g(n))$ se existem constantes positivas c, C e n_0 tais que $cg(n) \leq f(n) \leq Cg(n)$ para todo $n \geq n_0$.

Considere as funções $g(n) = 3\log(n^2) + 2$, $f(n) = 2\log n$ e $h(n) = 8\log n$ da Figura 5.2. Veja que

$$g(n) = 3\log(n^2) + 2 = 6\log n + 2 \leq 6\log n + 2\log n = 8\log n = h(n)\,,$$

onde não precisamos assumir nada de especial sobre o valor de n. Assim, tomando $C = 1$ e

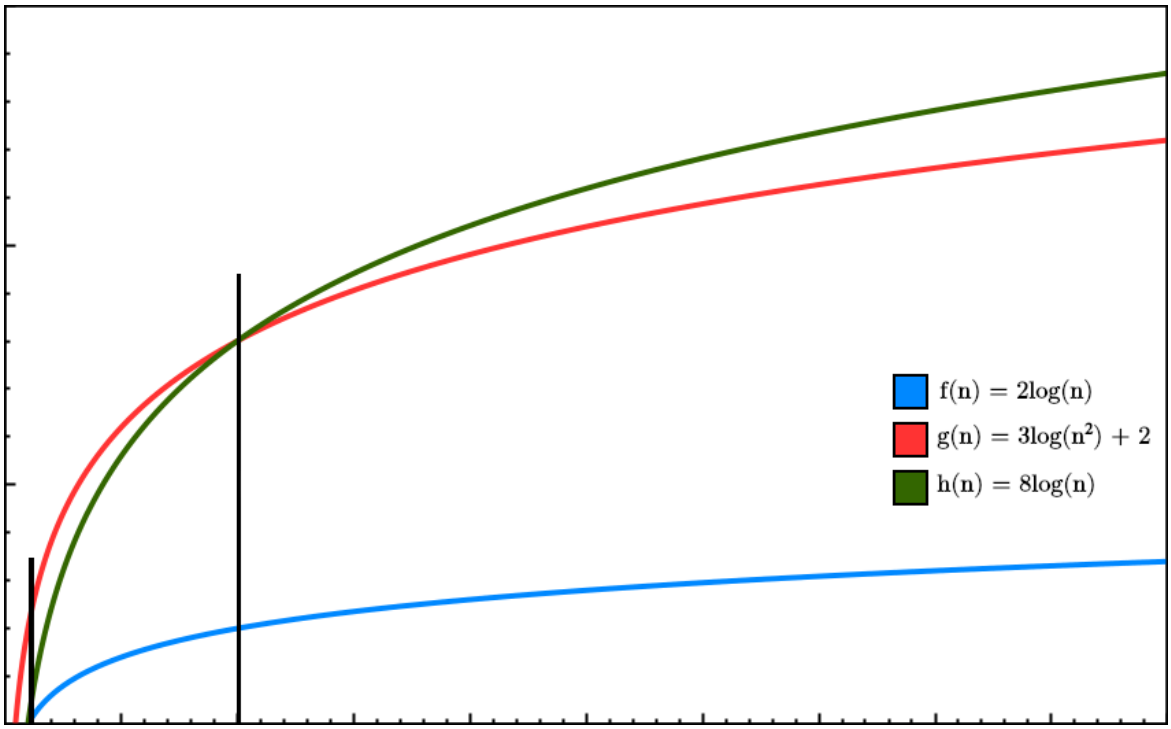


Figura 5.2: A função $g(n) = 3\log n^2 + 2$ é $\Theta(\log n)$, e isso pode ser visualizado no gráfico, pois a partir de um certo ponto (a barra vertical mais à direita), temos $g(n)$ sempre entre $2\log n$ e $8\log n$.

$n_0 = 1$, podemos afirmar, por definição, que $g(n)$ é $O(h(n))$. Note ainda que $h(n)$ é $O(\log n)$, bastando tomar a constante $C = 8$, de forma que $g(n)$ também é $O(\log n)$. Por outro lado,

$$g(n) = 3\log(n^2) + 2 = 6\log n + 2 \geq 6\log n = 3f(n)\,,$$

onde também não precisamos assumir nada de especial sobre o valor de n. Então, tomando $c = 3$ e $n_0 = 1$, podemos afirmar, por definição, que $g(n)$ é $\Omega(f(n))$. Também podemos ver que $f(n)$ é $\Omega(\log n)$, o que implica em $g(n)$ ser $\Omega(\log n)$. Assim, acabamos de mostrar que $g(n)$ é $\Theta(\log n)$.

É comum utilizar $O(1)$, $\Omega(1)$ e $\Theta(1)$ para indicar funções constantes, como um valor c (independente de n) ou mesmo cn^0 (um polinômio de grau 0). Por exemplo, $f(n) = 346$ é $\Theta(1)$, pois $346 \leq c \cdot 1$, bastando tomar $c = 346$ (ou 348, ou 360), e também pois $346 \geq c \cdot 1$, bastando também tomar $c = 346$ (ou 341, ou 320).

Vamos trabalhar com mais alguns exemplos para entender melhor as notações O, Ω e Θ.

Teorema 6.3: ***Teorema Mestre***

Se $f(n) = 10n^2 + 5n + 3$, então $f(n) = \Theta(n^2)$.

Demonstração. Para mostrar que $f(n) = \Theta(n^2)$, vamos mostrar que $f(n) = O(n^2)$ e $f(n) = \Omega(n^2)$. Verifiquemos primeiramente que $f(n) = O(n^2)$. Se tomarmos $n_0 = 1$, então note que, como queremos $f(n) \leq Cn^2$ para todo $n \geq n_0 = 1$, precisamos obter uma constante C tal que $10n^2 + 5n + 3 \leq Cn^2$. Mas então basta que

$$C \geq \frac{10n^2 + 5n + 3}{n^2} = 10 + \frac{5}{n} + \frac{3}{n^2}\,.$$

Como para $n \geq 1$ temos

$$10 + \frac{5}{n} + \frac{3}{n^2} \leq 10 + 5 + 3 = 18\,,$$

basta tomar $n_0 = 1$ e $C = 18$. Assim, temos

$$C = 18 = 10 + 5 + 3 \geq 10 + \frac{5}{n} + \frac{3}{n^2} = \frac{10n^2 + 5n + 3}{n^2}\,,$$

como queríamos. Logo, concluímos que $f(n) \leq 18n^2$ para todo $n \geq 1$ e, portanto, $f(n) = O(n^2)$.

Agora vamos verificar que $f(n) = \Omega(n^2)$. Se tomarmos $n_0 = 1$, então note que, como queremos $f(n) \geq cn^2$ para todo $n \geq n_0 = 1$, precisamos obter uma constante c tal que $10n^2 + 5n + 3 \geq cn^2$. Mas então basta que

$$c \leq 10 + \frac{5}{n} + \frac{3}{n^2}\,.$$

Como para $n \geq 1$ temos

$$10 + \frac{5}{n} + \frac{3}{n^2} \geq 10\,,$$

basta tomar $n_0 = 1$ e $c = 10$. Concluímos então que $f(n) \geq 10n^2$ para todo $n \geq 1$ e, portanto, $f(n) = \Omega(n^2)$.

Como mostramos que $f(n) = O(n^2)$ e $f(n) = \Omega(n^2)$, então concluímos que $f(n) = \Theta(n^2)$. □

Perceba que na prova do Fato 5.3 traçamos uma simples estratégia para encontrar um valor apropriado para as constantes. Os valores para n_0 escolhido nos dois casos foi 1, mas algumas vezes é mais conveniente ou somente é possível escolher um valor maior para n_0. Considere o exemplo a seguir.

Fato 5.4

Se $f(n) = 5\log n + \sqrt{n}$, então $f(n) = O(\sqrt{n})$.

Demonstração. Comece percebendo que $f(n) = O(n)$, pois sabemos que $\log n$ e $\sqrt{n}$ são menores que n para valores grandes de n (na verdade, para qualquer $n \geq 2$). Porém, é possível melhorar esse limitante para $f(n) = O(\sqrt{n})$. De fato, basta obter C e n_0 tais que para $n \geq n_0$ temos $5\log n + \sqrt{n} \leq C\sqrt{n}$. Logo, queremos que

$$C \geq \frac{5\log n}{\sqrt{n}} + 1\,. \tag{5.1}$$

Mas nesse caso precisamos ter cuidado ao escolher n_0, pois com $n_0 = 1$, temos $5(\log 1)/\sqrt{1} + 1 = 1$, o que pode nos levar a pensar que $C = 1$ é uma boa escolha para C. Com essa escolha, precisamos que a desigualdade (5.1) seja válida para todo $n \geq n_0 = 1$. Porém, se $n = 2$, então (5.1) não é válida, uma vez que $5(\log 2)/\sqrt{2} + 1 > 1$.

Para facilitar, podemos observar que, para todo $n \geq 16$, temos $(\log n)/\sqrt{n} \leq 1$, de modo que a desigualdade (5.1) é válida, i.e., $(5\log n)/\sqrt{n} + 1 \leq 6$. Portanto, tomando $n_0 = 16$ e $C = 6$, mostramos que $f(n) = O(\sqrt{n})$. □

A estratégia utilizada nas demonstrações dos Fatos 5.3 e 5.4 de isolar a constante e analisar a expressão restante não é única. Veja o próximo exemplo.

Fato 5.5

Se $f(n) = 5\log n + \sqrt{n}$, então $f(n) = O(\sqrt{n})$.

Demonstração. Podemos observar facilmente que $\log n \leq \sqrt{n}$ sempre que $n \geq 16$. Assim,

$$5\log n + \sqrt{n} \leq 5\sqrt{n} + \sqrt{n} = 6\sqrt{n}\,, \tag{5.2}$$

onde a desigualdade vale sempre que $n \geq 16$. Como chegamos a uma expressão da forma $f(n) \leq C\sqrt{n}$, concluímos nossa demonstração. Portanto, tomando $n_0 = 16$ e $C = 6$, mostramos que $f(n) = O(\sqrt{n})$. □

Uma terceira estratégia ainda pode ser vista no próximo exemplo.

Fato 5.6

Se $f(n) = 5\log n + \sqrt{n}$, então $f(n) = O(\sqrt{n})$.

Demonstração. Para mostrar esse resultado, basta obter C e n_0 tais que para $n \geq n_0$ temos $5 \log n + \sqrt{n} \leq C\sqrt{n}$. Logo, queremos que

$$C \geq \frac{5 \log n}{\sqrt{n}} + 1\,. \tag{5.3}$$

Note que

$$\lim_{n\to\infty} \left(\frac{5 \log n}{\sqrt{n}} + 1 \right) = \lim_{n\to\infty} \left(\frac{5 \log n}{\sqrt{n}} \right) + \lim_{n\to\infty} 1 \tag{5.4}$$

$$= \lim_{n\to\infty} \left(\frac{5 \frac{1}{n \ln 2}}{\frac{1}{2\sqrt{n}}} \right) + 1 \tag{5.5}$$

$$= \lim_{n\to\infty} \left(\frac{10/\ln 2}{\sqrt{n}} \right) + 1 = 0 + 1 = 1\,, \tag{5.6}$$

onde usamos a regra de L'Hôpital na segunda igualdade. Sabendo que quando $n = 1$ temos $5(\log 1)/\sqrt{1}+1 = 1$ e usando o resultado acima, que nos mostra que a expressão $(5 \log n)/\sqrt{n}+1$ tende a 1, provamos que é possível encontrar um C que seja maior do que essa expressão a partir de algum $n = n_0$. □

Perceba que podem existir diversas possibilidades de escolha para n_0 e C: pela definição, basta que encontremos alguma. Por exemplo, na prova do Fato 5.4, usar $n_0 = 3454$ e $C = 2$ também funciona para mostrar que $5 \log n + \sqrt{n} = O(\sqrt{n})$. Outra escolha possível seria $n_0 = 1$ e $C = 11$. Não é difícil mostrar que $f(n) = \Omega(\sqrt{n})$.

Outros exemplos de limitantes seguem abaixo, onde a e b são inteiros positivos. É um bom exercício formalizá-los:

- $\log_a n = \Theta(\log_b n)$.
- $\log_a n = O(n^\varepsilon)$ para qualquer $\varepsilon > 0$.
- $(n + a)^b = \Theta(n^b)$.
- $2^{n+a} = \Theta(2^n)$.
- $2^{an} \neq O(2^n)$.
- $7n^2 \neq O(n)$.

Vamos utilizar a definição da notação assintótica para mostrar que $7n^2 \neq O(n)$.

Fato 5.7

Se $f(n) = 7n^2$ então $f(n) \neq O(n)$.

Demonstração. Lembre que $f(n) = O(g(n))$ se existem constantes positivas C e n_0 tais que se $n \geq n_0$, então $0 \leq f(n) \leq Cg(n)$. Suponha, por contradição, que $7n^2 = O(n)$, i.e., que existem tais constantes C e n_0 tais que se $n \geq n_0$, então

$$7n^2 \leq Cn\,.$$

Nosso objetivo agora é chegar a uma contradição. Note que, isolando o C na equação acima, para todo $n \geq n_0$, temos $C \geq 7n$, o que é um absurdo, pois claramente $7n$ cresce indefinidamente, de forma que não é possível que C, constante positiva, seja ainda maior do que isso para todo $n \geq n_0$. □

5.1.1 Relações entre as notações O, Ω e Θ

No teorema enunciado a seguir descrevemos propriedades importantes acerca das relações entre as notações assintóticas O, Ω e Θ.

Definição 9.2

Sejam $f(n)$, $g(n)$ e $h(n)$ funções positivas. Temos que

1. $f(n) = \Theta(f(n))$;
2. $f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$;
3. $f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$;
4. Se $f(n) = O(g(n))$ e $g(n) = \Theta(h(n))$, então $f(n) = O(h(n))$;
   O mesmo vale substituindo O por Ω;
5. Se $f(n) = \Theta(g(n))$ e $g(n) = O(h(n))$, então $f(n) = O(h(n))$;
   O mesmo vale substituindo O por Ω;
6. $f(n) = O(g(n) + h(n))$ se e somente se $f(n) = O(g(n)) + O(h(n))$;
   O mesmo vale substituindo O por Ω ou por Θ;
7. Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, então $f(n) = O(h(n))$;
   O mesmo vale substituindo O por Ω ou por Θ.

Demonstração. Vamos mostrar que os itens enunciados no teorema são válidos.

Item 1. Esse item é simples, pois para qualquer $n \geq 1$ temos que $f(n) = 1 \cdot f(n)$, de modo que para $n_0 = 1$, $c = 1$ e $C = 1$ temos que para todo $n \geq n_0$ vale que

$$cf(n) \leq f(n) \leq Cf(n)\,,$$

de onde concluímos que $f(n) = \Theta(f(n))$.

Item 2. Note que basta provar uma das implicações (a prova da outra implicação é idêntica). Provaremos que se $f(n) = \Theta(g(n))$ então $g(n) = \Theta(f(n))$. Se $f(n) = \Theta(g(n))$, então temos que existem constantes positivas c, C e n_0 tais que

$$cg(n) \leq f(n) \leq Cg(n) \tag{5.7}$$

para todo $n \geq n_0$. Assim, analisando as desigualdades em (5.7), concluímos que

$$\left(\frac{1}{C}\right) f(n) \leq g(n) \leq \left(\frac{1}{c}\right) f(n)$$

para todo $n \geq n_0$. Portanto, existem constantes n_0, $c' = 1/C$ e $C' = 1/c$ tais que $c'f(n) \leq g(n) \leq C'f(n)$ para todo $n \geq n_0$.

Item 3. Vamos provar uma das implicações (a prova da outra implicação é análoga). Se $f(n) = O(g(n))$, então temos que existem constantes positivas C e n_0 tais que $f(n) \leq Cg(n)$ para todo $n \geq n_0$. Portanto, temos que $g(n) \geq (1/C)f(n)$ para todo $n \geq n_0$, de onde concluímos que $g(n) = \Omega(f(n))$.

Item 4. Se $f(n) = O(g(n))$, então temos que existem constantes positivas C e n_0 tais que $f(n) \leq Cg(n)$ para todo $n \geq n_0$. Se $g(n) = \Theta(h(n))$, então temos que existem constantes positivas d, D e n_0' tais que $dh(n) \leq g(n) \leq Dh(n)$ para todo $n \geq n_0'$. Então $f(n) \leq Cg(n) \leq CDh(n)$ para todo $n \geq \max\{n_0, n_0'\}$, de onde concluímos que $f(n) = O(h(n))$.

Item 5. Se $f(n) = \Theta(g(n))$, então temos que existem constantes positivas c, C e n_0 tais que $cg(n) \leq f(n) \leq Cg(n)$ para todo $n \geq n_0$. Se $g(n) = O(h(n))$, então temos que existem constantes positivas D e n_0' tais que $g(n) \leq Dh(n)$ para todo $n \geq n_0'$. Então $f(n) \leq Cg(n) \leq CDh(n)$ para todo $n \geq \max\{n_0, n_0'\}$, de onde concluímos que $f(n) = O(h(n))$.

Item 6. Vamos provar uma das implicações (a prova da outra implicação é análoga). Se $f(n) = O(g(n) + h(n))$, então temos que existem constantes positivas C e n_0 tais que $f(n) \leq C(g(n) + h(n))$ para todo $n \geq n_0$. Mas então $f(n) \leq Cg(n) + Ch(n)$ para todo $n \geq n_0$, de forma que $f(n) = O(g(n)) + O(h(n))$.

Item 7. Análoga às provas dos itens 4 e 5. □

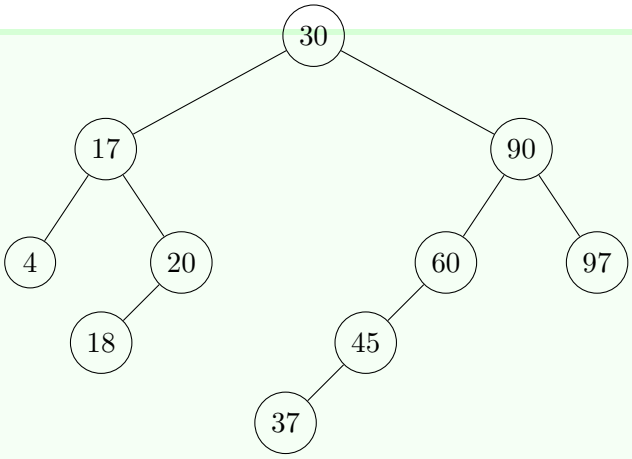
Note que se uma função $f(n)$ é uma soma de funções logarítmicas, exponenciais e polinômios em n, então sempre temos que $f(n)$ vai ser $\Theta(g(n))$, onde $g(n)$ é o termo de $f(n)$ com maior taxa de crescimento (desconsiderando constantes). Por exemplo, se

$$f(n) = 4\log n + 1000(\log n)^{100} + \sqrt{n} + n^3/10 + 5n^5 + n^8/27\,,$$

então sabemos que $f(n) = \Theta(n^8)$.

5.2 Notações o e ω

Apesar das notações assintóticas descritas até aqui fornecerem informações importantes acerca do crescimento das funções, muitas vezes elas não são tão precisas quanto gostaríamos. Por exemplo, temos que $2n^2 = O(n^2)$ e $4n = O(n^2)$. Apesar dessas duas funções terem ordem de complexidade $O(n^2)$, somente a primeira é "justa". Para descrever melhor essa situação, temos as notações *o-pequeno* e *ω-pequeno*.



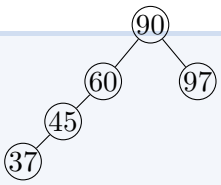
Seja n um inteiro positivo e sejam $f(n)$ e $g(n)$ funções positivas. Dizemos que

- $f(n) = o(g(n))$ se para toda constante $c > 0$ existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$ para todo $n \geq n_0$;
- $f(n) = \omega(g(n))$ se para toda constante $C > 0$ existe $n_0 > 0$ tal que $f(n) > Cg(n) \geq 0$ para todo $n \geq n_0$.

Por exemplo, $2n = o(n^2)$ mas $2n^2 \neq o(n^2)$. O que acontece é que, se $f(n) = o(g(n))$, então $f(n)$ é insignificante com relação a $g(n)$, para n grande. Alternativamente, podemos dizer que $f(n) = o(g(n))$ quando $\lim_{n\to\infty}(f(n)/g(n)) = 0$. Por exemplo, $2n^2 = \omega(n)$ mas

$2n^2 \neq \omega(n^2)$.

Vamos ver um exemplo para ilustrar como podemos mostrar que $f(n) = o(g(n))$ para duas funções f e g.



$10n + 3\log n = o(n^2)$.

Demonstração. Seja $f(n) = 10n + 3\log n$. Precisamos mostrar que, para qualquer constante positiva c, existe um n_0 tal que $10n + 3\log n < cn^2$ para todo $n \geq n_0$. Assim, seja $c > 0$ uma constante qualquer. Primeiramente note que $10n + 3\log n < 13n$ e que se $n > 13/c$, então

$$10n + 3\log n < 13n < cn \leq cn^2\,.$$

Portanto, acabamos de provar o que precisávamos (com $n_0 = (13/c) + 1$). □

Note que com uma análise similar à feita na prova acima podemos provar que $10n + 3\log n = o(n^{1+\varepsilon})$ para todo $\varepsilon > 0$. Basta que, para todo $c > 0$, façamos $n > (13/c)^{1/\varepsilon}$.

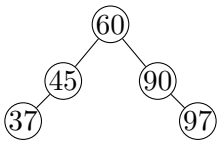
Outros exemplos de limitantes seguem abaixo, onde a e b são inteiros positivos.

- $\log_a n \neq o(\log_b n)$.
- $\log_a n \neq \omega(\log_b n)$.
- $\log_a n = o(n^\varepsilon)$ para qualquer $\varepsilon > 0$.
- $an = o(n^{1+\varepsilon})$ para qualquer $\varepsilon > 0$.
- $an = \omega(n^{1-\varepsilon})$ para qualquer $\varepsilon > 0$.
- $1000n^2 = o((\log n)n^2)$.

5.3 Relações entre as notações assintóticas

Muitas dessas comparações assintóticas têm propriedades importantes. No que segue, sejam $f(n)$, $g(n)$ e $h(n)$ assintoticamente positivas. Todas as cinco notações descritas são *transitivas*, e.g., se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, então temos $f(n) = O(h(n))$. *Reflexividade* vale para O, Ω e Θ, e.g., $f(n) = O(f(n))$. Temos também a *simetria* com a notação Θ, i.e.,

$f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$. Por fim, a *simetria transposta* vale para os pares $\{O, \Omega\}$ e $\{o, \omega\}$, i.e., $f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$, e $f(n) = o(g(n))$ se e somente se $g(n) = \omega(f(n))$.



Tempo com notação assintótica

A partir de agora, usaremos apenas notação assintótica para nos referirmos aos tempos de execução dos algoritmos. Em outras palavras, não vamos mais contar explicitamente a quantidade de passos básicos feitas por um algoritmo, escrevendo uma expressão que use, além do tamanho da entrada, o tempo t de execução de cada passo. A ideia é conseguir analisar um algoritmo de forma bem mais rápida.

Por exemplo, vejamos a BuscaLinear novamente, que está repetida no Algoritmo 6.1, por comodidade. Nossa intenção é olhar para esse algoritmo e dizer "a BuscaLinear tem tempo $O(n)$", "a BuscaLinear tem tempo $\Omega(1)$", "a BuscaLinear tem tempo $\Theta(n)$ no pior caso", e "a BuscaLinear tem tempo $\Theta(1)$ no melhor caso". A ideia é que vamos aplicar notação assintótica nas expressões de tempo que conseguimos no Capítulo 4. E a partir deste capítulo, não vamos mais descrever as expressões explicitamente (usando t), mas apenas as expressões em notação assintótica. Por isso, é muito importante ter certeza de que você entendeu *por que* podemos dizer aquelas frases.

Algoritmo 6.1: BuscaLinear(A, n, k)

```
1 i = 1
2 enquanto i ≤ n faça
3     se A[i] == k então
4         devolve i
5     i = i + 1
6 devolve −1
```

No Capítulo 4 vimos que o tempo de execução dele quando x está em A é dado pela função $T_{BL}(n) = 5tp_x$, onde p_x é a posição do elemento x no vetor A e t é uma constante que

indica o tempo que um passo básico demora para ser executado. Também vimos que o tempo é dado pela função $T_{BL}(n) = 5tn + 3t$ quando x não está em A. Analisamos, na Seção 4.1, os tempos no melhor caso, pior caso e caso médio dessa busca, de onde conseguimos três expressões diferentes.

No melhor caso, a busca linear leva tempo $T_{BL}(n) = 5t$. Veja que a expressão $5t$ já é uma constante, de forma que podemos escrever $5t = \Theta(1)$.

Agora considere o pior caso, em que vimos que a busca linear leva tempo $T_{BL}(n) = 5tn + 3t$. Veja que $5tn + 3t \leq 5tn + 3tn = 8tn$ para qualquer $n \geq 1$. Assim, tomando $C = 8t$ e $n_0 = 1$, podemos concluir que $5tn + 3t = O(n)$. Também é verdade que $5tn + 3t \leq 5tn + 3tn \leq 5tn^3 + 3tn^3 = 8tn^3$ sempre que $n \geq 1$, de onde podemos concluir que $5tn + 3t$ é $O(n^3)$. Podemos ainda concluir que $5tn + 3t$ é $O(n^5)$, $O(n^{10})$, $O(2^n)$, $O(n^n)$. Acontece que o limite $O(n)$ é mais justo, pois $5tn+3t$ é $\Omega(n)$. Isso por sua vez vale porque $5tn+3t \geq 5tn$ para qualquer $n \geq 1$, e podemos tomar $c = 5t$ e $n_0 = 1$ para chegar a essa conclusão. Também é verdade que $5tn + 3t$ é $\Omega(1)$, $\Omega(\log n)$ e $\Omega(\sqrt{n})$. Sabendo que $5tn + 3t$ é $O(n)$ e $\Omega(n)$, podemos concluir que também é $\Theta(n)$.

Agora veja que $5tn + 3t$ *não é* $\Omega(n^2)$, por exemplo. Se isso fosse verdade, então pela definição deveriam existir constantes c e n_0 tais que $5tn + 3t \geq cn^2$ sempre que $n \geq n_0$. Mas veja que isso é equivalente a dizer que deveria valer que $\frac{5t}{n} + \frac{3t}{n^2} \geq c$, o que é impossível, pois c é uma constante positiva (estritamente maior do que 0) e $\frac{5t}{c} + \frac{3t}{n^2}$ tende a 0 conforme n cresce indefinidamente, de forma que c nunca poderá ser sempre menor do que essa expressão. Como entramos em contradição, nossa suposição é falsa, o que indica que c e n_0 não podem existir.

De forma equivalente, podemos mostrar que $5tn+3t$ não é $O(\sqrt{n})$, porque se fosse, haveria constantes C e n_0 tais que $5tn + 3t \leq C\sqrt{n}$ sempre que $n \geq n_0$. Mas isso é equivalente a dizer que $5t\sqrt{n} + \frac{3t}{\sqrt{n}} \leq C$ sempre que $n \geq n_0$, o que é impossível, pois C é uma constante positiva e $5t\sqrt{n} + \frac{3t}{\sqrt{n}}$ cresce indefinidamente conforme n cresce indefinidamente, de forma que C nunca poderá ser sempre maior do que essa expressão. Novamente, a contradição nos leva a afirmar que C e n_0 não podem existir.

Dos resultados acima e de outros similares, podemos afirmar que a busca linear leva tempo $\Omega(n)$ no pior caso, leva tempo $\Omega(1)$ no pior caso, leva tempo $O(n)$ no pior caso, leva tempo $O(n^3)$ no pior caso, leva tempo $O(n^5)$ no pior caso, leva tempo $O(1)$ no melhor caso, leva tempo $O(n)$ no melhor caso, leva tempo $O(n^2)$ no melhor caso, leva tempo $\Omega(1)$ no melhor caso. Contudo, usando a notação Θ só é possível dizer que ela leva tempo tempo $\Theta(1)$ no melhor caso e $\Theta(n)$ no pior caso.

Um erro muito comum quando se usa notação assintótica é associar tempo de melhor caso com notação Ω e tempo de pior caso com notação O. E isso está muito errado, porque

note como as três notações são definidas em termos de funções e não de tempo de execução e muito menos de tempos em casos específicos. Assim, podemos utilizar *todas as notações* para analisar tempos de execução de melhor caso, pior caso ou caso médio de algoritmos, como fizemos nos exemplos acima, da busca linear. Por isso, sempre é importante explicitar a qual desses tempos estamos nos referimos.

Por outro lado, é importante perceber que esse erro não ocorre sem fundamento. Lembre-se, as análises de tempo de melhor e pior caso nos dão algumas garantias com relação ao tempo de execução do algoritmo sobre qualquer instância. Qualquer instância de entrada executará em tempo maior ou igual ao tempo do melhor caso e executará em tempo menor ou igual ao tempo do pior caso. Dos resultados anteriores vemos que não é errado dizer que no pior caso a busca linear leva tempo $\Omega(n)$. Também é verdade que no pior caso a busca linear leva tempo $\Omega(1)$. Acontece que esse tipo de informação não diz muita coisa sobre o algoritmo todo, ela apenas diz algo sobre o pior caso. De forma equivalente, dizer que o melhor caso da busca leva tempo $O(n)$ é verdade, mas não é tão acurado e não nos diz muito sobre o algoritmo todo. No que segue assumimos que n é grande o suficiente.

Se um algoritmo tem tempo de execução $T(n)$ no pior caso e sabemos que $T(n) = O(g(n))$, então para a instância de tamanho n em que o algoritmo é mais lento, ele leva tempo no máximo $Cg(n)$, onde C é constante. Portanto, podemos concluir que para **qualquer instância** de tamanho n o algoritmo leva tempo no máximo da ordem de $g(n)$. Por outro lado, se dizemos que $T(n) = \Omega(g(n))$ é o tempo de execução de pior caso de um algoritmo, então não temos muita informação útil. Nesse caso, sabemos somente que para a instância I_n de tamanho n em que o algoritmo é mais lento, o algoritmo leva tempo pelo menos $Cg(n)$, onde C é constante. Mas isso não implica nada sobre quaisquer outras instâncias do algoritmo, nem informa nada a respeito do tempo máximo de execução para a instância I_n.

Se um algoritmo tem tempo de execução $T(n)$ no melhor caso, uma informação importante é mostrar que $T(n) = \Omega(g(n))$, pois isso afirma que para a instância de tamanho n em que o algoritmo é mais rápido, ele leva tempo no mínimo $cg(n)$, onde c é constante. Isso também afirma que, para **qualquer instância** de tamanho n, o algoritmo leva tempo no mínimo da ordem de $g(n)$. Porém, se sabemos somente que $T(n) = O(g(n))$, então a única informação que temos é que para a instância de tamanho n em que o algoritmo é mais rápido, ele leva tempo no máximo $Cg(n)$, onde C é constante. Isso não diz nada sobre o tempo de execução do algoritmo para outras instâncias.

A BuscaLinear tem tempo $O(n)$ (sem mencionar qual caso) porque *o tempo de qualquer instância é no máximo* o tempo de execução do pior caso. Assim, se o tempo do algoritmo sobre qualquer instância de tamanho n é $T(n)$, então vale que $T(n) \leq 5tn + 3t \leq 8tn$, ou seja, $T(n)$ é $O(n)$ (ou, também podemos escrever $T(n) = O(n)$). É importante entender que,

apenas sabendo que $T(n) \leq 5tn + 3t$, não conseguimos dizer nada sobre $T(n)$ em notação Ω, porque para usar essa notação sobre uma função precisamos conhecer um limitante inferior para ela (e $5tn + 3t$ é limitante superior).

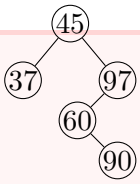
Podemos dizer que a BuscaLinear tem tempo $\Omega(1)$ (sem mencionar qual caso) porque o *tempo de qualquer instância é no mínimo* o tempo de execução do melhor caso. Assim, se o tempo do algoritmo sobre qualquer instância de tamanho n é $T(n)$, então vale que $T(n) \geq 5t$, ou seja, $T(n)$ é $\Omega(1)$ (ou, também podemos escrever $T(n) = \Omega(1)$). Note que, como *não conseguimos dizer que a busca tem tempo* $\Omega(n)$ *(por quê?), então não podemos dizer ela tem tempo* $\Theta(n)$. Podemos apenas dizer que ela tem tempo $\Theta(n)$ *no pior caso.*

Antes de prosseguir com mais exemplos, perceba que não precisávamos ter encontrado explicitamente as expressões $5tn + 3t$ e $5t$. Como mencionamos no início do capítulo, a ideia é fazer uma análise *mais rápida* e nós não vamos mais trabalhar com essas expressões. Olhe novamente para o algoritmo e perceba que cada linha, sozinha, executa em tempo constante (isto é, $\Theta(1)$). Ademais, a linha 2 domina o tempo de execução, pois ela executa o maior número de vezes. Assim, o tempo realmente depende de *quantas vezes* o laço será executado. Perceba que ele executa uma vez para cada valor de i, que começa em 1, é incrementado de uma em uma unidade, e pode ter valor *no máximo* $n + 1$. Ou seja, são *no máximo* $\Theta(n)$ execuções do laço. Como podemos ter bem menos valores de i, só conseguimos dizer que são $O(n)$ execuções (agora, sem o uso do termo "no máximo"). Por isso podemos dizer que o algoritmo leva tempo $O(n)$.

Agora volte ao algoritmo da BuscaBinaria e verifique que podemos dizer que ele leva tempo $O(\log n)$ (ou, $\Theta(\log n)$ no pior caso). Com essa nova nomenclatura e lembrando que a BuscaLinear e a BuscaBinaria resolvem o mesmo problema, de procurar um número em um vetor já ordenado, podemos compará-las dizendo que "a BuscaBinaria é *assintoticamente* mais eficiente do que a BuscaLinear". Isso porque a primeira tem tempo de pior caso proporcional a $\log n$, o que é bem melhor do que n. Agora, o termo *assintoticamente* é importante, porque para valores bem pequenos de n uma busca linear pode ser melhor do que a binária.

6.1 Exemplo completo de solução de problema

Nesta seção, considere o seguinte problema.



Dado um vetor A de tamanho n e um inteiro k, verificar se existem posições i e j, $i \neq j$, tais que $A[i] + A[j] = k$.

Uma vez que temos um novo problema, o **primeiro passo** é verificar se o entendemos corretamente. Por isso, sempre certifique-se de que você consegue fazer exemplos e resolvê-los. Por exemplo, se $A = (3, 6, 2, 12, 4, 9, 10, 1, 23, 5)$ e $k = 10$, então a resposta é `sim`, pois $A[2] = 6$ e $A[5] = 4$ somam 10. Veja que $A[6] = 9$ e $A[8] = 1$ também somam 10 (o problema apenas quer saber existem duas tais posições). Se $k = 234$ a resposta é claramente `não`.

O **segundo passo** é tentar criar um algoritmo que o resolva. Uma primeira ideia é verificar se esse problema é parecido com algum outro que já resolvemos, mesmo que apenas em partes, pois em caso positivo poderemos utilizar algoritmos que já temos em sua solução.

Veja que estamos *buscando* por duas posições no vetor, e já sabemos resolver um problema de busca quando isso envolve um único elemento. Será que conseguimos modificar esse problema de alguma forma que o problema da busca de um único elemento apareça? Veja que procurar por posições i e j tais que $A[i] + A[j] = k$ é equivalente a procurar por uma única posição i tal que $A[i] = k - A[j]$, se tivermos um valor fixo de j. Chamamos isso de *redução entre problemas*: usaremos um algoritmo que resolve um deles para poder criar um algoritmo para o outro[1]. Uma formalização dessa ideia encontra-se no Algoritmo 6.2.

Algoritmo 6.2: SomaK(A, n, k)

```
1 para j = 1 até n, incrementando faça
2     i = BuscaLinear(A, n, k − A[j])
3     se i ≠ −1 e i ≠ j então
4         devolve i, j
5 devolve −1, −1
```

O **terceiro passo** é verificar se esse algoritmo está de fato correto. Ele devolve uma solução correta para *qualquer* entrada? Em particular, teste-o sobre os exemplos que você mesmo criou no primeiro passo. Depois disso, um bom exercício é verificar a corretude dele usando uma invariante de laço. Note que esta invariante pode considerar que a chamada

[1] O conceito de redução será formalizado no Capítulo 28.

à BuscaLinear funciona corretamente, pois já provamos isso. Observe que tomamos o cuidado de verificar se $i \neq j$ na linha 3, porque se o vetor é, por exemplo, $A = (1, 3, 2)$ e $k = 6$, então a resposta deve ser negativa, pois não existem duas posições diferentes que somam 6. Sem a linha mencionada, o algoritmo iria devolver as posições $2, 2$, o que não é uma resposta válida.

O **quarto passo**, assumindo então que este algoritmo funciona, é verificar seu tempo de execução. Observe que o tamanho da entrada é n, a quantidade de elementos no vetor. Em uma análise rápida, veja que a linha 1 executa $O(n)$ vezes ao todo, porque j assume *no máximo* $n + 1$ valores diferentes (se a linha 4 executa, o algoritmo para). Isso significa que o conteúdo desse laço **para** executa $O(n)$ vezes (é importante entender que *não* conseguimos dizer $\Theta(n)$ aqui). Assim, se x é o tempo que leva para uma execução do corpo do laço acontecer, então o algoritmo todo leva tempo $xO(n)$. Resta encontrar x.

Considere então uma única execução do corpo do laço **para** externo. Na linha 2 fazemos uma chamada a outro algoritmo. Essa chamada por si só leva tempo constante e a atribuição à variável i também, porém a execução do algoritmo chamado leva mais tempo. Como já vimos o tempo de execução de BuscaLinear, sabemos portanto que essa linha leva tempo $O(n)$. O restante do laço leva tempo constante, pois temos apenas testes e eventualmente uma operação de retorno. Considerando a discussão do parágrafo anterior, concluímos que SomaK leva tempo $O(n^2)$.

O **quinto passo** é se perguntar se esse é o melhor que podemos fazer para esse problema. Veja que o tamanho da entrada é n, e no entanto o algoritmo tem tempo quadrático de execução. Tempos melhores de execução teriam ordem de $\log n$, n, $n \log n$, $n^{3/2}$, para citar alguns. Certamente não poderíamos fazer muito melhor do que tempo proporcional a n (como $\log n$, por exemplo), porque a própria entrada é da ordem de n e parece razoável supor que ao menos teremos que percorrê-la. Ainda sim, parece haver possibilidade de melhoria.

Note que o algoritmo busca pelo valor $k - A[j]$, para cada valor de j, testando todos os valores possíveis de i (pois a busca linear percorre todo o vetor dado). Um problema aqui é que, por exemplo, supondo $n > 10$, quando $j = 3$, a chamada a BuscaLinear irá considerar a posição 6, verificando se $A[6] == k - A[3]$. Mas quando $j = 6$, a chamada irá considerar a posição 3, verificando se $A[3] == k - A[6]$. Ou seja, há muita repetição de testes. Será que esse pode ser o motivo do algoritmo levar tempo quadrático?

Agora, voltamos ao segundo passo, para tentar criar um algoritmo para o problema. Vamos então tentar resolver o problema diretamente, sem ajuda de algoritmos para outros problemas. Veja que a forma mais direta de resolvê-lo é acessar cada par possível de posições i e j, com $i < j$ para não haver repetição, verificando se a soma dos elementos nelas é k. O Algoritmo 6.3 formaliza essa ideia. Outro bom exercício é verificar, também com uma

invariante de laço, que ele corretamente resolve o problema.

Algoritmo 6.3: SomaK_v2(A, n, k)

```
1 para i = 1 até n − 1, incrementando faça
2     para j = i + 1 até n, incrementando faça
3         se A[i] + A[j] == k então
4             devolve i, j
5 devolve −1, −1
```

Novamente, considerando que esse algoritmo está correto, o próximo passo é verificar seu tempo de execução. Similar à versão anterior, vemos que o laço **para** externo, da linha 1, executa $O(n)$ vezes, e que resta descobrir quanto tempo leva uma execução do corpo do laço.

Considere então uma única execução do corpo do laço **para** externo. Veja que a linha 2 do laço **para** interno também executa $O(n)$ vezes, também porque j assume no máximo n valores diferentes (também não conseguimos dizer $\Theta(n)$ aqui). Assim, o corpo desse laço **para** interno executa $O(n)$ vezes. Como esse corpo leva tempo constante para ser executado (ele só tem um teste, uma soma e uma comparação – talvez um comando de retorno), então o tempo de uma execução do laço interno é $O(n)$. Considerando a discussão do parágrafo anterior, concluímos que SomaK_v2 leva tempo $O(n^2)$. Então, assintoticamente esse algoritmo não é melhor do que o anterior. Mas talvez na prática, ou em média, ele se saia melhor.

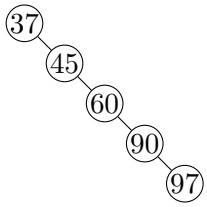
Talvez tenha te incomodado um pouco o fato de que a análise do laço **para** interno foi um tanto descuidada. Dissemos que j assume no máximo n valores diferentes e por isso limitamos o tempo do laço interno por $O(n)$. Mas veja que podemos fazer uma análise mais justa, porque j assume no máximo $n - i + 1$ valores em cada iteração do laço **para** externo. Como i varia entre 1 e $n-1$, no máximo, então a quantidade total valores que j assume, por toda execução do algoritmo, é no máximo

$$\begin{aligned}\sum_{i=1}^{n-1}(n-i+1) &= \left(\sum_{i=1}^{n-1} n\right) - \left(\sum_{i=1}^{n-1} i\right) + \left(\sum_{i=1}^{n-1} 1\right) \\ &= n(n-1) - \frac{n(n-1)}{2} + (n-1) = \frac{n^2}{2} + \frac{n}{2} - 1\,.\end{aligned}$$

Portanto, mesmo com essa análise mais justa, ainda chegamos à conclusão de que o tempo do algoritmo é $O(n^2)$.

Veja que não conseguimos dizer que o algoritmo é $\Theta(n^2)$, porque no melhor caso os elementos estão nas posições 1 e 2 e o algoritmo executa em tempo $\Theta(1)$.

Lembre-se sempre de se perguntar se não dá para fazer algo melhor. Para esse problema, é sim possível melhorar o tempo de execução do pior caso, para $O(n \log n)$. Até o fim deste livro você será capaz de pensar nessa solução melhor.



Recursividade

Você quis dizer: *recursividade*.

Google

Ao desenvolver um algoritmo, muitas vezes precisamos executar uma tarefa repetidamente, utilizando para isso estruturas de repetição **para** ou **enquanto**, ou precisamos tomar decisões condicionais, utilizando operações da forma "se ... senão ... então". Em geral, todas essas operações são rapidamente assimiladas pois fazem parte do cotidiano de qualquer pessoa, dado que muitas vezes precisamos tomar decisões condicionais ou executar tarefas repetidamente. Porém, para desenvolver alguns algoritmos é necessário fazer uso da *recursão*.

Qualquer função que chama a si mesma é recursiva, mas recursão é muito mais do que isso. Ela é uma técnica muito poderosa para solução de problemas. A ideia é reduzir uma instância em instâncias menores, do mesmo problema, que por sua vez também são reduzidas, e assim por diante, até que elas sejam tão pequenas que possam ser resolvidas diretamente.

Antes de continuarmos, é importante avisar que nesse livro, muitas vezes usaremos os termos "problema" e "subproblema" para nos referenciar a "uma instância do problema" e "uma instância menor do problema", respectivamente. Isso ficará claro pelo contexto.

Diversos problemas têm essa característica: toda instância contém uma instância menor do mesmo problema (estrutura recursiva). Mas veja que não basta apenas reduzir o tamanho da instância. Deve-se encontrar uma instância menor (ou até mais de uma) cuja solução ajude a instância maior a ser resolvida.

Vamos ver um exemplo concreto dessa discussão. Considere novamente o problema da busca de um elemento x em um vetor A que contém n elementos. Vamos nos referir a esse

problema como "x está em $A[1..n]$?". Inicialmente veja que temos um problema de tamanho n: é o tamanho do vetor e nos diz todas as possíveis posições em que x poderia estar. Note ainda que qualquer subvetor de A é também um vetor e, portanto, uma entrada válida para o problema. Por exemplo, "x está em $A[5..\frac{n}{2}]$?" é um problema válido e é um subproblema do nosso problema inicial: tem tamanho menor do que n. Também podemos dizer isso de "x está em $A[n-10..n]$?", ou "x está em $A[\frac{n}{4}..\frac{2n}{3}]$?". Assim, se nós temos um algoritmo que resolve o problema de buscar x em um vetor, esse algoritmo pode ser usado para resolver qualquer um desses subproblemas.

No entanto, de que adianta resolver esses subproblemas? Nosso objetivo era resolver "x está em $A[1..n]$?", e resolver esses subproblemas não parece ajudar muito nessa tarefa. O subproblema "x está em $A[1..n-1]$?", por outro lado, é mais útil, porque se sabemos resolvê-lo e sabemos a resposta para o teste "x é igual a $A[n]$?", então podemos combiná-las para resolver nosso problema original.

Agora, como procurar x em $A[1..n-1]$? Veja que podemos fazer exatamente o mesmo procedimento: desconsideramos o último elemento desse vetor para gerar um vetor menor, resolvemos o problema de procurar x nesse vetor menor, comparamos x com o último elemento, $A[n-1]$, e combinamos as respostas obtidas.

Como estamos sempre reduzindo o tamanho do vetor de entrada por uma unidade, em algum momento *obrigatoriamente* chegaremos a um vetor que não tem elemento nenhum. E nesse caso, a solução é bem simples: x não pode estar nele pois ele é vazio. Esse caso simples que pode ser resolvido diretamente é o que chamamos de *caso base*.

O Algoritmo 7.1 formaliza essa ideia.

Algoritmo 7.1: BuscaLinearRecursiva(A, n, x)

```
1 se n == 0 então
2     devolve −1
3 se A[n] == x então
4     devolve n
5 devolve BuscaLinearRecursiva(A, n − 1, x)
```

De forma geral, problemas que apresentam estrutura recursiva podem ser resolvidos com os seguintes passos:

(i) se a instância for suficientemente pequena, resolva-a diretamente (casos base),

(ii) caso contrário, divida a instância em instâncias menores, resolva-as usando os passos (i) e (ii) e combine as soluções, encontrando uma solução para a instância original.

7.1 Corretude de algoritmos recursivos

Por que a recursão funciona? Por que o algoritmo que vimos acima, da busca linear recursiva, realmente decide se x pertence a A para *qualquer* vetor A e valor x? Se nosso vetor é vazio, note que a chamada BuscaLinearRecursiva(A, 0, x) funciona corretamente, pois devolve -1. Se nosso vetor tem um elemento, quando fazemos uma chamada BuscaLinearRecursiva(A, 1, x), comparamos x com $A[1]$. Caso sejam iguais, a busca para e devolve 1; caso sejam diferentes, uma chamada BuscaLinearRecursiva(A, 0, x) é feita e acabamos de verificar que esta funciona corretamente. Se nosso vetor tem dois elementos, quando fazemos uma chamada BuscaLinearRecursiva(A, 2, x), comparamos x com $A[2]$. Caso sejam iguais, a busca para e devolve 2; caso sejam diferentes, uma chamada BuscaLinearRecursiva(A, 1, x) é feita e já verificamos que esta funciona. E assim por diante.

Verificar se o algoritmo está correto quando $n = 0$, depois quando $n = 1$, depois quando $n = 2$, etc., não é algo viável. Em primeiro lugar porque se $n = 1000$, essa tarefa se torna extremamente tediosa e longa. Em segundo lugar porque não sabemos o valor de n, de forma que essa tarefa é na verdade impossível! Acontece que não precisamos verificar *explicitamente* que o algoritmo funciona para cada valor de n que é dado na instância de entrada. Basta verificar se ele vale para o caso base (ou casos base) e que, supondo que ele vale para chamadas a valores menores do que n, verificar que ele valerá para n. Esse é o princípio da indução.

Em geral, mostramos que algoritmos recursivos estão corretos fazendo uma prova por indução no tamanho da entrada. No caso da BuscaLinearRecursiva, queremos mostrar que "para qualquer vetor de tamanho n, o algoritmo devolve a posição de x no vetor se ele existir, ou devolve -1". Quando $n = 0$, o algoritmo devolve -1 e, de fato, x não existe no vetor, que é vazio.

Considere então um $n > 0$ e suponha que para qualquer vetor de tamanho k, com $0 \leq k < n$, o algoritmo devolve a posição de x se ele existir, ou devolve -1. Na chamada para um vetor de tamanho n, o algoritmo começa comparando x com $A[n]$. Se forem iguais, o algoritmo devolve n, que é a posição de x em A. Se não forem, o algoritmo faz uma chamada a BuscaLinearRecursiva(A, $n-1$, x). Por hipótese, essa chamada corretamente devolve a posição de x em $A[1..n-1]$, se ele existir, ou devolve -1 se ele não existir em $A[1..n-1]$. Esse resultado combinado ao que já sabemos sobre $A[n]$ mostra que a chamada atual funciona corretamente.

Vamos analisar mais exemplos de algoritmos recursivos a seguir.

7.2 Fatorial de um número

Uma função bem conhecida na matemática é o fatorial de um inteiro não negativo n. O fatorial de um número n, denotado por $n!$, é definido como o produto de todos os inteiros entre 1 e n, se $n > 0$, isto é, $n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1$, e $0!$ é definido como sendo 1.

Note que dentro da solução de $n!$ temos a solução de problemas menores. Por exemplo, a solução para $3!$ está contida ali: $3 \cdot 2 \cdot 1$. Em particular, a solução para $(n-1)!$ também, que se resolvido, basta multiplicá-lo por n para obter a solução do problema original, $n!$. Assim, podemos escrever $n!$ da seguinte forma recursiva:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0\,. \end{cases}$$

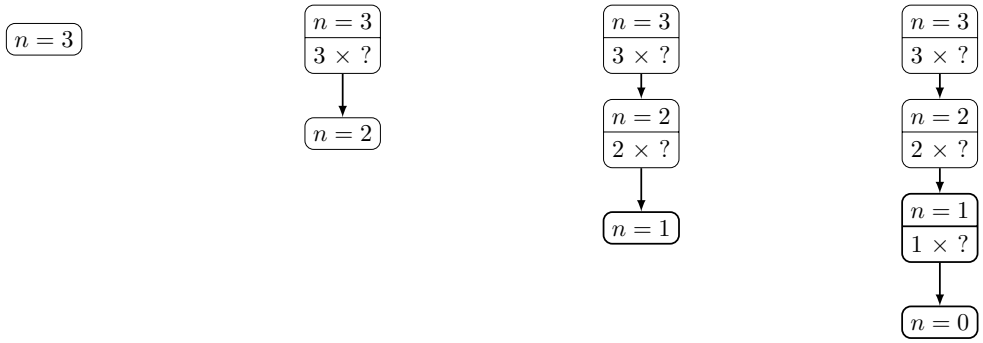
Essa definição diretamente inspira o Algoritmo 7.2, que é recursivo.

Algoritmo 7.2: FATORIAL(n)

1 **se** $n == 0$ **então**
2   **devolve** 1
3 **devolve** $n \times$ FATORIAL($n-1$)

Por exemplo, ao chamar "FATORIAL(3)", o algoritmo vai falhar no teste da linha 1 e vai executar a linha 3, fazendo "3 × FATORIAL(2)". Antes de poder retornar, é necessário calcular FATORIAL(2). Nesse ponto, o computador salva o estado atual na pilha de execução e faz uma chamada "FATORIAL(2)", que vai falhar no teste da linha 1 novamente e vai executar a linha 3para devolver "2 × FATORIAL(1)". Novamente, o estado atual é salvo na pilha de execução e uma chamada "FATORIAL(1)" é realizada. O teste da linha 2 falha novamente e a linha 3 será executada, para devolver "1 × FATORIAL(0)", momento em que a pilha de execução é novamente salva e a chamada "FATORIAL(0)" é feita. Essa chamada recursiva será a última, pois agora o teste da linha 1 dá verdadeiro e a linha 2 será executada, de forma que essa chamada simplesmente devolve o valor 1. Assim, a pilha de execução começa a ser desempilhada, e o resultado final devolvido pelo algoritmo será $3 \cdot (2 \cdot (1 \cdot 1))$. Veja a Figura 7.1 para um esquema de execução desse exemplo. A Figura 7.2 mostra a árvore de recursão completa.

Pelo exemplo descrito no parágrafo anterior, conseguimos perceber que a execução de um programa recursivo precisa salvar vários estados do programa ao mesmo tempo, de modo que isso pode aumentar a necessidade de memória consideravelmente. Por outro lado, em

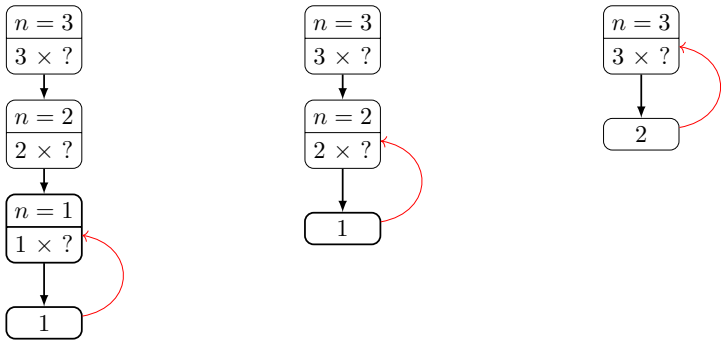


(a) Chama FATORIAL(3).

(b) $3 \neq 0$: chama FATORIAL(2).

(c) $2 \neq 0$: chama FATORIAL(1).

(d) $1 \neq 0$: chama FATORIAL(0).



(e) Caso base: devolve 1.

(f) Multiplica e devolve 1.

(g) Multiplica e devolve 2.

(h) Multiplica e devolve 6.

Figura 7.1: Exemplo de execução de FATORIAL(3) (Algoritmo 7.2). Cada nó é uma chamada ao algoritmo. Setas vermelhas indicam o retorno de uma função.

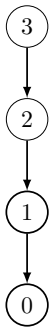


Figura 7.2: Árvore de recursão completa de FATORIAL(3) e POTENCIAV1(4,3). Cada nó é rotulado com o tamanho do problema (n) correspondente.

muitos problemas, uma solução recursiva é bem mais simples e intuitiva do que uma iterativa correspondente. Esses fatores também devem ser levados em consideração na hora de escolher qual algoritmo utilizar.

Vamos por fim provar por indução que FATORIAL está correto, isto é, que para qualquer valor $n \geq 0$, vale que FATORIAL(n) devolve $n!$ corretamente. Primeiro note que se $n = 0$, o algoritmo devolve 1. De fato, $0! = 1$, então nesse caso ele funciona corretamente.

Agora considere um $n > 1$. Vamos supor que FATORIAL(k) corretamente devolve $k!$ nos casos em que $0 \leq k < n$. Na chamada FATORIAL(n), o que o algoritmo faz é devolver a multiplicação de n por FATORIAL($n-1$). Como $n - 1 < n$, por hipótese sabemos que essa chamada recursiva corretamente devolve $(n-1)!$. Como $n \cdot (n-1)! = n!$, temos que a chamada atual funciona corretamente também.

7.3 Potência de um número

A n-ésima potência de um número x, denotada x^n, é definida como a multiplicação de x por si mesmo n vezes, onde assumimos que $x^0 = 1$:

$$x^n = \underbrace{x \cdot x \cdot x \cdot \cdots \cdot x}_{n \ fatores} .$$

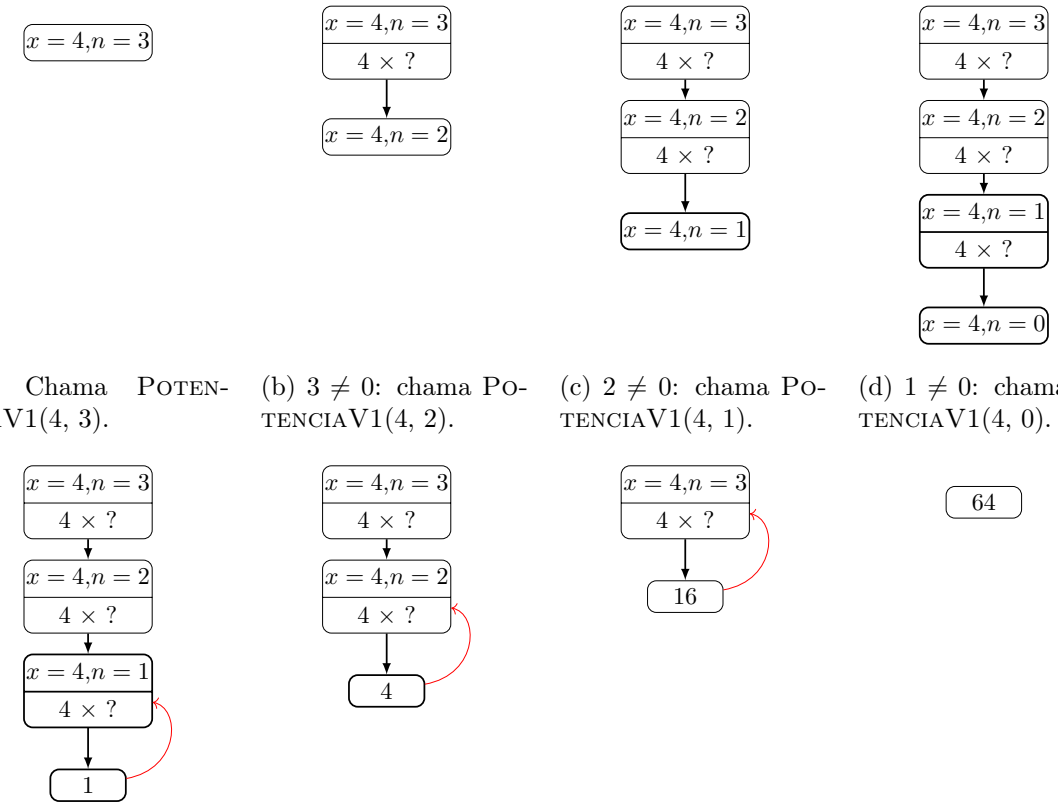
Será que é possível resolver o problema de calcular x^n de forma recursiva? Perceba que a resposta para essa pergunta é sim, pois o valor de x^n pode ser escrito como combinação de potências menores de x. De fato, note que x^n é o mesmo que x^{n-1} multiplicado por x. Com isso, podemos escrever x^n da seguinte forma recursiva:

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ x \cdot x^{n-1} & \text{se } n > 0 . \end{cases}$$

Essa definição sugere diretamente um algoritmo recursivo para calcular a n-ésima potência de um número, que é descrito no Algoritmo 7.3. A verificação de sua corretude por indução segue diretamente. Veja a Figura 7.3 para um exemplo simples de execução. A Figura 7.2 mostra a árvore de recursão completa.

Algoritmo 7.3: POTENCIAV1(x, n)

```
1 se n == 0 então
2     devolve 1
3 devolve x × PotenciaV1(x, n − 1)
```



(a) Chama PotenciaV1(4, 3).

(b) $3 \neq 0$: chama PotenciaV1(4, 2).

(c) $2 \neq 0$: chama PotenciaV1(4, 1).

(d) $1 \neq 0$: chama PotenciaV1(4, 0).

(e) Caso base: devolve 1.

(f) Multiplica e devolve 4.

(g) Multiplica e devolve 16.

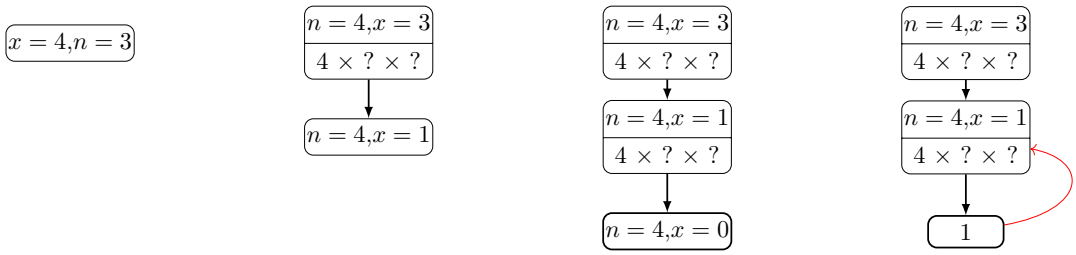
(h) Multiplica e devolve 64.

Figura 7.3: Exemplo de execução de PotenciaV1(4, 3) (Algoritmo 7.3). Cada nó é uma chamada ao algoritmo. Setas vermelhas indicam o retorno de uma função.

Existe ainda outra forma de definir x^n recursivamente, notando que $x^n = (x^{n/2})^2$ quando n é par e que $x^n = x \cdot (x^{(n-1)/2})^2$ quando n é ímpar. A verificação de sua corretude por indução também segue diretamente. Veja essa ideia formalizada no Algoritmo 7.4. A Figura 7.4 mostra um exemplo simples de execução enquanto a Figura 7.5 mostra a árvore de recursão completa.

Com base no esquema feito na Figura 7.5, podemos perceber que o Algoritmo 7.4 é muito ineficiente. Isso também pode ser visto na própria descrição do algoritmo onde, por exemplo, há duas chamadas PotenciaV2(x, $n/2$). Acontece que como esse é um algoritmo *determinístico*, a chamada PotenciaV2(x, $n/2$) sempre devolve o mesmo valor, de modo que não faz sentido realizá-la duas vezes. Uma versão mais eficiente do Algoritmo 7.4 é descrita no Algoritmo 7.5 e um exemplo de sua execução encontra-se na Figura 7.4.

Com esses dois algoritmos para resolver o problema da potência, qual deles é melhor? Qual

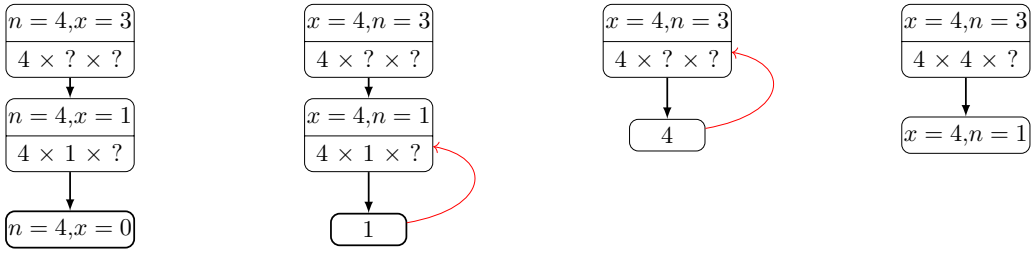


(a) Chama PotenciaV2(4, 3).

(b) $3 \neq 0$ e ímpar: chama PotenciaV2(4, 1).

(c) $1 \neq 0$ e ímpar: chama PotenciaV2(4, 0).

(d) Caso base: devolve 1.

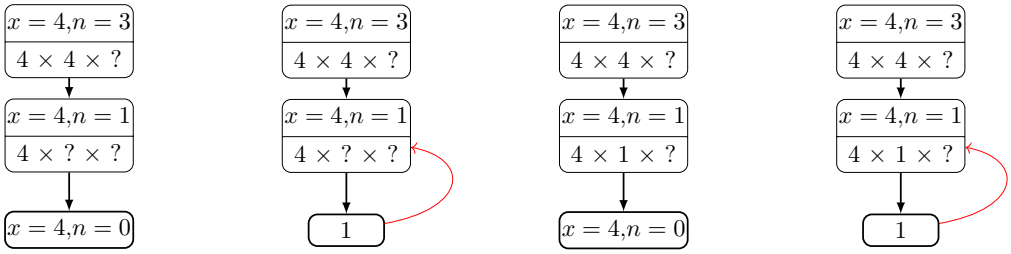


(e) Chama PotenciaV2(4, 0).

(f) Caso base: devolve 1.

(g) Multiplica e devolve 4.

(h) Chama PotenciaV2(4, 1).

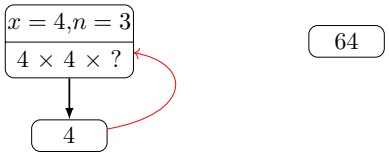


(i) $1 \neq 0$ e ímpar: chama PotenciaV2(4, 0).

(j) Caso base: devolve 1.

(k) Chama PotenciaV2(4, 0).

(l) Caso base: devolve 1.



(m) Multiplica e devolve 4.

(n) Multiplica e devolve 64.

Figura 7.4: Exemplo de execução de PotenciaV2(4, 3) (Algoritmo 7.4). Cada nó é uma chamada ao algoritmo. Setas vermelhas indicam o retorno de uma função.

Algoritmo 7.4: PotenciaV2(x, n)

```
1 se n == 0 então
2     devolve 1
3 se n é par então
4     devolve PotenciaV2(x, n/2) · PotenciaV2(x, n/2)
5 senão
6     devolve x · PotenciaV2(x, (n − 1)/2) · PotenciaV2(x, (n − 1)/2)
```

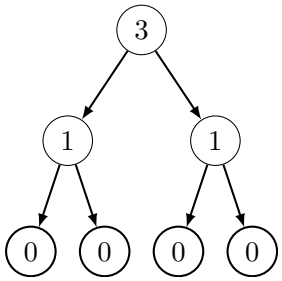


Figura 7.5: Árvore de recursão completa de PotenciaV2(4,3). Cada nó é rotulado com o tamanho do problema (n) correspondente.

Algoritmo 7.5: PotenciaV3(x, n)

```
1 se n == 0 então
2     devolve 1
3 se n é par então
4     aux = PotenciaV3(x, n/2)
5     devolve aux · aux
6 senão
7     aux = PotenciaV3(x, (n − 1)/2)
8     devolve x · aux · aux
```

é o mais eficiente? Agora que temos recursão envolvida, precisamos tomar alguns cuidados para calcular o tempo de execução desses algoritmos. No caso de PotenciaV1, podemos ver que o valor de n é decrescido de uma unidade a cada chamada, e que só não fazemos uma chamada recursiva quando $n = 0$. Assim, existem n chamadas recursivas sendo feitas. O tempo gasto dentro de uma única chamada é constante (alguns testes, multiplicações e atribuições). Assim, PotenciaV1(x, n) leva tempo $\Theta(n)$, que é exponencial no tamanho da entrada, que é $\log n$ bits. No caso de PotenciaV3, o valor de n é reduzido aproximadamente pela metade a cada chamada, e só não fazemos uma chamada recursiva quando $n = 0$. Assim, existem por volta de $\log n$ chamadas recursivas. Também temos que o tempo gasto dentro de uma única chamada é constante, de forma que PotenciaV3(x, n) leva tempo $\Theta(\log n)$, que é linear no tamanho da entrada. Com isso temos que PotenciaV3 é o mais eficiente dos dois.

Esses dois algoritmos são pequenos e não é difícil fazer sua análise da forma como fizemos acima. Isso infelizmente não é verdade sempre. Por outro lado, existe uma forma bem simples de analisar o tempo de execução de algoritmos recursivos, que será vista no próximo capítulo.

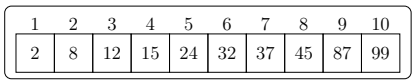
7.4 Busca binária

Considere o problema da busca em um vetor ordenado (ordem não-decrescente) A com n elementos. Se $A[\lfloor n/2 \rfloor] = x$, então a busca está encerrada. Caso contrário, se $x < A[\lfloor n/2 \rfloor]$, então basta verificar se o vetor $A[1..\lfloor n/2 \rfloor - 1]$ contém x. Isso é exatamente o mesmo problema (procurar x em um vetor), só que menor. Assim, essa busca pode ser feita recursivamente. Se $x > A[\lfloor n/2 \rfloor]$, devemos verificar recursivamente o vetor $A[\lfloor n/2 \rfloor + 1..n]$. Veja que apenas um dos três casos ocorre e que essa estratégia é a mesma da busca binária feita de forma iterativa. Podemos parar de realizar chamadas recursivas quando não há elementos no vetor. O Algoritmo 7.6 formaliza essa ideia. Para executá-lo basta fazer uma chamada BuscaBinariaRecursiva(A, 1, n, x). A Figura 7.6 mostra um exemplo de execução desse algoritmo enquanto a Figura 7.7 mostra a árvore de recursão completa para o mesmo exemplo.

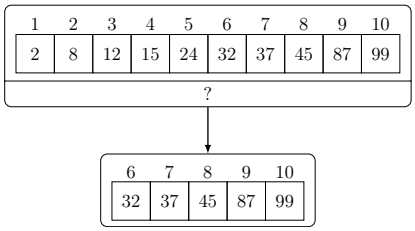
Para mostrar que BuscaBinariaRecursiva está correto, podemos fazer uma prova por indução no tamanho do vetor. Especificamente, queremos mostrar que se $n = dir - esq + 1$, então BuscaBinariaRecursiva(A, esq, dir, x) devolve -1 se x não está em $A[esq..dir]$ e devolve i se $A[i] = x$, para qualquer valor de n.

Quando $n = 0$, veja que $0 = dir - esq + 1$ implica em $dir + 1 = esq$, ou seja, $esq > dir$. Veja que nesse caso, o algoritmo devolve -1, o que é o esperado, pois $A[esq..dir]$ é vazio e certamente não contém x.

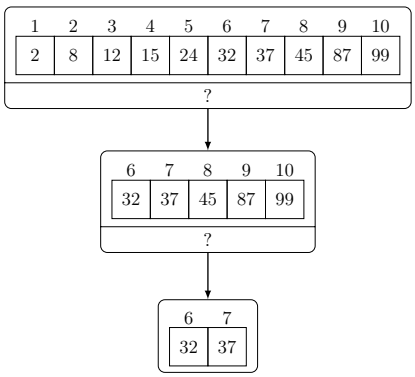
Considere agora algum $n > 0$ e suponha que o algoritmo corretamente decide se x pertence



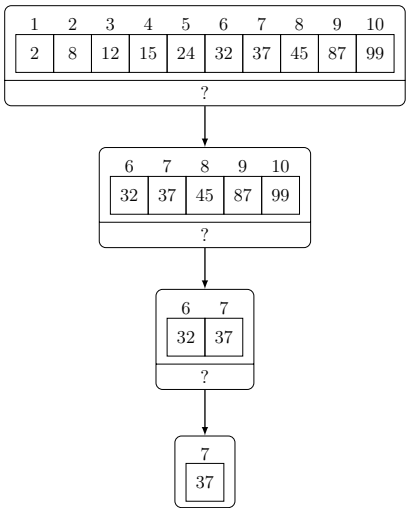
(a) BuscaBinariaRecursiva$(A, 1, 10, 37)$.



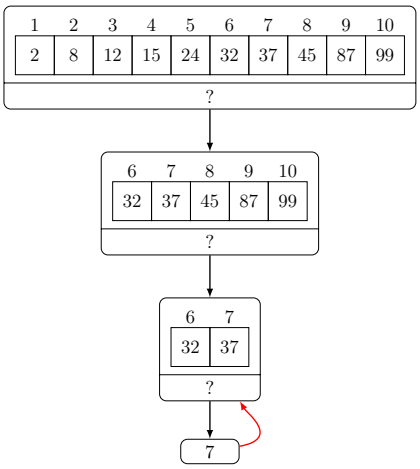
(b) $A[5] < 37$: chama BuscaBinariaRecursiva$(A, 6, 10, 37)$.



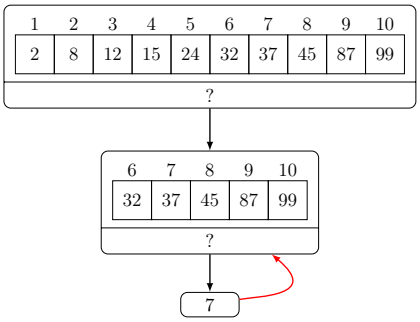
(c) $A[8] > 37$: chama BuscaBinariaRecursiva$(A, 6, 7, 37)$.



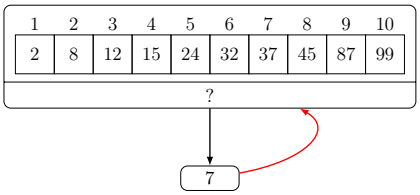
(d) $A[6] < 37$: chama BuscaBinariaRecursiva$(A, 7, 7, 37)$.



(e) $A[7] = 37$, devolve 7.



(f) Fim de BuscaBinariaRecursiva$(A, 6, 7, 37)$: devolve 7.



(g) Fim de BuscaBinariaRecursiva$(A, 6, 10, 37)$: devolve 7.

(h) Fim de BuscaBinariaRecursiva$(A, 1, 10, 37)$: devolve 7.

Figura 7.6: Exemplo de execução de BuscaBinariaRecursiva$(A, 1, 10, 37)$ (Algoritmo 7.6) para $A = (2, 8, 12, 15, 24, 32, 37, 45, 87, 99)$. Cada nó é uma chamada ao algoritmo. Setas vermelhas indicam o retorno de uma função.

Algoritmo 7.6: BuscaBinariaRecursiva(A, esq, dir, x)

```
1 se esq > dir então
2     devolve −1
3 meio = ⌊(esq + dir)/2⌋
4 se A[meio] == x então
5     devolve meio
6 senão se x < A[meio] então
7     devolve BuscaBinariaRecursiva(A, esq, meio − 1, x)
8 senão
9     devolve BuscaBinariaRecursiva(A, meio + 1, dir, x)
```

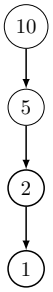


Figura 7.7: Árvore de recursão completa de BuscaBinariaRecursiva(A, 1, 10, 37) para $A = (2, 8, 12, 15, 24, 32, 37, 45, 87, 99)$. Cada nó é rotulado com o tamanho do problema $(dir - esq + 1)$ correspondente.

a qualquer vetor de tamanho k, para $0 \leq k < n$. Na chamada em que $n = dir - esq_1$, como $n > 0$, temos que $dir + 1 > esq$, ou $dir \geq esq$. Então o que o algoritmo faz inicialmente é calcular $meio = \lfloor (esq + dir)/2 \rfloor$ e comparar $A[meio]$ com x. Se $A[meio] = x$, então o algoritmo devolve $meio$ e, portanto, funciona corretamente nesse caso.

Se $A[meio] > x$, ele devolve o mesmo que a chamada recursiva BuscaBinariaRecursiva(A, esq, $meio - 1$, x) devolver. Note que

$$meio - 1 - esq + 1 = \left\lfloor \frac{esq + dir}{2} \right\rfloor - esq \leq \frac{esq + dir}{2} - esq = \frac{dir - esq}{2} = \frac{n-1}{2}$$

e que $(n-1)/2 < n$ sempre que $n > -1$. Então, podemos usar a hipótese de indução, de forma que a chamada BuscaBinariaRecursiva(A, esq, $meio - 1$, x) corretamente detecta se x pertence a $A[esq..meio - 1]$. Como o vetor está ordenado e $x < A[meio]$, a chamada atual também detecta se x pertence a $A[esq..dir]$. Se $A[meio] < x$ a análise é similar.

Como esse procedimento analisa, passo a passo, somente metade do tamanho do vetor do passo anterior, temos que, no pior caso, $\log n$ chamadas recursivas serão realizadas. Assim, como cada chamada recursiva leva tempo constante, o algoritmo tem tempo $O(\log n)$. Veja que não podemos utilizar a notação Θ, pois o algoritmo pode parar antes de realizar as no máximo $\log n$ chamadas recursivas e, por isso, nosso limite superior é da ordem de $\log n$, mas o limite superior é 1.

7.5 Algoritmos recursivos $\times$ algoritmos iterativos

Quando utilizar um algoritmo recursivo ou um algoritmo iterativo? Vamos discutir algumas vantagens e desvantagens de cada tipo de procedimento.

A utilização de um algoritmo recursivo tem a vantagem de, em geral, ser simples e oferecer códigos claros e concisos. Assim, alguns problemas que podem parecer complexos de início, acabam tendo uma solução simples e elegante, enquanto que algoritmos iterativos longos requerem experiência por parte do programador para serem entendidos. Por outro lado, uma solução recursiva pode ocupar muita memória, dado que o computador precisa manter vários estados do algoritmo gravados na pilha de execução do programa. Muitas pessoas acreditam que algoritmos recursivos são, em geral, mais lentos do que algoritmos iterativos para o mesmo problema, mas a verdade é que isso depende muito do compilador utilizado e do problema em si. Alguns compiladores conseguem lidar de forma rápida com as chamadas a funções e com o gerenciamento da pilha de execução.

Algoritmos recursivos eliminam a necessidade de se manter o controle sobre diversas variáveis que possam existir em um algoritmo iterativo para o mesmo problema. Porém,

pequenos erros de implementação podem levar a infinitas chamadas recursivas, de modo que o programa não encerraria sua execução.

Nem sempre a simplicidade de um algoritmo recursivo justifica o seu uso. Um exemplo claro é dado pelo problema de se calcular termos da sequência de Fibonacci, que é a sequência infinita de números 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... Por definição, o n-ésimo número da sequência, escrito como F_n, é dado por

$$F_n = \begin{cases} 1 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ F_{n-1} + F_{n-2} & \text{se } n > 2\,. \end{cases} \tag{7.1}$$

Não é claro da definição, mas F_{30} é maior do que 1 milhão, F_{100} é um número com 21 dígitos e, em geral, $F_n \approx 2^{0.684n}$. Ou seja, F_n é um valor exponencial em n.

Problema 7.1: *Número de Fibonacci*

Dado um inteiro $n \geq 0$, encontrar F_n.

O Algoritmo 7.7 calcula recursivamente F_n para um n dado como entrada.

Algoritmo 7.7: FibonacciRecursivo(n)

```
1 se n ≤ 2 então
2     devolve 1
3 devolve FibonacciRecursivo(n − 1) + FibonacciRecursivo(n − 2)
```

Apesar de sua simplicidade, o procedimento acima é muito ineficiente e isso pode ser percebido na Figura 7.8, onde vemos que muito trabalho é repetido. Seja $T(n)$ o tempo necessário para computar F_n. É possível mostrar que $T(n) = O(2^n)$ e $T(n) = \Omega(\sqrt{2}^n)$, ou seja, o tempo é exponencial em n. Na prática, isso significa que se tivermos um computador que executa 4 bilhões de instruções por segundo (nada que os computadores existentes não possam fazer), levaria menos de 1 segundo para calcular F_{10} e cerca de 10^{21} milênios para calcular F_{200}. Mesmo se o computador fosse capaz de realizar 40 trilhões de instruções por segundo, ainda precisaríamos de cerca de $5 \cdot 10^{17}$ milênios para calcular F_{200}. De fato, quando FibonacciRecursivo($n-1$) + FibonacciRecursivo($n-2$) é executado, além da chamada a FibonacciRecursivo($n-2$) que é feita, a chamada a FibonacciRecursivo($n-1$) fará mais uma chamada a FibonacciRecursivo($n-2$), mesmo que ele já tenho sido calculado antes, e esse fenômeno cresce exponencialmente até chegar à base da recursão.

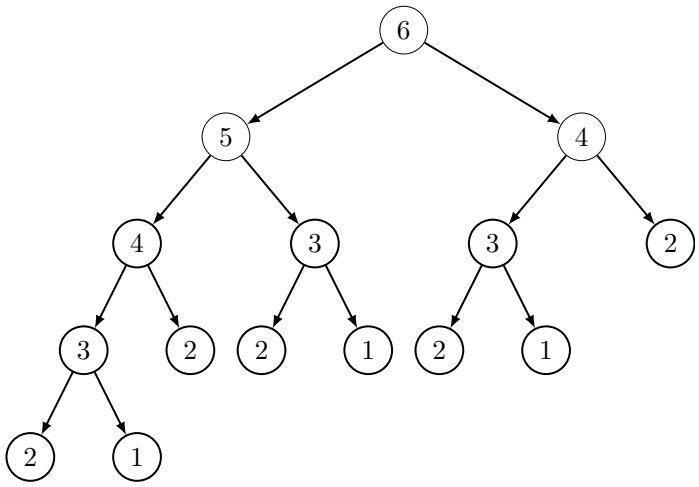


Figura 7.8: Árvore de recursão completa de FibonacciRecursivo(6) (Algoritmo 7.7). Cada nó representa uma chamada ao algoritmo e é rotulado com o tamanho do problema correspondente. Note como várias chamadas são repetidas.

É possível implementar um algoritmo iterativo simples que resolve o problema do número de Fibonacci e que é executado em tempo polinomial. Na prática, isso significa que os mesmos dois computadores mencionados acima conseguem calcular F_{200} e mesmo $F_{1000000}$ em menos de 1 segundo. Para isso, basta utilizar um vetor, como mostra o Algoritmo 7.8.

Esse exemplo clássico também mostra como as estruturas de dados podem ter grande impacto na análise de algoritmos. Na Parte III veremos várias estruturas de dados que devem ser de conhecimento de todo bom desenvolvedor de algoritmos. Na Parte IV apresentamos diversos algoritmos recursivos para resolver o problema de ordenação dos elementos de um vetor. Ao longo deste livro muitos outros algoritmos recursivos serão discutidos.

Algoritmo 7.8: Fibonacci(n)

1 **se** $n \leq 2$ **então**
2 ⌊ **devolve** 1
3 Seja $F[1..n]$ um vetor de tamanho n
4 $F[1] = 1$
5 $F[2] = 1$
6 **para** $i = 3$ *até* n, *incrementando* **faça**
7 ⌊ $F[i] = F[i-1] + F[i-2]$
8 **devolve** $F[n]$

Capítulo 8

Recorrências

Relações como $f(n) = f(n-1) + f(n-2)$, $T(n) = 2T(n/2) + n$ ou $T(n) \leq T(n/3) + T(n/4) + 3\log n$ são chamadas de *recorrências*, que são equações ou inequações que descrevem uma função em termos de seus valores para instâncias menores. Recorrências são muito comuns e úteis para descrever o tempo de execução de algoritmos recursivos. Portanto, elas são compostas de duas partes que indicam, respectivamente, o tempo gasto quando não há recursão (caso base) e o tempo gasto quando há recursão, que consiste no tempo das chamadas recursivas juntamente com o tempo gasto no restante da chamada atual.

Por exemplo, considere novamente os algoritmos PotenciaV1 e PotenciaV3 definidos na Seção 7.3 para resolver o problema de calcular x^n, a n-ésima potência de x. Se usamos $T(n)$ para representar o tempo de execução de PotenciaV1(x, n), então

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{caso contrário}\,, \end{cases}$$

onde, no caso em que $n > 0$, $T(n-1)$ se refere ao tempo gasto na execução de PotenciaV1(x, $n-1$) e $\Theta(1)$ se refere ao tempo gasto no restante da chamada atual, onde fazemos um teste, uma chamada a função, uma multiplicação e uma operação de retorno[1].

[1] Lembre-se que uma chamada a uma função leva tempo constante, mas executá-la leva outro tempo.

De forma similar, se $T'(n)$ representa o tempo de execução de PotenciaV3(x, n), então

$$T'(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T'(\frac{n}{2}) + \Theta(1) & \text{se } n > 0 \text{ e } n \text{ é par} \\ T'(\frac{n-1}{2}) + \Theta(1) & \text{se } n > 0 \text{ e } n \text{ é ímpar}\,. \end{cases}$$

Também podemos descrever o tempo de execução do Algoritmo 7.7, FibonacciRecursivo, utilizando uma recorrência:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 2 \\ T(n-1) + T(n-2) + O(n) & \text{caso contrário}\,, \end{cases}$$

onde, no caso em que $n > 2$, $T(n-1)$ se refere ao tempo de execução de FibonacciRecursivo($n-1$), $T(n-2)$ se refere ao tempo de execução de FibonacciRecursivo($n-2$) e $O(n)$ se refere ao tempo gasto no restante da chamada, onde fazemos duas chamadas a funções, uma comparação, uma operação de retorno (estes todos em tempo constante) e a soma de F_{n-1} com F_{n-2}, pois já vimos que esses valores são da ordem de 2^{n-1} e 2^{n-2}, respectivamente. Somar dois números tão grandes pode levar um tempo muito maior, proporcional à quantidade de bits necessários para armazená-los[2].

Em geral, o tempo gasto nos casos base dos algoritmos é constante ($\Theta(1)$), de forma que é comum descrevemos apenas a segunda parte. Por exemplo, descrevemos o tempo de execução $T(n)$ do Algoritmo 7.6, BuscaBinariaRecursiva, apenas como $T(n) \leq T(n/2) + \Theta(1)$.

Acontece que informações do tipo "o tempo de execução do algoritmo é $T(n/3)+T(n/4)+\Theta(n)$" não nos dizem muita coisa. Gostaríamos portanto de *resolver* a recorrência, encontrando uma expressão que não depende da própria função, para que de fato possamos observar sua taxa de crescimento.

Neste capítulo apresentaremos quatro métodos para resolução de recorrências: (i) substituição, (ii) iterativo, (iii) árvore de recursão e (iv) mestre. Cada método se adequa melhor dependendo do formato da recorrência e, por isso, é importante saber bem quando usar cada um deles. Como recorrências são funções definidas recursivamente em termos de si mesmas para valores menores, se expandirmos recorrências até que cheguemos ao caso base da recursão, muitas vezes teremos realizado uma quantidade logarítmica de passos recursivos. Assim, é natural que termos logarítmicos apareçam durante a resolução de recorrências. Somatórios dos tempos de execução realizados fora das chamadas recursivas também irão aparecer.

[2]Nesses casos, certamente essa quantidade de bits será maior do que 32 ou 64, que é a quantidade máxima que os computadores reais costumam conseguir manipular em operações básicas.

Assim, pode ser útil revisar esses conceitos antes (veja a Parte I)

8.1 Método da substituição

Esse método consiste em provar por indução matemática que uma recorrência $T(n)$ é limitada (inferiormente e/ou superiormente) por alguma função $f(n)$. Um ponto importante é que é necessário que se saiba qual é a função $f(n)$ de antemão. O método da árvore de recursão, descrito mais adiante (veja Seção 8.3), pode fornecer uma estimativa para $f(n)$.

Outro ponto importante é que no passo indutivo é necessário provar **exatamente** o que foi suposto, com a mesma constante. Por exemplo, quando nos propomos a provar por indução que $T(n) \leq f(n)$, *precisamos provar que* $T(n) \leq f(n)$, e não está correto quando provamos que $T(n) \leq f(n) + 10$ ou $T(n) \leq f(n) + g(n)$. Esse é um ponto de muita confusão nesse método, porque acabamos misturando os conceitos de notação assintótica. Como $f(n) + 10 = O(f(n))$ e, se $g(n) = O(f(n))$, então $f(n) + g(n) = O(f(n))$, acabamos achando que a prova acabou. Mas veja que não nos propusemos a provar que $T(n) \leq O(f(n))$, e sim que $T(n) \leq f(n)$.

Considere um algoritmo com tempo de execução $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$. Por simplicidade, vamos assumir agora que n é uma potência de 2. Logo, podemos considerar $T(n) = 2T(n/2) + n$, pois temos que $n/2^i$ é um inteiro, para todo i com $1 \leq i \leq \log n$.

Nossa intuição inicial é que $T(n) = O(n^2)$. Uma forma de mostrar isso é provando que

$$\text{existem constantes } c \text{ e } n_0 \text{ tais que, se } n \geq n_0, \text{ então } T(n) \leq cn^2, \tag{8.1}$$

e essa expressão sim pode ser provada por indução (novamente, precisamos mostrar que $T(n) \leq cn^2$ e não que $T(n) \leq (c+1)n^2$ ou outra variação). Via de regra assumiremos $T(1) = 1$, a menos que indiquemos algo diferente. Durante a prova, ficará claro quais os valores de c e n_0 necessários para que (8.1) aconteça.

Comecemos considerando $n = 1$ como caso base. Como $T(1) = 1$ e queremos que $T(1) \leq c1^2$, precisamos ter $c \geq 1$ para esse caso valer. Agora suponha que para qualquer m, com $1 \leq m < n$, temos que $T(m) \leq cm^2$. Precisamos mostrar que $T(n) \leq cn^2$ quando $n > 1$. Veja que só sabemos que $T(n) = 2T(n/2) + n$. Porém, $n/2 < n$ quando $n > 1$, o que significa

que podemos usar a hipótese de que $T(m) \leq cm^2$ para $m < n$, chegando a

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &\leq 2\left(c\left(\frac{n}{2}\right)^2\right) + n \\ &= 2\left(c\frac{n^2}{4}\right) + n \\ &= c\frac{n^2}{2} + n \\ &\leq cn^2\,, \end{aligned}$$

onde a última desigualdade vale sempre que $c(n^2/2) + n \leq cn^2$, o que é verdade quando $c \geq 2/n$. Como $2/n \leq 2$ para qualquer valor de $n \geq 1$, basta escolher um valor para c que seja sempre maior ou igual a 2 (assim, será sempre maior ou igual à expressão $2/n$). Juntando isso com o fato de que o caso base precisa que $c \geq 1$ para funcionar, escolhemos $c = 4$ (ou podemos escolher $c = 2$, $c = 3$, $c = 5$, ...). Todas as nossas conclusões foram possíveis para todo $n \geq 1$. Com isso, mostramos por indução em n que $T(n) \leq 4n^2$ sempre que $n \geq n_0 = 1$, de onde concluímos que $T(n) = O(n^2)$.

Há ainda uma pergunta importante a ser feita: será que é possível provar um limitante superior assintótico melhor que n^2?[3] Por exemplo, será que se $T(n) = 2T(n/2) + n$, então temos $T(n) = O(n \log n)$?

Novamente, utilizaremos o método da substituição. Uma forma de mostrar que $T(n) = O(n \log n)$ é provando que

existem constantes c e n_0 tais que, se $n \geq n_0$, então $T(n) \leq cn \log n$.

Lembre que assumimos $T(1) = 1$. Quando $n = 1$, como $T(1) = 1$ e queremos que $T(1) \leq c\,1 \log 1 = 0$, temos um problema. Porém, em análise assintótica estamos preocupados somente com valores suficientemente grandes de n (maiores do que um n_0). Consideremos então $n = 2$ como nosso caso base. Temos $T(2) = 2T(1) + 2 = 4$ e queremos que $T(2) \leq c\,2 \log 2$. Nesse caso, sempre que $c \geq 2$ o caso base valerá.

Suponha agora que, para algum m, com $2 \leq m < n$, temos que vale $T(m) \leq cm \log m$. Precisamos mostrar que $T(n) \leq cn \log n$ quando $n > 2$. Sabemos apenas que $T(n) = 2T(n/2) + n$, mas podemos usar a hipótese de que $T(n/2) \leq c(n/2)\log(n/2)$, pois $n/2 < n$

[3] Aqui queremos obter um limitante $f(n)$ tal que $f(n) = o(n^2)$.

quando $n > 2$. Temos então

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&\leq 2\left(c\frac{n}{2}\log\frac{n}{2}\right) + n \\
&= cn\log\frac{n}{2} + n \\
&= cn(\log n - \log 2) + n \\
&= cn\log n - cn + n \\
&\leq cn\log n\,,
\end{aligned}$$

onde a última inequação vale sempre que $-cn + n \leq 0$, o que é verdade sempre que $c \geq 1$. Como o caso base precisa que $c \geq 2$ para funcionar e o passo precisa que $c \geq 1$, podemos escolher, por exemplo, $c = 3$. Com isso, mostramos que $T(n) \leq 3n\log n$ sempre que $n \geq n_0 = 2$, de onde concluímos que $T(n) = O(n\log n)$.

O fato de que se $T(n) = 2T(n/2) + n$, então temos $T(n) = O(n\log n)$, não indica que não podemos diminuir ainda mais esse limite. Para garantir que a ordem de grandeza de $T(n)$ é $n\log n$, precisamos mostrar que $T(n) = \Theta(n\log n)$. Para isso, uma vez que temos o resultado com a notação O, resta mostrar que $T(n) = \Omega(n\log n)$. Vamos então utilizar o método da substituição para mostrar que $T(n) \geq cn\log n$ sempre que $n \geq n_0$, para constantes c e n_0.

Considerando $n = 1$, temos que $T(1) = 1$ e $1 \geq c\ 1\log 1$ qualquer que seja o valor de c pois $1 > 0$. Suponha agora que para todo m, com $1 \leq m < n$, temos $T(m) \geq cm\log m$. Assim,

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&\geq 2\left(c\frac{n}{2}\log\frac{n}{2}\right) + n \\
&= cn(\log n - \log 2) + n \\
&= cn\log n - cn + n \\
&\geq cn\log n\,,
\end{aligned}$$

onde a última inequação vale sempre que $-cn + n \geq 0$, o que é verdade sempre que $c \leq 1$. Portanto, escolhendo $c = 1$ e $n_0 = 1$, mostramos que $T(n) = \Omega(n\log n)$.

8.1.1 Diversas formas de obter o mesmo resultado

Podem existir diversas formas de encontrar um limitante assintótico utilizando indução. Lembre-se que anteriormente, mostrar que $T(n) = O(n\log n)$, escolhemos mostrar que

$T(n) \leq dn \log n$. Mostraremos agora que $T(n) = O(n \log n)$ provando que $T(n) \leq c(n \log n + n)$.

A base da indução nesse caso é $T(1) = 1$, e vemos que $T(1) \leq c(1 \log 1 + 1)$ sempre que $c \geq 1$. Suponha agora que para todo m, com $1 \leq m < n$, temos $T(m) \leq c(m \log m + m)$. Assim,

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&\leq 2c\left(\frac{n}{2}\log\frac{n}{2} + \frac{n}{2}\right) + n \\
&= cn \log\frac{n}{2} + cn + n \\
&= cn(\log n - \log 2) + cn + n \\
&= cn \log n - cn + cn + n \\
&= cn \log n + n \\
&\leq cn \log n + cn \\
&= c(n \log n + n)\,,
\end{aligned}$$

onde a penúltima inequação vale quando $c \geq 1$. Para que o passo e o caso base valham, tomamos $c = 1$ e $n_0 = 1$. Assim, $T(n) \leq n \log n + n$. Como $n \log n + n \leq n \log n + n \log n = 2n \log n$, temos que $n \log n + n = O(n \log n)$, o que significa que $T(n) = O(n \log n)$.

8.1.2 Ajustando os palpites

Algumas vezes quando queremos provar que $T(n) = O(f(n))$ (ou $T(n) = \Omega(f(n))$) para alguma função $f(n)$, podemos ter problemas para obter êxito caso nosso palpite esteja errado. Porém, como visto na seção anterior, existem diversas formas de se obter o mesmo resultado. Assim, é possível que de fato $T(n) = O(f(n))$ (ou $T(n) = \Omega(f(n))$) mas que o palpite para mostrar tal resultado precise de um leve ajuste.

Considere $T(n) = 3T(n/3) + 1$. Podemos imaginar que esse é o tempo de execução de um algoritmo recursivo sobre um vetor que a cada chamada divide o vetor em 3 partes de tamanho $n/3$, fazendo três chamadas recursivas sobre estes, e o restante não envolvido nas chamadas recursivas é realizado em tempo constante. Assim, temos a impressão de estar "visitando" cada elemento do vetor uma única vez, de forma que um bom palpite é que $T(n) = O(n)$. Para mostrar que o palpite está correto, vamos tentar provar que $T(n) \leq cn$

para alguma constante positiva c, por indução em n. No passo indutivo, temos

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{3}\right) + 1 \\ &\leq 3\left(c\frac{n}{3}\right) + 1 \\ &= cn + 1\,, \end{aligned}$$

o que não prova o que desejamos, pois para completar a prova por indução precisamos mostrar que $T(n) \leq cn$ (e não $cn + 1$, como foi feito), e claramente $cn + 1$ não é menor ou igual a cn.

Acontece que é verdade que $T(n) = O(n)$, mas o problema é que a expressão que escolhemos para provar nosso palpite não foi "forte" o suficiente. Podemos perceber isso também pois na tentativa falha de provar $T(n) \leq cn$, sobrou apenas uma constante ($cn + 1$). Como corriqueiro em provas por indução, precisamos fortalecer a hipótese indutiva.

Vamos tentar agora provar que $T(n) \leq cn - d$, onde c e d são constantes. Note que provando isso estaremos provando que $T(n) = O(n)$ de fato, pois $cn - d \leq cn$ e $cn = O(n)$. No passo indutivo, temos

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{3}\right) + 1 \\ &\leq 3\left(c\frac{n}{3} - d\right) + 1 \\ &= cn - 3d + 1 \\ &\leq cn - d\,, \end{aligned}$$

onde o último passo vale sempre que $-3d + 1 \leq -d$, e isso é verdade sempre que $d \geq 1/2$. Assim, como no caso base ($n = 1$) temos $T(1) = 1 \leq c - d$ sempre que $c \geq d + 1$, podemos escolher $d = 1/2$ e $c = 3/2$ para finalizar a prova que $T(n) = O(n)$.

8.1.3 Desconsiderando pisos e tetos

Vimos que $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n = \Theta(n \log n)$ sempre que n é uma potência de 2. Mostraremos a seguir que geralmente podemos assumir que n é uma potência de 2 (ou uma potência conveniente para a recorrência em questão), de modo que em recorrências do tipo $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$ não há perda de generalidade ao desconsiderar pisos e tetos.

Suponha que $n \geq 3$ não é uma potência de 2 e considere a recorrência $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$. Como n não é uma potência de 2, então deve existir um inteiro $k \geq 2$ tal que $2^{k-1} < n < 2^k$. Como o tempo de execução cresce com o crescimento da instância de

entrada, podemos assumir que $T(2^{k-1}) \leq T(n) \leq T(2^k)$. Já provamos que $T(n) = \Theta(n \log n)$ no caso em que n é potência de 2. Em particular, $T(2^k) \leq d2^k \log(2^k)$ para alguma constante d e $T(2^{k-1}) \geq d'2^{k-1} \log(2^{k-1})$ para alguma constante d'. Assim,

$$\begin{aligned} T(n) \leq T(2^k) &\leq d2^k \log(2^k) \\ &= (2d)2^{k-1} \log(2 \cdot 2^{k-1}) \\ &< (2d)n(\log 2 + \log n) \\ &< (2d)n(\log n + \log n) \\ &= (4d)n \log n\,. \end{aligned}$$

Similarmente,

$$\begin{aligned} T(n) \geq T(2^{k-1}) &\geq d'2^{k-1} \log(2^{k-1}) \\ &= \frac{d'}{2} 2^k (\log(2^k) - 1) \\ &> \frac{d'}{2} n \left(\log n - \frac{9 \log n}{10} \right) \\ &= \left(\frac{d'}{20} \right) n \log n\,. \end{aligned}$$

Como existem constantes $d'/20$ e $4d$ tais que para todo $n \geq 3$ temos $(d'/20)n \log n \leq T(n) \leq (4d)n \log n$, então $T(n) = \Theta(n \log n)$ quando n não é potência de 2. Logo, é suficiente considerar somente valores de n que são potências de 2.

Análises semelhantes funcionam para a grande maioria das recorrências consideradas em análises de tempo de execução de algoritmos. Em particular, é possível mostrar que podemos desconsiderar pisos e tetos em recorrências do tipo $T(n) = a(T(\lfloor n/b \rfloor) + T(\lceil n/c \rceil)) + f(n)$ para constantes $a > 0$ e $b, c > 1$.

Portanto, geralmente vamos assumir que n é potência de algum inteiro positivo, sempre que for conveniente para a análise, de modo que em geral desconsideraremos pisos e tetos.

8.1.4 Mais exemplos

Discutiremos agora alguns exemplos que nos ajudarão a entender todas as particularidades que podem surgir na aplicação do método da substituição.

Exemplo 1. $T(n) = 4T(n/2) + n^3$.

Vamos provar que $T(n) = \Theta(n^3)$. Primeiramente, mostraremos que $T(n) = O(n^3)$ e, para isso, vamos provar que $T(n) \leq cn^3$ para alguma constante apropriada c.

Note que $T(1) = 1 \leq c \cdot 1^3$ desde que $c \geq 1$. Suponha que $T(m) \leq cm^3$ para todo $2 \leq m < n$. Assim, temos que

$$\begin{aligned}
T(n) &= 4T\left(\frac{n}{2}\right) + n^3 \\
&\leq 4\left(c\left(\frac{n}{2}\right)^3\right) + n^3 \\
&= 4c\left(\frac{n^3}{8}\right) + n^3 \\
&= c\frac{n^3}{2} + n^3 \\
&\leq cn^3\,,
\end{aligned}$$

onde a última desigualdade vale sempre que $c \geq 2$. Portanto, fazendo $c = 2$ (ou qualquer valor maior), acabamos de provar por indução que $T(n) \leq cn^3 = O(n^3)$.

Para provar que $T(n) = \Omega(n^3)$, vamos provar que $T(n) \geq dn^3$ para algum d apropriado. Primeiro note que $T(1) = 1 \geq d \cdot 1^3$ desde que $d \leq 1$. Suponha que $T(m) \geq dm^3$ para todo $2 \leq m < n$. Assim, temos que

$$\begin{aligned}
T(n) &= 4T\left(\frac{n}{2}\right) + n^3 \\
&\geq \frac{4dn^3}{8} + n^3 \\
&\geq dn^3\,,
\end{aligned}$$

onde a última desigualdade vale sempre que $d \leq 2$. Portanto, fazendo $d = 1$, acabamos de provar por indução que $T(n) \geq dn^3 = \Omega(n^3)$.

Exemplo 2. $T(n) = 4T(n/16) + 5\sqrt{n}$.

Comecemos provando que $T(n) \leq c\sqrt{n}\log n$ para um c apropriado. Assumimos que $n \geq 16$. Para o caso base temos $T(16) = 4+5\sqrt{16} = 24 \leq c\sqrt{16}\log 16$, onde a última desigualdade vale sempre que $c \geq 3/2$. Suponha que $T(m) \leq c\sqrt{m}\log m$ para todo $16 \leq m < n$. Assim,

$$\begin{aligned}
T(n) &= 4T\left(\frac{n}{16}\right) + 5\sqrt{n} \\
&\leq 4\left(c\frac{\sqrt{n}}{\sqrt{16}}(\log n - \log 16)\right) + 5\sqrt{n} \\
&= c\sqrt{n}\log n - 4c\sqrt{n} + 5\sqrt{n} \\
&\leq c\sqrt{n}\log n\,,
\end{aligned}$$

onde a última desigualdade vale se $c \geq 5/4$. Como $3/2 > 5/4$, basta tomar $c = 3/2$ para concluir que $T(n) = O(\sqrt{n}\log n)$. A prova de que $T(n) = \Omega(\sqrt{n}\log n)$ é similar à prova feita para o limitante superior, de modo que a deixamos por conta do leitor.

Exemplo 3. $T(n) \leq T(n/2) + 1$.

Temos agora o caso onde $T(n)$ é o tempo de execução do algoritmo de busca binária. Mostraremos que $T(n) = O(\log n)$. Para $n = 2$ temos $T(2) = 2 \leq c = c\log 2$ sempre que $c \geq 2$. Suponha que $T(m) \leq c\log m$ para todo $2 \leq m < n$. Logo,

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + 1 \\ &\leq c\log n - c + 1 \\ &\leq c\log n\,, \end{aligned}$$

onde a última desigualdade vale para $c \geq 1$. Assim, $T(n) = O(\log n)$.

Exemplo 4. $T(n) = T(\lfloor n/2 \rfloor + 2) + 1$, onde assumimos $T(4) = 1$.

Temos agora o caso onde $T(n)$ é muito semelhante ao tempo de execução do algoritmo de busca binária. Logo, nosso palpite é que $T(n) = O(\log n)$, o que de fato é correto. Porém, para a análise funcionar corretamente precisamos de cautela. Vamos mostrar duas formas de analisar essa recorrência.

Primeiro vamos mostrar que $T(n) \leq c\log n$ para um valor de c apropriado. Seja $n \geq 4$ e note que $T(4) = 1 \leq c\log 4$ para $c \geq 1/2$. Suponha que $T(m) \leq c\log m$ para todo $4 \leq m < n$. Temos

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor + 2\right) + 1 \\ &\leq c\log\left(\frac{n}{2} + 2\right) + 1 \\ &= c\log\left(\frac{n+4}{2}\right) + 1 \\ &= c\log(n+4) - c + 1 \\ &\leq c\log\frac{3n}{2} - c + 1 \\ &= c\log n + c\log 3 - 2c + 1 \\ &= c\log n - c(2 - \log 3) + 1 \\ &\leq c\log n\,, \end{aligned}$$

onde a penúltima desigualdade vale para $n \geq 8$ e a última desigualdade vale sempre que

$c \geq 1/(2 - \log 3)$. Portanto, temos $T(n) = O(\log n)$.

Veremos agora uma outra abordagem, onde fortalecemos a hipótese de indução. Provaremos que $T(n) \leq c \log(n - a)$ para valores apropriados de a e c. No passo da indução, temos

$$\begin{aligned}
T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor + 2\right) + 1 \\
&\leq c \log\left(\frac{n}{2} + 2 - a\right) + 1 \\
&= c \log\left(\frac{n-a}{2}\right) + 1 \\
&= c \log(n - a) - c + 1 \\
&\leq c \log(n - a)\,,
\end{aligned}$$

onde a primeira desigualdade vale para $a \geq 4$ e a última desigualdade vale para $c \geq 1$. Assim, faça $a = 4$ e note que $T(6) = T(5) + 1 = T(4) + 2 = 3 \leq c \log(6 - 4)$ para todo $c \geq 3$. Portanto, fazendo $a = 4$ e $c \geq 3$, mostramos que $T(n) \leq c \log(n - a)$ para todo $n \geq 6$, de onde concluímos que $T(n) = O(\log n)$.

8.2 Método iterativo

Esse método consiste em expandir a recorrência até se chegar no caso base, que sabemos como calcular diretamente. Em geral, vamos utilizar como caso base $T(1) = 1$.

Como um primeiro exemplo, considere $T(n) \leq T(n/2) + 1$, que é o tempo de execução do algoritmo de busca binária. Expandindo:

$$\begin{aligned}
T(n) &\leq T\left(\frac{n}{2}\right) + 1 \\
&\leq \left(T\left(\frac{n/2}{2}\right) + 1\right) + 1 = T\left(\frac{n}{2^2}\right) + 2 \\
&\leq \left(T\left(\frac{n/2^2}{2}\right) + 1\right) + 2 = T\left(\frac{n}{2^3}\right) + 3 \\
&\vdots \\
&= T\left(\frac{n}{2^i}\right) + i\,.
\end{aligned}$$

Sabendo que $T(1) = 1$, essa expansão para quando $T(n/2^i) = T(1)$, que ocorre quando

$i = \log n$. Assim,

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2^i}\right) + i \\ &= T\left(\frac{n}{2^{\log n}}\right) + \log n \\ &= T(1) + \log n \\ &= O(\log n)\,. \end{aligned}$$

Para um segundo exemplo, considere $T(n) = 2T(n/2) + n$. Temos

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(2T\left(\frac{n/2}{2}\right) + \frac{n}{2}\right) + n = 2^2T\left(\frac{n}{2^2}\right) + 2n \\ &= 2^3T\left(\frac{n}{2^3}\right) + 3n \\ &\vdots \\ &= 2^iT\left(\frac{n}{2^i}\right) + in\,. \end{aligned}$$

Note que $n/2^i = 1$ quando $i = \log n$, de onde temos que

$$\begin{aligned} T(n) &= 2^{\log n}T\left(\frac{n}{2^{\log n}}\right) + n\log n \\ &= nT(1) + n\log n \\ &= n + n\log n = \Theta(n\log n)\,. \end{aligned}$$

Como veremos na Parte IV, *Insertion sort* e *Mergesort* são dois algoritmos que resolvem o problema de ordenação e têm, respectivamente, tempos de execução de pior caso $T_1(n) = \Theta(n^2)$ e $T_2(n) = 2T(n/2) + n$. Como acabamos de verificar, temos $T_2(n) = \Theta(n\log n)$, de modo que podemos concluir que, no pior caso, *Mergesort* é mais eficiente que *Insertion sort*.

Analisaremos agora um último exemplo, que representa o tempo de execução de um algoritmo que sempre divide o problema em 2 subproblemas de tamanho $n/3$ e cada chamada recursiva é executada em tempo constante. Assim, seja $T(n) = 2T(n/3) + 1$. Seguindo a

mesma estratégia dos exemplos anteriores, obtemos

$$\begin{aligned}
T(n) &= 2T\Big(\frac{n}{3}\Big) + 1 \\
&= 2\left(2T\Big(\frac{n/3}{3}\Big) + 1\right) + 1 = 2^2T\Big(\frac{n}{3^2}\Big) + (1+2) \\
&= 2^3T\Big(\frac{n}{3^3}\Big) + (1+2+2^2) \\
&\vdots \\
&= 2^iT\Big(\frac{n}{3^i}\Big) + \sum_{j=0}^{i-1} 2^j \\
&= 2^iT\Big(\frac{n}{3^i}\Big) + 2^i - 1\,.
\end{aligned}$$

Teremos $T(n/3^i) = 1$ quando $i = \log_3 n$, de onde concluímos que

$$\begin{aligned}
T(n) &= 2 \cdot 2^{\log_3 n} - 1 \\
&= 2 \cdot n^{\log_3 2} - 1 \\
&= 2 \cdot n^{\log 2/\log 3} - 1 \\
&= \Theta(n^{1/\log 3})\,.
\end{aligned}$$

É interessante resolver uma recorrência em que o tamanho do problema não é reduzido por divisão, mas por subtração, como $T(n) = 2T(n-1) + n$.

8.2.1 Limitantes assintóticos inferiores e superiores

Se quisermos apenas provar que $T(n) = O(f(n))$ em vez de $\Theta(f(n))$, podemos utilizar limitantes superiores em vez de igualdades. Analogamente, para mostrar que $T(n) = \Omega(f(n))$, podemos utilizar limitantes inferiores em vez de igualdades.

Por exemplo, para $T(n) = 2T(n/3) + 1$, se quisermos mostrar apenas que $T(n) = \Omega(n^{1/\log 3})$, podemos utilizar limitantes inferiores para nos ajudar na análise. O ponto principal é, ao expandir a recorrência $T(n)$, entender qual é o termo que "domina" assintoticamente

$T(n)$, i.e., qual é o termo que determina a ordem de complexidade de $T(n)$. Note que

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{3}\right) + 1 \\
&= 2\left(2T\left(\frac{n/3}{3}\right) + 1\right) + 1 \geq 2^2 T\left(\frac{n}{3^2}\right) + 2 \\
&\geq 2^3 T\left(\frac{n}{3^3}\right) + 3 \\
&\vdots \\
&\geq 2^i T\left(\frac{n}{3^i}\right) + i\,.
\end{aligned}$$

Teremos $T(n/3^i) = 1$ quando $i = \log_3 n$, de onde concluímos que

$$\begin{aligned}
T(n) &\geq 2^{\log_3 n} + \log_3 n \\
&= n^{1/\log 3} + \log_3 n \\
&= \Omega(n^{1/\log 3})\,.
\end{aligned}$$

Nem sempre o método iterativo para resolução de recorrências funciona bem. Quando o tempo de execução de um algoritmo é descrito por uma recorrência não tão balanceada como as dos exemplos dados, pode ser difícil executar esse método. Outro ponto fraco é que rapidamente os cálculos podem ficar complicados.

8.3 Método da árvore de recursão

Este é talvez o mais intuitivo dos métodos, que consiste em analisar a *árvore de recursão* do algoritmo, que é uma árvore onde cada nó representa um subproblema em alguma chamada recursiva. Esse nó é rotulado com o tempo feito naquela chamada, desconsiderando os tempos nas chamadas recursivas que ela faz. Seus filhos são os subproblemas que foram gerados nas chamadas recursivas feitas por ele. Assim, se somarmos os custos dentro de cada nível, obtendo o custo total por nível, e então somarmos os custos de todos os níveis, obtemos a solução da recorrência.

A Figura 8.1 mostra o início da construção da árvore de recursão de $T(n) = 2T(n/2) + n$, que é mostrada por completo na Figura 8.2. Cada nó contém o tempo feito na chamada representada pelo mesmo desconsiderando o tempo das chamadas recursivas. No lado direito temos os níveis da árvore (que vão até $\log n$ pois cada subproblema é reduzido pela metade)

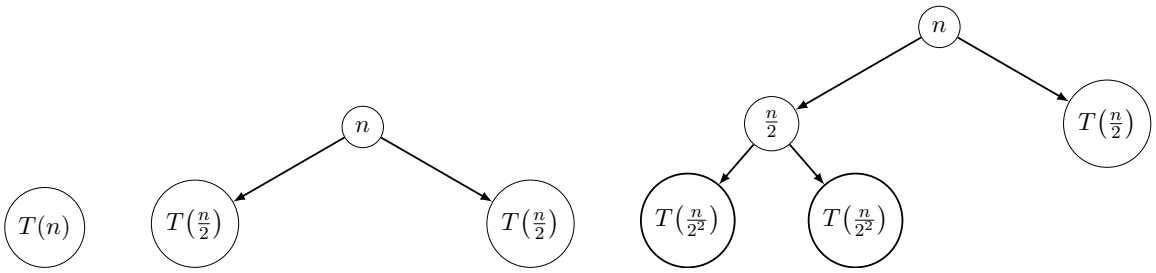


Figura 8.1: Começo da construção da árvore de recursão para $T(n) = 2T(n/2) + n$.

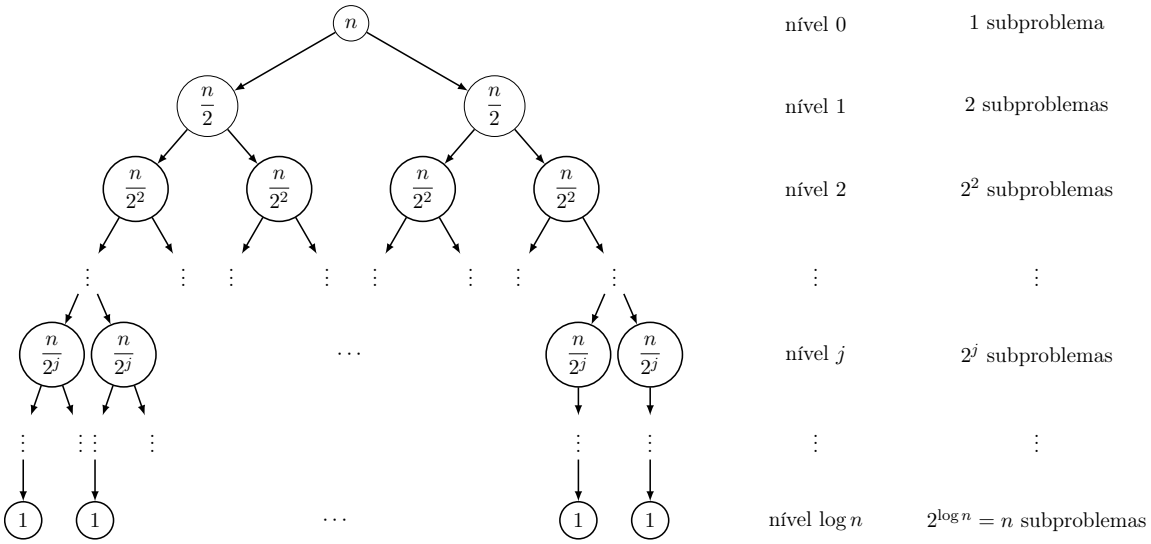


Figura 8.2: Árvore de recursão para $T(n) = 2T(n/2) + n$.

e a quantidade de subproblemas por nível. Assim, temos que

$$\begin{aligned} T(n) &= \sum_{j=0}^{\log n} \left(2^j \cdot \frac{n}{2^j}\right) \\ &= \sum_{j=0}^{\log n} n \\ &= n(\log n + 1)\,. \end{aligned}$$

Não é difícil mostrar que $n(\log n + 1) = \Theta(n \log n)$. Assim, essa árvore de recursão fornece o palpite que $T(n) = \Theta(n \log n)$.

Na Figura 8.3 temos a árvore de recursão para $T(n) = 2T(n/3) + 1$. Somando os custos

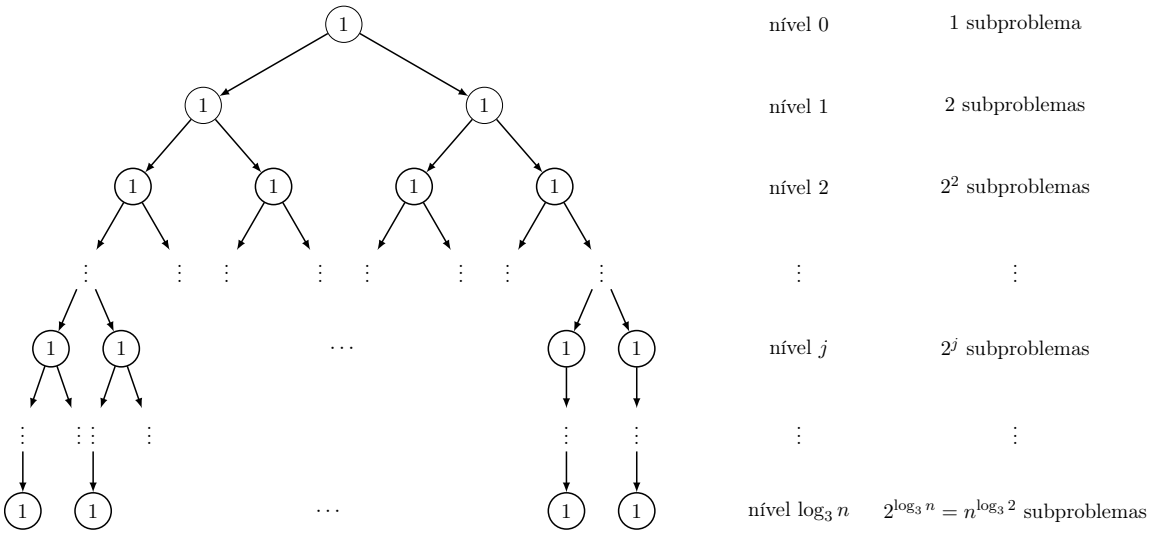


Figura 8.3: Árvore de recursão para $T(n) = 2T(n/3) + 1$.

por nível, temos que

$$\begin{aligned} T(n) &= \sum_{j=0}^{\log_3 n} \left(2^j \cdot 1\right) \\ &= \frac{2^{\log_3 n+1} - 1}{2-1} \\ &= 2n^{\log_3 2} - 1\,, \end{aligned}$$

de forma que $T(n) = \Theta(n^{\log_3 2})$.

Na Figura 8.4 temos a árvore de recursão para $T(n) = 3T(n/2) + n$. Somando os custos por nível, temos que

$$\begin{aligned} T(n) &= \sum_{j=0}^{\log n} \left(3^j \cdot \frac{n}{2^j}\right) = n \sum_{j=0}^{\log n} \left(\frac{3}{2}\right)^j \\ &= n\left(\frac{(3/2)^{\log n+1} - 1}{3/2 - 1}\right) = 2n\left(\frac{3}{2}\left(\frac{3}{2}\right)^{\log n} - 1\right) \\ &= 3\,n\,n^{\log(3/2)} - 2n = 3n^{\log(3/2)+1} - 2n = 3n^{\log 3} - 2n\,. \end{aligned}$$

Como $3n^{\log 3} - 2n \leq 3n^{\log 3}$ e $3n^{\log 3} - 2n \geq 3n^{\log 3} - 2n^{\log 3} = n^{\log 3}$, temos que $T(n) = \Theta(n^{\log 3})$.

Geralmente o método da árvore de recursão é utilizado para fornecer um bom palpite

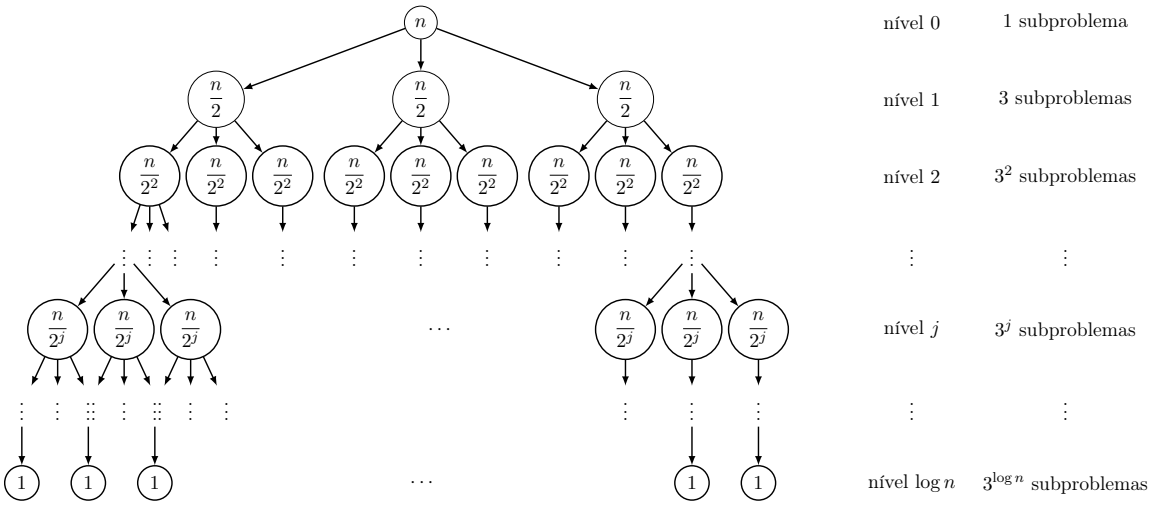


Figura 8.4: Árvore de recursão para $T(n) = 3T(n/2) + n$.

para o método da substituição, de modo que é permitida uma certa "frouxidão" na análise. Considere, por exemplo, a recorrência $T(n) = T(n/3) + T(2n/3) + 1$. Podemos aproximar $T(n)$ pelos resultados de $T'(n) = 2T'(n/3)+1$ e $T''(n) = 2T''(2n/3)+1$, pois $T'(n) \leq T(n) \leq T''(n)$. Porém, uma análise cuidadosa da árvore de recursão e dos custos associados a cada nível pode servir como uma prova direta para a solução da recorrência em questão.

8.4 Método mestre

O método mestre faz uso do Teorema 8.1 abaixo para resolver recorrências do tipo $T(n) = aT(n/b) + f(n)$, para $a \geq 1$, $b > 1$, e $f(n)$ positiva. Esse resultado formaliza uma análise cuidadosa feita utilizando árvores de recorrência. Na Figura 8.5 temos uma análise da árvore de recorrência de $T(n) = aT(n/b) + f(n)$.

Note que temos

$$\begin{aligned}
a^0 + a^1 + \cdots + a^{\log_b n} &= \frac{a^{1+\log_b n} - 1}{a - 1} \\
&= \frac{a^{\log_b b + \log_b n} - 1}{a - 1} \\
&= \frac{a^{\log_b (bn)} - 1}{a - 1} \\
&= \frac{(bn)^{\log_b a} - 1}{a - 1} \\
&= \Theta\left(n^{\log_b a}\right) .
\end{aligned}$$

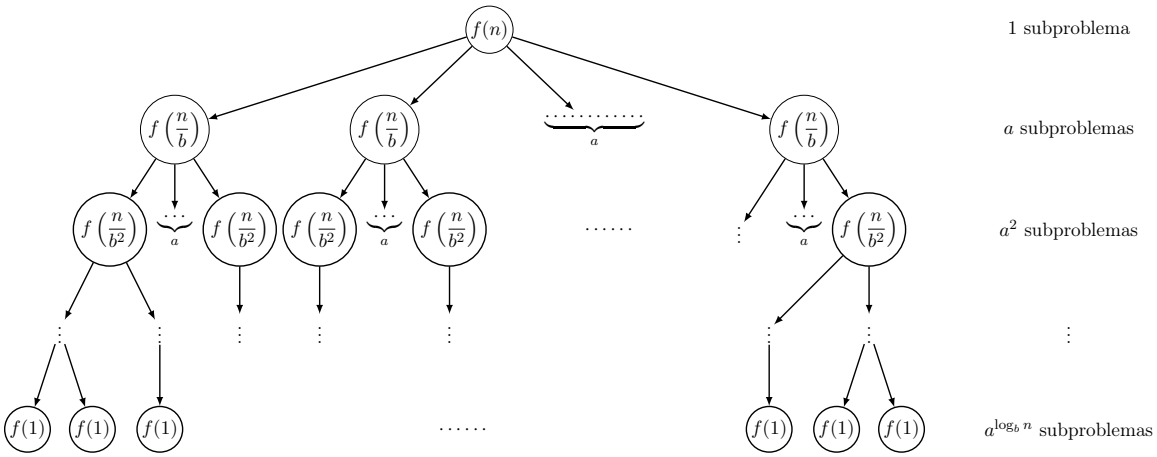


Figura 8.5: Árvore de recorrência para $T(n) = aT(n/b) + f(n)$.

Portanto, considerando somente o tempo para dividir o problema em subproblemas recursivamente, temos que é gasto tempo $\Theta\big(n^{\log_b a}\big)$. A ideia envolvida no Teorema Mestre, que será apresentado a seguir, analisa situações dependendo da diferença entre $f(n)$ e $n^{\log_b a}$.

Teorema 8.1: ***Teorema Mestre***

Sejam $a \geq 1$ e $b > 1$ constantes e seja $f(n)$ uma função. Para $T(n) = aT(n/b) + f(n)$, vale que

(1) se $f(n) = O(n^{\log_b a - \varepsilon})$ para alguma constante $\varepsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$;

(2) se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$;

(3) se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ para alguma constante $\varepsilon > 0$ e para n suficientemente grande temos $af(n/b) \leq cf(n)$ para alguma constante $c < 1$, então $T(n) = \Theta(f(n))$.

Mas qual a intuição por trás desse resultado? Imagine um algoritmo com tempo de execução $T(n) = aT(n/b) + f(n)$. Primeiramente, lembre que a árvore de recorrência descrita na Figura 8.5 sugere que o valor de $T(n)$ depende de quão grande ou pequeno $f(n)$ é com relação a $n^{\log_b a}$. Se a função $f(n)$ sempre assume valores "pequenos" (aqui, pequeno significa $f(n) = O(n^{\log_b a - \varepsilon})$), então é de se esperar que o mais custoso para o algoritmo seja dividir cada instância do problema em a partes de uma fração $1/b$ dessa instância. Assim, nesse caso, o algoritmo vai ser executado recursivamente $\log_b n$ vezes até que se chegue à base da recursão, gastando para isso tempo da ordem de $a^{\log_b n} = n^{\log_b a}$, como indicado pelo item (1).

O item (3) corresponde ao caso em que $f(n)$ é "grande" comparado com o tempo gasto para dividir o problema em a partes de uma fração $1/b$ da instância em questão. Portanto, faz sentido que $f(n)$ determine o tempo de execução do algoritmo nesse caso, que é a conclusão obtida no item (3). O caso intermediário, no item (2), corresponde ao caso em que a função $f(n)$ e dividir o problema recursivamente são ambos essenciais no tempo de execução do algoritmo.

Infelizmente, existem alguns casos não cobertos pelo Teorema Mestre, mas mesmo nesses casos conseguimos utilizar o teorema para conseguir limitantes superiores e/ou inferiores. Entre os casos (1) e (2) existe um intervalo em que o Teorema Mestre não fornece nenhuma informação, que é quando $f(n)$ é assintoticamente menor que $n^{\log_b a}$, mas assintoticamente maior que $n^{\log_b a-\varepsilon}$ para todo $\varepsilon > 0$, e.g., $f(n) = \Theta(n^{\log_b a}/\log n)$ ou $\Theta(n^{\log_b a}/\log(\log n))$. De modo similar, existe um intervalo sem informações entre (2) e (3).

Existe ainda um outro caso em que não é possível aplicar o Teorema Mestre a uma recorrência do tipo $T(n) = aT(n/b) + f(n)$. Pode ser o caso que $f(n) = \Omega(n^{\log_b a+\varepsilon})$ mas a condição $af(n/b) \leq cf(n)$ do item (3) não é satisfeita. Felizmente, essa condição é geralmente satisfeita em recorrências que representam tempo de execução de algoritmos. Desse modo, para algumas funções $f(n)$ podemos considerar uma versão simplificada do Teorema Mestre, que dispensa a condição extra no item (3). Veremos essa versão na Seção 8.4.1.

Antes disso, a seguir temos um exemplo de recorrência que não satisfaz a condição extra do item (3) do Teorema 8.1. Ressaltamos que é improvável que tal recorrência descreva o tempo de execução de um algoritmo.

Exemplo 1. $T(n) = T(n/2) + n(2 - \cos n)$.

Primeiro vamos verificar em que caso estaríamos no Teorema Mestre. De fato, como $a = 1$ e $b = 2$, temos $n^{\log_b a} = 1$. Assim, como $f(n) = n(2 - \cos n) \geq n$, temos $f(n) = \Omega(n^{\log_b a+\varepsilon})$ para qualquer $0 < \varepsilon < 1$.

Vamos agora verificar se é possível obter a condição extra do caso (3). Precisamos mostrar que $f(n/2) \leq cf(n)$ para algum $c < 1$ e todo n suficientemente grande. Vamos usar o fato que $\cos(2k\pi) = 1$ para qualquer inteiro k, e que $\cos(k\pi) = -1$ para todo inteiro ímpar k. Seja $n = 2k\pi$ para qualquer inteiro ímpar $k \geq 3$. Assim, temos

$$c \geq \frac{f(n/2)}{f(n)} = \frac{(n/2)\,(2 - \cos(k\pi))}{n(2 - \cos(2k\pi))} = \frac{2 - \cos(k\pi)}{2(2 - \cos(2k\pi))} = \frac{3}{2}\,.$$

Logo, para infinitos valores de n, a constante c precisa ser pelo menos $3/2$, e portanto não é possível obter a condição extra no caso (3). Assim, não há como aplicar o Teorema Mestre à recorrência $T(n) = T(n/2) + n(2 - \cos n)$.

8.4.1 Versão simplificada do método mestre

Seja $f(n)$ um polinômio de grau k cujo coeficiente do monômio de maior grau é positivo (para k constante), i.e., $f(n) = \sum_{i=0}^{k} a_i n^i$, onde $a_0, a_1, \ldots, a_k$ são constantes e $a_k > 0$.

Teorema 8.2: ***Teorema Mestre - Versão simplificada***

Sejam $a \geq 1$, $b > 1$ e $k \geq 0$ constantes e seja $f(n)$ um polinômio de grau k cujo coeficiente do monômio de maior grau é positivo. Para $T(n) = aT(n/b) + f(n)$, vale que

(1) se $f(n) = O(n^{\log_b a - \varepsilon})$ para alguma constante $\varepsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$;

(2) se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$;

(3) se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ para alguma constante $\varepsilon > 0$, então $T(n) = \Theta(f(n))$.

Demonstração. Vamos provar que, para $f(n)$ como no enunciado, se $f(n) = \Omega(n^{\log_b a + \varepsilon})$, então para todo n suficientemente grande temos $af(n/b) \leq cf(n)$ para alguma constante $c < 1$. Dessa forma, o resultado segue diretamente do Teorema 8.1.

Primeiro note que como $f(n) = \sum_{i=0}^{k} a_i n^i = \Omega(n^{\log_b a + \varepsilon})$ temos $k = \log_b a + \varepsilon$. Resta provar que $af(n/b) \leq cf(n)$ para algum $c < 1$. Logo, basta provar que $cf(n) - af(n/b) \geq 0$ para algum $c < 1$. Assim,

$$\begin{aligned}
cf(n) - af(n/b) &= c\sum_{i=0}^{k} a_i n^i - a\sum_{i=0}^{k} a_i \frac{n^i}{b^i} \\
&= a_k \left(c - \frac{a}{b^k}\right) n^k + \sum_{i=0}^{k-1} a_i \left(c - \frac{a}{b^i}\right) n^i \\
&\geq a_k \left(c - \frac{a}{b^k}\right) n^k - \sum_{i=0}^{k-1} a_i \left(\frac{a}{b^i}\right) n^i \\
&\geq a_k \left(c - \frac{a}{b^k}\right) n n^{k-1} - \left(a \sum_{i=0}^{k-1} a_i\right) n^{k-1} \\
&= (c_1 n) n^{k-1} - (c_2) n^{k-1},
\end{aligned}$$

onde c_1 e c_2 são constantes e na última desigualdade utilizamos o fato de $b > 1$ (assim, $b^i > 1$ para todo $i \geq 0$). Logo, para $n \geq c_2/c_1$, temos que $cf(n) - af(n/b) \geq 0$. □

Abaixo mostramos uma segunda prova para o Teorema 8.2. Reformulamos seu enunciado com base nas seguintes observações. Primeiro, sendo $f(n) = \sum_{i=0}^{k} a_i n^i$, onde $a_0, a_1, \ldots, a_k$

são constantes e $a_k > 0$, não é difícil mostrar que $f(n) = \Theta(n^k)$. Segundo, se $\Theta(n^k) = O(n^{\log_b a - \varepsilon})$ para algum $\varepsilon > 0$, então essencialmente estamos assumindo $n^k \leq n^{\log_b a - \varepsilon}$. Mas $n^{\log_b a - \varepsilon} < n^{\log_b a}$ pois $\varepsilon > 0$, ou seja, estamos assumindo $n^k < n^{\log_b a}$, que equivale a assumir $b^k < a$. Com argumentos semelhantes, assumir $\Theta(n^k) = \Theta(n^{\log_b a})$ significa essencialmente assumir $b^k = a$, e assumir $\Theta(n^k) = \Omega(n^{\log_b a + \varepsilon})$ significa essencialmente assumir $b^k > a$.

Teorema 8.3: ***Teorema Mestre - Versão simplificada***

Sejam $a \geq 1$, $b > 1$ e $k \geq 0$ constantes. Para $T(n) = aT(n/b) + \Theta(n^k)$, vale que

(1) se $a > b^k$, então $T(n) = \Theta(n^{\log_b a})$;

(2) se $a = b^k$, então $T(n) = \Theta(n^k \log n)$;

(3) se $a < b^k$, então $T(n) = \Theta(n^k)$.

Demonstração. Como $T(n) = aT(n/b) + \Theta(n^k)$, isso significa que existem constantes c_1 e c_2 para as quais vale que:

1. $T(n) \leq aT(n/b) + c_1 n^k$; e

2. $T(n) \geq aT(n/b) + c_2 n^k$.

Vamos assumir que $T(1) = 1$ em qualquer caso.

Considere inicialmente que o item 1 vale, isto é, $T(n) \leq aT(n/b) + c_1 n^k$. Ao analisar a árvore de recorrência para $T(n)$, percebemos que a cada nível o tamanho do problema diminui por um fator b, de forma que o último nível é $\log_b n$. Também notamos que um certo nível j possui a^j subproblemas de tamanho n/b^j cada.

Dessa forma, o total de tempo gasto em um nível j é $\leq a^j c_1 (n/b^j)^k = c_1 n^k (a/b^k)^j$. Somando o tempo gasto em todos os níveis, temos o tempo total do algoritmo, que é

$$T(n) \leq \sum_{j=0}^{\log_b n} c_1 n^k \left(\frac{a}{b^k}\right)^j = c_1 n^k \sum_{j=0}^{\log_b n} \left(\frac{a}{b^k}\right)^j , \tag{8.2}$$

de onde vemos que o tempo depende da relação entre a e b^k. Assim,

(1) se $a > b^k$, temos $a/b^k > 1$, e a equação (8.2) pode ser desenvolvida da seguinte forma:

$$\begin{aligned}
T(n) &\leq c_1 n^k \left(\frac{\left(\frac{a}{b^k}\right)^{\log_b n+1} - 1}{\frac{a}{b^k} - 1} \right) = \frac{c_1 n^k}{\frac{a}{b^k} - 1} \left(\left(\frac{a}{b^k}\right)^{\log_b n+1} - 1 \right) \\
&\leq \frac{c_1 n^k}{\frac{a}{b^k} - 1} \left(\frac{a}{b^k}\right)^{\log_b n+1} = \frac{a c_1 n^k}{\left(\frac{a}{b^k} - 1\right) b^k} \left(\frac{a}{b^k}\right)^{\log_b n} \\
&= \frac{a c_1 n^k}{\left(\frac{a}{b^k} - 1\right) b^k} n^{\log_b a/b^k} = c' n^k \frac{n^{\log_b a}}{n^{\log_b b^k}} \\
&= c' n^{\log_b a} ,
\end{aligned}$$

onde $c' = (ac_1)/((a/b^k - 1)b^k)$ é constante. Ou seja, acabamos de mostrar que se $a > b^k$, então $T(n) = O(n^{\log_b a})$.

(2) se $a = b^k$, temos $a/b^k = 1$, e a equação (8.2) pode ser desenvolvida da seguinte forma:

$$\begin{aligned}
T(n) &\leq c_1 n^k (\log_b n + 1) = c_1 n^k \log_b n + c_1 n^k \\
&\leq c_1 n^k \log_b n + c_1 n^k \log_b n = 2 c_1 n^k \log_b n ,
\end{aligned}$$

sempre que $n \geq b$. Ou seja, acabamos de mostrar que se $a = b^k$, então $T(n) = O(n^k \log n)$.

(3) se $a < b^k$, temos $a/b^k < 1$, e a equação (8.2) pode ser desenvolvida da seguinte forma:

$$T(n) \leq c_1 n^k \left(\frac{1 - \left(\frac{a}{b^k}\right)^{\log_b n+1}}{1 - \frac{a}{b^k}} \right) = \frac{c_1 n^k}{1 - \frac{a}{b^k}} \left(1 - \left(\frac{a}{b^k}\right)^{\log_b n+1} \right) \leq \frac{c_1 n^k}{1 - \frac{a}{b^k}} = c' n^k ,$$

onde $c' = c_1/(1 - a/b^k)$ é constante. Ou seja, acabamos de mostrar que se $a < b^k$, então $T(n) = O(n^k)$.

Considere agora que o item 2 vale, isto é, $T(n) \geq aT(n/b) + c_2 n^k$. De forma semelhante, ao analisar a árvore de recorrência para $T(n)$, somando o tempo gasto em todos os níveis, temos que

$$T(n) \geq \sum_{j=0}^{\log_b n} c_2 n^k \left(\frac{a}{b^k}\right)^j = c_2 n^k \sum_{j=0}^{\log_b n} \left(\frac{a}{b^k}\right)^j , \tag{8.3}$$

de onde vemos que o tempo também depende da relação entre a e b^k. Não é difícil mostrar que

(1) se $a > b^k$, então $T(n) = \Omega(n^{\log_b a})$,

(2) se $a = b^k$, então $T(n) = \Omega(n^k \log n)$, e

(3) se $a < b^k$, então $T(n) = \Omega(n^k)$,

o que conclui o resultado. $\square$

8.4.2 Resolvendo recorrências com o método mestre

Vamos analisar alguns exemplos de recorrências onde aplicaremos o Teorema Mestre para resolvê-las.

Exemplo 1. $T(n) = 2T(n/2) + n$.

Claramente, temos $a = 2$, $b = 2$ e $f(n) = n$. Como $f(n) = n = n^{\log_2 2}$, o caso do Teorema Mestre em que esses parâmetros se encaixam é o caso (2). Assim, pelo Teorema Mestre, $T(n) = \Theta(n \log n)$.

Exemplo 2. $T(n) = 4T(n/10) + 5\sqrt{n}$.

Neste caso temos $a = 4$, $b = 10$ e $f(n) = 5\sqrt{n}$. Assim, $\log_b a = \log_{10} 4 \approx 0,6$. Como $5\sqrt{n} = 5n^{0,5} = O(n^{0,6-0,1})$, estamos no caso (1) do Teorema Mestre. Logo, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_{10} 4})$.

Exemplo 3. $T(n) = 4T(n/16) + 5\sqrt{n}$.

Note que $a = 4$, $b = 16$ e $f(n) = 5\sqrt{n}$. Assim, $\log_b a = \log_{16} 4 = 1/2$. Como $5\sqrt{n} = 5n^{0,5} = \Theta(n^{\log_b a})$, estamos no caso (2) do Teorema Mestre. Logo, $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^{\log_{16} 4} \log n) = \Theta(\sqrt{n} \log n)$.

Exemplo 4. $T(n) = 4T(n/2) + 10n^3$.

Neste caso temos $a = 4$, $b = 2$ e $f(n) = 10n^3$. Assim, $\log_b a = \log_2 4 = 2$. Como $10n^3 = \Omega(n^{2+1})$, estamos no caso (3) do Teorema Mestre. Logo, concluímos que $T(n) = \Theta(n^3)$.

Exemplo 5. $T(n) = 5T(n/4) + n$.

Temos $a = 5$, $b = 4$ e $f(n) = n$. Assim, $\log_b a = \log_4 5$. Como $\log_4 5 > 1$, temos que $f(n) = n = O(n^{\log_4 5 - \varepsilon})$ para $\varepsilon = \log_4 5 - 1 > 0$. Logo, estamos no caso (1) do Teorema Mestre. Assim, concluímos que $T(n) = \Theta(n^{\log_4 5})$.

8.4.3 Ajustes para aplicar o método mestre

Dada uma recorrência $T(n) = aT(n/b) + f(n)$, existem duas possibilidades em que o Teorema Mestre (Teorema 8.1) não é aplicável (diretamente):

(i) nenhuma das três condições assintóticas no teorema é válida para $f(n)$; ou

(ii) $f(n) = \Omega(n^{\log_b a+\varepsilon})$ para alguma constante $\varepsilon > 0$, mas não existe $c < 1$ tal que $af(n/b) \leq cf(n)$ para todo n suficientemente grande.

Para afirmar que o Teorema Mestre não vale devido à (i), temos que verificar que valem as três seguintes afirmações: 1) $f(n) \neq \Theta(n^{\log_b a})$; 2) $f(n) \neq O(n^{\log_b a-\varepsilon})$ para qualquer $\varepsilon > 0$; e 3) $f(n) \neq \Omega(n^{\log_b a+\varepsilon})$. Lembre que, dado que temos a versão simplificada do Teorema Mestre (Teorema 8.2), não precisamos verificar o item (ii), pois essa condição é sempre satisfeita para polinômios $f(n)$ com coeficientes não negativos.

No que segue mostraremos que não é possível aplicar o Teorema Mestre diretamente a algumas recorrências, mas sempre é possível conseguir limitantes superiores e inferiores analisando recorrências levemente modificadas.

Exemplo 1. $T(n) = 2T(n/2) + n \log n$.

Começamos notando que $a = 2$, $b = 2$ e $f(n) = n \log n$. Para todo n suficientemente grande e qualquer constante C vale que $n \log n \geq Cn$. Assim, para qualquer $\varepsilon > 0$, temos que $n \log n \neq O(n^{1-\varepsilon})$, de onde concluímos que a recorrência $T(n)$ não se encaixa no caso (1). Como $n \log n \neq \Theta(n)$, também não podemos utilizar o caso (2). Por fim, como $\log n \neq \Omega(n^{\varepsilon})$ para qualquer $\varepsilon > 0$, temos que $n \log n \neq \Omega(n^{1+\varepsilon})$, de onde concluímos que o caso (3) do Teorema Mestre também não se aplica.

Exemplo 2. $T(n) = 5T(n/8) + n^{\log_8 5} \log n$.

Começamos notando que $a = 5$, $b = 8$ e $f(n) = n^{\log_8 5} \log n$. Para todo n suficientemente grande e qualquer constante C vale que $n^{\log_8 5} \log n \geq Cn^{\log_8 5}$. Assim, para qualquer $\varepsilon > 0$, temos que $n^{\log_8 5} \log n \neq O(n^{\log_8 5-\varepsilon})$, de onde concluímos que a recorrência $T(n)$ não se encaixa no caso (1). Como $n^{\log_8 5} \log n \neq \Theta(n^{\log_8 5})$, também não podemos utilizar o caso (2). Por fim, como $\log n \neq \Omega(n^{\varepsilon})$ para qualquer $\varepsilon > 0$, temos que $n^{\log_8 5} \log n \neq \Omega(n^{\log_8 5+\varepsilon})$, de onde concluímos que o caso (3) do Teorema Mestre também não se aplica.

Exemplo 3. $T(n) = 3T(n/9) + \sqrt{n} \log n$.

Começamos notando que $a = 3$, $b = 9$ e $f(n) = \sqrt{n} \log n$. Logo, $n^{\log_b a} = \sqrt{n}$. Para todo n suficientemente grande e qualquer constante C vale que $\sqrt{n} \log n \geq C\sqrt{n}$. Assim, para qualquer $\varepsilon > 0$, temos que $\sqrt{n} \log n \neq O(\sqrt{n}/n^{\varepsilon})$, de onde concluímos que a recorrência $T(n)$ não se encaixa no caso (1). Como $\sqrt{n} \log n \neq \Theta(\sqrt{n})$, também não podemos utilizar o caso (2). Por fim, como $\log n \neq \Omega(n^{\varepsilon})$ para qualquer $\varepsilon > 0$, temos que $\sqrt{n} \log n \neq \Omega(\sqrt{n}n^{\varepsilon})$, de onde concluímos que o caso (3) do Teorema Mestre também não se aplica.

Exemplo 4. $T(n) = 16T(n/4) + n^2/\log n$.

Começamos notando que $a = 16$, $b = 4$ e $f(n) = n^2/\log n$. Logo, $n^{\log_b a} = n^2$. Para todo n suficientemente grande e qualquer constante C vale que $n \geq C \log n$. Assim, para qualquer $\varepsilon > 0$, temos que $n^2/\log n \neq O(n^{2-\varepsilon})$, de onde concluímos que a recorrência $T(n)$ não se encaixa no caso (1). Como $n^2/\log n \neq \Theta(n^2)$, também não podemos utilizar o caso (2). Por fim, como $n^2/\log n \neq \Omega(n^{2+\varepsilon})$ para qualquer $\varepsilon > 0$, concluímos que o caso (3) do Teorema Mestre também não se aplica.

Como vimos, não é possível aplicar o Teorema Mestre diretamente às recorrências descritas nos exemplos acima. Porém, podemos ajustar as recorrências e conseguir bons limitantes assintóticos utilizando o Teorema Mestre. Por exemplo, para a recorrência $T(n) = 16T(n/4) + n^2/\log n$ dada acima, claramente temos que $T(n) \leq 16T(n/4) + n^2$, de modo que podemos aplicar o Teorema Mestre na recorrência $T'(n) = 16T'(n/4) + n^2$. Como $n^2 = n^{\log_4 16}$, pelo caso (2) do Teorema Mestre, temos que $T'(n) = \Theta(n^2 \log n)$. Portanto, como $T(n) \leq T'(n)$, concluímos que $T(n) = O(n^2 \log n)$, obtendo um limitante assintótico superior para $T(n)$. Por outro lado, temos que $T(n) = 16T(n/4) + n^2/\log n \geq T''(n) = 16T''(n/4) + n$. Pelo caso (1) do Teorema Mestre, temos que $T''(n) = \Theta(n^2)$. Portanto, como $T(n) \geq T''(n)$, concluímos que $T(n) = \Omega(n^2)$. Dessa forma, apesar de não sabermos exatamente qual é a ordem de grandeza de $T(n)$, temos uma boa estimativa, dado que mostramos que essa ordem de grandeza está entre n^2 e $n^2 \log n$.

Existem outros métodos para resolver equações de recorrência mais gerais que equações do tipo $T(n) = aT(n/b) + f(n)$. Um exemplo importante é o método de Akra-Bazzi, que consegue resolver equações não tão balanceadas, como $T(n) = T(n/3) + T(2n/3) + \Theta(n)$, mas não entraremos em detalhes desse método aqui.

Parte

III

Estruturas de dados

"Computer programs usually operate on tables of information. In most cases these tables are not simply amorphous masses of numerical values; they involve important *structural relationships* between the data elements."

Knuth — The Art of Computer Programming, 1997.

Nesta parte

Algoritmos geralmente precisam manipular conjuntos de dados que podem crescer, diminuir ou sofrer diversas modificações durante sua execução. Um *tipo abstrato de dados* é um conjunto de dados, as relações entre eles e as funções e operações que podem ser aplicadas aos dados. Uma *estrutura de dados* é uma implementação de um tipo abstrato de dados.

O segredo de muitos algoritmos é o uso de uma boa estrutura de dados. Como vimos na Seção 7.5, o uso de uma boa estrutura pode ter grande impacto na velocidade de um programa. Estruturas diferentes suportam operações diferentes em tempos diferentes, de forma que nenhuma estrutura funciona bem em todas as circunstâncias. Assim, é importante conhecer as qualidades e limitações de várias delas. Nas seções a seguir discutiremos os tipos abstratos e as estruturas de dados mais recorrentes em algoritmos.

Capítulo 9

Estruturas lineares

Neste capítulo veremos as estruturas de dados mais simples e clássicas, que formam a base para muitos dos algoritmos vistos neste livro.

9.1 Vetor

Um *vetor* é uma coleção de elementos de um mesmo tipo que são referenciados por um identificador único. Esses elementos ocupam posições *contíguas* na memória, o que permite acesso direto (em tempo constante, $\Theta(1)$) a qualquer elemento por meio de um índice inteiro.

Denotamos um vetor A com *capacidade* para m elementos por $A[1..m]$. Se o vetor armazena n elementos, dizemos que *seu tamanho* é n e o denotamos também por $A[1..n]$ ou por $A = (a_1, a_2, \ldots, a_n)$. Denotamos por $A[i]$ o elemento que está armazenado na i-ésima posição de A (i.e., a_i), para todo i com $1 \leq i \leq n$. Para quaisquer i e j em que $1 \leq i < j \leq n$, denotamos por $A[i..j]$ o subvetor de A que contém os elementos $A[i], A[i+1], \ldots, A[j]$.

A seguir temos uma representação gráfica do vetor $A = (12, 99, 37, 24, 2, 15)$:

	1	2	3	4	5	6
A	12	99	37	24	2	15

Como já foi discutido no Capítulo 2, o tempo para buscar um elemento em um vetor de tamanho n é $O(n)$ se usarmos o algoritmo de busca linear pois, no pior caso, ela precisa acessar todos os elementos armazenados no vetor. A inserção de um novo elemento x em um vetor A de tamanho n pode ser feita em tempo constante, $\Theta(1)$, ao inseri-lo na primeira posição disponível, a posição $n+1$. Já a remoção de algum elemento do vetor envolve saber

em que posição i encontra-se tal elemento. Sabendo que o vetor tem n elementos, então podemos simplesmente copiar o elemento $A[n]$ para a posição i. Assim, se a posição i já for indicada, a remoção leva tempo $\Theta(1)$, mas caso contrário uma busca pelo elemento ainda precisa ser feita, levando assim tempo $O(n)$.

Veja que, se o vetor estiver ordenado, então os tempos mencionados acima mudam. A busca binária nos garante que o tempo de busca em um vetor de tamanho n é $O(\log n)$. A inserção, no entanto, não pode mais ser feita em tempo constante em uma posição qualquer, pois precisamos garantir que o vetor continuará ordenado. Assim, potencialmente precisaremos deslocar vários elementos do vetor durante uma inserção, de forma que ela leva tempo $O(n)$. De forma similar, a remoção de um elemento que está em uma posição i precisa de tempo $O(n)$ para deslocar os elementos à direita dessa posição e assim manter o vetor ordenado.

O fato do vetor estar ordenado ainda nos permite realizar a operação de encontrar o k-ésimo menor elemento do vetor em tempo $\Theta(1)$. Se o vetor não estiver ordenado, existe um algoritmo que consegue realizar tal operação em tempo $O(n)$.

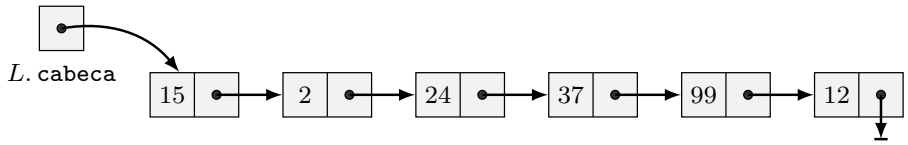
9.2 Lista encadeada

Uma *lista encadeada* é uma estrutura de dados linear onde cada elemento é armazenado em um *nó*, que armazena também *endereços* para outros nós da lista. Por isso, cada nó de uma lista pode estar em uma posição diferente da memória, sendo diferente de um vetor, onde os elementos são armazenados de forma contínua. Na forma mais simples, têm-se acesso apenas ao primeiro nó da lista. Em qualquer variação, têm-se acesso a um número constante de nós apenas (o primeiro nó e o último nó, por exemplo). Assim, listas não permitem acesso direto a um elemento: para acessar o k-ésimo elemento da lista, deve-se acessar o primeiro, que dá acesso ao segundo, que dá acesso ao terceiro, e assim sucessivamente, até que o $(k-1)$-ésimo elemento dá acesso ao k-ésimo.

Consideramos que cada nó contém um atributo *chave* e, como sempre, pode conter outros atributos importantes. Iremos inserir, remover ou modificar elementos de uma lista baseados nos atributos chave, que devem conter números inteiros. Outros atributos importantes que sempre existem são os endereços para outros nós da lista e sua quantidade e significado dependem de qual variação da lista estamos lidando. Em uma *lista encadeada simples* existe apenas um atributo de endereço, chamado `proximo`, que dá acesso ao nó que está imediatamente após o nó atual na lista. Em uma *lista duplamente encadeada* existe, além do atributo `proximo`, o atributo `anterior`, que dá acesso ao nó que está imediatamente antes do nó atual na lista. Seja x um nó qualquer. Se $x.\texttt{anterior} = \text{NULO}$, então x não tem predecessor, de

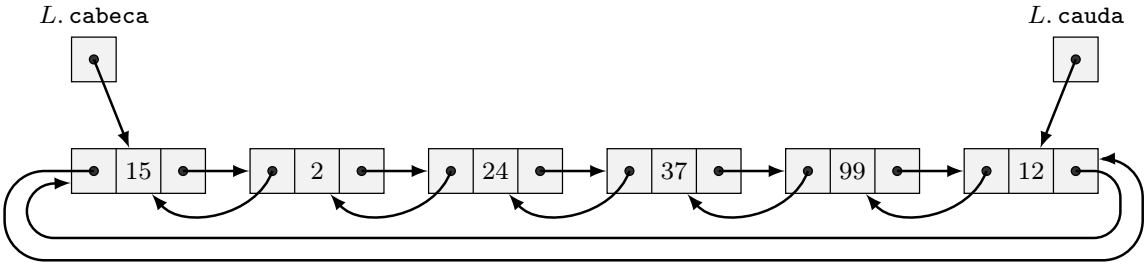
modo que é o primeiro nó da lista, a *cabeça* da lista. Se $x.$`proximo` = NULO, então x não tem sucessor e é chamado de *cauda* da lista, sendo o último nó da mesma. Em uma *lista circular*, o atributo `proximo` da cauda aponta para a cabeça da lista, enquanto que o atributo *anterior* da cabeça aponta para a cauda. Dada uma lista L, o atributo $L.$`cabeca` é o primeiro nó de L, sendo que $L.$`cabeca` = NULO quando a lista estiver vazia.

A seguir temos uma representação gráfica de uma lista encadeada simples que contém os elementos $12, 99, 37, 24, 2, 15$:



Veja que o acesso ao elemento que contém chave 24 (terceiro da lista) é feito de forma indireta: $L.$`cabeca.proximo.proximo`.

A seguir temos uma representação gráfica de uma lista duplamente encadeada circular que contém os elementos $12, 99, 37, 24, 2, 15$:



A seguir vamos descrever os procedimentos de busca, inserção e remoção em uma lista duplamente encadeada, não ordenada e não-circular.

O procedimento BuscaNaLista mostrado no Algoritmo 9.1 realiza uma busca pelo primeiro nó que possui chave k na lista L. Primeiramente, a cabeça da lista L é analisada e em seguida os elementos da lista são analisados, um a um, até que k seja encontrado ou até que a lista seja completamente verificada. No pior caso, toda a lista deve ser verificada, de modo que o tempo de execução de BuscaNaLista é $O(n)$ para uma lista com n elementos.

A inserção de um elemento em uma lista é realizada, em geral, no começo da lista. Para inserir no começo, já temos de antemão a posição em que o elemento será inserido, que é $L.$`cabeca`. No Algoritmo 9.2 inserimos um nó x na lista L. Portanto, caso L não seja vazia, o ponteiro $x.$`proximo` deve apontar para a atual cabeça de L e $L.$`cabeca.anterior` deve apontar para x. Caso L seja vazia, então $x.$`proximo` aponta para NULO. Como x será a cabeça de L, o ponteiro $x.$`anterior` deve apontar para NULO. Para algumas aplicações pode

Algoritmo 9.1: BuscaNaLista(L, k)

```
1 x = L.cabeca
  /* Seguimos nós por meio dos endereços de proximo até chegar ao fim da lista
     ou encontrar o elemento                                                  */
2 enquanto x ≠ NULO e x.chave ≠ k faça
3     x = x.proximo
4 devolve x
```

ser útil que exista um ponteiro que sempre aponta para a cauda de uma lista L. Assim, vamos manter um ponteiro $L.$`cauda`, que aponta para o último elemento de L, onde $L.$`cauda` $=$ NULO quando L é uma lista vazia.

Algoritmo 9.2: InsereNoInicioLista(L, x)

```
1 x.anterior = NULO
2 x.proximo = L.cabeca
3 se L.cabeca ≠ NULO então
      /* Se há elemento na cabeça de L, este deve ter como anterior o nó x  */
4     L.cabeca.anterior = x
5 senão
      /* Se não há cabeça, também não há cauda, e o nó x será o último      */
6     L.cauda = x
  /* Em qualquer caso, x será a nova cabeça da lista                        */
7 L.cabeca = x
```

Como somente uma quantidade constante de operações é executada, o procedimento InsereNoInicioLista é executado em tempo $\Theta(1)$ para uma lista com n elementos. Note que o procedimento de inserção em uma lista encadeada ordenada levaria tempo $O(n)$, pois precisaríamos inserir x na posição correta dentro da lista, tendo que percorrer toda a lista no pior caso.

Pode ser que uma aplicação necessite inserir elementos no fim de uma lista. Por exemplo, inserir no fim de uma lista facilita a implementação da estrutura de dados 'fila' com o uso de listas encadeadas. Outro exemplo é na obtenção de informações importantes durante a execução da busca em profundidade em grafos. O uso do ponteiro $L.$`cauda` torna essa tarefa análoga à inserção no início de uma lista. O procedimento InsereNoFimLista, mostrado no Algoritmo 9.3, realiza essa tarefa.

Finalmente, o Algoritmo 9.4 mostra o procedimento RemoveDaLista, que remove um nó

Algoritmo 9.3: InsereNoFimLista(L, x)

```
1 x.anterior = L.cauda
2 x.proximo = NULO
3 se L.cauda ≠ NULO então
      /* Se há elemento na cauda de L, este deve ter como próximo o nó x    */
4     L.cauda.proximo = x
5 senão
      /* Se não há cauda, também não há cabeça, e o nó x será o primeiro    */
6     L.cabeca = x
  /* Em qualquer caso, x será a nova cauda da lista                         */
7 L.cauda = x
```

com chave k de uma lista L. A remoção é simples, sendo necessário efetuar uma busca para encontrar o nó x com chave k e atualizar os ponteiros x.`anterior`.`proximo` e x.`proximo`.`anterior`, tendo cuidado com os casos onde x é a cabeça ou a cauda de L. Caso utilizemos uma lista ligada L em que inserções são feitas no fim da lista, precisamos garantir que vamos manter L.`cauda` atualizado quando removemos o último elemento de L.

Como somente uma busca por uma chave k e uma quantidade constante de operações é efetuada, a remoção leva tempo $O(n)$ no pior caso, como o algoritmo de busca.

Algoritmo 9.4: RemoveDaLista(L, k)

```
x = L.cabeca
enquanto x ≠ NULO e x.chave ≠ k faça
    x = x.proximo
se x = NULO então
    devolve NULO
/* Se x é a cauda de L, então o penúltimo nó passa a ser L.cauda */
se x.proximo == NULO então
    L.cauda = x.anterior
senão
    x.proximo.anterior = x.anterior
/* Se x é a cabeça de L, então o segundo nó passa a ser L.cabeca */
se x.anterior == NULO então
    L.cabeca = x.proximo
senão
    x.anterior.proximo = x.proximo
```

Capítulo 10

Pilha e fila

Este capítulo apresenta dois tipos abstratos de dados muito básicos e que são muito importantes na descrição de vários algoritmos. Ambos oferecem operações de adição e remoção de um elemento, sendo que a operação de remoção deve sempre remover um elemento especial (não há escolha sobre em qual elemento deve ser removido).

10.1 Pilha

Pilha é uma coleção dinâmica de dados cuja operação de remoção deve remover o elemento que está na coleção há menos tempo. Essa política de remoção é conhecida como "LIFO", acrônimo para "*last in, first out*". Independente da implementação, é possível realizar ambas operações de remoção e inserção em tempo $\Theta(1)$.

Existem inúmeras aplicações para pilhas. Por exemplo, podemos verificar se uma palavra é um palíndromo ao inserir as letras em ordem e depois realizar a remoção uma a uma, verificando se a palavra formada é a mesma que a inicial. Uma outra aplicação é a operação de "desfazer/refazer", presente em vários programas e aplicativos. Toda mudança é colocada em uma pilha, de modo que cada remoção da pilha fornece a última modificação realizada. Vale mencionar também que pilhas são úteis na implementação de algoritmos de busca em profundidade em grafos.

Vamos mostrar como implementar uma pilha utilizando um vetor $P[1..m]$ com capacidade para m elementos. Ressaltamos que existem ainda outras formas de implementar pilhas. Por exemplo, poderíamos utilizar listas encadeadas para realizar essa tarefa.

Dado um vetor $P[1..m]$, manteremos um atributo $P.\texttt{topo}$ que deve sempre conter o índice

do elemento que foi inserido por último, e inicialmente é 0. O atributo $P.\texttt{capacidade}$ contém a capacidade total do vetor, que é m. Manteremos a invariante de que o vetor $P[1..P.\texttt{topo}]$ armazena os elementos da pilha em questão, onde $P[1]$ contém o elemento inserido há mais tempo na pilha e $P[P.\texttt{topo}]$ contém o mais recente. Note que o tamanho atual da pilha é dado por $P.\texttt{topo}$.

A seguir temos uma representação gráfica de uma pilha implementada em um vetor P, com $P.\texttt{capacidade} = 10$ e $P.\texttt{topo} = 4$:

	1	2	3	4	5	6	7	8	9	10
P	4	7	1	9						

$P.\texttt{topo}$

Quando inserimos um elemento x em uma pilha P, dizemos que estamos *empilhando* x em P. Similarmente, ao remover um elemento de P nós *desempilhamos* de P. Os dois procedimentos que detalham essas operações, EMPILHA e DESEMPILHA, são dados nos Algoritmos 10.1 e 10.2, respectivamente. Eles são bem simples e, como dito acima, levam tempo $\Theta(1)$ para serem executadas. Veja a Figura 10.1 para um exemplo dessas operações.

Para empilhar x em P, o procedimento EMPILHA primeiro verifica se o vetor está cheio e, caso ainda haja espaço, incrementa o valor de $P.\texttt{topo}$ e insere x em $P[P.\texttt{topo}]$.

Algoritmo 10.1: EMPILHA(P, x)

```
1 se P.topo ≠ P.capacidade então
2     P.topo = P.topo + 1
3     P[P.topo] = x
```

Para desempilhar, basta verificar se a pilha está vazia e, caso contrário, decrementar de uma unidade o valor de $P.\texttt{topo}$, devolvendo o elemento que estava no topo da pilha.

Algoritmo 10.2: DESEMPILHA(P)

```
1 se P.topo ≠ 0 então
2     x = P[P.topo]
3     P.topo = P.topo − 1
4     devolve x
5 devolve NULO
```

Um outro procedimento interessante de se ter disponível é o CONSULTA, que simplesmente devolve o valor armazenado em $P[P.\texttt{topo}]$, sem modificar sua estrutura.

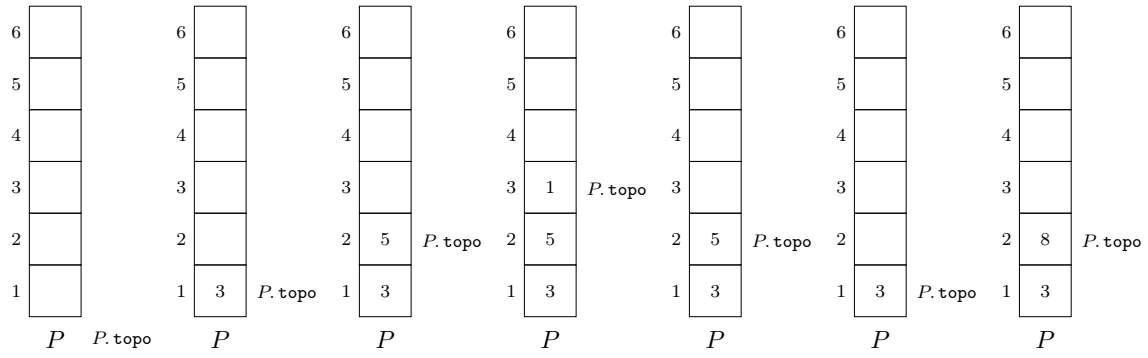


Figura 10.1: Operações em uma pilha P inicialmente vazia. Da esquerda para a direita, mostramos a pilha pós cada uma das seguintes operações: EMPILHA(P, 3), EMPILHA(P, 5), EMPILHA(P, 1), DESEMPILHA(P), DESEMPILHA(P), EMPILHA(P, 8).

10.2 Fila

Fila é uma coleção dinâmica de dados cuja operação de remoção deve remover o elemento que está na coleção há mais tempo. Essa política de remoção é conhecida como "FIFO", acrônimo para "*first in, first out*". Independente da implementação, é possível realizar ambas as operações oferecidas, de remoção e inserção, em tempo $\Theta(1)$.

O conceito de fila é amplamente utilizado em aplicações práticas. Por exemplo, qualquer sistema que controla ordem de atendimento pode ser implementado utilizando-se filas. Elas também são úteis para manter a ordem de documentos que são enviados a uma impressora. De forma mais geral, filas podem ser utilizadas em algoritmos que precisam controlar acesso a recursos, de modo que a ordem de acesso é definida pelo momento em que o recurso foi solicitado. Outra aplicação é a implementação de busca em largura em grafos.

Como acontece com pilhas, filas podem ser implementadas de diversas formas. A seguir vamos mostrar como implementar uma fila utilizando um vetor $F[1..m]$ com capacidade para m elementos. Teremos um atributo $F.\texttt{cabeca}$, que deve sempre armazenar o índice do elemento que está há mais tempo na fila. Teremos também um atributo $F.\texttt{cauda}$, que deve sempre armazenar o índice seguinte ao último elemento que foi inserido na fila. O vetor será utilizado de forma circular, o que significa que as operações de soma e subtração nos valores de $F.\texttt{cabeca}$ e $F.\texttt{cauda}$ são feitas módulo $F.\texttt{capacidade} = m$. Com isso, manteremos a invariante de que se $F.\texttt{cabeca} < F.\texttt{cauda}$, então os elementos da fila encontram-se em $F[F.\texttt{cabeca}\,..F.\texttt{cauda}-1]$, e se $F.\texttt{cabeca} > F.\texttt{cauda}$, então os elementos encontram-se em $F[F.\texttt{cabeca}\,..F.\texttt{capacidade}]$ e $F[1..F.\texttt{cauda}-1]$.

A seguir temos uma representação gráfica de uma fila implementada em um vetor F, com $F.\texttt{capacidade} = 10$, $F.\texttt{cabeca} = 3$ e $F.\texttt{cauda} = 7$:

	1	2	3	4	5	6	7	8	9	10
F			4	7	1	9				
			$F.\texttt{cabeca}$			$F.\texttt{cauda}$				

Na inicialização da estrutura, faremos $F.\texttt{cabeca} = F.\texttt{cauda} = 1$. Teremos ainda um campo $F.\texttt{tamanho}$, que é inicializado com 0, e indicará a quantidade de elementos efetivamente armazenados em F.

Quando inserimos um elemento x na fila F, dizemos que estamos *enfileirando* x em F. Similarmente, ao remover um elemento de F nós estamos *desenfileirando* de F. O dois procedimentos que detalham essas operações, Enfileira e Desenfileira, são dado respectivamente nos Algoritmos 10.3 e 10.4 e levam tempo $\Theta(1)$ para serem executadas. Veja a Figura 10.2 para um exemplo dessas operações.

Para enfileirar x em F, o procedimento Enfileira primeiro verifica se o vetor está cheio e, caso haja espaço, insere o elemento x em $F[F.\texttt{cauda}]$ e incrementa $F.\texttt{cauda}$ e $F.\texttt{tamanho}$.

Algoritmo 10.3: Enfileira(F, x)

```
1 se F.tamanho ≠ F.capacidade então
2     F[F.cauda] = x
3     se F.cauda == F.capacidade então
4         F.cauda = 1
5     senão
6         F.cauda = F.cauda +1
7     F.tamanho = F.tamanho +1
```

Para desenfileirar, basta verificar se a fila está vazia e, caso contrário, devolver o elemento em $F[F.\texttt{cabeca}]$. Veja que o valor de $F.\texttt{cabeca}$ precisa ser incrementado e $F.\texttt{tamanho}$ precisa der decrementado. Um outro procedimento interessante é o Consulta, que apenas devolve o valor em $F[F.\texttt{cabeca}]$.

Algoritmo 10.4: DESENFILEIRA(F)

```
se F.tamanho ≠ 0 então
    x = F[F.cabeca]
    se F.cabeca == F.capacidade então
        F.cabeca = 1
    senão
        F.cabeca = F.cabeca +1
    F.tamanho = F.tamanho −1
    devolve (x)
devolve NULO
```

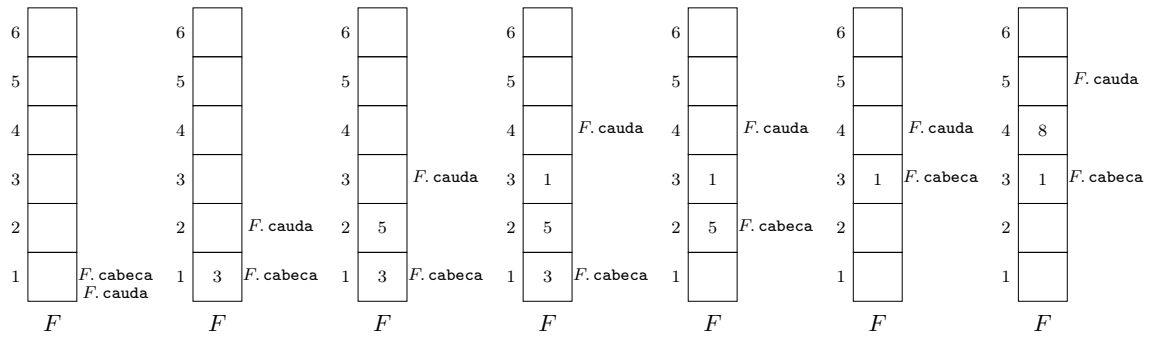


Figura 10.2: Operações em uma fila F inicialmente vazia. Da esquerda para a direita, mostramos a fila pós cada uma das seguintes operações: ENFILEIRA(F, 3), ENFILEIRA(F, 5), ENFILEIRA(F, 1), DESENFILEIRA(F), DESENFILEIRA(F), ENFILEIRA(F, 8).

Capítulo 11

Árvores

Árvores são estruturas não lineares constituídas de nós, onde cada nó x contém um elemento cuja chave está armazenada em $x.$`chave` e pode ter uma ou mais ligações para outros nós. Mais especificamente, árvores são estruturas hierárquicas nas quais cada nó nos dá acesso aos nós imediatamente "abaixo" dele na hierarquia, que são seus *filhos*. Um nó que não possui filhos é uma *folha* da árvore. Um nó especial é a *raiz*, que é o topo da hierarquia e está presente no *nível 0* da árvore. Para cada nó x que não é raiz da árvore, imediatamente acima de x nessa hierarquia existe somente um nó que tem ligação com x, chamado de *pai* de x. Note que a raiz de uma árvore é o único nó que não possui pai.

Se x é um nó qualquer, um *descendente* de x é qualquer nó que possa ser alcançado a partir de x seguindo ligações por meio de filhos. Um *ancestral* de x é qualquer nó a partir do qual pode-se seguir por filhos e alcançar x. Um nó x e seus descendentes formam uma *subárvore*, chamada *subárvore enraizada em* x.

Um nó que é filho da raiz está no nível 1, um nó que é filho de um filho da raiz está no nível 2, e assim por diante. Formalmente, o *nível* (ou profundidade) de um nó x é a quantidade de ligações no caminho (único) entre a raiz da árvore e x. A *altura* de um nó x é a quantidade de ligações no maior caminho entre x e uma folha. A altura da raiz define a *altura da árvore*. Equivalentemente, a altura de uma árvore é igual ao maior nível de uma folha. Veja na Figura 11.1 um exemplo de árvore e exemplos das nomenclaturas acima.

Uma árvore pode ser definida recursivamente da seguinte forma: uma árvore é vazia ou é um nó que possui ligações para outras árvores. De fato, muitos algoritmos que lidam com árvores são recursivos e ficam muito mais simples quando escritos dessa forma.

Em uma árvore, somente temos acesso direto ao nó raiz e qualquer manipulação, portanto,

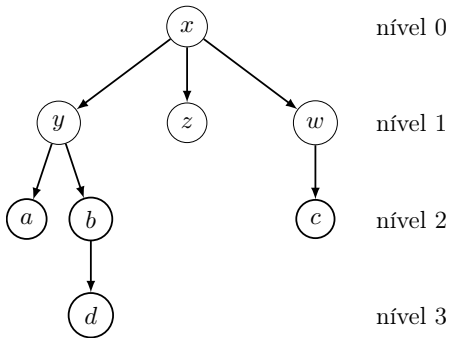


Figura 11.1: Árvore com 4 níveis e altura 3, onde (i) x é o nó raiz (nível 0), (ii) y, z e w são filhos de x, (iii) y é pai de a e b, (iv) a, d, z e c são folhas. Note que a altura do nó z é zero e a altura do nó y é 2. Os nós a, b e d são descendentes de y e, juntamente com y, formam a subárvore enraizada em y

deve utilizar as ligações entre os nós. Note que qualquer busca por uma chave precisa ser feita percorrendo-se a árvore inteira (no pior caso). Assim, no pior caso uma busca em árvore é executada em tempo linear na quantidade de nós, do mesmo modo que uma busca em uma lista ligada.

Na Seção 11.1 apresentamos árvores binárias de busca, que são árvores onde os elementos são distribuídos dependendo da relação entre as chaves dos elementos armazenados. Isso nos possibilita realizar procedimentos de busca, inserção e remoção de uma forma sistemática em tempo proporcional à altura da árvore. Na Seção 11.2 apresentamos algumas árvores binárias de busca que, mesmo após inserções e remoções, têm altura $O(\log n)$, onde n é a quantidade de elementos armazenados.

11.1 Árvores binárias de busca

Árvores binárias são árvores em que qualquer nó possui *no máximo* dois filhos. Por isso, podemos referir aos dois filhos de um nó sempre como *filho esquerdo* e *filho direito*. É possível também definir árvore binária de uma forma recursiva.

Definição 11.1

Uma *árvore binária* é uma árvore vazia ou é um nó que possui ligações para duas árvores binárias.

O filho esquerdo (resp. direito) de um nó x é raiz da *subárvore esquerda* (resp. *direita*) de x. Formalmente, se x é um nó, então x contém os atributos $x.$`chave`, $x.$`esq` e $x.$`dir`,

onde $x.\texttt{chave}$ contém a chave do elemento x e em $x.\texttt{esq}$ e $x.\texttt{dir}$ temos, respectivamente, as raízes das subárvores esquerda e direita (ou NULO, caso x não tenha aquele filho). Para toda folha x temos $x.\texttt{esq} = x.\texttt{dir} =$ NULO. Por clareza, vamos assumir que todas as chaves dos nós de uma árvore são diferentes entre si.

A seguinte definição apresenta uma importante árvore binária, que nos permite realizar diversas operações em tempo proporcional à altura da árvore.

Definição 11.2

Uma *árvore binária de busca* é uma árvore binária em que, para cada nó x, todos os nós da subárvore esquerda de x possuem chaves menores do que $x.\texttt{chave}$ e todos os nós da subárvore direita possuem chaves maiores do que $x.\texttt{chave}$.

Nas Seções 11.1.1, 11.1.2 e 11.1.3 apresentamos as operações de busca, inserção e remoção de uma chave, respectivamente, em árvores binárias de busca, todas com tempo proporcional à altura da árvore. Outras operações úteis são apresentadas na Seção 11.1.4.

11.1.1 Busca em árvores binárias de busca

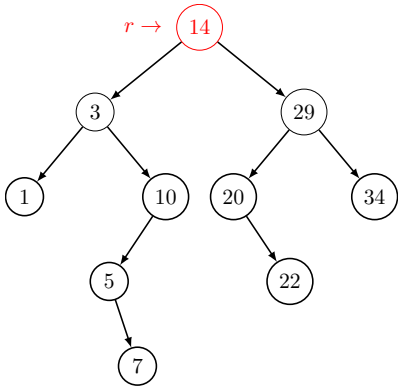
Dado o endereço r da raiz de uma árvore binária de busca (ABB) e uma chave k a ser buscada, o procedimento BuscaABB(r, k) devolve um ponteiro para o nó que contém a chave k ou NULO, caso não exista elemento com chave k na árvore dada.

A ideia do algoritmo é semelhante à ideia utilizada na busca binária em vetores ordenados. Começamos comparando k com $r.\texttt{chave}$, de forma que temos 3 resultados possíveis: (i) caso $k = r.\texttt{chave}$, encontramos o elemento e o ponteiro para r é devolvido pelo algoritmo; (ii) caso $k < r.\texttt{chave}$, sabemos que se um elemento de chave k estiver na árvore, então ele deve estar na subárvore esquerda, de forma que podemos executar o mesmo procedimento para procurar k em $r.\texttt{esq}$; e (iii) caso $k > r.\texttt{chave}$, então o elemento de chave k só pode estar na subárvore direita, de forma que executamos o procedimento sobre $r.\texttt{dir}$. Esse procedimento está formalizado no Algoritmo 11.1 e a figura 11.2 mostra um exemplo de execução.

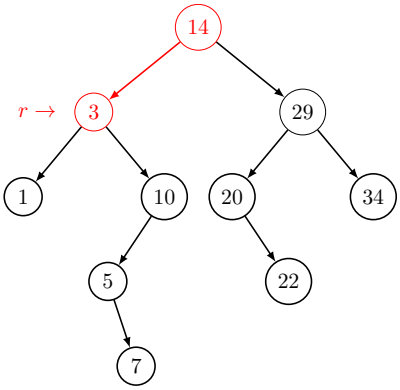
É um bom exercício provar que o algoritmo funciona corretamente por indução na altura do nó r dado como parâmetro. Note que, a cada chamada recursiva, o algoritmo "desce" na árvore, indo em direção às folhas, cujos filhos certamente satisfazem uma condição do caso base. Portanto, os nós encontrados durante a execução do algoritmo formam um caminho descendente começando na raiz da árvore. Assim, o tempo de execução de BuscaABB em uma árvore de altura h é $O(h)$, i.e., no pior caso é proporcional à altura da árvore. Note que se h for assintoticamente menor que n (i.e., $h = o(n)$), então pode ser mais eficiente realizar uma busca nessa árvore do que em uma lista ligada que armazene os mesmos elementos.

Algoritmo 11.1: BUSCAABB(r, k)

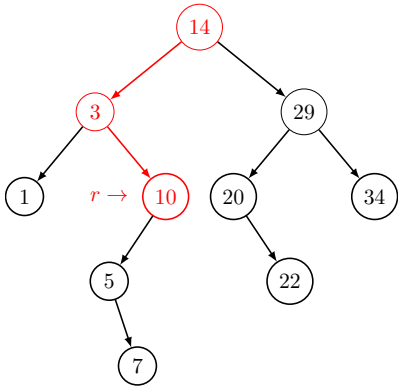
```
1 se r == NULO ou k == r.chave então
2     devolve r
3 senão se k < r.chave então
4     devolve BUSCAABB(r.esq, k)
5 senão
6     devolve BUSCAABB(r.dir, k)
```



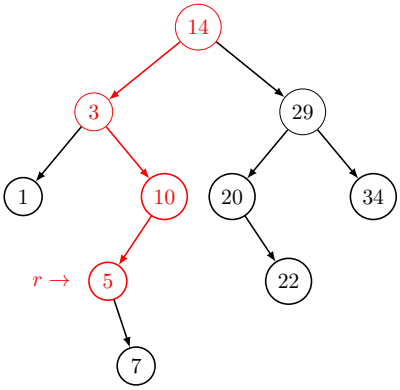
(a) $5 < 14$, chama BUSCAABB(r.esq, 5).



(b) $5 > 3$, chama BUSCAABB(r.dir, 5).



(c) $5 < 10$, chama BUSCAABB(r.esq, 5).



(d) Encontrou 5.

Figura 11.2: Execução de BUSCAABB(r, 5). O caminho percorrido está destacado em vermelho.

11.1.2 Inserção em árvores binárias de busca

Dado o endereço r da raiz de uma árvore binária de busca e o endereço de um nó x, o procedimento InsereABB(r, x) tenta inserir x na árvore, devolvendo o endereço para a raiz da árvore depois de modificada. Vamos assumir que o novo nó x que deseja-se inserir tem $x.\texttt{esq} = x.\texttt{dir} =$ NULO.

Se a árvore está inicialmente vazia, então o nó x será a nova raiz. Caso contrário, o primeiro passo do algoritmo é buscar por $x.\texttt{chave}$ na árvore. Se $x.\texttt{chave}$ não estiver na árvore, então a busca terminou em um nó y que será o pai de x: se $x.\texttt{chave} < y.\texttt{chave}$, então inserimos x à esquerda de y e caso contrário o inserimos à direita. Note que qualquer busca posterior por $x.\texttt{chave}$ vai percorrer exatamente o mesmo caminho e chegar corretamente a x. Portanto, essa inserção mantém a invariante de que a árvore é binária de busca. Não é difícil perceber que o tempo de execução desse algoritmo também é $O(h)$. O Algoritmo 11.2 mostra o procedimento InsereABB, e a figura 11.3 mostra um exemplo de execução.

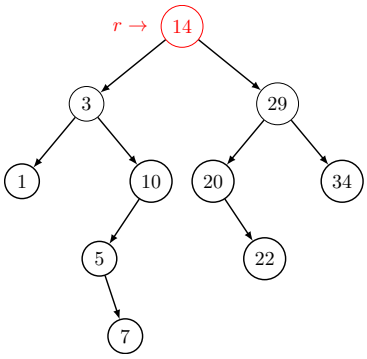
Algoritmo 11.2: InsereABB(r, x)

```
1 se r == NULO então
2     devolve x
3 se x.chave < r.chave então
4     r.esq = InsereABB(r.esq, x)
5 senão se x.chave > r.chave então
6     r.dir = InsereABB(r.dir, x)
7 devolve r
```

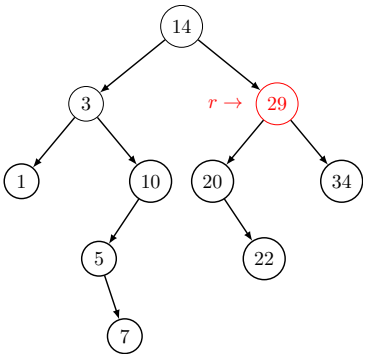
11.1.3 Remoção em árvores binárias de busca

Dado o endereço r da raiz de uma árvore binária de busca e uma chave k, o procedimento RemoveABB(r, k) remove o nó x tal que $x.\texttt{chave} = k$ da árvore (se ele existir) e devolve o endereço da para a raiz da árvore depois de modificada. Esse procedimento deve ser bem cuidadoso, pois precisa garantir que a árvore continue sendo de busca após sua execução.

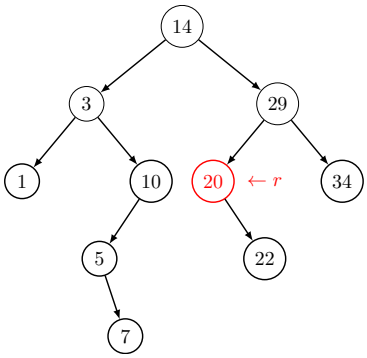
Observe que remoção de um nó x depende da quantidade de filhos de x. Quando ele não possui filhos, basta remover x, modificando o atributo $z.\texttt{esq}$ ou $z.\texttt{dir}$ de seu pai z, caso este exista, para ser NULO em vez de x. Se x possui somente um filho w, então basta colocar w no lugar de x, modificando o atributo $z.\texttt{esq}$ ou $z.\texttt{dir}$ do pai z de x, caso este exista, para apontar para w em vez de x. O caso que requer um pouco mais de atenção ocorre quando x possui dois filhos. Nesse caso, um bom candidato a substituir x na árvore é seu *sucessor* y,



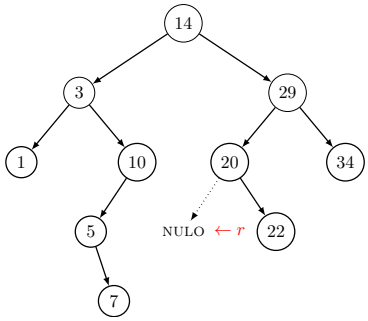
(a) 18 > 14, chama INSERE-ABB(r. `dir`, 18).



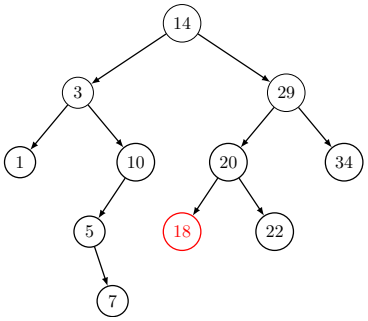
(b) 18 < 29, chama INSERE-ABB(r. `esq`, 18).



(c) 18 < 20, chama INSERE-ABB(r. `esq`, 18).

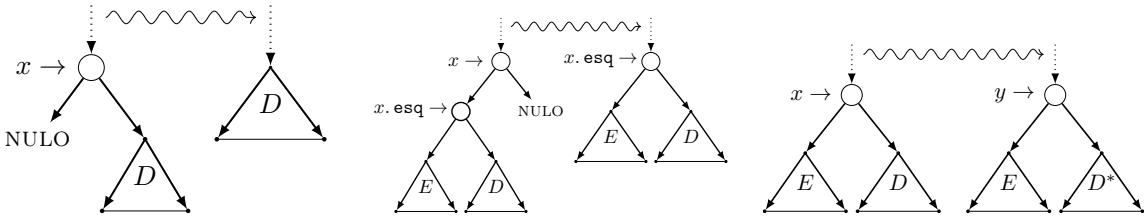


(d) r = NULO, cria nó para o 18.



(e) Ligações são atualizadas recursivamente.

Figura 11.3: Execução de INSEREABB(r, 18).



(a) Caso 1: x não tem filho esquerdo.

(b) Caso 2: x não tem filho direito mas tem esquerdo.

(c) Caso 3: x tem os dois filhos. O nó y sucessor está contido na subárvore D e é removido da mesma.

Figura 11.4: Esquema dos três casos de remoção de um nó x em uma árvore binária de busca.

que é o nó cuja chave é o menor valor que é maior do que $x.\texttt{chave}$. Isso porque com x na árvore, temos $x.\texttt{esq}.\texttt{chave} \leq x.\texttt{chave} \leq x.\texttt{dir}.\texttt{chave}$. Ao substituir x pelo seu sucessor, continuaremos tendo $x.\texttt{esq}.\texttt{chave} \leq y.\texttt{chave} \leq x.\texttt{dir}.\texttt{chave}$, mantendo a característica da busca. Observe que, pela propriedade de árvore binária de busca, o nó y que é sucessor de qualquer nó x que tem dois filhos sempre está contido na subárvore direita de x.

Assim, vamos dividir o procedimento de remoção de um nó x nos seguintes três casos, que não são explicitamente os três mencionados acima, mas lidam com todas as particularidades mencionadas:

1. se x não possui filho esquerdo, então substituímos x por $x.\texttt{dir}$ (note que caso x não tenha filho direito, teremos x substituído por NULO, que é o esperado);

2. se x tem filho esquerdo mas não tem filho direito, então substituímos x por $x.\texttt{esq}$;

3. se x tem os dois filhos, então substituímos x por seu sucessor y e removemos y de seu local atual (o que pode ser feito recursivamente).

Note que o procedimento que acabamos de descrever mantém a propriedade de busca da árvore. A Figura 11.4 exemplifica todos os casos descritos acima.

Antes de apresentarmos o algoritmo para remoção de um nó da árvore, precisamos saber como encontrar o sucessor de um elemento x em uma árvore binária de busca. Veja que quando precisamos encontrar o sucessor y de x, sabemos que x possui os dois filhos e que $y.\texttt{chave}$ é o menor elemento da subárvore com raiz $x.\texttt{dir}$. Assim, basta encontrarmos o menor elemento de uma subárvore binária de busca. Para encontrar o menor elemento de uma árvore com raiz r, executamos MinimoABB(r), que é apresentado no Algoritmo 11.3.

Quando r é a raiz da árvore, o procedimento MinimoABB segue um caminho de r até uma folha, seguindo sempre pelo filho esquerdo. Dessa forma, o tempo de execução de MinimoABB(r) é, no pior caso, proporcional à altura da árvore.

Algoritmo 11.3: MINIMOABB(r)

```
1 se r.esq ≠ NULO então
2     devolve MINIMOABB(r.esq)
3 devolve r
```

Voltemos nossa atenção ao procedimento REMOVEABB, que remove um elemento com chave k de uma árvore binária de busca cuja raiz é x. Ele é formalizado no Algoritmo 11.4, que contém os três casos de remoção que discutimos acima. Ele devolve a raiz da árvore modificada após a remoção do elemento.

Algoritmo 11.4: REMOVEABB(x, k)

```
1  se x == NULO então
2      devolve NULO
3  se k < x.chave então
4      x.esq = REMOVEABB(x.esq, k)
5  senão se k > x.chave então
6      x.dir = REMOVEABB(x.dir, k)
   /* Aqui temos x.chave = k                                          */
7  senão
8      se x.esq == NULO então
9          x = x.dir
10     senão se x.dir == NULO então
11         x = x.esq
       /* O sucessor de x é o nó de menor chave da subárvore direita, que
          precisa ser removido de lá                                   */
12     senão
13         y = MINIMOABB(x.dir)
14         x.chave = y.chave
15         x.dir =REMOVEABB(x.dir, y.chave)
16 devolve x
```

Note que caso uma execução do algoritmo entre no terceiro caso (teste da linha 12), onde o nó que contém a chave a ser removida possui dois filhos, o algoritmo executa REMOVEABB(x.dir, y.chave) na linha 15. Dentro dessa chamada, o algoritmo simplesmente vai

"descendo para a esquerda" na árvore, i.e., executa repetidas chamadas a RemoveABB na linha 4, até que entra no **senão** da linha 7, onde vai entrar no primeiro caso, por se tratar de uma folha. Resumindo, uma vez que o algoritmo entra no caso 3, o próximo caso que entrará será o caso 1 e então a remoção é finalizada.

Resumindo, a ideia de execução de RemoveABB(r, k) é como segue: até que o nó x com chave k seja encontrado, realizamos uma busca por k de forma recursiva, gastando tempo proporcional a um caminho de r até x. Após encontrar x, pode ser necessário executar MinimoABB($x.$`dir`) para encontrar o sucessor y de x, o que leva tempo proporcional à altura de x, e executar RemoveABB($x.$`dir`, $y.$`chave`), que também leva tempo proporcional à altura de x, porque essa chamada a RemoveABB vai fazer o mesmo percurso que MinimoABB fez (ambos estão em busca de y). Como y não pode ter filho esquerdo, sua remoção é feita diretamente. Portanto, o tempo de execução de RemoveABB(r, k) é, ao todo, $O(h)$, onde h é a altura da árvore. A Figura 11.5 exemplifica a execução do algoritmo de remoção.

11.1.4 Outras operações sobre árvores binárias de busca

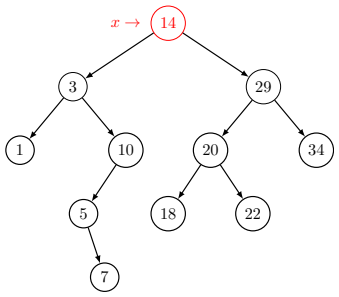
Diversas outras operações podem ser realizadas em árvores binárias de busca de forma eficiente:

- Encontrar o menor elemento: basta seguir os filhos esquerdos a partir da raiz até chegar em um nó que não tem filho esquerdo – este contém o menor elemento da árvore. Tempo necessário: $O(h)$.

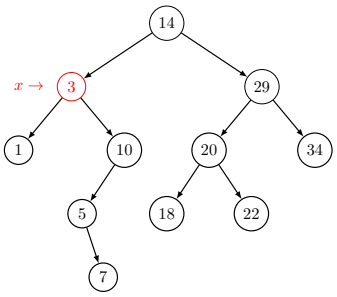
- Encontrar o maior elemento: basta seguir os filhos direitos a partir da raiz até chegar em um nó que não tem filho direito – este contém o maior elemento da árvore. Tempo necessário: $O(h)$.

- O sucessor de um elemento k (o menor elemento que é maior do que k): seja x o nó tal que $x.$`chave` $= k$. Pela estrutura da árvore, se x tem um filho direito, então o sucessor de k é o menor elemento armazenado nessa subárvore direita. Caso x não tenha filho direito, então o primeiro nó que contém um elemento maior do que k deve estar em um ancestral de x: é o nó de menor chave cujo filho esquerdo também é ancestral de x. Veja a Figura 11.6 para exemplos de elementos sucessores. Tempo necessário: $O(h)$.

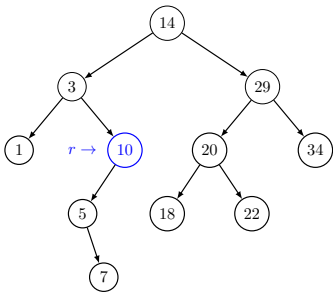
- O predecessor de um elemento k: se x é o nó cuja chave é k, o predecessor de k é o maior elemento da subárvore enraizada no filho esquerdo de x ou então é o maior ancestral cujo filho direito também é ancestral de x. Tempo necessário: $O(h)$.



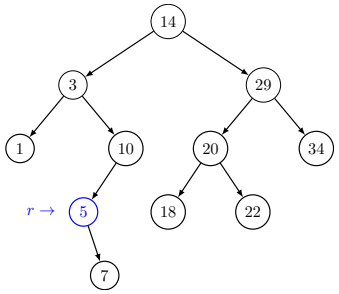
(a) $3 < 14$, chama REMOVEABB($x.\texttt{esq}$, 3).



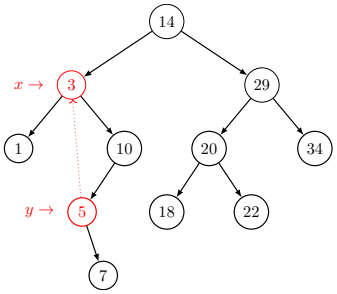
(b) Encontrou 3 e tem 2 filhos. Chama MINIMOABB($x.\texttt{dir}$).



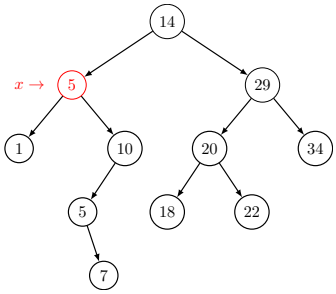
(c) $r.\texttt{esq} \neq$ NULO. Chama MINIMOABB($r.\texttt{esq}$).



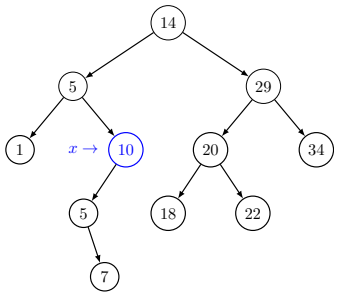
(d) $r.\texttt{esq} =$ NULO. Devolve r.



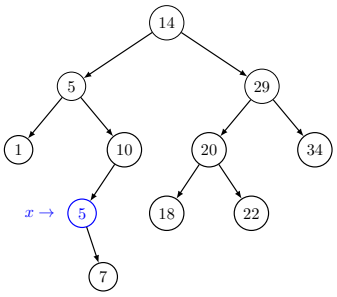
(e) Copia chave de y para x.



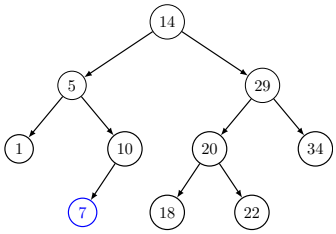
(f) Chama REMOVEABB($x.\texttt{dir}, 5$).



(g) $5 < 10$, chama REMOVEABB($x.\texttt{esq}, 5$).



(h) Encontrou 5 e tem 1 filho. Apenas remove x.



(i) Ligações são arrumadas recursivamente.

Figura 11.5: Execução de REMOVEABB(r, 3).

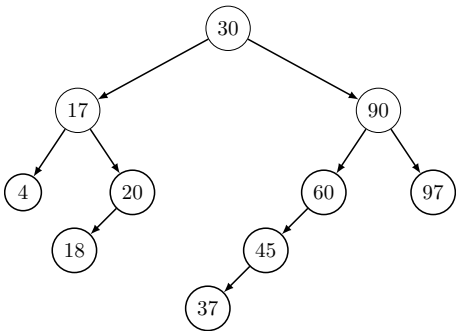


Figura 11.6: Exemplo de árvore binária de busca onde o sucessor de 30 é o 37 (menor nó da subárvore enraizada em 90) e o sucessor de 20 é o 30 (menor ancestral do 20 cujo filho esquerdo, o 17, também é ancestral do 20).

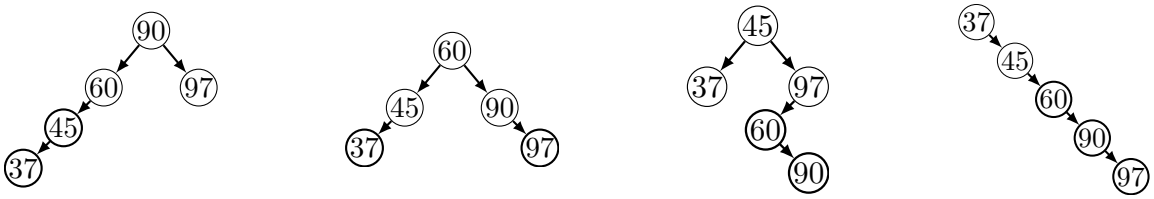


Figura 11.7: Árvores distintas formadas pela inserção dos elementos 37, 45, 60, 90 e 97 em diferentes ordens. Da esquerda para direita as ordens são (90, 60, 97, 45, 37), (60, 45, 37, 90, 97), (45, 37, 97, 60, 90), (37, 45, 60, 90, 97)

11.1.5 Altura de uma árvore binária de busca

Comece lembrando que a altura de uma árvore binária é determinada pela quantidade de ligações no maior caminho entre a raiz e uma folha da árvore. Desde que respeite a propriedade de busca, a inserção de elementos para criar uma árvore binária de busca pode ser feita em qualquer ordem. Um mesmo conjunto de elementos, dependendo da ordem na qual são inseridos em uma árvore, pode dar origem a árvores diferentes (veja a Figura 11.7). Por exemplo, considere a situação em que criaremos uma árvore binária de busca com chaves $x_1, \ldots, x_n$, onde $x_1 < \ldots < x_n$. A ordem em que os elementos serão inseridos é x_1, $x_2, \ldots, x_n$. Esse procedimento irá gerar uma árvore com altura n, composta por um único caminho. Note que uma mesma árvore pode ser gerada por diferentes ordens de inserções. Por exemplo, inserir elementos com chaves 10, 15, 18 e 20 em uma árvore inicialmente vazia utilizando qualquer uma das ordens (18, 15, 20, 10), (18, 15, 10, 20) e (18, 20, 15, 10) gera a mesma árvore.

Dada uma árvore binária de busca com n elementos e altura h, vimos que as operações de busca, inserção e remoção são executadas em tempo $O(h)$. Porém, como vimos, uma árvore binária de busca com n nós pode ter altura $h = n$ e, portanto, em tais árvores essas operações não podem ser executadas mais eficientemente que em uma lista ligada. Uma solução natural

para ganhar em eficiência é de alguma forma garantir que a altura da árvore seja tão pequena quanto possível.

Vamos estimar qual a menor altura possível de uma árvore binária. Como cada nó tem no máximo dois filhos, dada uma árvore binária de altura h com n nós, sabemos que $n \leq 2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$. Assim, temos $2^{h+1} \geq n + 1$, de onde concluímos que (aplicando log dos dois lados) $h \geq \log(n+1) - 1$. Portanto, obtemos o seguinte resultado

Teorema 11.3

A altura de uma árvore binária com n nós é $\Omega(\log n)$.

Uma árvore binária é dita *completa* se todos os seus níveis estão completamente preenchidos. Note que árvores binárias completas com altura h possuem exatamente $2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$ nós. Similar ao que fizemos para obter o Teorema 11.3, concluímos que a altura h de uma árvore completa T com n nós é dada por $h = \log(n+1) - 1 = O(\log n)$. Uma árvore binária com altura h é dita *quase completa* se os níveis $0, 1, \ldots, h-1$ têm todos os nós possíveis (i,e, o último nível é o único que pode não estar preenchido totalmente). Claramente, toda árvore completa é quase completa.

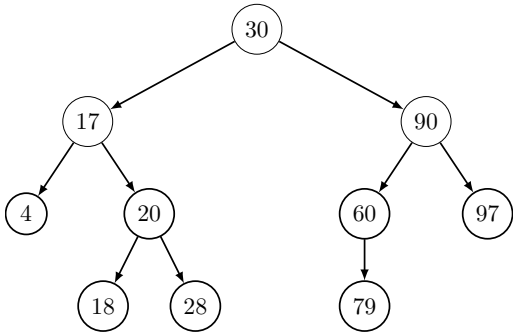


Figura 11.8: Uma árvore binária quase completa.

Como todas as operações discutidas nesta seção são executadas em tempo proporcional à altura da árvore e vimos que qualquer árvore binária tem altura $\Omega(\log n)$, idealmente queremos árvores binárias de altura $O(\log n)$.

11.2 Árvores balanceadas

Árvores balanceadas são árvores de busca com que garantem manter altura $O(\log n)$ quando n elementos estão armazenados. Ademais, essa propriedade se mantém após qualquer sequência de inserções e remoções. Note que ao manter a altura $O(\log n)$, diretamente garantimos que as operações de busca, inserção e remoção, e as operações vistas na Seção 11.1.4 são todas $O(\log n)$.

Esta seção ainda está em construção.

Capítulo 12

Fila de prioridades

Uma *fila de prioridades* é uma coleção dinâmica de elementos que possuem prioridades associadas e cuja operação de remoção deve remover o elemento que possui maior prioridade. Ela é um tipo abstrato de dados que oferece, além da remoção de elementos, consulta ao elemento de maior prioridade, inserção de um novo elemento, alteração da prioridade de um elemento já armazenado e construção a partir de um conjunto pré-existente de elementos.

É importante perceber que o termo *prioridade* é usado de maneira genérica, no sentido de que ter maior prioridade não significa necessariamente que o *valor* indicativo da prioridade é o maior. Por exemplo, se falamos de atendimento em um banco e a prioridade de atendimento é indicada pela idade da pessoa, então tem maior prioridade a pessoa que tiver maior idade. Por outro lado, se falamos de gerenciamento de estoque de remédios em uma farmácia e a prioridade de compra é indicada pela quantidade em estoque, então tem maior prioridade o remédio que estiver em menor quantidade.

O conceito de fila de prioridades é muito utilizado em aplicações práticas. Além das mencionadas acima, elas são usadas para gerar implementações eficientes de alguns algoritmos clássicos, como Dijkstra e Kruskal. Filas de prioridades podem ser implementadas de diversas formas, como por exemplo em um vetor ou lista ligada ordenados pela prioridade dos elementos, em uma árvore de busca ou mesmo em uma fila. Essas implementações, no entanto, não fornecem uma implementação eficiente de todas as operações desejadas.

A seguir apresentamos a estrutura de dados *heap binário*, que permite implementar as operações desejadas de modo eficiente. Utilizando heaps binários é possível realizar inserção, remoção e alteração de um elemento em tempo $O(\log n)$, consulta pelo elemento de maior prioridade em tempo $\Theta(1)$ e construção a partir de um conjunto já existente em tempo $O(n)$.

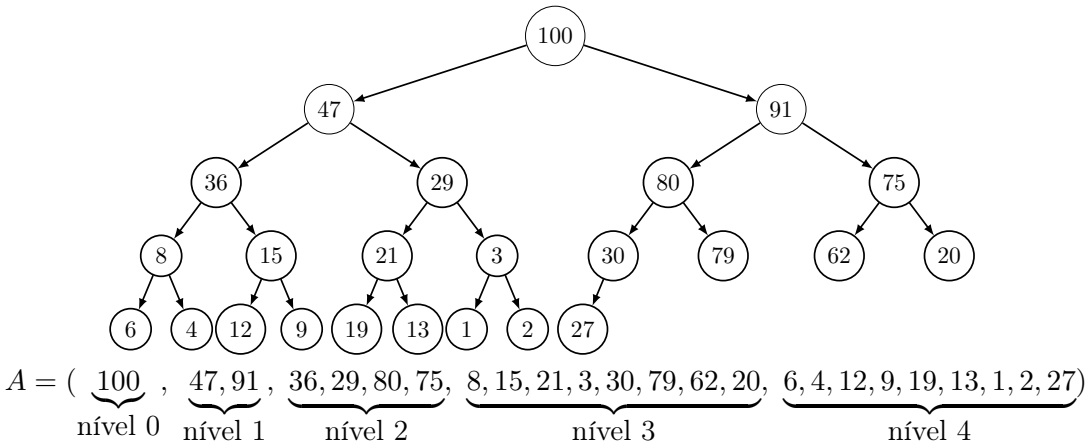


Figura 12.1: Exemplo de um vetor A observado como árvore binária. A árvore enraizada na posição 4 possui os elementos 36, 15, 8, 12, 9, 6 e 4 (que estão nas posições 4, 8, 9, 16, 17, 18 e 19 do vetor).

12.1 Heap binário

Qualquer vetor A com n elementos pode ser visto de forma conceitual como uma árvore binária quase completa em que todos os níveis estão cheios, exceto talvez pelo último, que é preenchido de forma contígua da esquerda para a direita. Formalmente, isso é feito da seguinte forma. O elemento na posição i tem filho esquerdo na posição $2i$, se $2i \leq n$, filho direito na posição $2i + 1$, se $2i + 1 \leq n$, e pai na posição $\lfloor i/2 \rfloor$, se $i > 1$. Assim, ao percorrer o vetor A da esquerda para a direita, estamos acessando todos os nós de um nível ℓ consecutivamente antes de acessar os nós do nível $\ell + 1$. Além disso, um elemento na posição i de A tem altura $\lfloor \log(n/i) \rfloor$ e está no nível $\lfloor \log i \rfloor$. Perceba que podemos falar sobre uma subárvore enraizada em um elemento $A[i]$ ou em uma posição i. Também podemos falar de uma subárvore formada por um subvetor $A[1..k]$ para qualquer $1 \leq k \leq n$. Veja a Figura 12.1 para um exemplo.

Um *heap (binário)* é uma estrutura de dados que implementa o tipo abstrato de dados fila de prioridades. Em geral, um heap é implementado em um vetor, que é a estrutura que usaremos. Por isso, no que segue vamos que cada elemento x possui um atributo $x.$`prioridade`, que guarda o valor referente à prioridade de x, e um atributo $x.$`indice`, que guarda o índice em que x está armazenado no vetor. Esse último atributo é importante para operações de alteração de prioridades, uma vez que, como veremos, heaps não fornecem a operação de busca de forma eficiente. Formalmente, dizemos que um vetor H *é um heap* se ele satisfaz a *propriedade de heap*.

Definição 12.1: ***Propriedade de heap***

Em um heap, todo nó deve ter prioridade maior ou igual à prioridade de seus filhos, se eles existirem.

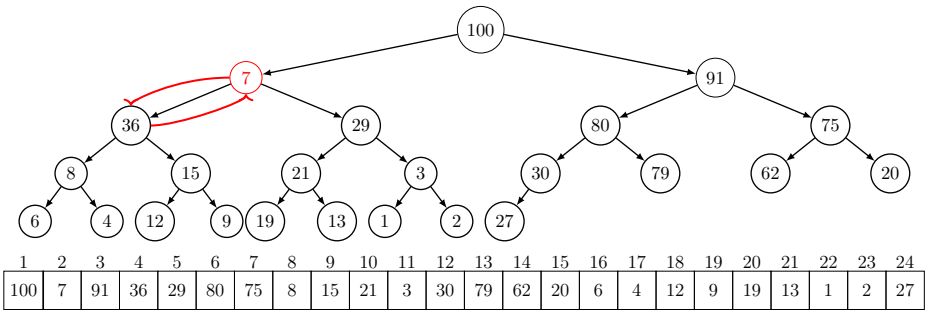
Em outras palavras, um vetor H com n elementos é um heap se para todo i, com $2 \leq i \leq n$, temos $H[\lfloor i/2 \rfloor].\texttt{prioridade} \geq H[i].\texttt{prioridade}$, i.e., a prioridade do pai é sempre maior ou igual à prioridade de seus filhos. Se considerarmos que os valores no vetor da Figura 12.1 são as prioridades dos elementos, então note que tal vetor é um heap.

No que segue, seja H um vetor que armazena $n = H.\texttt{tamanho}$ elementos em que assumimos que a quantidade máxima possível de elementos que podem ser armazenados em H está salva no atributo $H.\texttt{capacidade}$. Assim, vamos considerar neste momento que as primeiras n posições do vetor H formam um heap.

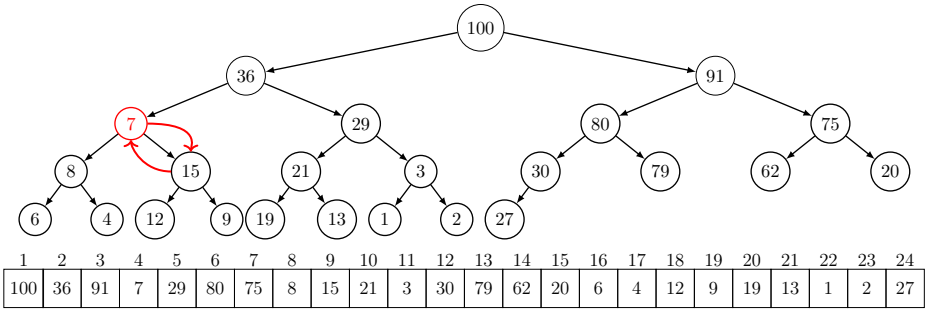
Perceba que a propriedade de heap garante que $H[1]$ sempre armazena o elemento de maior prioridade do heap. Assim, a operação de consulta ao elemento de maior prioridade, ConsultaHeap(H), se dá em tempo $\Theta(1)$. Essa operação devolve o elemento de maior prioridade, mas não faz modificações na estrutura. Nas seções seguintes, discutiremos cada uma das outras quatro operações fornecidas pela estrutura (remoção, inserção, construção e alteração). Antes disso, precisamos definir dois procedimentos muito importantes que serão utilizados por todas elas.

As quatro operações principais fornecidas por uma fila de prioridades podem perturbar a estrutura, de forma que precisamos ser capazes de restaurar a propriedade de heap, se necessário. Os procedimentos CorrigeHeapDescendo e CorrigeHeapSubindo, formalizados nos Algoritmos 12.1 e 12.2, respectivamente, e discutidos a seguir, têm como objetivo restaurar a propriedade de heap quando apenas um dos elementos está causando a falha.

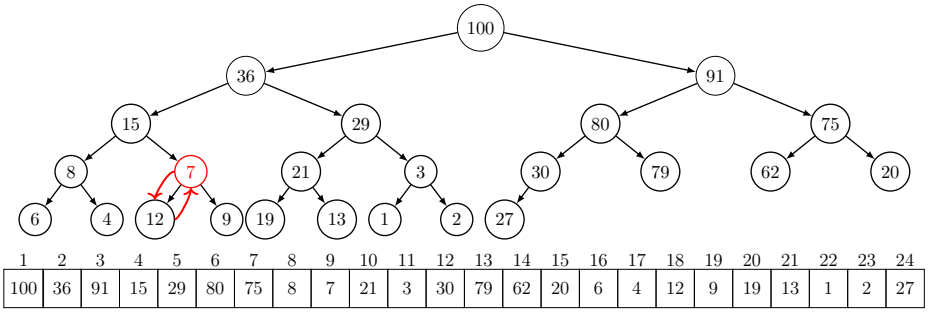
O algoritmo CorrigeHeapDescendo recebe um vetor H e um índice i tal que as subárvores enraizadas em $H[2i]$ e $H[2i+1]$ já são heaps. O objetivo dele é transformar a árvore enraizada em $H[i]$ em heap. Veja que se $H[i]$ não tem prioridade maior ou igual à de seus filhos, então basta trocá-lo com o filho que tem maior prioridade para restaurar *localmente* a propriedade. Potencialmente, o filho alterado pode ter causado falha na prioridade também. Por isso, fazemos trocas sucessivas entre pais e filhos até que atingimos um nó folha ou até que não tenhamos mais falha na propriedade. Esse comportamento dá a sensação de que o elemento que estava na posição i inicialmente está "descendo" por um ramo da árvore. Observe que durante essas trocas, os índices onde os elementos estão armazenados mudam, de forma que precisamos mantê-los atualizados também. A Figura 12.2 mostra um exemplo de execução desse algoritmo.



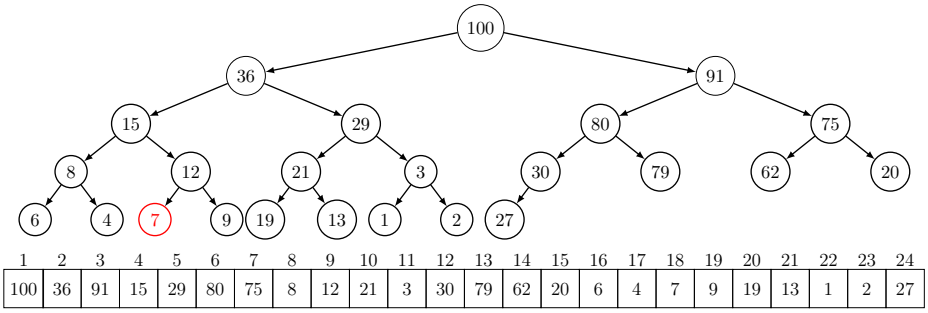
(a) $H[4] > H[2]$: troca e chama CorrigeHeapDescendo$(H, 4)$.



(b) $H[9] > H[4]$: troca e chama CorrigeHeapDescendo$(H, 9)$.



(c) $H[18] > H[9]$: troca e chama CorrigeHeapDescendo$(H, 18)$.



(d) Posição 18 é folha: faz nada.

Figura 12.2: Execução de CorrigeHeapDescendo$(H,\ 2)$ (Algoritmo 12.1) sobre o vetor $H = (100, 7, 91, 36, 29, 80, 75, 8, 15, 21, 3, 30, 79, 62, 20, 6, 4, 12, 9, 19, 13, 1, 2, 27)$. As setas vermelhas indicam trocas de valores. Essas trocas podem ser visualizadas na árvore, porém lembre-se que são feitas no vetor.

Algoritmo 12.1: CorrigeHeapDescendo(H, i)

```
  /* Guardar em maior o índice do maior dentre o pai e os dois filhos (se eles
     existirem)                                                              */
1 maior = i
2 se 2i ≤ H.tamanho e H[2i].prioridade > H[maior].prioridade então
3     maior = 2i
4 se 2i + 1 ≤ H.tamanho e H[2i + 1].prioridade > H[maior].prioridade então
5     maior = 2i + 1
  /* Trocar maior filho com o pai, se o pai já não for o maior               */
6 se maior ≠ i então
7     troca H[i].indice com H[maior].indice
8     troca H[i] com H[maior]
      /* A troca pode ter estragado a subárvore enraizada em maior           */
9     CorrigeHeapDescendo(H, maior)
```

O Teorema 12.2 a seguir mostra que o CorrigeHeapDescendo de fato consegue transformar a árvore enraizada em $H[i]$ em um heap.

Teorema 12.2: ***Corretude de* CorrigeHeapDescendo**

O algoritmo CorrigeHeapDescendo recebe um vetor H e um índice i tal que as subárvores enraizadas em $H[2i]$ e $H[2i+1]$ são heaps, e modifica H de modo que a árvore enraizada em $H[i]$ é um heap.

Demonstração. Seja h_x a altura de um nó que está na posição x na heap (isto é, $h_x = \lfloor \log(n/x) \rfloor$).

Vamos provar o resultado por indução na altura h_i do nó i.

Se $h_i = 0$, o nó deve ser uma folha, que por definição é uma heap (de tamanho 1). O algoritmo não faz nada nesse caso, já que folhas não possuem filhos e, portanto, está correto.

Suponha que o CorrigeHeapDescendo(H, k) corretamente transforma $H[k]$ em heap se $H[2k]$ e $H[2k+1]$ já forem heaps, para todo nó na posição k tal que $h_k < h_i$. Precisamos mostrar que CorrigeHeapDescendo(H, i) funciona corretamente, i.e., a árvore com raiz $H[i]$ é um heap ao fim da execução se inicialmente $H[2i]$ e $H[2i+1]$ eram heaps.

Considere uma execução de CorrigeHeapDescendo(H, i). Note que se $H[i]$ tem prioridade maior ou igual a seus filhos, então os testes nas linhas 2, 4 e 6 serão falsos e o

algoritmo não faz nada, o que é o esperado nesse caso, uma vez que as árvores com raiz em $H[2i]$ e $H[2i+1]$ já são heaps.

Assuma agora que $H[i]$ tem prioridade menor do que a de algum dos seus filhos. Caso $H[2i]$ seja o filho de maior prioridade, o teste na linha 2 será verdadeiro e teremos $maior = 2i$. Como $maior \neq i$, o algoritmo troca $H[i]$ com $H[maior]$ e executa CorrigeHeapDescendo(H, $maior$). Como qualquer filho de i tem altura menor do que a de i, temos $h_{maior} < h_i$. Assim, pela hipótese de indução, CorrigeHeapDescendo(H, $maior$) funciona corretamente, de onde concluímos que a árvore com raiz em $H[2i]$ é heap. Como $H[i]$ tem agora prioridade maior do que as prioridades de $H[2i]$ e $H[2i+1]$ e a árvore em $H[2i+1]$ já era heap, concluímos que a árvore enraizada em $H[i]$ agora é um heap.

A prova é análoga quando $H[2i+1]$ é o filho de maior prioridade de $H[i]$. □

Vamos analisar agora o tempo de execução de CorrigeHeapDescendo(H, i) em que $n = H.$`tamanho`. O ponto chave é perceber que, a cada chamada recursiva, CorrigeHeapDescendo acessa um elemento que está em uma altura menor na árvore, acessando apenas nós que fazem parte de um caminho que vai de i até uma folha. Assim, o algoritmo tem tempo proporcional à altura do nó i na árvore, isto é, $O(\log(n/i))$. Como a altura de qualquer nó é no máximo a altura h da árvore, e em cada passo somente tempo constante é gasto, concluímos que o tempo de execução total é $O(h)$. Como um heap pode ser visto como uma árvore binária quase completa, que tem altura $O(\log n)$ (veja Seção 11.1.4), o tempo de execução de CorrigeHeapDescendo também é, portanto, $O(\log n)$.

Vamos fazer uma análise mais detalhada do tempo de execução $T(n)$ de CorrigeHeapDescendo sobre um vetor com n elementos. Note que a cada chamada recursiva o problema diminui consideravelmente de tamanho. Se estamos na iteração correspondente a um elemento $H[i]$, a próxima chamada recursiva será na subárvore cuja raiz é um filho de $H[i]$. Mas qual o pior caso possível? No pior caso, se o problema inicial tem tamanho n, o subproblema seguinte possui tamanho no máximo $2n/3$. Isso segue do fato de possivelmente analisarmos a subárvore cuja raiz é o filho esquerdo de $H[1]$ (i.e., enraizada em $H[2]$) e o último nível da árvore estar cheio até a metade. Assim, a subárvore com raiz no índice 2 possui aproximadamente $2/3$ dos nós, enquanto que a subárvore com raiz em 3 possui aproximadamente $1/3$ dos nós. Em todos os próximos passos, os subproblemas são divididos na metade do tamanho da instância atual. Como queremos um limitante superior, podemos dizer que o tempo de execução $T(n)$ de CorrigeHeapDescendo é descrito pela recorrência

$T(n) \leq T(2n/3) + 1$. E assim, podemos utilizar o método da iteração da seguinte forma:

$$\begin{aligned}
T(n) &\leq T\left(\frac{2n}{3}\right) + 1 \\
&\leq \left(T\left(\frac{2(2n/3)}{3}\right) + 1\right) + 1 = T\left(\left(\frac{2}{3}\right)^2 n\right) + 2 \\
&\vdots \\
&\leq T\left(\left(\frac{2}{3}\right)^i n\right) + i = T\left(\frac{n}{(3/2)^i}\right) + i\,.
\end{aligned}$$

Fazendo $i = \log_{3/2} n$ e assumindo $T(1) = 1$, temos $T(n) \leq 1 + \log_{3/2} n = O(\log n)$.

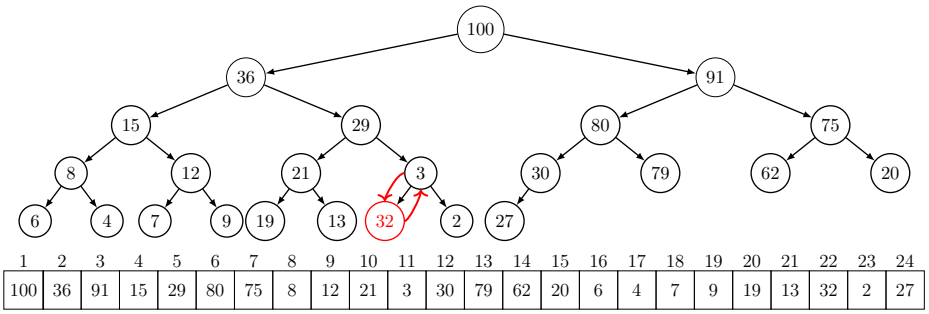
Podemos também aplicar o Teorema Mestre. Sabemos que o tempo $T(n)$ de CorrigeHeapDescendo é no máximo $T(2n/3) + 1$. Podemos então aplicar o Teorema Mestre à recorrência $T'(n) = T'(2n/3) + 1$ para obter um limitante superior para $T(n)$. Como $a = 1$, $b = 3/2$ e $f(n) = 1$, temos que $f(n) = \Theta(n^{\log_{3/2} 1})$. Assim, utilizando o caso (2) do Teorema Mestre, concluímos que $T'(n) = \Theta(\log n)$. Portanto, $T(n) = O(\log n)$.

O outro algoritmo importante para recuperação da propriedade de heap que mencionamos é o CorrigeHeapSubindo. Ele recebe um vetor H e um índice i tal que o subvetor $H[1..i-1]$ já é heap. O objetivo é fazer com que o subvetor $H[1..i]$ seja heap também. Veja que se $H[i]$ não tem prioridade menor ou igual à do seu pai, basta trocá-lo com seu pai para restaurar localmente a propriedade de heap. Potencialmente, o pai pode ter causado falha na propriedade também. Por isso, fazemos trocas sucessivas entre filhos e pais até que atingimos a raiz ou até que não tenhamos mais falha na propriedade de heap. Esse comportamento dá a sensação de que o elemento que estava na posição i inicialmente está "subindo" por um ramo da árvore. A Figura 12.3 mostra um exemplo de execução desse algoritmo.

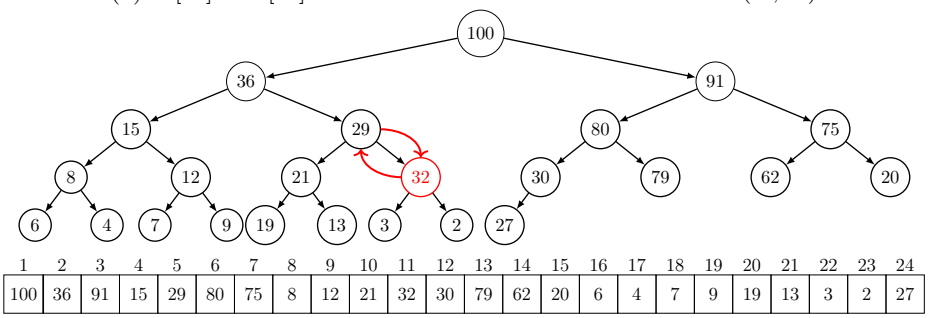
Algoritmo 12.2: CorrigeHeapSubindo(H, i)

```
   /* Se i tiver um pai e a prioridade do pai for menor, trocar          */
1  pai = ⌊i/2⌋
2  se i ≥ 2 e H[i].prioridade > H[pai].prioridade então
3      troca H[i].indice com H[pai].indice
4      troca H[i] com H[pai]
       /* A troca pode ter estragado a subárvore enraizada em pai        */
5      CorrigeHeapSubindo(H, pai)
```

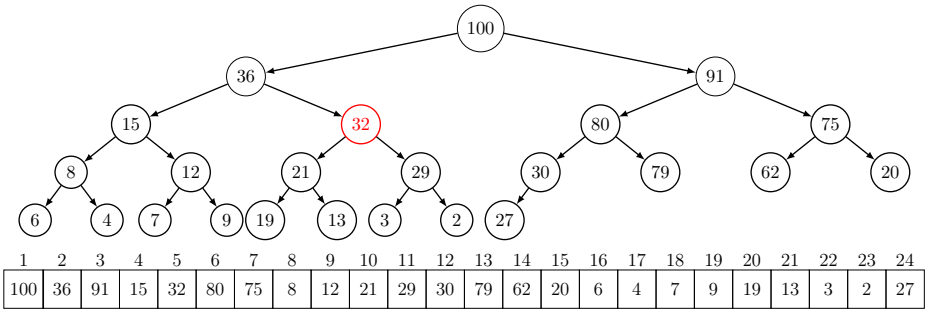
O Teorema 12.3 a seguir mostra que CorrigeHeapSubindo$H$, i de fato consegue trans-



(a) $H[22] > H[11]$: troca e chama CorrigeHeapSubindo$(H, 11)$.



(b) $H[11] > H[5]$: troca e chama CorrigeHeapSubindo$(H, 5)$.



(c) $H[2] < H[5]$: faz nada.

Figura 12.3: Execução de CorrigeHeapSubindo$(H,\ 22)$ (Algoritmo 12.2) sobre o vetor $H = (100, 36, 91, 15, 29, 80, 75, 8, 12, 21, 3, 30, 79, 62, 20, 6, 4, 7, 9, 19, 13, 32, 2, 27)$. As setas vermelhas indicam trocas de valores.

formar o subvetor $H[1..i]$ em um heap.

Teorema 12.3: ***Corretude de*** **CorrigeHeapSubindo**

O algoritmo CorrigeHeapSubindo recebe um vetor H e um índice i tal que o subvetor $H[1..i-1]$ é heap, e modifica H de modo que o subvetor $H[1..i]$ é um heap.

Demonstração. Seja ℓ_x o nível de um nó que está na posição x do heap (isto é, $\ell_x = \lfloor \log x \rfloor$). Vamos provar o resultado por indução no nível ℓ_i do nó i.

Se $\ell_i = 0$, então o nó deve ser a raiz, $H[1]$, que é um heap (de tamanho 1). O algoritmo não faz nada nesse caso, pois a raiz não tem pai, e, portanto, está correto.

Suponha que CorrigeHeapSubindo(H, k) corretamente transforma $H[1..k]$ em heap se $H[1..k-1]$ já for heap, para todo k tal que $\ell_k < \ell_i$.

Considere então uma execução de CorrigeHeapSubindo(H, i). Note que se $H[i]$ tem prioridade menor ou igual à de seu pai, então o teste na linha 2 falha e o algoritmo não faz nada. Nesse caso, como $H[1..i-1]$ já é heap, teremos que $H[1..i]$ é heap ao fim, e o algoritmo funciona corretamente.

Assuma agora que $H[i]$ tem prioridade maior do que a de seu pai e seja $p = \lfloor i/2 \rfloor$. O algoritmo então troca $H[i]$ com $H[p]$ e executa CorrigeHeapSubindo(H, p). Como o pai de i está em um nível menor do que o nível de i, temos que $\ell_p < \ell_i$. Assim, pela hipótese de indução, CorrigeHeapSubindo(H, p) funciona corretamente e concluímos que $H[1..p]$ é heap. Como $H[i]$ tem agora prioridade menor ou igual à prioridade de $H[p]$, $H[1..i-1]$ já era heap antes e os elementos de $H[p+1..i-1]$ não foram mexidos, concluímos que $H[1..i]$ agora é heap. □

Para a análise do tempo de execução de CorrigeHeapSubindo(H, i), perceba que, a cada chamada recursiva, o algoritmo acessa um elemento que está em um nível a menos na árvore, acessando apenas nós que fazem parte de um caminho que vai de i até a raiz. Assim, o algoritmo tem tempo proporcional ao nível do nó i na árvore, isto é, $O(\log i)$. Como o nível de qualquer nó é no máximo a altura h da árvore, e em cada passo somente tempo constante é gasto, concluímos que o tempo de execução total é $O(h)$, ou seja, $O(\log n)$.

12.1.1 Construção de um heap binário

Suponha que temos um vetor H com capacidade total $H.$`capacidade` em que suas primeiras $n = H.$`tamanho` posições são as únicas que contêm elementos (note que estamos falando de um vetor qualquer, que não necessariamente satisfaz a propriedade de heap). O objetivo do procedimento ConstroiHeap é transformar H em heap.

Note que os últimos $\lceil n/2 \rceil + 1$ elementos de H são folhas e, portanto, são heaps de tamanho 1. O elemento $H[\lfloor n/2 \rfloor]$, que é o primeiro elemento que tem filhos, pode não ser um heap. No entanto, como seus filhos são, podemos utilizar o algoritmo CorrigeHeapDescendo para corrigir a situação. O mesmo vale para o elemento $H[\lfloor n/2 \rfloor - 1]$ e todos os outros elementos que são pais de folhas. Com isso teremos vários heaps de altura 1, de forma que podemos aplicar o CorrigeHeapDescendo aos elementos pais dessas também. O Algoritmo 12.3 formaliza essa ideia.

Algoritmo 12.3: ConstroiHeap(H, n)

```
1 H.tamanho = n
  /* Cada elemento começa no índice em que está atualmente                    */
2 para i = 1 até H.tamanho, incrementando faça
3     H[i].indice = i
  /* Corrigimos cada subárvore, a partir do primeiro índice que não é folha */
4 para i = ⌊H.tamanho/2⌋ até 1, decrementando faça
5     CorrigeHeapDescendo(H, i)
```

A Figura 12.4 tem um exemplo de execução da rotina ConstroiHeap. Antes de estimarmos o tempo de execução do algoritmo, vamos mostrar que ele funciona corretamente no Teorema 12.4 a seguir.

Teorema 12.4

O algoritmo ConstroiHeap(H, n) transforma qualquer vetor H em um heap.

Demonstração. O algoritmo começa sua execução determinando o valor de $H.\texttt{tamanho}$ e inicializando os índices. Iremos mostrar que a seguinte frase sobre o segundo laço **para** (da linha 4) é uma invariante de laço.

Invariante: *Segundo laço para* – ConstroiHeap

$P(x) =$ "Antes da iteração em que $i = x$ começar, a árvore enraizada em $H[j]$ é um heap para todo j tal que $x + 1 \leq j \leq n = H.\texttt{tamanho}$."

Inicialmente, temos $i = \lfloor n/2 \rfloor$ no início do segundo laço **para**. Note que para qualquer j tal que $\lfloor n/2 \rfloor + 1 \leq j \leq n$, a árvore com raiz em $H[j]$ contém somente $H[j]$ como nó, pois como $j > \lfloor n/2 \rfloor$, o elemento $H[j]$ é folha e não tem filhos. Assim, de fato a árvore com raiz

em $H[j]$ é um heap e $P(\lfloor n/2 \rfloor)$ vale.

Considere agora que estamos na iteração em que $i = i'$ e suponha que $P(i')$ vale, i.e., para todo j tal que $i' + 1 \leq j \leq n$, a árvore com raiz $H[j]$ é um heap. Precisamos provar a frase é verdadeira na iteração posterior, isto é, que $P(i' - 1)$ vale (pois o valor de i é decrementado).

Se $H[i']$ tem filhos, então esses filhos são raízes de heaps devido a $P(i')$ ser válida. Assim, como a chamada a CorrigeHeapDescendo(H, i') na linha 5 funciona corretamente, ela transforma a árvore com raiz $H[i']$ em um heap. Assim, para todo j tal que $i' \leq j \leq n$, a árvore com raiz $H[j]$ é um heap, ao final dessa iteração e, consequentemente, antes do início da próxima. Assim, $P(i' - 1)$ de fato vale e a frase é realmente uma invariante de laço.

Ao fim da execução do algoritmo temos $i = 0$, de modo que, como $P(0)$ vale, a árvore com raiz em $H[1]$ é um heap. □

No que segue seja $T(n)$ o tempo de execução de ConstroiHeap em um vetor H com n elementos. Uma simples análise permite concluir que $T(n) = O(n \log n)$, pois o laço **para** é executado $n/2$ vezes e, em cada uma dessas execuções, a rotina CorrigeHeapDescendo, que leva tempo $O(\log n)$ é executada. Logo, concluímos que $T(n) = O(n \log n)$.

Uma análise mais cuidadosa, no entanto, fornece um limitante melhor que $O(n \log n)$. Primeiro vamos observar que em uma árvore com n elementos existem no máximo $\lceil n/2^{h+1} \rceil$ elementos com altura h. Verificaremos isso por indução na altura h. As folhas são os elementos com altura $h = 0$. Como temos $\lceil n/2 \rceil = \lceil n/2^{0+1} \rceil$ folhas, então a base está verificada. Seja $1 \leq h \leq \lfloor \log n \rfloor$ e suponha que existem no máximo $\lceil n/2^h \rceil$ elementos com altura $h - 1$. Note que na altura h existe no máximo metade da quantidade máxima possível de elementos de altura $h - 1$. Assim, utilizando a hipótese indutiva, na altura h temos no máximo $\lceil \lceil n/2^h \rceil / 2 \rceil$ elementos, que implica que existem no máximo $\lceil n/2^{h+1} \rceil$ elementos com altura h.

Como visto anteriormente, o tempo de execução do CorrigeHeapDescendo(H, i) é, na verdade, proporcional à altura do elemento i. Assim, para cada elemento de altura h, a chamada a CorrigeHeapDescendo correspondente executa em tempo $O(h)$, de forma que cada uma dessas chamadas é executada em tempo no máximo $Ch \leq C(h + 1)$ para alguma

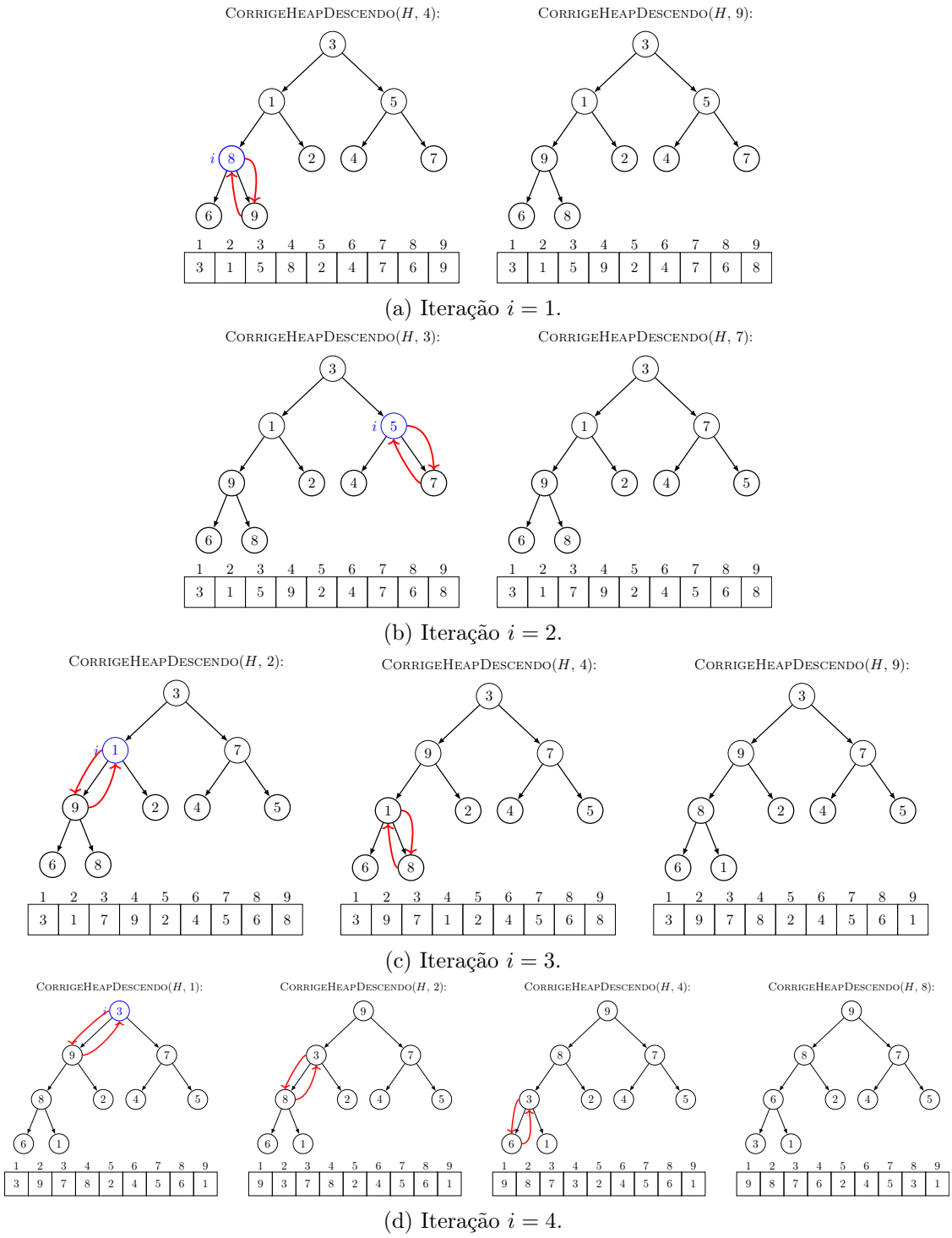


Figura 12.4: Execução de ConstroiHeap(H, 9) (Algoritmo 12.3) sobre o vetor $H = (3, 1, 5, 8, 2, 4, 7, 6, 9)$. Cada iteração do segundo laço **para** é representada em uma linha. As setas vermelhas indicam trocas de valores.

constante $C > 0$. Portanto, o tempo de execução $T(n)$ de ConstroiHeap é dado por:

$$
\begin{aligned}
T(n) &\leq \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil C(h+1) \\
&= Cn \sum_{h=0}^{\lfloor \log n \rfloor} (h+1) \left(\frac{1}{2}\right)^{h+1} \\
&= Cn \sum_{i=1}^{\lfloor \log n \rfloor + 1} i \left(\frac{1}{2}\right)^{i} \\
&= Cn \left(\frac{(1/2) - (\lfloor \log n \rfloor + 2)(1/2)^{\lfloor \log n \rfloor + 2} + (\lfloor \log n \rfloor + 1)(1/2)^{\lfloor \log n \rfloor + 3}}{(1-1/2)^2} \right) \\
&= 4Cn \left(\frac{1}{2} - \lfloor \log n \rfloor \frac{1}{4} \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor} - 2\frac{1}{4} \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor} + \lfloor \log n \rfloor \frac{1}{8} \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor} + \frac{1}{8} \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor} \right) \\
&= Cn \left(2 - \lfloor \log n \rfloor \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor} - 2 \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor} + \lfloor \log n \rfloor \frac{1}{2} \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor} + \frac{1}{2} \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor} \right) \\
&= Cn \left(2 - \frac{1}{2} \lfloor \log n \rfloor \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor} - \frac{3}{2} \left(\frac{1}{2}\right)^{\lfloor \log n \rfloor} \right) \leq 2Cn\,.
\end{aligned}
$$

Portanto, $T(n) = O(n)$.

Outra forma mais simples de observar o resultado acima é notar que

$$
Cn \sum_{i=1}^{\lfloor \log n \rfloor + 1} i \left(\frac{1}{2}\right)^{i} \leq Cn \sum_{i=1}^{\infty} i \left(\frac{1}{2}\right)^{i} = Cn \left(\frac{1/2}{(1-1/2)^2} \right) = 2Cn\,.
$$

12.1.2 Inserção em um heap binário

Para inserir um novo elemento x em uma heap H, primeiro verificamos se há espaço em H para isso, lembrando que H comporta no máximo $H.\texttt{capacidade}$ elementos. Se sim, então inserimos x na primeira posição disponível, $H[H.\texttt{tamanho}+1]$, o que potencialmente destruirá a propriedade de heap. No entanto, como $H[1..H.\texttt{tamanho}]$ já era heap, podemos simplesmente fazer uma chamada a CorrigeHeapSubindo para restaurar a propriedade em $H[1..H.\texttt{tamanho}+1]$.

O Algoritmo 12.4 formaliza essa ideia, do procedimento InsereNaHeap. Ele recebe um elemento x novo (que, portanto, tem atributos $x.\texttt{prioridade}$ e $x.\texttt{indice}$).

Como CorrigeHeapSubindo(H, $H.\texttt{tamanho}$) é executado em tempo $O(\log n)$, com $n = H.\texttt{tamanho}$, o tempo de execução de InsereNaHeap é $O(\log n)$.

Algoritmo 12.4: InsereNaHeap(H, x)

```
/* Inserimos o novo elemento na primeira posição disponível, e corrigimos, se
   necessário, fazendo troca com o pai                                      */
1 se H.tamanho < H.capacidade então
2     H.tamanho = H.tamanho +1
3     x.indice = H.tamanho
4     H[H.tamanho] = x
5     CorrigeHeapSubindo(H, H.tamanho)
```

12.1.3 Remoção em um heap binário

Sabendo que o elemento de maior prioridade em um heap H está em $H[1]$, se quisermos removê-lo, precisamos fazer isso de modo que ao fim da operação H ainda seja um heap. Dado que H já é heap, podemos tentar remover $H[1]$ sem modificar muito a estrutura trocando $H[1]$ com $H[H.\texttt{tamanho}]$, o que potencialmente destrói a propriedade de heap na posição 1, mas apenas nesta posição. Como essa é a única posição que está causando problemas e ambos $H[2]$ e $H[3]$ já eram heaps, aplicamos CorrigeHeapDescendo(H, 1) para restaurar a propriedade. O Algoritmo 12.5 formaliza essa ideia.

Algoritmo 12.5: RemoveDaHeap(H)

```
/* Removemos o primeiro elemento, copiamos o último elemento para a posição 1
   e corrigimos, se necessário, fazendo troca com o filho                   */
1 x = NULO
2 se H.tamanho ≥ 1 então
3     x = H[1]
4     H[H.tamanho].indice = 1
5     H[1] = H[H.tamanho]
6     H.tamanho = H.tamanho −1
7     CorrigeHeapDescendo(H, 1)
8 devolve x
```

Note que CorrigeHeapDescendo(H, 1) é executado em tempo $O(\log n)$ para $n = H.\texttt{tamanho}$. Logo, o tempo de execução de RemoveDaHeap(H) é $O(\log n)$ também.

12.1.4 Alteração em um heap binário

Ao alterarmos a prioridade de um elemento armazenado em uma heap H, podemos estar destruindo a propriedade de heap. No entanto, como H já é heap, fizemos isso em uma única posição específica. Veja que se o elemento ficou com prioridade maior do que tinha antes, então talvez esteja em conflito com seu pai, de forma que basta usar o algoritmo CorrigeHeapSubindo. Caso contrário, se o elemento ficou com prioridade menor do que tinha antes, então talvez esteja em conflito com algum filho, de forma que basta usar o algoritmo CorrigeHeapDescendo. O Algoritmo 12.6 formaliza essa ideia, do procedimento AlteraHeap. Ele recebe a posição i do elemento que deve ter sua prioridade alterada para um novo valor k.

Algoritmo 12.6: AlteraHeap(H, i, k)

```
/* A nova prioridade pode ser conflitante com o pai do elemento ou algum
   filho, dependendo da sua relação com a prioridade anterior            */
1 aux = H[i].prioridade
2 H[i].prioridade = k
3 se aux < k então
4 │ CorrigeHeapSubindo(H, i)
5 se aux > k então
6 │ CorrigeHeapDescendo(H, i)
```

Note que se sabemos que x é o elemento do conjunto de elementos armazenados em H que queremos alterar, então sua posição em H é facilmente recuperada, pois está armazenada em x.`indice`, uma vez que a estrutura heap não suporta busca de maneira eficiente.

A operação mais custosa do algoritmo AlteraHeap é uma chamada a CorrigeHeapSubindo ou a CorrigeHeapDescendo, de forma que o tempo de execução dele é $O(\log n)$.

Capítulo 13

Disjoint Set

Um *disjoint set* é um tipo abstrato de dados que mantém uma coleção de elementos particionados em grupos. Formalmente, dizemos que $A_1, A_2, \ldots, A_m$ é uma *partição* de uma coleção B se $A_i \cap A_j = \emptyset$ para todo $i \neq j$ e $\cup_{i=1}^{m} A_i = B$. Um *disjoint set* fornece operações de criação de um novo grupo, união de dois grupos existentes e busca pelo grupo que contém um determinado elemento.

Uma forma possível de implementar um *disjoint set* é usando uma árvore para representar cada grupo. Cada nó dessa árvore é um elemento do grupo e pode-se usar a raiz da árvore como *representante* do grupo. Assim, a criação de um novo grupo pode ser feita gerando-se uma árvore com apenas um nó, a união pode ser feita fazendo com que a raiz de uma árvore aponte para a raiz da outra, e a busca pelo grupo que contém um elemento pode ser feita percorrendo o caminho do elemento até a raiz. Perceba que as duas primeiras operações são eficientes, podendo ser realizadas em tempo constante, mas a operação de busca pode levar tempo $O(n)$ se a sequência de operações de união que construiu uma árvore criar uma estrutura linear com n nós.

É possível, no entanto, implementar um *disjoint set* garantindo tempo médio $O(\alpha(n))$ por operação, onde $\alpha(n)$ é a inversa da função Ackermann que, para todos os valores práticos de n, é no máximo 5.

13.1 Union-Find

A estrutura de dados conhecida como *union-find* mantém uma partição de um grupo de elementos e permite as seguintes operações:

- MakeSet(x): cria um grupo novo contendo somente o elemento x;
- FindSet(x): devolve qual é o grupo que contém o elemento x;
- Union(x, y): gera um grupo obtido da união dos grupos que contêm os elementos x e y.

A seguir vamos descrever uma possível implementação da estrutura. Ela considera que cada grupo tem um representante, que é um membro do grupo e que irá identificar o grupo.

Consideraremos que um elemento x possui atributos $x.$`representante`, que armazena o representante do grupo onde x está, e $x.$`tamanho`, que armazena o tamanho do grupo representado por x. Precisaremos ainda de um vetor L de listas encadeadas tal que $L[x]$ é uma lista encadeada que armazena todos os elementos que estão no grupo representado pelo elemento x. O atributo $L[x].$`cabeca` aponta para o primeiro nó da lista e o atributo $L[x].$`cauda` aponta para o último.

Note que a operação MakeSet(x) pode ser implementada em tempo constante, como mostra o Algoritmo 13.1.

Algoritmo 13.1: MakeSet(x)

```
1 x.representante = x
2 x.tamanho = 1
3 L[x].cabeca = x
4 L[x].cauda = x
```

A operação FindSet(x) também pode ser implementada em tempo constante, conforme mostra o Algoritmo 13.2.

Algoritmo 13.2: FindSet(x)

```
1 devolve x.representante
```

Quando a operação de união de dois grupos é requerida, fazemos com que o grupo de menor tamanho passe a ter o mesmo representante que o grupo de maior tamanho. Para isso, acessamos os elementos do grupo de menor tamanho e atualizamos seus atributos. Veja o Algoritmo 13.3.

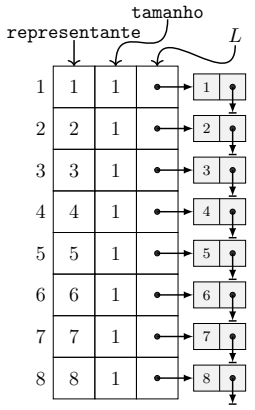
Perceba que graças à manutenção das listas ligadas em L, acessamos realmente apenas os elementos do menor dos grupo para atualizar seus atributos nos laços **para**. Todas as operações levam tempo constante para serem executadas. Assim, perceba que o tempo de execução de Union(x, y) é dominado pela quantidade de atualizações de representantes,

Algoritmo 13.3: UNION(x, y)

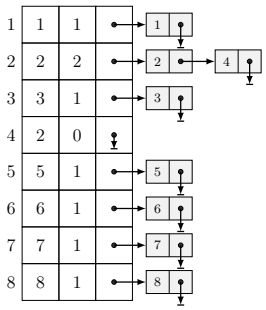
```
X = FindSet(x)
Y = FindSet(y)
/* O grupo de menor tamanho vai ser representado pelo outro grupo */
se X.tamanho < Y.tamanho então
    /* Apenas os elementos do grupo menor precisam ter seu representante
       modificado */
    para todo v em L[X] faça
        v.representante = Y
    /* Os tamanhos dos grupos envolvidos precisam ser atualizados */
    Y.tamanho = X.tamanho + Y.tamanho
    X.tamanho = 0
    /* A lista com os elementos do grupo menor deve ir para o fim da lista
       com os elementos do grupo maior */
    L[Y].cauda.proximo = L[X].cabeca
    L[Y].cauda = L[X].cauda
    L[X].cabeca = NULO
senão
    para todo v em L[Y] faça
        v.representante = X
    X.tamanho = X.tamanho + Y.tamanho
    Y.tamanho = 0
    L[X].cauda.proximo = L[Y].cabeca
    L[X].cauda = L[Y].cauda
    L[Y].cabeca = NULO
```

feitas nas linhas 5 e 13. Ademais, apenas um dos dois laços será executado, de forma que uma única chamada a UNION(x, y) leva tempo $\Theta(t)$, onde $t = \min\{x.\texttt{representante.tamanho}, y.\texttt{representante.tamanho}\}$.

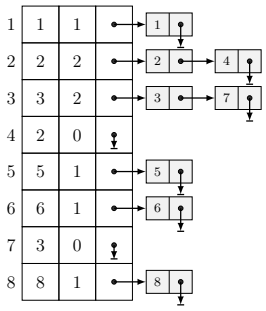
A Figura 13.1 apresenta um exemplo de operações de união feitas sobre um conjunto de dados considerando a implementação dada acima.



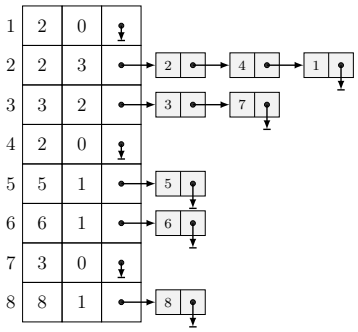
(a) MakeSet sobre todos os elementos.



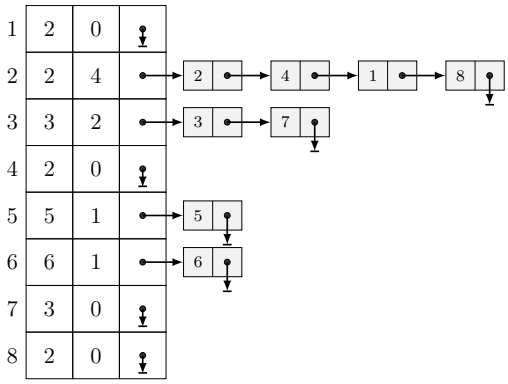
(b) Union(2, 4).



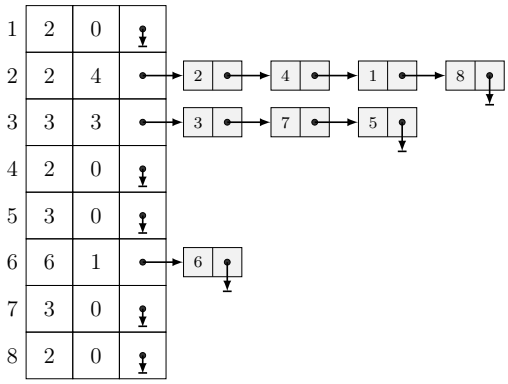
(c) Union(7, 3).



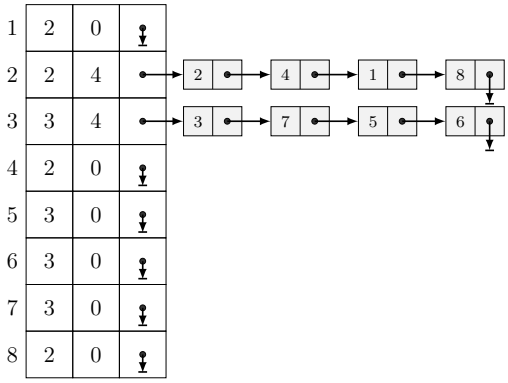
(d) Union(1, 2).



(e) Union(4, 8).



(f) Union(3, 5).



(g) Union(5, 6).

Figura 13.1: Execução de algumas chamadas a Union (Algoritmo 13.3) sobre a coleção $\{1, 2, 3, 4, 5, 6, 7, 8\}$.

Capítulo 14

Tabelas hash

Suponha que queremos projetar um sistema que armazena dados de funcionários usando como chave seus CPFs. Esse sistema vai precisar fazer inserções, remoções e buscas (todas dependentes do CPF dos funcionários). Note que podemos usar um vetor ou lista ligada para isso, porém neste caso a busca é feita em tempo linear, o que pode ser custoso na prática se o número n de funcionários armazenados for muito grande. Se usarmos um vetor ordenado, a busca pode ser melhorada para ter tempo $O(\log n)$, mas inserções e remoções passam a ser custosas. Uma outra opção é usar uma árvore binária de busca balanceada, que garante tempo $O(\log n)$ em qualquer uma das três operações. Uma terceira solução é criar um vetor grande o suficiente para que ele seja indexado pelos CPFs. Essa estratégia, chamada *endereçamento direto*, é ótima pois garante que as três operações serão executadas em tempo $\Theta(1)$.

Acontece que um CPF tem 11 dígitos, sendo 9 válidos e 2 de verificação, de forma que podemos ter 9^{10} possíveis números diferentes (algo na casa dos bilhões). Logo, endereçamento direto não é viável. Por outro lado, a empresa precisa armazenar a informação de n funcionários apenas, o que é um valor bem menor. Temos ainda uma quarta opção: *tabelas hash*.

Uma *tabela hash* é uma estrutura de dados que mapeia chaves a elementos. Ela implementa eficientemente – em tempo médio $O(1)$ – as operações de busca, inserção e remoção. Ela usa uma *função hash*, que recebe como entrada uma chave (um CPF, no exemplo acima) e devolve um número pequeno (entre 1 e m), que serve como índice da tabela que vai armazenar os elementos de fato (que tem tamanho m). Assim, se h é uma função hash, um elemento de chave k vai ser armazenado (falando de forma bem geral) na posição $h(k)$.

Note, no entanto, que sendo o universo U de chaves grande (tamanho M) e o tamanho m da tabela bem menor do que M, não importa como seja a função h: várias chaves serão mapeadas para a mesma posição – o que é chamado de *colisão*. Aliás, vale mencionar que mesmo se o contrário fosse verdade ainda teríamos colisões: por exemplo, se 2450 chaves forem mapeadas pela função hash para uma tabela de tamanho 1 milhão, mesmo com uma distribuição aleatória perfeitamente uniforme, de acordo com o Paradoxo do Aniversário, existe uma chance de aproximadamente 95% de que pelo menos duas chaves serão mapeadas para a mesma posição.

Temos então que lidar com dois problemas quando se fala em tabelas hash: (i) escolher uma função hash que minimize o número de colisões, e (ii) lidar com as colisões, que são inevitáveis.

Se bem implementada e considerando que os dados não são problemáticos, as operações de busca, inserção e remoção podem ser feitas em tempo $O(1)$ no caso médio.

Parte

IV

Algoritmos de ordenação

“`enquanto emOrdem(vetor) == false:`
`    embaralha(vetor)`”

Algoritmo Bogosort

Nesta parte

O *problema da ordenação* é um dos mais básicos e mais estudados em computação. Ele consiste em, dada uma lista de elementos, ordená-los de acordo com alguma ordem pré-estabelecida.

Algoritmos que resolvem o problema de ordenação são simples e fornecem uma base para várias ideias de projeto de algoritmos. Além disso, vários outros problemas se tornam mais simples de tratar quando os dados estão ordenados.

Existem inúmeros algoritmos de ordenação. Veremos os mais clássicos nas seções a seguir, considerando a seguinte definição do problema.

Problema 14.1: *Ordenação*

Dado um vetor $A = (a_1, a_2, \ldots, a_n)$ com n números, obter uma permutação desses números $(a'_1, a'_2, \ldots, a'_n)$ de modo que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Perceba que em um vetor ordenado, todos os elementos à esquerda de um certo elemento são menores ou iguais a ele e todos à direita são maiores ou iguais a ele. Esse argumento simples será muito usado nas discussões sobre os algoritmos que veremos.

Note que estamos considerando um vetor que contém números, mas poderíamos supor que o vetor contém registros e assumir que existe um campo de tipo comparável em cada registro (que forneça uma noção de ordem, por exemplo numérica ou lexicográfica).

Dentre características importantes de algoritmos de ordenação, podemos destacar duas. Um algoritmo é dito *in-place* se utiliza somente espaço constante além dos dados de entrada e é dito *estável* se a ordem em que chaves de mesmo valor aparecem na saída são a mesma da entrada. Discutiremos essas propriedades e a aplicabilidade e tempo de execução dos algoritmos que serão apresentados nas seções a seguir.

Capítulo 15

Ordenação por inserção

Algoritmos de ordenação por inserção consideram um elemento por vez e os inserem na posição correta de ordenação *relativa aos elementos que já foram considerados*. Neste capítulo veremos dois desses algoritmos, o *Insertion Sort* e o *Shellsort*.

15.1 Insertion sort

O funcionamento do *Insertion sort* foi mencionado brevemente na introdução desse livro. Para ordenar um conjunto de cartas, há quem prefira manter a pilha de cartas sobre a mesa e olhar uma por vez, colocando-a de forma ordenada com relação às cartas que já estão em sua mão. Sabendo que as cartas em sua mão estão ordenadas, qualquer carta nova que você receba pode ser facilmente inserida em uma posição de forma a ainda manter as cartas ordenadas, pois só há uma posição possível para ela (a menos de naipes), que seria a posição em que toda carta de valor menor fique à esquerda (ou ela seja a menor de todas) e toda carta de valor maior fique à direita (ou ela seja a maior de todas).

Formalmente, dado um vetor $A[1..n]$ com n números, a ideia do *Insertion sort* é executar n rodadas de instruções onde, a cada rodada temos um subvetor de A ordenado que contém um elemento a mais do que o subvetor da rodada anterior. Mais precisamente, ao fim na i-ésima rodada, o algoritmo garante que o subvetor $A[1..i]$ está ordenado. Sabendo que o subvetor $A[1..i-1]$ está ordenado, é fácil "encaixar" o elemento $A[i]$ na posição correta para deixar o subvetor $A[1..i]$ ordenado: compare $A[i]$ com $A[i-1]$, com $A[i-2]$, e assim por diante, até encontrar um índice j tal que $A[j] \leq A[i]$, caso em que a posição correta de $A[i]$ é $j+1$, ou até descobrir que $A[1] > A[i]$, caso em que a posição correta de $A[i]$ é 1. Encontrada

a posição $j+1$ em que $A[i]$ deveria estar, é necessário deslocar todo o subvetor $A[j+1..i-1]$ uma posição para a direita, para que $A[i]$ possa ser copiado na posição $j+1$. Com isso, todo o subvetor $A[1..i]$ ficará ordenado. Veja no Algoritmo 15.1 um pseudocódigo desse algoritmo, o InsertionSort. Perceba que o passo de deslocamento do subvetor $A[j+1..i-1]$ mencionado acima é feito indiretamente, durante a própria procura pelo índice j.

Algoritmo 15.1: InsertionSort(A, n)

```
1 para i = 2 até n, incrementando faça
2     atual = A[i]
3     j = i − 1
4     enquanto j > 0 e A[j] > atual faça
5         A[j + 1] = A[j]
6         j = j − 1
7     A[j + 1] = atual
```

É possível perceber, pelo funcionamento simples do algoritmo, que o InsertionSort é *in-place* e estável. A Figura 15.1 mostra um exemplo de execução do mesmo.

Para mostrar que o InsertionSort funciona corretamente, isto é, que para qualquer vetor A com n elementos dado na entrada, ele ordena os elementos de A de forma não-decrescente, vamos precisar de duas invariantes de laço. A primeira delas é referente ao laço **enquanto**, da linha 4.

Invariante: *Laço enquanto* – InsertionSort

$R(y)$ = "Antes da iteração em que $j = y$ começar, os elementos contidos em $A[y+2..i]$ são maiores do que *atual* e estão na mesma ordem relativa em que estavam no início do laço."

Vamos primeiro verificar que a frase acima realmente é uma invariante do laço **enquanto**. Antes do laço começar, $j = i-1$, de forma que $A[j+2..i]$ é um vetor vazio e, portanto, $R(i-1)$ vale trivialmente.

Agora suponha que estamos em uma iteração em que $j = j'$ e suponha que $R(j')$ vale, i.e., os elementos de $A[j'+2..i]$ são maiores do que *atual* e estão na mesma ordem relativa em que estavam no início do laço. Precisamos mostrar que a frase é verdadeira para a iteração posterior, isto é, que $R(j'-1)$ vale (pois o valor de j é decrementado). Como o laço começou, sabemos que $j' > 0$ e que $A[j'] > atual$. Nesse momento, fazemos $A[j'+1] = A[j']$, o que faz com que $A[j'+1..i]$ contenha elementos maiores do que *atual*. Como os elementos em

1	2	3	4	5	6	7	8
7	3	1	10	2	8	15	6
$\uparrow j$	$\uparrow i$						
7	7	1	10	2	8	15	6
$\uparrow j$	$\uparrow i$						
3	7	1	10	2	8	15	6
$\uparrow j$	$\uparrow i$						

(a) $i = 2$, $atual = 3$.

1	2	3	4	5	6	7	8
3	7	1	10	2	8	15	6
	$\uparrow j$	$\uparrow i$					
3	7	7	10	2	8	15	6
$\uparrow j$		$\uparrow i$					
3	3	7	10	2	8	15	6
$\uparrow j$		$\uparrow i$					
1	3	7	10	2	8	15	6
$\uparrow j$		$\uparrow i$					

(b) $i = 3$, $atual = 1$.

1	2	3	4	5	6	7	8
1	3	7	10	2	8	15	6
		$\uparrow j$	$\uparrow i$				

(c) $i = 4$, $atual = 10$.

1	2	3	4	5	6	7	8
1	3	7	10	2	8	15	6
			$\uparrow j$	$\uparrow i$			
1	3	7	10	10	8	15	6
		$\uparrow j$		$\uparrow i$			
1	3	7	7	10	8	15	6
	$\uparrow j$			$\uparrow i$			
1	3	3	7	10	8	15	6
$\uparrow j$				$\uparrow i$			
1	2	3	7	10	8	15	6
$\uparrow j$				$\uparrow i$			

(d) $i = 5$, $atual = 2$.

1	2	3	4	5	6	7	8
1	2	3	7	10	8	15	6
				$\uparrow j$	$\uparrow i$		
1	2	3	7	10	10	15	6
			$\uparrow j$		$\uparrow i$		
1	2	3	7	8	10	15	6
			$\uparrow j$		$\uparrow i$		

(e) $i = 6$, $atual = 8$.

1	2	3	4	5	6	7	8
1	2	3	7	8	10	15	6
					$\uparrow j$	$\uparrow i$	

(f) $i = 7$, $atual = 15$.

1	2	3	4	5	6	7	8
1	2	3	7	8	10	15	6
						$\uparrow j$	$\uparrow i$
1	2	3	7	8	10	15	15
					$\uparrow j$		$\uparrow i$
1	2	3	7	8	10	10	15
				$\uparrow j$			$\uparrow i$
1	2	3	7	8	8	10	15
			$\uparrow j$				$\uparrow i$
1	2	3	7	7	8	10	15
		$\uparrow j$					$\uparrow i$
1	2	3	6	7	8	10	15
		$\uparrow j$					$\uparrow i$

(g) $i = 8$, $atual = 6$.

Figura 15.1: Execução de InsertionSort(A, 8) (Algoritmo 15.1), com $A = (7, 3, 1, 10, 2, 8, 15, 6)$.

$A[j'+2..i]$ não foram modificados e $A[j']$ foi apenas copiado para a posição à sua direita, então os elementos $A[j'+1..i]$ estão na mesma ordem relativa em que estavam no início do laço. Portanto, $R(j'-1)$ é verdadeira. Assim, a frase de fato é uma invariante.

Perceba que ela nos permite concluir que quando o laço termina, supondo que $j = r$, vale $R(r)$, isto é, que $A[r+2..i]$ contém elementos maiores do que *atual*. Vamos precisar desta informação para provar que a seguinte frase é uma invariante do laço **para** da linha 1.

Invariante: *Laço para* – InsertionSort

$P(x)$ = "Antes da iteração em que $i = x$ começar, o subvetor $A[1..x-1]$ contém os elementos contidos originalmente em $A[1..x-1]$ em ordem não-decrescente."

Em primeiro lugar, essa frase é válida antes da primeira iteração, quando $i = 2$, uma vez que o vetor $A[1..i-1] = A[1]$ contém somente um elemento e, portanto, sempre está ordenado.

Agora precisamos mostrar que se ela vale quando $i = i'$, então ela vale quando $i = i'+1$. Assim, suponha que estamos em uma iteração em que $i = i'$ e suponha que $P(i')$ vale, isto é, que o vetor $A[1..i'-1]$ contém os elementos originais em ordem não-decrescente. Nessa iteração, fazemos $atual = A[i']$ e, como visto acima, sabemos que quando o laço **enquanto** termina temos a garantia que $A[r+2..i]$ contém elementos maiores do que *atual*, em que r é o valor da variável j ao fim do laço. Ademais, sabemos que $r = 0$ ou $A[r] \leq atual$. Se $r = 0$, então $A[2..i]$ contém elementos maiores do que *atual* e a linha 7 faz $A[1] = atual$, deixando $A[1..i']$ ordenado. Se $A[r] \leq atual$, então como sabemos que $A[1..i'-1]$ estava ordenado no início dessa iteração (porque supomos que $P(i')$ vale), temos que os elementos em $A[1..r]$ são todos menores ou iguais a *atual*. Como a linha 7 faz $A[r+1] = atual$, teremos $A[1..i']$ ordenado. Ou seja, $P(i'+1)$ vale, o que termina a demonstração de que a frase acima é de fato uma invariante.

Mas lembre-se que nosso objetivo na verdade é mostrar que ao final da execução do algoritmo, o vetor A está ordenado. A invariante do laço **para** nos diz que antes da iteração em que $i = n+1$, o subvetor $A[1..i-1] = A[1..n]$ contém os elementos contidos originalmente em $A[1..n]$ em ordem não-decrescente. Como a iteração em que $i = n+1$ não executa, o que temos é que isso vale ao fim do laço. Assim, quando o laço termina, o vetor todo está em ordem não-decrescente com todos os elementos originais, de onde concluímos que o algoritmo está correto.

Com relação ao tempo de execução, note que todas as instruções de todas as linhas do InsertionSort são executadas em tempo constante, de modo que o que vai determinar o tempo de execução do algoritmo é a quantidade de vezes que os laços **para** e **enquanto** são

executados. Os comandos internos ao laço **para** são executados $n-1$ vezes (a quantidade de valores diferentes que i assume), independente da entrada, mas a quantidade de execuções dos comandos do laço **enquanto** depende da distribuição dos elementos dentro do vetor A. No melhor caso, o teste do laço **enquanto** é executado somente uma vez para cada valor de i do laço **para**, totalizando $n-1 = \Theta(n)$ execuções. Esse caso ocorre quando A já está ordenado. No pior caso, o teste do laço **enquanto** é executado i vezes para cada valor de i do laço **para**, totalizando $2+\cdots+n = n(n+1)/2-1 = \Theta(n^2)$ execuções. Esse caso ocorre quando A está em ordem decrescente. No caso médio, qualquer uma das $n!$ permutações dos n elementos pode ser o vetor de entrada. Nesse caso, cada número tem a mesma probabilidade de estar em quaisquer das posições do vetor. Assim, em média metade dos elementos em $A[1..i-1]$ são menores do que $A[i]$, de modo que o laço **enquanto** é executado cerca de $i/2$ vezes, em média. Portanto, temos em média por volta de $n(n-1)/4$ execuções do laço **enquanto**, de onde vemos que, no caso médio, o tempo é $\Theta(n^2)$. Podemos concluir, portanto, que o tempo do InsertionSort é $\Omega(n)$ e $O(n^2)$ (ou, $\Theta(n)$ no melhor caso e $\Theta(n^2)$ no pior caso).

15.2 Shellsort

O *Shellsort* é uma variação do *Insertion sort* que faz comparação de elementos mais distantes e não apenas vizinhos.

A seguinte definição é muito importante para definirmos o funcionamento desse algoritmo. Dizemos que um vetor está *h-ordenado* se, a partir de qualquer posição, considerar todo elemento a cada h posições leva a uma sequência ordenada. Por exemplo, o vetor $A = (1,3,5,8,4,15,20,7,9,6)$ está 5-ordenado, pois as sequências de elementos $(1,15)$, $(3,20)$, $(5,7)$, $(8,9)$ e $(4,6)$ estão ordenadas. Já o vetor $A = (1,3,5,6,4,9,8,7,15,20)$ está 3-ordenado, pois $(1,6,8,20)$, $(3,4,7)$, $(5,9,15)$, $(6,8,20)$, $(4,7)$, $(9,15)$ e $(8,20)$ são sequências ordenadas de elementos que estão à distância 3 entre si. Note que um vetor 1-ordenado está totalmente ordenado.

A ideia do *Shellsort* é iterativamente h-ordenar o vetor de entrada com uma sequência de valores de h que termina em 1. Ele usa o fato de que é fácil h'-ordenar um vetor que já está h-ordenado, para $h' < h$. Esse algoritmo se comporta exatamente como o *Insertion sort* quando $h = 1$. O procedimento Shellsort é formalizado no Algoritmo 15.2. Ele recebe o vetor A com n números a serem ordenados e um vetor H com m inteiros. Ele assume que H mantém uma sequência decrescente de inteiros menores do que n tal que $H[m] = 1$.

Note que o tempo de execução do Shellsort depende drasticamente dos valores em H. Uma questão em aberto ainda hoje é determinar sua complexidade de tempo. Knuth por exemplo propôs a sequência $1, 4, 13, 40, 121, 246, \ldots$ e ela dá bons resultados na prática e faz

Algoritmo 15.2: SHELLSORT(A, n, H, m)

```
1 para t = 1 até m, incrementando faça
2     para i = H[t] + 1 até n, incrementando faça
3         atual = A[i]
4         j = i − 1
5         enquanto j ≥ H[t] e A[j − H[t] + 1] > atual faça
6             A[j + 1] = A[j − H[t] + 1]
7             j = j − H[t]
8         A[j + 1] = atual
```

$O(n^{3/2})$ comparações. Uma sequência do tipo $1, 2, 4, 8, 16, \ldots$ dá resultados muito ruins, já que elementos em posições ímpares não são comparados com elementos em posições pares até a última iteração.

Capítulo 16

Ordenação por intercalação

O algoritmo que veremos nesse capítulo faz ordenação por intercalação de elementos, usando o paradigma de *divisão e conquista* (veja Capítulo 20). Dado um vetor A com n números, esse algoritmo divide A em duas partes de tamanho $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$, ordena as duas partes recursivamente e depois intercala o conteúdo das duas partes ordenadas em uma única parte ordenada. Esse algoritmo foi inventado por Jon von Neumann em 1945.

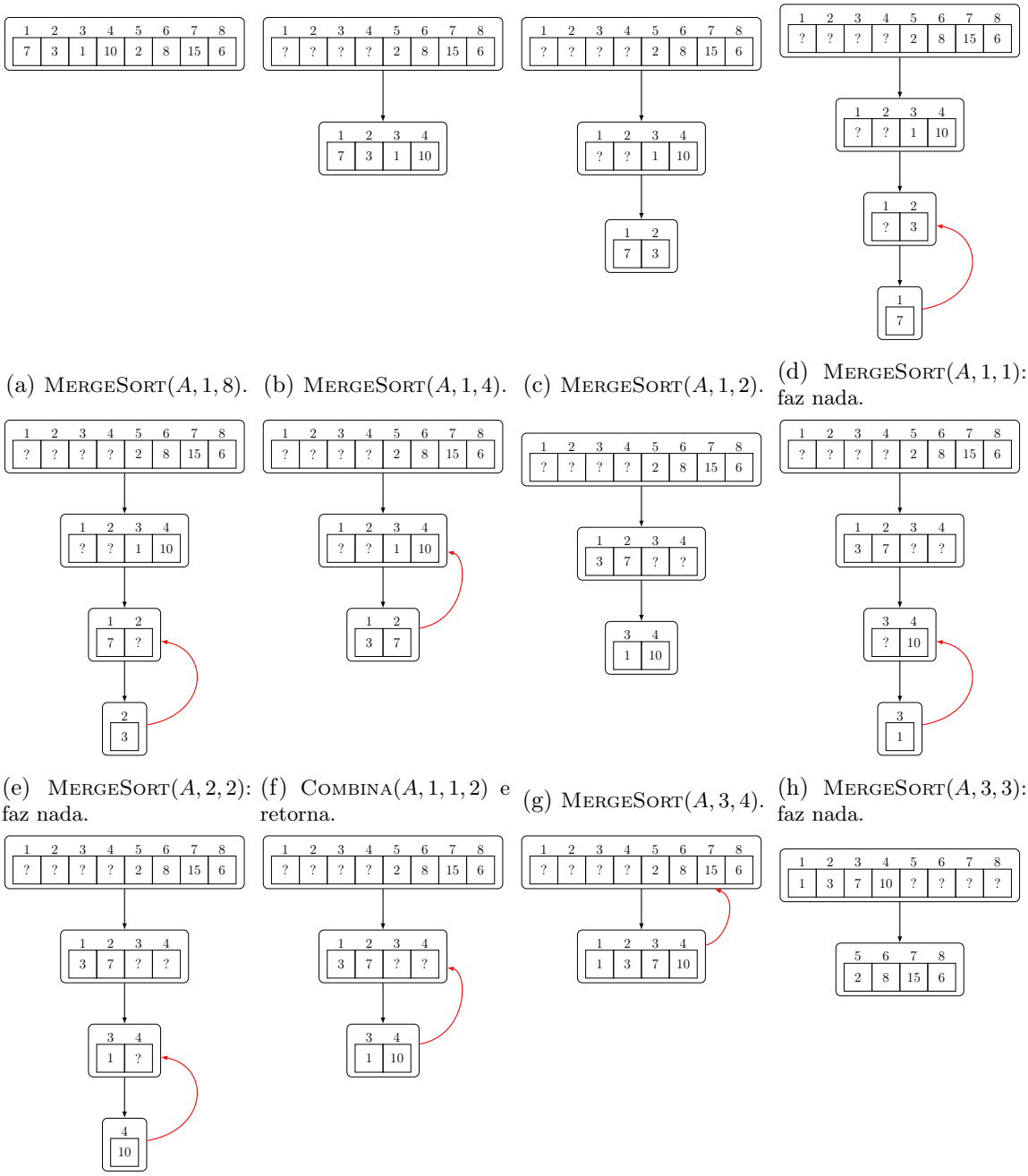
O procedimento, MergeSort, é dado no Algoritmo 16.1, onde Combina é um procedimento para combinar duas partes ordenadas em uma só parte ordenada e será visto com mais detalhes adiante. Como o procedimento recursivamente acessa partes do vetor, ele recebe A e duas posições $inicio$ e fim, e seu objetivo é ordenar o subvetor $A[inicio..fim]$. Assim, para ordenar um vetor A inteiro de n posições, basta executar MergeSort(A, 1, n).

Algoritmo 16.1: MergeSort(A, $inicio$, fim)

```
1 se inicio < fim então
2     meio = ⌊(inicio + fim)/2⌋
3     MergeSort(A, inicio, meio)
4     MergeSort(A, meio + 1, fim)
5     Combina(A, inicio, meio, fim)
```

As Figuras 16.1 e 16.2 mostram um exemplo de execução do algoritmo MergeSort. A Figura 16.3 mostra a árvore de recursão completa.

Veja que a execução do MergeSort em si é bem simples. A operação chave aqui é realizada pelo Combina. Esse algoritmo recebe o vetor A e posições $inicio$, $meio$, fim, e considera que $A[inicio..meio]$ e $A[meio + 1..fim]$ estão ordenados. O objetivo é deixar

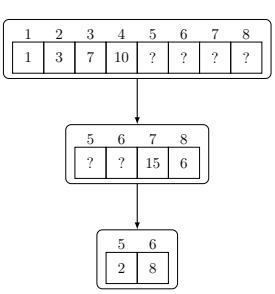


(a) MergeSort$(A, 1, 8)$. (b) MergeSort$(A, 1, 4)$. (c) MergeSort$(A, 1, 2)$. (d) MergeSort$(A, 1, 1)$: faz nada.

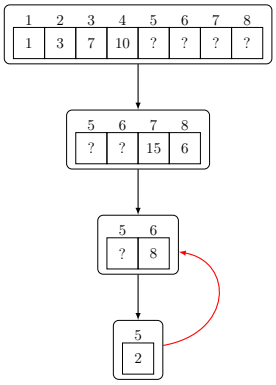
(e) MergeSort$(A, 2, 2)$: faz nada. (f) Combina$(A, 1, 1, 2)$ e retorna. (g) MergeSort$(A, 3, 4)$. (h) MergeSort$(A, 3, 3)$: faz nada.

(i) MergeSort$(A, 4, 4)$: faz nada. (j) Combina$(A, 3, 3, 4)$ e retorna. (k) Combina$(A, 1, 2, 4)$ e retorna. (l) MergeSort$(A, 5, 8)$.

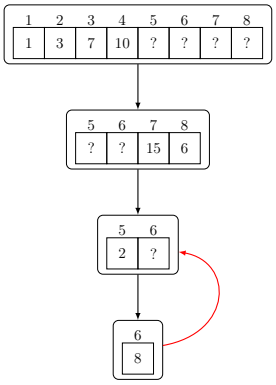
Figura 16.1: Parte 1 da execução de MergeSort$(A,\ 1,\ 8)$ (Algoritmo 16.1) para $A = (7, 3, 1, 10, 2, 8, 15, 6)$.



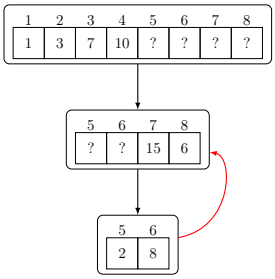
(a) MergeSort$(A, 5, 6)$.



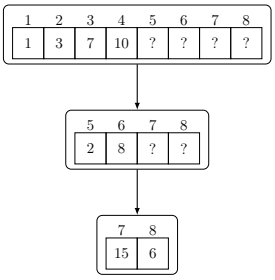
(b) MergeSort$(A, 5, 5)$: faz nada.



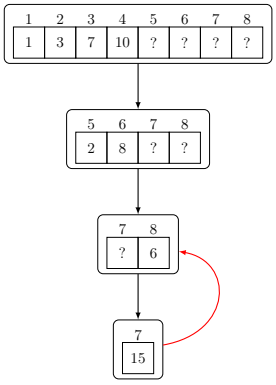
(c) MergeSort$(A, 6, 6)$: faz nada.



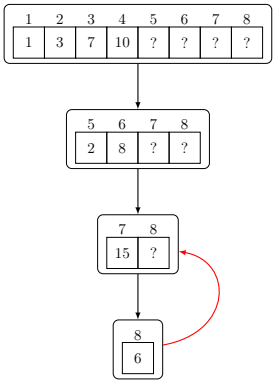
(d) Combina$(A, 5, 5, 6)$ e retorna.



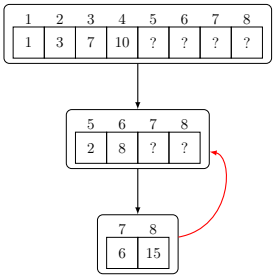
(e) MergeSort$(A, 7, 8)$.



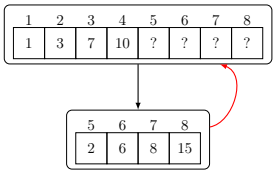
(f) MergeSort$(A, 7, 7)$: faz nada.



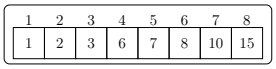
(g) MergeSort$(A, 8, 8)$: faz nada.



(h) Combina$(A, 7, 7, 8)$ e retorna.



(i) Combina$(A, 5, 6, 8)$ e retorna.



(j) Combina$(A, 1, 4, 8)$ e retorna.

Figura 16.2: Parte 2 da execução de MergeSort$(A,\ 1,\ 8)$ (Algoritmo 16.1) para $A = (7, 3, 1, 10, 2, 8, 15, 6)$.

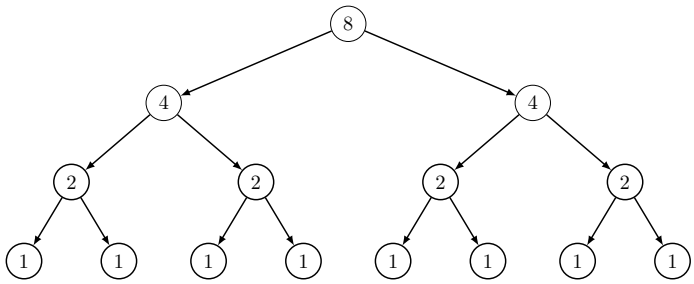


Figura 16.3: Árvore de recursão completa de MergeSort(A, 1, 8). Cada nó é rotulado com o tamanho do problema ($fim - inicio + 1$) correspondente.

$A[inicio..fim]$ ordenado. Como o conteúdo a ser deixado em $A[inicio..fim]$ já está armazenado nesse mesmo subvetor, esse procedimento faz uso de dois vetores auxiliares B e C, para manter uma cópia de $A[inicio..meio]$ e $A[meio + 1..fim]$, respectivamente.

O fato dos dois vetores B e C já estarem ordenados nos dá algumas garantias. Veja que o menor de todos os elementos que estão em B e C, que será colocado em $A[inicio]$, só pode ser $B[1]$ ou $C[1]$: o menor dentre os dois. Se $B[1] < C[1]$, então o elemento a ser colocado em $A[inicio + 1]$ só pode ser $B[2]$ ou $C[1]$: o menor dentre esses dois. Mas se $C[1] < B[1]$, então o elemento que vai para $A[inicio + 1]$ só pode ser $B[1]$ ou $C[2]$: o menor dentre esses. A garantia mais importante é que uma vez que um elemento $B[i]$ ou $C[j]$ é copiado para sua posição final em A, esse elemento não precisa mais ser considerado. É possível, portanto, realizar todo esse procedimento fazendo uma única passagem por cada elemento de B e C.

Pela discussão acima, vemos que precisamos manter um índice i para acessar elementos a serem copiadas de B, um índice j para acessar elementos em C e um índice k para acessar o vetor A. A cada iteração, precisamos colocar um elemento em $A[k]$, que será o menor dentre $B[i]$ e $C[j]$. Se $B[i]$ (resp. $C[j]$) for copiado, incrementamos i (resp. j) para que esse elemento não seja considerado novamente. Veja o procedimento Combina formalizado no Algoritmo 16.2. Como ele utiliza vetores auxiliares, o MergeSort não é um algoritmo in-place. Na Figura 16.4 temos uma simulação da execução do Combina.

O Teorema 16.1 a seguir mostra que o algoritmo Combina de fato funciona corretamente.

Teorema 16.1

Seja $A[inicio..fim]$ um vetor e $meio$ uma posição tal que $inicio \leq meio < fim$. Se $A[inicio..meio]$ e $A[meio + 1..fim]$ estão ordenados, então o algoritmo Combina(A, $inicio$, $meio$, fim) corretamente ordena $A[inicio..fim]$.

Demonstração. Vamos analisar primeiro o primeiro laço **enquanto**, da linha 11. Observe

Algoritmo 16.2: COMBINA(A, $inicio$, $meio$, fim)

```
1  n1 = meio − inicio + 1 /* Qtd. de elementos em A[inicio..meio] */
2  n2 = fim − meio /* Qtd. de elementos em A[meio + 1..fim] */
3  Crie vetores auxiliares B[1..n1] e C[1..n2]
4  para i = 1 até n1, incrementando faça
       /* Copiando o conteúdo de A[inicio..meio] para B */
5      B[i] = A[inicio + i − 1]
6  para j = 1 até n2, incrementando faça
       /* Copiando o conteúdo A[meio + 1..fim] para C */
7      C[j] = A[meio + j]
8  i = 1 /* i manterá o índice em B do menor elemento ainda não copiado */
9  j = 1 /* j manterá o índice em C do menor elemento ainda não copiado */
10 k = inicio /* k manterá o índice em A da posição para onde um elemento será
       copiado */
11 enquanto i ≤ n1 e j ≤ n2 faça
       /* Copia o menor dentre B[i] e C[j] para A[k] */
12     se B[i] ≤ C[j] então
13         A[k] = B[i]
14         i = i + 1
15     senão
16         A[k] = C[j]
17         j = j + 1
18     k = k + 1
19 enquanto i ≤ n1 faça
       /* Termina de copiar elementos de B, se houver */
20     A[k] = B[i]
21     i = i + 1
22     k = k + 1
23 enquanto j ≤ n2 faça
       /* Termina de copiar elementos de C, se houver */
24     A[k] = C[j]
25     j = j + 1
26     k = k + 1
```

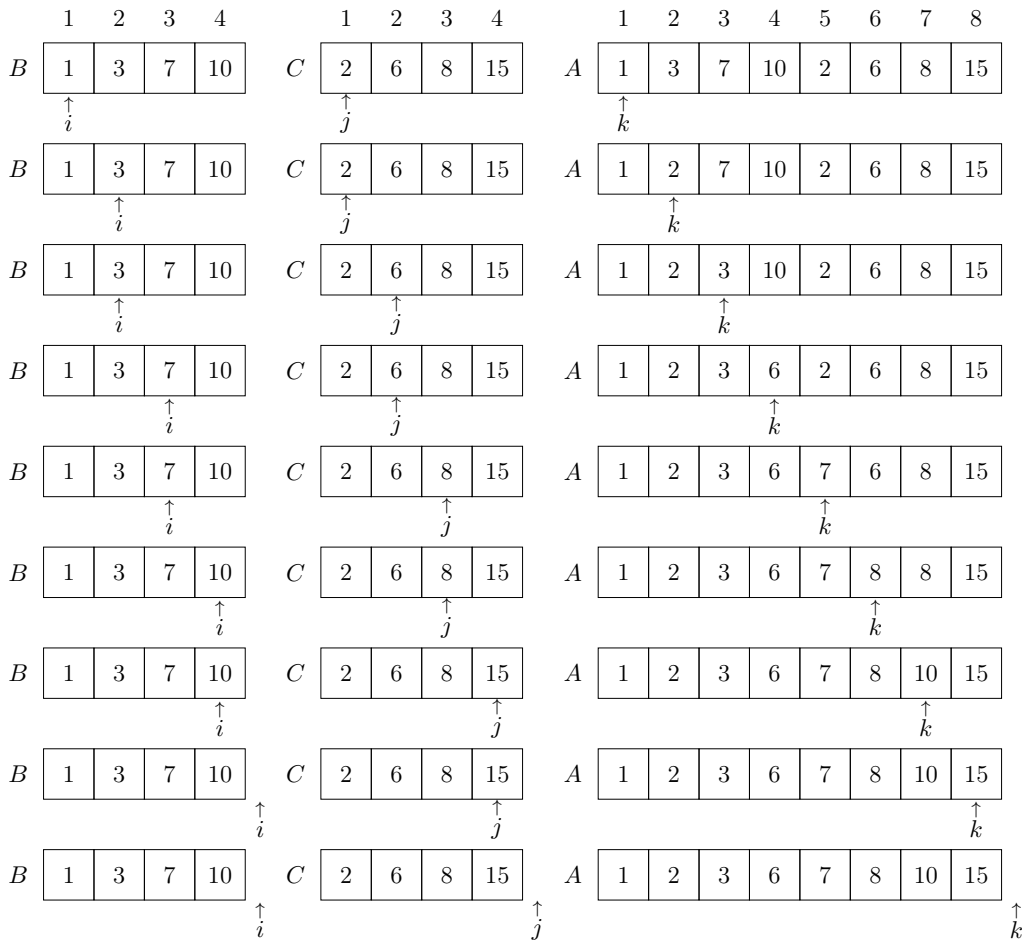


Figura 16.4: Execução de Combina(A, 1, 4, 8) (Algoritmo 16.2) sobre o vetor $A = (1, 3, 7, 10, 2, 6, 8, 15)$.

que, pelo funcionamento do algoritmo, uma vez que um elemento de B ou C é copiado para A, ele não mudará de lugar depois. Precisamos então garantir que para o elemento $A[k]$ valerá que ele é maior ou igual aos elementos em $A[inicio..k-1]$ e que é menor ou igual aos elementos em $A[k+1..fim]$. Uma das formas de fazer isso é mostrando que a seguinte frase é uma invariante de laço.

Invariante: *Primeiro laço enquanto* – Combina

$P(x, y, z)$ = "Antes da iteração em que $k = x$, $i = y$ e $j = z$ começar, temos que (i) o vetor $A[inicio..x-1]$ está ordenado, e (ii) os elementos de $B[y..n_1]$ e $C[z..n_2]$ são maiores ou iguais aos elementos de $A[inicio..x-1]$."

Antes do laço começar, $i = 1$, $j = 1$ e $k = inicio$. Veja que $P(inicio, 1, 1)$ = "Antes da iteração em que $k = inicio$, $i = 1$ e $j = 1$ começar, temos que (i) o vetor $A[inicio..inicio-1]$ está ordenado, e (ii) os elementos de $B[1..n_1]$ e $C[1..n_2]$ são maiores ou iguais aos elementos de $A[inicio..inicio-1]$" de fato é verdade, pois $A[inicio..inicio-1]$ é vazio, e portanto está ordenado, e os elementos em $B[1..n_1]$ e $C[1..n_2]$ são de fato maiores ou iguais aos elementos de um conjunto vazio.

Considere agora que a iteração atual tem $k = k'$, $i = i'$ e $j = j'$ no início dela. Suponha que $P(k', i', j')$ vale. Precisamos mostrar que $P(k'', i'', j'')$ vale, onde k'', i'' e j'' são os os valores de k, i e j, respectivamente,no início da próxima iteração.

Nessa iteração, duas coisas podem acontecer: $B[i'] \leq C[j']$ ou $B[i'] > C[j']$.

Considere primeiro que $B[i'] \leq C[j']$. Nesse caso, copiamos $B[i']$ para $A[k']$ e incrementamos apenas os valores das variáveis i e k. Assim, temos $k'' = k' + 1$, $i'' = i' + 1$ e $j'' = j'$. Como $P(k', i', j')$ vale no início da iteração, então $B[i']$ é maior do que os elementos que estão em $A[inicio..k'-1]$, de forma que $A[inicio..k'] = A[inicio..k''-1]$ fica em ordem ao fim da iteração. Como $B[i'] \leq C[j']$ e B e C estão em ordem, então os elementos em $B[i'+1..n_1] = B[i''..n_1]$ e $C[j'..n_2] = C[j''..n_2]$ são maiores do que os elementos em $A[inicio..k''-1]$. Então, nesse caso, temos que $P(k'', i'', j'')$ vale.

Se $B[i'] > C[j']$, então com uma análise similar podemos mostrar que $P(k'', i'', j'')$ vale. Nesse caso, note que $k'' = k' + 1$, $i'' = i'$ e $j'' = j' + 1$.

Vamos então utilizar essa invariante para mostrar que o algoritmo Combina está correto. Quando o laço **enquanto** da linha 11 acabar, suponha que k_f, i_f e j_f são os valores de k, i e j, respectivamente. A invariante nos diz que

o vetor $A[inicio..k_f-1]$ está ordenado e os elementos de $B[i_f..n_1]$ e $C[j_f..n_2]$ são maiores ou iguais aos elementos de $A[inicio..k_f-1]$.

Mas note que o laço pode ter acabado porque $i_f = n_1 + 1$ ou então porque $j_f = n_2 + 1$, de forma que um dentre $B[i_f..n_1]$ ou $C[j_f..n_2]$ é vazio.

Se $i_f = n_1 + 1$, então $B[i_f..n_1]$ é vazio, ou seja, já B foi totalmente copiado para A e $C[j_f..n_2]$ ainda não foi. O teste do segundo laço **enquanto**, da linha 19, falha. O terceiro laço **enquanto**, da linha 23, será executado e copiará $C[j_f..n_2]$ para A a partir da posição k_f. Como $C[j_f..n_2]$ só contém elementos maiores do que $A[inicio..k_f - 1]$, então $A[inicio..fim]$ ficará totalmente em ordem (pois $fim - inicio + 1 = n_1 + n_2$).

Uma análise similar pode ser feita para o caso do laço **enquanto** da linha 11 ter terminado porque $j_f = n_2 + 1$.

Combina(A, $inicio$, $meio$, fim), portanto, termina com $A[inicio..fim]$ ordenado. □

Com relação ao tempo de execução, considere uma execução de Combina ao receber um vetor A e parâmetros $inicio$, $meio$ e fim como entrada. Note que além das linhas que são executadas em tempo constante, o laço **para** na linha 4 é executado $n_1 = meio - inicio + 1$ vezes, o laço **para** na linha 6 é executado $n_2 = fim - meio$ vezes, e os laços **enquanto** das linhas 11, 19 e 23 são executados ao todo $n_1 + n_2 = fim - inicio + 1$ vezes (podemos notar isso pela quantidade de valores diferentes que k assume). Se $R(n)$ é o tempo de execução de Combina(A, $inicio$, $meio$, fim) onde $n = fim - inicio + 1$, então claramente temos $R(n) = \Theta(n)$.

Agora podemos analisar o MergeSort. O Teorema 16.2 a seguir mostra que ele está correto, isto é, para qualquer vetor A e posições $inicio \leq fim$, o algoritmo corretamente ordena o vetor $A[inicio..fim]$. Isso diretamente implica que a chamada MergeSort(A, 1, n) ordena por completo um vetor $A[1..n]$.

Teorema 16.2

Seja A um vetor qualquer e $inicio$ e fim duas posições. O algoritmo MergeSort(A, $inicio$, fim) corretamente ordena $A[inicio..fim]$.

Demonstração. Vamos provar que o algoritmo está correto por indução no tamanho $n = fim - inicio + 1$ do vetor.

Quando $n = 0$, temos que $n = fim - inicio + 1$ implica em $inicio = fim + 1$, ou $inicio > fim$, e quando $n = 1$, isso implica em $inicio = fim$. Veja que quando $inicio \geq fim$ o MergeSort não faz nada. De fato, se $inicio > fim$, $A[inicio..fim]$ é vazio e, por vacuidade, está ordenado. Se $inicio = fim$, $A[inicio..fim]$ contém um elemento e, portanto, também está ordenado. Então MergeSort funciona no caso base.

Considere então que $n \geq 2$, o que faz $n = fim - inicio + 1$ implicar em $inicio < fim$.

Suponha que MergeSort corretamente ordena qualquer vetor com k elementos, onde $0 \leq k < n$. Precisamos provar que ele ordena o vetor com n elementos.

A primeira coisa que MergeSort faz nesse caso é calcular a posição $meio = \lfloor (inicio + fim)/2 \rfloor$, do elemento central de $A[inicio..fim]$. Em seguida, faz uma chamada a MergeSort(A, $inicio$, $meio$), isto é, uma chamada passando um vetor com $meio - inicio + 1$ elementos. Veja que

$$\begin{aligned} meio - inicio + 1 &= \left\lfloor \frac{inicio + fim}{2} \right\rfloor - inicio + 1 \\ &\leq \frac{inicio + fim}{2} - inicio + 1 \\ &= \frac{fim - inicio + 2}{2} = \frac{n+1}{2}\,. \end{aligned}$$

E $(n+1)/2 < n$ sempre que $n > 1$. Assim, essa chamada de fato reduz o tamanho do vetor inicial e, por hipótese, corretamente ordena $A[inicio..meio]$.

Em seguida, outra chamada recursiva é feita, a MergeSort(A, $meio + 1$, fim), que é uma chamada passando um vetor com $fim - meio$ elementos. Veja que

$$\begin{aligned} fim - meio &= fim - \left\lfloor \frac{inicio + fim}{2} \right\rfloor \\ &\leq fim - \left(\frac{inicio + fim}{2} - 1 \right) \\ &= \frac{fim - inicio + 2}{2} = \frac{n+1}{2}\,. \end{aligned}$$

Novamente, $(n+1)/2 < n$ sempre que $n > 1$. Essa chamada, também por hipótese, corretamente ordena $A[meio + 1..fim]$.

O próximo passo do algoritmo é chamar Combina(A, $inicio$, $meio$, fim). Como vimos no Teorema 16.1, Combina funciona sempre que $A[inicio..meio]$ e $A[meio+1..fim]$ já estão ordenados, o que é o caso, como visto acima. Logo, $A[inicio..fim]$ termina totalmente ordenado. □

Vamos agora analisar o tempo de execução do MergeSort quando ele é utilizado para ordenar um vetor com n elementos. Como o vetor da entrada é dividido ao meio no algoritmo, seu tempo de execução $T(n)$ é dado por $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$, onde $\Theta(n)$ é o tempo $R(n)$ do Combina, visto acima. Como estamos preocupados em fazer uma análise assintótica, podemos substituir $\Theta(n)$ por n apenas, o que não fará diferença no resultado obtido. Também podemos desconsiderar pisos e tetos, como visto na Seção 8.1.3, de forma

que o tempo do MergeSort pode ser descrito por

$$T(n) = 2T(n/2) + n\,,$$

para $n > 1$, e $T(n) = 1$ para $n = 1$. Assim, como visto no Capítulo 8, o tempo de execução de MergeSort é $T(n) = \Theta(n \log n)$.

Capítulo 17

Ordenação por seleção

Neste capítulo vamos introduzir dois algoritmos para o problema de ordenação que utilizam a ideia de ordenação por seleção. Em ambos, consideramos uma posição i do vetor por vez, selecionamos o i-ésimo menor elemento do vetor e o colocamos em i, posição final desse elemento no vetor ordenado.

17.1 Selection sort

O *Selection sort* é um algoritmo que sempre mantém o vetor de entrada $A[1..n]$ dividido em dois subvetores contíguos separados por uma posição i, um à direita e outro à esquerda, estando um deles ordenado. Aqui consideraremos uma implementação onde o subvetor da esquerda, $A[1..i]$, contém os menores elementos da entrada ainda não ordenados e o subvetor da direita, $A[i+1..n]$, contém os maiores elementos da entrada já ordenados. A cada iteração, o maior elemento x do subvetor $A[1..i]$ é encontrado e colocado na posição i, de forma que o subvetor da direita é aumentado em uma unidade[1].

O Algoritmo 17.1 descreve o procedimento SelectionSort e possui uma estrutura muito simples, contendo dois laços **para** aninhados. O primeiro laço, indexado por i, é executado $n-1$ vezes e, em cada iteração, aumenta o subvetor da direita, que já estava ordenado, em uma unidade. Ademais, esse subvetor da direita sempre contém os maiores elementos de A. Para aumentar esse subvetor, o maior elemento que não está nele é adicionado ao início dele. A Figura 17.1 mostra um exemplo de execução do algoritmo SelectionSort.

[1] Não é difícil adaptar toda a discussão que faremos considerando que o subvetor $A[1..i-1]$ da esquerda contém os menores elementos ordenados e o da direita contém os elementos não ordenados. Com isso, a cada iteração, o menor elemento do subvetor $A[i..n]$ deve ser encontrado e colocado na posição i.

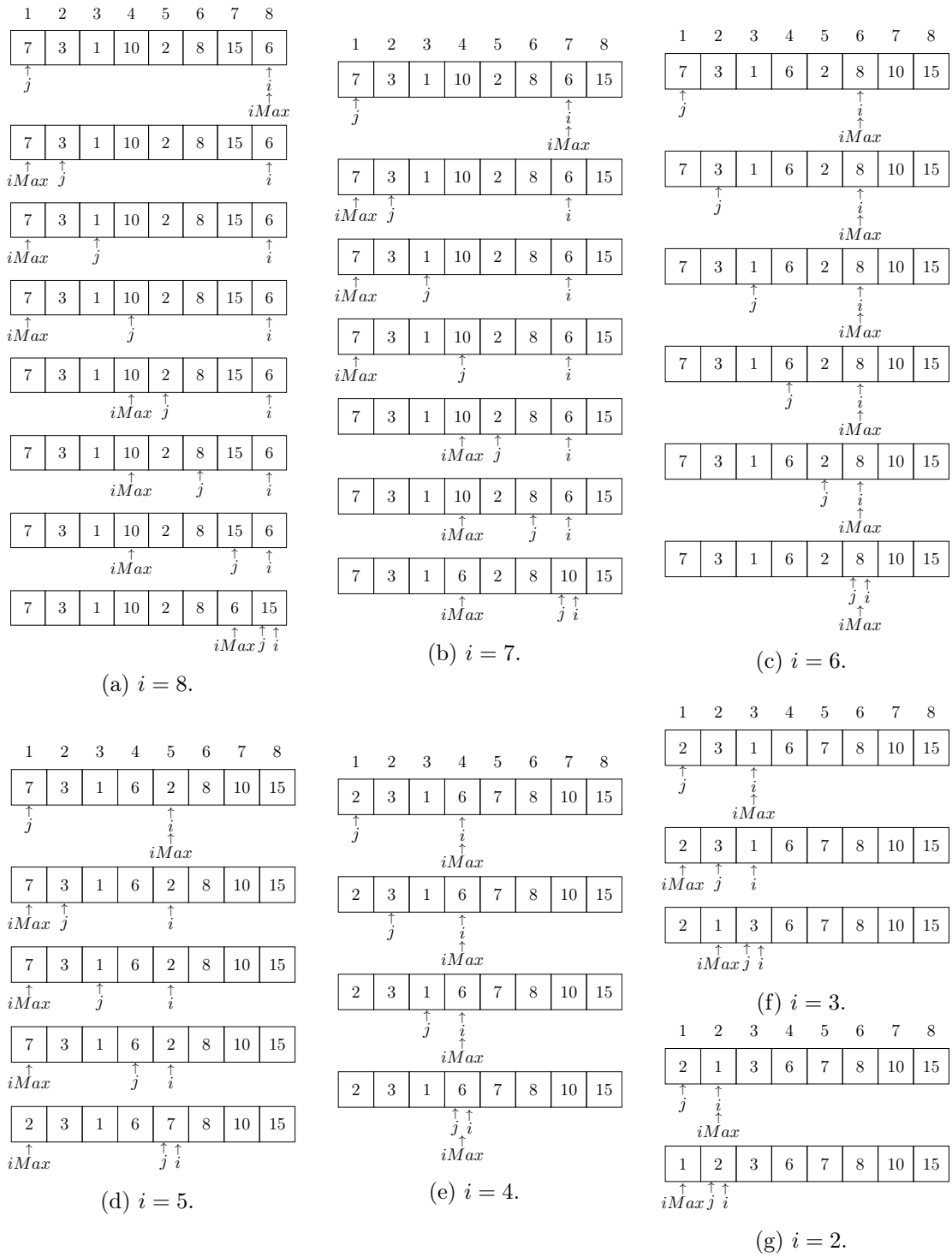


Figura 17.1: Execução de SelectionSort(A, 8) (Algoritmo 17.1), com $A = (7, 3, 1, 10, 2, 8, 15, 6)$.

Algoritmo 17.1: SELECTIONSORT(A, n)

```
1 para i = n até 2, decrementando faça
2     iMax = i
3     para j = 1 até i − 1, incrementando faça
4         se A[j] > A[iMax] então
5             iMax = j
6     troca A[iMax] com A[i]
7 devolve A
```

O Teorema 17.1 a seguir prova que o algoritmo funciona corretamente, isto é, que ordena qualquer vetor A e n dados na entrada, ele corretamente deixa os n elementos de A em ordem não-decrescente.

Teorema 17.1

O algoritmo SELECTIONSORT ordena qualquer vetor A com n elementos de modo não-decrescente.

Demonstração. Vamos inicialmente provar que a seguinte frase é uma invariante do laço **para** mais interno, da linha 3.

Invariante: *Segundo laço para* – SELECTIONSORT

$R(y)$ = "Antes da iteração em que $j = y$ começar, $A[indiceMax]$ é maior ou igual a qualquer elemento em $A[1..y-1]$."

Antes do laço começar, temos $j = 1$ e $indiceMax = i$. De fato, podemos dizer que $A[i]$ é maior ou igual a qualquer elemento em $A[1..0]$, que é vazio. Assim, $R(1)$ é válido.

Agora suponha que estamos em uma iteração em que $j = j'$. Suponha que $R(j')$ vale, isto é, que no início da iteração $A[indiceMax]$ é maior ou igual a qualquer elemento em $A[1..j'-1]$. Durante a iteração, duas coisas podem ocorrer. Se $A[j'] > A[indiceMax]$, então atualizaremos a variável $indiceMax$ para ter valor igual a j'. Note que como $A[indiceMax]$ era maior ou igual a todos os elementos em $A[1..j'-1]$ no início da iteração, então ao fim teremos que $A[indiceMax] = A[j']$ é maior ou igual aos elementos em $A[1..j']$. Por outro lado, se $A[j'] \leq A[indiceMax]$, então o valor da variável $indiceMax$ não muda mas agora temos garantia de que $A[indiceMax]$ é maior ou igual aos elementos em $A[1..j']$. Em qualquer

caso, mostramos que $R(j'+1)$ é válida e, portanto, a frase acima é uma invariante.

Ela nos permite concluir que quando o laço termina, momento em que $j = i$, vale $R(i)$, isto é, que $A[indiceMax]$ é maior ou igual a qualquer elemento de $A[1..i-1]$. Usaremos este fato para mostrar que a frase a seguir é uma invariante para o laço externo, da linha 1.

Invariante: *Primeiro laço para* – SelectionSort

$P(x) =$ "Antes da iteração em que $i = x$ começar, o subvetor $A[x+1..n]$ está ordenado de modo não-decrescente e contém os $n - x$ maiores elementos de A."

Antes da primeira iteração, quando $i = n$, temos que $P(n)$ é trivialmente verdadeira pois $A[n+1..n]$ é um vetor sem elementos.

Considere agora uma iteração qualquer em que $i = i'$ e suponha que $P(i')$ vale, isto é, que o subvetor $A[i'+1..n]$ está ordenado de modo não-decrescente e contém os $n-i'$ maiores elementos de A. Precisamos mostrar que a frase será verdadeira para a próxima iteração, quando $i = i' - 1$, ou seja, que $P(i'-1)$ é verdadeiro.

Note que na iteração atual, pela invariante anterior, sabemos que quando o segundo laço **para** (da linha 3) termina, o valor em $indiceMax$ é tal que $A[indiceMaior]$ é maior ou igual a qualquer elemento em $A[1..i'-1]$. Na linha 6, trocamos tal valor com o elemento $A[i']$. Como $P(i'-1)$ vale, sabemos que todos os elementos de $A[i'+1..n]$ são maiores do que $A[i']$, de forma que agora temos que $A[i'..n]$ está ordenado e contém os $n-i'+1$ maiores elementos de A, valendo assim $P(i'-1)$.

Agora que temos essa invariante, sabemos que ela vale para quando $i = 1$. Ela nos diz que o vetor $A[2..n]$ está ordenado com os maiores elementos de A. Logo, concluímos que o vetor $A[1..n]$ está ordenado ao fim da execução do algoritmo. □

Agora que sabemos que o algoritmo está correto, vamos analisar seu tempo de execução. Note que todas as linhas de SelectionSort(A, n) são executadas em tempo constante. As linhas 1, 2 e 6 executam em tempo $\Theta(n)$ cada. Já as linhas 3 e 4 executam $\Theta(i)$ vezes cada, para cada i entre 2 e n, totalizando tempo $\sum_{i=2}^{n} \Theta(i) = \Theta(n^2)$. A linha 5 executa $O(i)$ vezes, para cada i entre 2 e n, levando, portanto, tempo $O(n^2)$. Assim, o tempo total de execução de SelectionSort(A, n) é $\Theta(n^2)$.

17.2 Heapsort

O *Heapsort*, assim como o *Selection sort*, é um algoritmo que sempre mantém o vetor de entrada $A[1..n]$ dividido em dois subvetores contíguos separados por uma posição i, onde o

subvetor da esquerda, $A[1..i]$, contém os menores elementos da entrada ainda não ordenados e o subvetor da direita, $A[i+1..n]$, contém os maiores elementos da entrada já ordenados. A diferença está no fato do *Heapsort* utilizar a estrutura de dados *heap binário* (ou, simplesmente, heap) para repetidamente encontrar o maior elemento de $A[1..i]$ e colocá-lo na posição i (o *Selection sort* faz essa busca percorrendo todo o vetor $A[1..i]$). Com isso, seu tempo de execução de pior caso é $\Theta(n \log n)$, como o *Merge sort*. Dessa forma, o *Heapsort* pode ser visto como uma versão mais eficiente do *Selection sort*. O *Heapsort* é um algoritmo in-place, apesar de não ser estável.

Com relação à estrutura heap, o *Heapsort* faz uso especificamente apenas dos procedimentos CorrigeHeapDescendo e ConstroiHeap, definidos na Seção 12.1. Consideraremos aqui que os valores armazenados no vetor A de entrada diretamente indicam as suas prioridades. Por comodidade, reproduzimos esses dois procedimentos nos Algoritmos 17.2 e 17.3, adaptados com essa consideração das prioridades.

Algoritmo 17.2: CorrigeHeapDescendo(H, i)

```
1 maior = i
2 se 2i ≤ H.tamanho e H[2i] > H[maior] então
3     maior = 2i
4 se 2i + 1 ≤ H.tamanho e H[2i + 1] > H[maior] então
5     maior = 2i + 1
6 se maior ≠ i então
7     troca H[i] com H[maior]
8     CorrigeHeapDescendo(H, maior)
```

Algoritmo 17.3: ConstroiHeap(H)

```
1 para i = ⌊H.tamanho/2⌋ até 1, decrementando faça
2     CorrigeHeapDescendo(H, i)
```

Note que se um vetor A com n elementos é um heap, então $A[1]$ contém o maior elemento de $A[1..n]$. O primeiro passo do Heapsort é trocar $A[1]$ com $A[n]$, colocando assim o maior elemento em sua posição final após a ordenação. Como A era heap, potencialmente perdemos a propriedade em $A[1..n-1]$ ao fazer essa troca, porém devido a uma única posição. Assim, basta restaurar a propriedade de heap em $A[1..n-1]$ a partir da posição 1 para que $A[1..n-1]$ volte a ser heap. Agora, de forma equivalente, $A[1]$ contém o maior elemento de $A[1..n-1]$ e, portanto, podemos repetir o mesmo procedimento acima. Descrevemos formalmente o

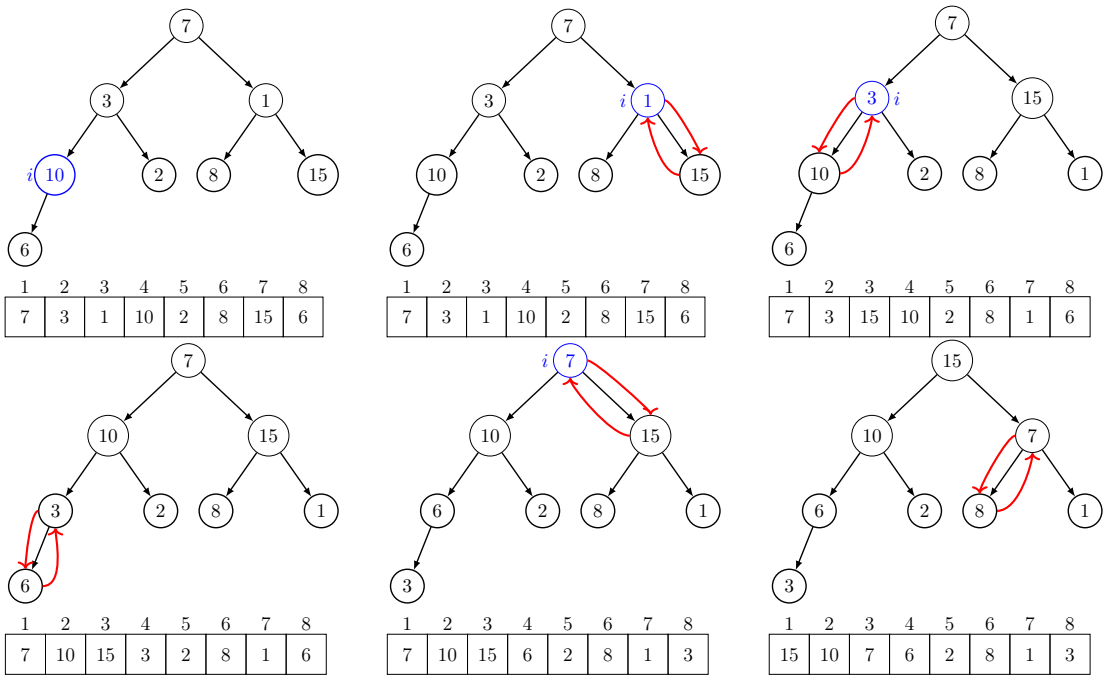


Figura 17.2: Parte 1 da execução de HEAPSORT(A, 8) (Algoritmo 17.4), com $A = (7, 3, 1, 10, 2, 8, 15, 6)$: chamada a CONSTROIHEAP(A).

procedimento HEAPSORT no Algoritmo 17.4. Lembre-se que $A.\texttt{tamanho}$ é a quantidade de elementos armazenados em A, isto é, n. As Figuras 17.2, 17.3 e 17.4 mostram um exemplo de execução do algoritmo HEAPSORT.

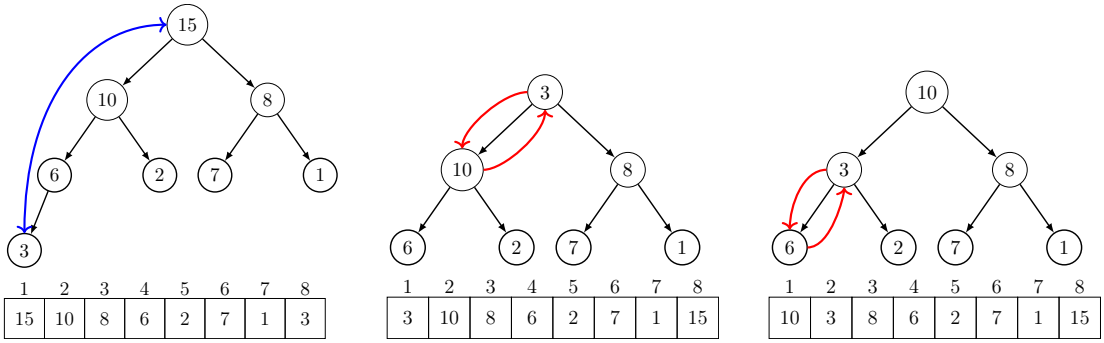
Algoritmo 17.4: HEAPSORT(A, n)

```
1 ConstroiHeap(A)
2 para i = n até 2, decrementando faça
3     troca A[1] com A[i]
4     A.tamanho = A.tamanho − 1
5     CorrigeHeapDescendo(A, 1)
```

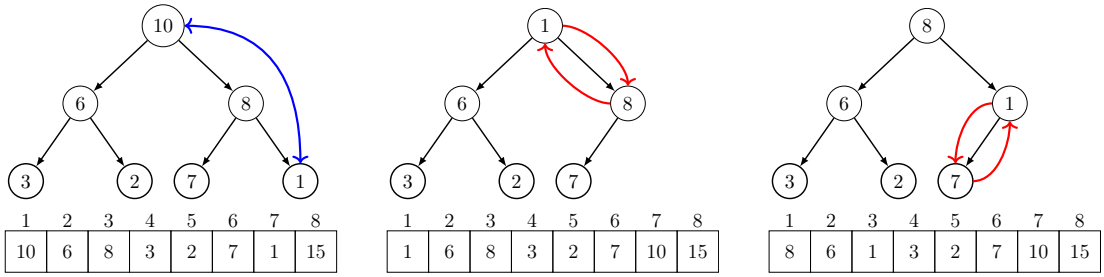
Uma vez provada a corretude de CONSTROIHEAP e CORRIGEHEAPDESCENDO, provar a corretude do HEAPSORT é mais fácil. Isso é feito no Teorema 17.2 a seguir.

Teorema 17.2

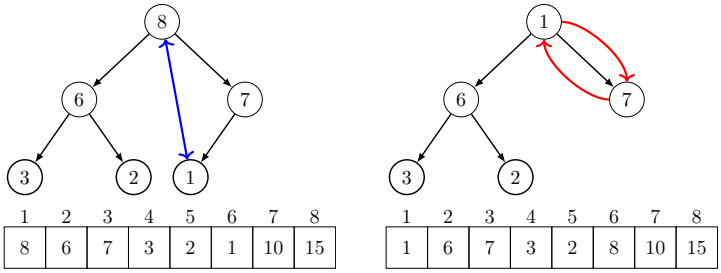
O algoritmo HEAPSORT ordena qualquer vetor A de n elementos de modo não-decrescente.



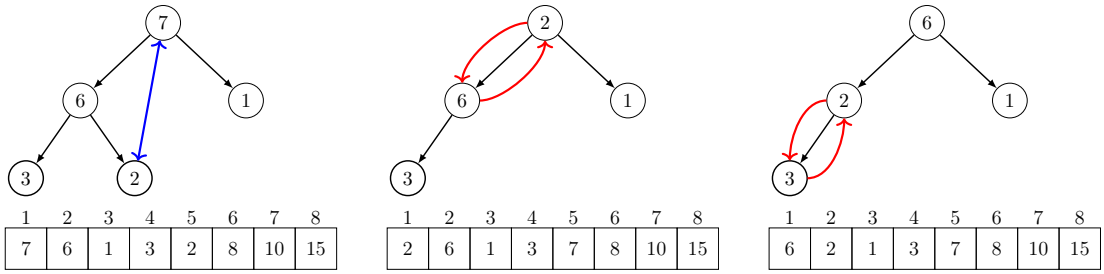
(a) Iteração $i = 8$. Troca $A[1]$ com $A[8]$, diminui heap e corrige descendo.



(b) Iteração $i = 7$. Troca $A[1]$ com $A[7]$, diminui heap e corrige descendo.

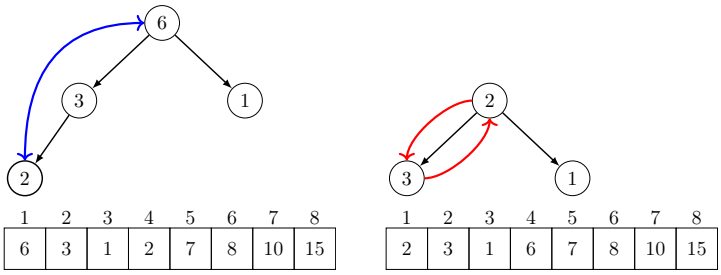


(c) Iteração $i = 6$. Troca $A[1]$ com $A[6]$, diminui heap e corrige descendo.

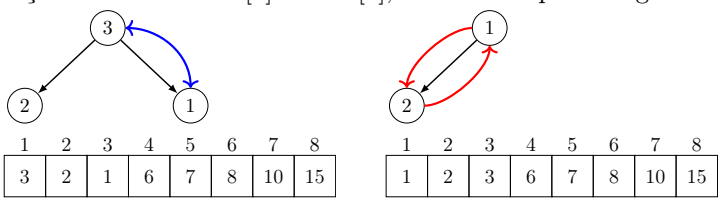


(d) Iteração $i = 5$. Troca $A[1]$ com $A[5]$, diminui heap e corrige descendo.

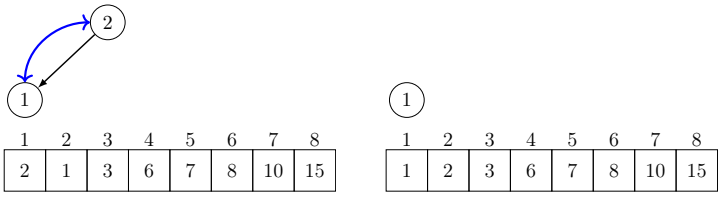
Figura 17.3: Parte 2 da execução de HEAPSORT(A, 8) (Algoritmo 17.4), com $A = (7, 3, 1, 10, 2, 8, 15, 6)$: laço para.



(a) Iteração $i = 4$. Troca $A[1]$ com $A[4]$, diminui heap e corrige descendo.



(b) Iteração $i = 3$. Troca $A[1]$ com $A[7]$, diminui heap e corrige descendo.



(c) Iteração $i = 2$. Troca $A[1]$ com $A[2]$, diminui heap e corrige descendo.

Figura 17.4: Parte 3 da execução de HEAPSORT(A, 8) (Algoritmo 17.4), com $A = (7, 3, 1, 10, 2, 8, 15, 6)$: laço para.

Demonstração. Vamos inicialmente mostrar que a seguinte frase é uma invariante para o laço **para** do algoritmo.

Invariante: *Laço para* – Heapsort

$P(x)$ = "Antes da iteração em que $i = x$ começar, temos que:

- O vetor $A[x+1..n]$ está ordenado de modo não-decrescente e contém os $n-x$ maiores elementos de A;
- $A.\texttt{tamanho} = x$ e o vetor $A[1..A.\texttt{tamanho}]$ é um heap."

Note que a linha 1 constrói um heap a partir do vetor A. Assim, antes da primeira iteração, quando $i = n$, a frase é de fato verdadeira.

Considere agora que estamos no início de uma iteração em que $i = i'$ e suponha que $P(i')$ vale, isto é, que antes dessa iteração começar temos que o subvetor $A[i'+1..n]$ está ordenado de modo não-decrescente e contém os $n-i'$ maiores elementos de A, e $A.\texttt{tamanho} = i'$ onde $A[1..A.\texttt{tamanho}]$ é um heap. Precisamos mostrar que a frase vale antes da próxima iteração, isto é, que $P(i'-1)$ é verdadeira (pois o laço decrementa i).

Note que a iteração atual começa trocando $A[1]$ com $A[i']$, colocando portanto o maior elemento de $A[1..i']$ em $A[i']$. Em seguida, diminui-se o valor de $A.\texttt{tamanho}$ em uma unidade, fazendo com que $A.\texttt{tamanho} = i'-1$. Por fim, chama-se CorrigeHeapDescendo(A, 1), transformando $A[1..i'-1]$ em heap, pois o único elemento de $A[1..A.\texttt{tamanho}]$ que pode não satisfazer a propriedade de heap é $A[1]$ e sabemos que CorrigeHeapDescendo(A, 1) funciona corretamente. Como o maior elemento de $A[1..i']$ agora está em $A[i']$ e dado que sabemos que $A[i'+1..n]$ está ordenado de modo não-decrescente e contém os $n-i'$ maiores elementos de A (porque $P(i')$ é verdadeira), concluímos que o vetor $A[i'..n]$ está ordenado de modo não-decrescente e contém os $n-i'+1$ maiores elementos de A ao fim da iteração. Assim, mostramos que $P(i'-1)$ é verdadeira, o que prova que a frase acima é uma invariante de fato.

Agora que temos uma invariante de laço, sabemos que ela vale para quando $i = 1$, em particular. Ela nos diz que que $A[2..n]$ está ordenado de modo não-decrescente e contém os maiores elementos de A. Como $A[2..n]$ contém os maiores elementos de A, o menor elemento certamente está em $A[1]$, de onde concluímos que A está totalmente ordenado. □

Sobre o tempo de execução, note que ConstroiHeap executa em tempo $O(n)$. Note ainda que a cada execução do laço **para**, a heap tem tamanho i e o CorrigeHeapDescendo é executado a partir da primeira posição do vetor, de forma que ele leva tempo $O(\log i)$. Como

são realizadas $n-1$ execuções do laço, com $2 \leq i \leq n$, o tempo total é dado por

$$\sum_{i=2}^{n} O(\log i) = O(n \log n)\,.$$

A expressão acima é válida pelo seguinte. Note que $\sum_{i=2}^{n} \log i = \log 2 + \log 3 + \cdots + \log n = \log(2 \cdot 3 \cdots n) = \log n!$. Além disso, $n! \leq n^n$, de forma que $\log n! \leq \log n^n$, o que significa que $\log n! \leq n \log n$ e, por isso, $\log n! = O(n \log n)$.

É possível ainda mostrar que no caso médio o HEAPSORT tem tempo de execução $O(n \log n)$ também.

Capítulo 18

Ordenação por troca

Os algoritmos que veremos nesse capítulo funcionam realizando sucessivas trocas de vários elementos até que algum seja colocado em sua posição correta final (relativa ao vetor completamente ordenado).

18.1 Quicksort

O *Quicksort* é um algoritmo que tem tempo de execução de pior caso $\Theta(n^2)$, o que é bem pior que o tempo $O(n \log n)$ gasto pelo *Heapsort* ou pelo *Mergesort*. No entanto, o *Quicksort* costuma ser a melhor escolha na prática. De fato, seu tempo de execução esperado é $\Theta(n \log n)$ e as constantes escondidas em $\Theta(n \log n)$ são bem pequenas. Esse algoritmo também faz uso do paradigma de divisão e conquista, assim como o *Mergesort*.

Seja $A[1..n]$ um vetor com n elementos. Dizemos que A está *particionado* com relação a um elemento, chamado *pivô*, se os elementos que são menores do que o pivô estão à esquerda dele e os outros elementos (maiores ou iguais) estão à direita dele. Note que o pivô está em sua posição correta final com relação ao vetor ordenado. A ideia do *Quicksort* é particionar o vetor e recursivamente ordenar as partes à direita e à esquerda do pivô, desconsiderando-o.

Formalmente, o algoritmo escolhe um elemento pivô qualquer (discutiremos adiante formas de escolha do pivô). Feito isso, ele particiona A com relação ao pivô deixando-o, digamos, na posição x. Assim, todos os elementos em $A[1..x-1]$ são menores ou iguais ao pivô e todos os elementos em $A[x+1..n]$ são maiores ou iguais ao pivô. O próximo passo é ordenar recursivamente os vetores $A[1..x-1]$ e $A[x+1..n]$, que efetivamente são menores do que o vetor original, pois removemos ao menos um elemento, o $A[x]$.

O procedimento, QUICKSORT, é formalizado no Algoritmo 18.1, onde PARTICIONA é um procedimento que particiona o vetor com relação a um pivô e será visto com mais detalhes adiante e ESCOLHEPIVO é um procedimento que faz a escolha de um elemento como pivô. Como QUICKSORT recursivamente acessa partes do vetor, ele recebe A e duas posições *inicio* e fim, e seu objetivo é ordenar o subvetor $A[inicio..fim]$. Assim, para ordenar um vetor A com n elementos, basta executar QUICKSORT(A, 1, n).

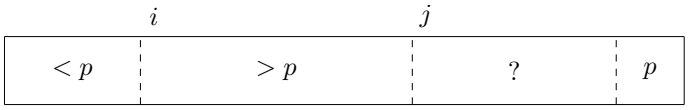
A Figura 18.1 mostra um exemplo de execução do algoritmo QUICKSORT. A Figura 18.2 mostra a árvore de recursão completa.

Algoritmo 18.1: QUICKSORT(A, *inicio*, fim)

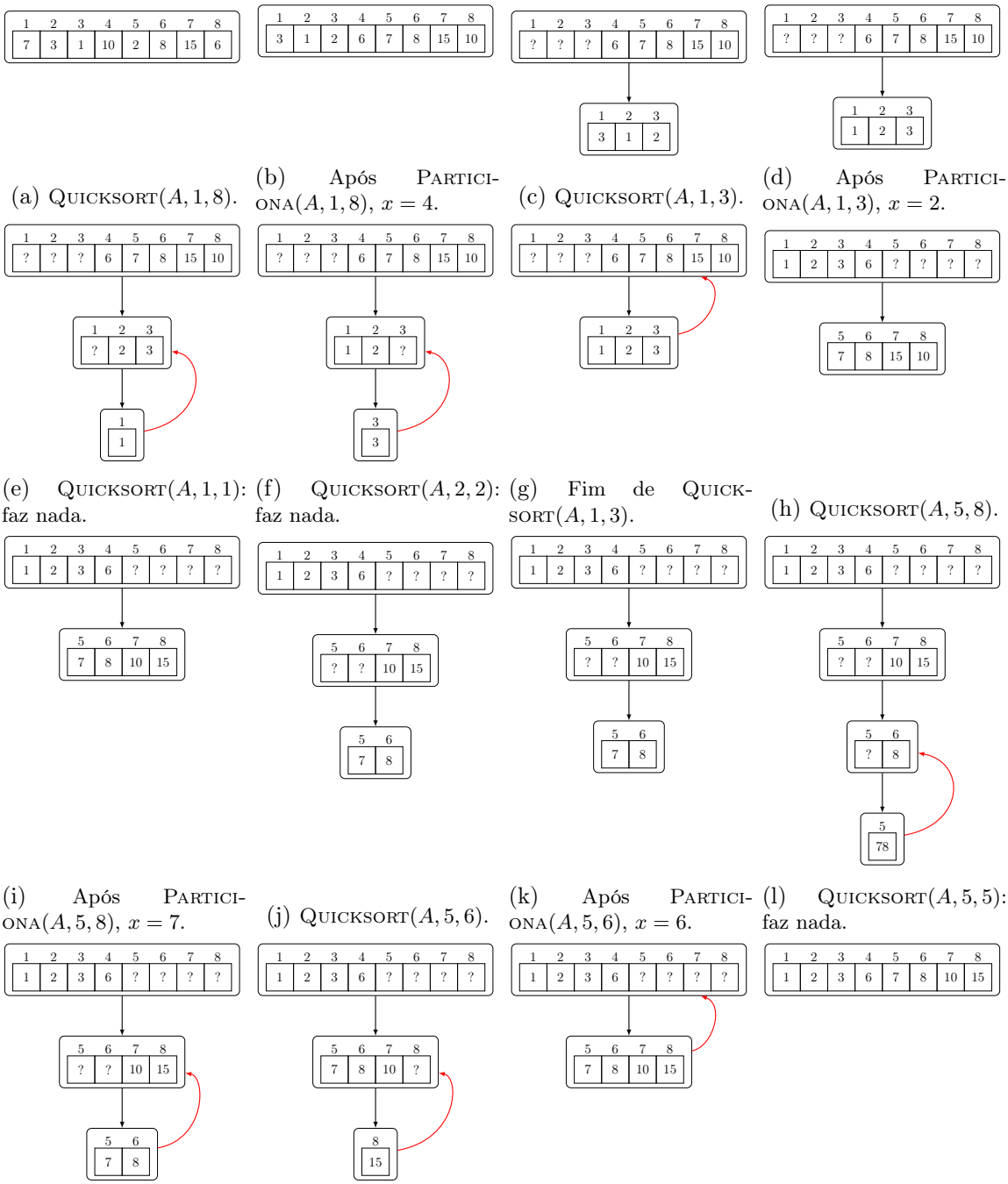
```
1 se inicio < fim então
2     p = EscolhePivo(A, inicio, fim)
3     troque A[p] com A[fim]
4     x = Particiona(A, inicio, fim)
5     Quicksort(A, inicio, x − 1)
6     Quicksort(A, x + 1, fim)
```

O procedimento PARTICIONA recebe o vetor A e as posições *inicio* e fim, e considera que o pivô é $A[fim]$. Seu objetivo é particionar $A[inicio..fim]$ com relação ao pivô. Ele devolve a posição final do pivô após a partição.

A ideia do PARTICIONA é fazer uma única varredura no vetor da esquerda para a direita. Assim, a qualquer momento, o que já foi visto está antes da posição atual e o que ainda será visto está depois. No subvetor que contém elementos já vistos, vamos manter uma divisão entre elementos que são menores do que o pivô e elementos que são maiores do que ele. Assim, a cada elemento acessado, basta decidir para qual dessas partes do vetor ele deverá ser colocado, baseando-se no fato do elemento ser maior ou menor do que o pivô. Precisamos, portanto, manter um índice j que irá indicar uma separação do vetor em duas partes: $A[inicio..j-1]$ contém elementos que já foram acessados e $A[j..fim-1]$ contém elementos que serão acessados. Também iremos manter um índice i que divide os elementos já acessados em duas partes: $A[inicio..i-1]$ contém elementos menores ou iguais ao pivô e $A[i..j-1]$ contém elementos maiores do que o pivô:



Como queremos realizar uma única varredura no vetor, precisamos decidir imediatamente o que fazer com $A[j]$. Se $A[j]$ é menor ou igual ao pivô, então ele deve ser colocado próximo



(a) Quicksort$(A, 1, 8)$.

(b) Após Particiona$(A, 1, 8)$, $x = 4$.

(c) Quicksort$(A, 1, 3)$.

(d) Após Particiona$(A, 1, 3)$, $x = 2$.

(e) Quicksort$(A, 1, 1)$: faz nada.

(f) Quicksort$(A, 2, 2)$: faz nada.

(g) Fim de Quicksort$(A, 1, 3)$.

(h) Quicksort$(A, 5, 8)$.

(i) Após Particiona$(A, 5, 8)$, $x = 7$.

(j) Quicksort$(A, 5, 6)$.

(k) Após Particiona$(A, 5, 6)$, $x = 6$.

(l) Quicksort$(A, 5, 5)$: faz nada.

(m) Fim de Quicksort$(A, 5, 6)$.

(n) Quicksort$(A, 8, 8)$: faz nada.

(o) Fim de Quicksort$(A, 5, 8)$.

(p) Fim de Quicksort$(A, 1, 8)$.

Figura 18.1: Execução de Quicksort$(A, 1, 8)$ (Algoritmo 18.1) para $A = (7, 3, 1, 10, 2, 8, 15, 6)$. Suponha que o pivô sempre é o último elemento do vetor.

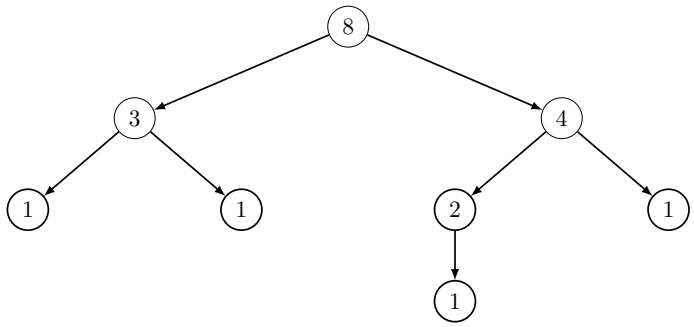


Figura 18.2: Árvore de recursão completa de QUICKSORT(A, 1, 8). Cada nó é rotulado com o tamanho do problema ($fim - inicio + 1$) correspondente.

aos elementos de $A[inicio..i-1]$. Se $A[j]$ é maior do que o pivô, então ele já está próximo aos elementos maiores, que estão em $A[i..j-1]$. O PARTICIONA é formalizado no Algoritmo 18.2 e um exemplo de sua execução é mostrado na Figura 18.3.

Algoritmo 18.2: PARTICIONA(A, $inicio$, fim)

```
1 pivo = A[fim]
2 i = inicio
3 para j = inicio até fim − 1, incrementando faça
4     se A[j] ≤ pivo então
5         troca A[i] e A[j]
6         i = i + 1
7 troca A[i] e A[fim]
8 devolve i
```

Vamos começar analisando o algoritmo PARTICIONA, que é um algoritmo iterativo simples. O Teorema 18.1 a seguir prova que ele funciona corretamente.

Teorema 18.1

O algoritmo PARTICIONA devolve um índice x tal que o *pivô* está na posição x, todo elemento em $A[1..x-1]$ é menor ou igual ao *pivô*, e todo elemento em $A[x+1..n]$ é maior que o *pivô*.

Demonstração. Vamos inicialmente provar que a seguinte frase é uma invariante do laço **para** do algoritmo.

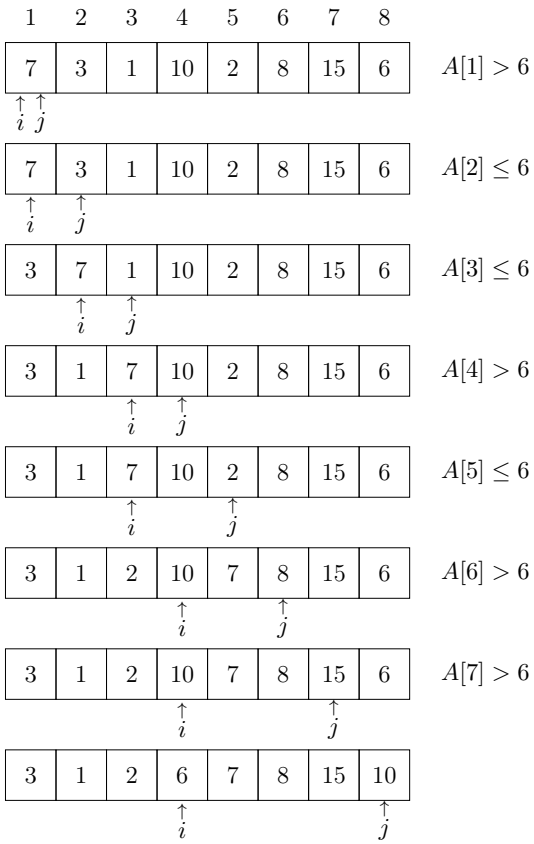


Figura 18.3: Execução de PARTICIONA(A, 1, 8) (Algoritmo 18.2), onde $A = (7, 3, 1, 10, 2, 8, 15, 6)$ e $pivo = 6$.

Invariante: *Laço para* – Particiona

$P(x, y)$ = "Antes da iteração em que $i = x$ e $j = y$ começar, temos $pivo = A[fim]$ e vale que

(i) os elementos de $A[inicio..x-1]$ são menores ou iguais a $pivo$;

(ii) os elementos de $A[x..y-1]$ são maiores do que $pivo$."

Como o pivô está inicialmente em $A[fim]$, não precisamos nos preocupar com a condição $pivo = A[fim]$ na frase por enquanto, dado que $A[fim]$ só é alterado após a execução do laço.

Antes da primeira iteração do laço **para** temos $i = inicio$ e $j = inicio$, logo as condições (i) e (ii) de $P(inicio, inicio)$ são trivialmente satisfeitas e, portanto, $P(inicio, inicio)$ vale.

Considere agora que estamos em uma iteração em que $j = j'$ e $i = i'$ e suponha que $P(i', j')$ vale, isto é, que $A[inicio..i'-1]$ contém elementos menores ou iguais a $pivo$ e $A[i'..j'-1]$ contém elementos maiores do que $pivo$. Precisamos provar que ela continua válida imediatamente antes da próxima iteração, ou seja, que $P(i'', j'+1)$ é verdadeira, onde i'' é o valor de i na iteração seguinte e $j'+1$ é o valor de j, pois o laço sempre incrementa j.

Na iteração atual, se $A[j'] > pivo$, então a única operação feita é alterar j' para $j'+1$. Como já sabíamos que $A[i'..j'-1]$ só tinha elementos maiores e vimos que $A[j'] > pivo$, então temos que $A[i'..j']$ contém elementos maiores do que $pivo$. Também vale que $A[inicio..i'-1]$ continua contendo elementos menores ou iguais ao fim da iteração. Como o valor de i não muda para a próxima iteração, acabamos de mostrar que $P(i', j'+1)$ é verdadeira.

Agora, se $A[j'] \leq pivo$, então trocamos $A[i']$ com $A[j']$ e incrementamos o valor da variável i, que agora vale $i'+1$. Assim, como antes sabíamos que $A[inicio..i'-1]$ tinha elementos menores ou iguais a $pivo$, agora sabemos que todo elemento em $A[inicio..i']$ é menor ou igual a $pivo$. Como sabíamos que $A[i'..j'-1]$ só tinha elementos maiores e fizemos a troca, agora sabemos que todo elemento em $A[i'+1..j']$ é maior do que $pivo$. Assim, temos que $P(i'+1, j'+1)$ é verdadeira.

Com isso, a frase acima é de fato uma invariante do laço **para**. Ela nos garante que ao fim da execução do laço, quando temos $j = fim$ e $i = x$, para algum valor x, trocar $A[x]$ com $A[fim]$ na linha 7 irá de fato particionar o vetor $A[1..n]$ com relação a $pivo$. □

Com relação ao tempo, claramente o laço **para** é executado $fim - inicio$ vezes, de forma que o tempo de execução de Particiona é $\Theta(fim - inicio)$, isto é, leva tempo $\Theta(n)$ se $n = fim - inicio + 1$ é a quantidade de elementos dados na entrada.

Para provar que o algoritmo Quicksort funciona corretamente, usaremos indução no

valor de $n = fim - inicio + 1$ (o tamanho do vetor). Perceba que a escolha do pivô não interfere na explicação do funcionamento ou da corretude do algoritmo. Você pode assumir por enquanto, se preferir, que EscolhePivo(A, $inicio$, fim) devolve o índice fim. Veja a prova completa no Teorema 18.2 a seguir.

Teorema 18.2: *Corretude de* Quicksort

O algoritmo Quicksort ordena qualquer vetor A de modo não-decrescente.

Demonstração. Vamos provar que o algoritmo está correto por indução no tamanho $n = fim - inicio + 1$ do vetor dado.

Quando $n \leq 1$, temos que $n = fim - inicio + 1 \leq 1$, o que implica em $fim \leq inicio$. Veja que quando isso acontece, o algoritmo não faz nada. De fato, se $fim \leq inicio$, então $A[inicio..fim]$ é um vetor com um ou zero elementos e, portanto, já ordenado. Logo, o algoritmo funciona corretamente nesse caso.

Considere então que $n > 1$ e suponha que o algoritmo corretamente ordena vetores com menos do que n elementos.

Veja que $n = fim - inicio + 1 > 1$ implica em $fim > inicio$. Então o algoritmo executa a linha 4, que devolve um índice x, com $inicio \leq x \leq fim$, tal que $A[x]$ é um elemento que está em sua posição final na ordenação desejada, todos os elementos de $A[inicio..x-1]$ são menores ou iguais a $A[x]$, e todos os elementos de $A[x+1..fim]$ são maiores do que $A[x]$, como mostrado no Teorema 18.1.

O algoritmo então chama Quicksort(A, $inicio$, $x-1$). Veja que o tamanho do vetor nessa chamada recursiva é $x-1-inicio+1 = x-inicio \leq fim-inicio < fim-inicio+1 = n$. Logo, podemos usar a hipótese de indução para afirmar que $A[inicio..x-1]$ estará ordenado após essa chamada.

Em seguida o algoritmo chama Quicksort(A, $x+1$, fim). Da mesma forma, $fim - (x+1)+1 = fim-x \leq fim-inicio < n$, de forma que após a execução da linha 6 sabemos, por hipótese de indução, que $A[x+1..fim]$ estará ordenado.

Portanto, todo o vetor A fica ordenado ao final da execução da chamada atual. □

18.1.1 Análise do tempo de execução

Perceba que o tempo de execução de Quicksort(A, $inicio$, fim) depende fortemente de como a partição é feita, o que depende da escolha do pivô. Seja $n = fim - inicio + 1$ a quantidade de elementos do vetor de entrada.

Suponha que EscolhePivo devolve o índice que contém o maior elemento armazenado

em $A[inicio..fim]$. Nesse caso, o vetor é sempre particionado em um subvetor de tamanho $n-1$ e outro de tamanho 0. Como o tempo de execução do PARTICIONA é $\Theta(m)$ quando m elementos lhe são passados, temos que, nesse caso, o tempo de execução de QUICKSORT é dado por $T(n) = T(n-1) + \Theta(n)$. Se esse fenômeno ocorre em todas as chamadas recursivas, então temos

$$\begin{aligned}
T(n) &= T(n-1) + n \\
&= T(n-2) + (n-1) + n \\
&\vdots \\
&= T(1) + \sum_{j=2}^{n} i \\
&= 1 + \frac{(n-1)(n+2)}{2} \\
&= \Theta(n^2)\,.
\end{aligned}$$

Intuitivamente, conseguimos perceber que esse é o pior caso possível. Formalmente, o tempo de execução de pior caso é dado por $T(n) = \max_{0 \le x \le n-1}(T(x) + T(n-x-1)) + n$. Vamos utilizar indução para mostrar que $T(n) \le n^2$. Supondo que $T(m) \le m^2$ para todo $m < n$, obtemos

$$\begin{aligned}
T(n) &\le \max_{0 \le x \le n-1}(cx^2 + c(n-x-1)^2) + n \\
&\le (n-1)^2 + n \\
&= n^2 - (2n-1) + n \\
&\le n^2\,,
\end{aligned}$$

onde o máximo na primeira linha é atingido quando $x = 0$ ou $x = n-1$. Para ver isso, seja $f(x) = x^2 + (n-x-1)^2$ e note que $f'(x) = 2x - 2(n-x-1)$, de modo que $f'((n-1)/2) = 0$. Assim, $(n-1)/2$ é um ponto máximo ou mínimo. Como $f''((n-1)/2) > 0$, temos que $(n-1)/2$ é ponto de mínimo de f. Portanto, os pontos máximos são $x = 0$ e $x = n-1$.

Por outro lado, pode ser que ESCOLHEPIVO sempre devolve o índice que contém a mediana dos elementos do vetor, de forma que a partição produza duas partes de mesmo tamanho, sendo o tempo de execução dado por $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.

Suponha agora que PARTICIONA divide o problema em um subproblema de tamanho

$(n-1)/1000$ e outro de tamanho $999(n-1)/1000$. Então o tempo de execução é dado por

$$\begin{aligned} T(n) &= T\left(\frac{n-1}{1000}\right) + T\left(\frac{999(n-1)}{1000}\right) + \Theta(n) \\ &\leq T\left(\frac{n}{1000}\right) + T\left(\frac{999n}{1000}\right) + \Theta(n)\,. \end{aligned}$$

É possível mostrar que temos $T(n) = O(n \log n)$.

De fato, para qualquer constante $k > 1$ (e.g., $k = 10^{100}$), se o algoritmo Particiona divide A em partes de tamanho aproximadamente n/k e $(k-1)n/k$, então o tempo de execução ainda é $O(n \log n)$. Vamos utilizar o método da substituição para mostrar que $T(n) = T(n/k) + T((k-1)n/k) + n$ tem solução $O(n \log n)$. Assuma que $T(n) \leq c$ para alguma constante $c \geq 1$ e todo $n \leq k-1$. Vamos provar que $T(n) = T(n/k)+T((k-1)n/k)+n$ é no máximo

$$dn \log n + n$$

para todo $n \geq k$ e alguma constante $d > 0$. Começamos notando que $T(k) \leq T(k-1) + T(1) + k \leq 2c + k \leq dk \log k + k$. Suponha que $T(m) \leq dm \log m + m$ para todo $k < m < n$ e vamos analisar $T(n)$:

$$\begin{aligned} T(n) &= T\left(\frac{n}{k}\right) + T\left(\frac{(k-1)n}{k}\right) + n \\ &\leq d\left(\frac{n}{k}\log\left(\frac{n}{k}\right)\right) + \frac{n}{k} + d\left(\frac{(k-1)n}{k}\log\left(\frac{(k-1)n}{k}\right)\right) + \frac{(k-1)n}{k} + n \\ &= d\left(\frac{n}{k}\log\left(\frac{n}{k}\right)\right) + d\left(\frac{(k-1)n}{k}\left(\log(k-1) + \log\left(\frac{n}{k}\right)\right)\right) + 2n \\ &= dn\log n + n - dn\log k + \left(\frac{d(k-1)n}{k}\log(k-1) + n\right) \\ &\leq dn\log n + n\,, \end{aligned}$$

onde a última desigualdade vale se $d \geq k/\log k$, pois para tal valor de d temos

$$dn \log k \geq \left(\frac{d(k-1)n}{k}\log(k-1) + n\right)\,.$$

Portanto, acabamos de mostrar que $T(n) = O(n \log n)$ quando o Quicksort divide o vetor A sempre em partes de tamanho aproximadamente n/k e $(k-1)n/k$.

A ideia por trás desse fato que, a princípio, pode parecer contraintuitivo, é que o tamanho da árvore de recursão é $\log_{k/(k-1)} n = \Theta(\log n)$ e, em cada passo, é executada uma quantidade de passos proporcional ao tamanho do vetor analisado, de forma que o tempo total de

execução é $O(n \log n)$. Com isso, vemos que se as divisões, a cada chamada recursiva, não deixarem um subvetor vazio muitas vezes, então isso já seria bom o suficiente para termos um bom tempo de execução (assintoticamente falando).

O problema da discussão que tivemos até agora é que é improvável que a partição seja *sempre* feita da mesma forma em todas as chamadas recursivas. Vamos agora analisar o que acontece no caso médio, quando cada uma das $n!$ possíveis ordenações dos elementos de A tem a mesma chance de ser a distribuição do vetor de entrada A. Suponha que EscolhePivo sempre devolve a posição fim.

Perceba que o tempo de execução de Quicksort é dominado pela quantidade de operações feitas na linha 4 de Particiona. Seja então X uma variável aleatória que conta o número de vezes que essa linha é executada durante uma execução completa do Quicksort, isto é, ela representa o número de comparações feitas durante toda a execução. Pela segunda observação acima, o tempo de execução do Quicksort é $T(n) \leq \mathbb{E}[X]$. Logo, basta encontrar um limitante superior para $\mathbb{E}[X]$.

Sejam $o_1, \ldots, o_n$ os elementos de A em sua ordenação final (após estarem ordenados de modo não-decrescente), i.e., $o_1 \leq o_2 \leq \cdots \leq o_n$ e não necessariamente $o_i = A[i]$. A primeira observação importante é que dois elementos o_i e o_j são comparados no máximo uma vez, pois elementos são comparados somente com o pivô e uma vez que algum elemento é escolhido como pivô ele é colocado em sua posição final e ignorado pelas chamadas posteriores. Então defina X_{ij} como a variável aleatória indicadora para o evento "o_i é comparado com o_j". Assim,

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \,.$$

Utilizando a linearidade da esperança, concluímos que

$$\begin{aligned} \mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{P}(o_i \text{ ser comparado com } o_j) \,. \end{aligned} \tag{18.1}$$

Vamos então calcular $\mathbb{P}(o_i \text{ ser comparado com } o_j)$. Comecemos notando que para o_i ser comparado com o_j, um dos dois precisa ser o primeiro elemento de $O_{ij} = \{o_i, o_{i+1}, \ldots, o_j\}$ a ser escolhido como pivô. De fato, caso o_k, com $i < k < j$, seja escolhido como pivô antes de o_i e o_j, então o_i e o_j irão para partes diferentes do vetor ao fim da chamada atual ao

algoritmo Particiona e nunca serão comparados durante toda a execução. Portanto,

$$
\begin{aligned}
&\mathbb{P}(o_i \text{ ser comparado com } o_j) \\
&\quad = 1 - \mathbb{P}(o_1 \text{ não ser comparado com } o_j) \\
&\quad = 1 - \mathbb{P}(\text{qualquer elemento em } O_{ij} \setminus \{o_i, o_j\} \text{ ser escolhido primeiro como pivô em } O_{ij}) \\
&\quad = 1 - \frac{j-i+1-2}{j-i+1} = \frac{2}{j-i+1}\,.
\end{aligned}
$$

Assim, voltando à (18.1), temos

$$
\begin{aligned}
\mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i+1} \frac{2}{k} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\log n) = O(n \log n)\,.
\end{aligned}
$$

Portanto, concluímos que o *tempo médio de execução* de Quicksort é $O(n \log n)$.

Se, em vez de escolhermos um elemento fixo para ser o pivô, escolhermos um dos elementos do vetor uniformemente ao acaso, então uma análise análoga à que fizemos aqui mostra que o *tempo esperado de execução* dessa versão aleatória de Quicksort é $O(n \log n)$. Assim, sem supor nada sobre a entrada do algoritmo, garantimos um tempo de execução esperado de $O(n \log n)$.

Capítulo 19

Ordenação sem comparação

Vimos, nos capítulos anteriores, alguns algoritmos com tempo de execução (de pior caso ou caso médio) $\Theta(n \log n)$. *Mergesort* e *Heapsort* têm esse limitante no pior caso e *Quicksort* possui tempo de execução esperado da ordem de $n \log n$. Acontece que todos os algoritmos anteriores são baseados *em comparações* entre os elementos de entrada.

Suponha um algoritmo correto para o problema da ordenação que recebe como entrada n números. Veja que, por ser correto, ele deve corretamente ordenar qualquer uma das $n!$ possíveis entradas. Suponha que esse algoritmo faz no máximo k comparações para ordenar qualquer uma dessas entradas. Como uma comparação tem dois resultados possíveis (sim ou não), podemos associar uma string binária de k bits com uma possível execução do algoritmo. Temos, portanto, no máximo 2^k possíveis execuções diferentes do algoritmo para todas as $n!$ entradas. Pelo Princípio da Casa dos Pombos e porque supomos que o algoritmo está correto, devemos ter $2^k \geq n!$ (uma execução diferente para cada entrada). Como $n! \geq (n/2)^{n/2}$, temos que $k \geq (n/2)\log(n/2)$, isto é, $k = \Omega(n \log n)$.

Pela discussão acima, temos que qualquer algoritmo baseado em comparações requer $\Omega(n \log n)$ comparações no pior caso. Portanto, *Mergesort* e *Heapsort* são assintoticamente ótimos.

Algumas vezes, no entanto, sabemos informações extras sobre os dados de entrada. Nesses casos, é possível obter um algoritmo de ordenação com tempo melhor, inclusive linear. Obviamente, pela discussão acima, tais algoritmos não podem ser baseados em comparações. Para exemplificar, vamos discutir o algoritmo *Counting sort* a seguir.

19.1 Counting sort

Seja $A[1..n]$ um vetor que contém somente números inteiros entre 0 e k. Sabendo desses limites nos valores dos elementos, é possível fazer uma ordenação baseada em contagem. Suponha que existem $cont_i$ elementos de valor $i-1$ em A. Veja que o vetor ordenado final deverá ter $cont_0$ elementos 0, seguidos por $cont_1$ elementos 1, e assim por diante, até ter $cont_k$ elementos k.

Uma forma de implementar essa ideia seria, portanto, percorrer o vetor A e adicionando uma unidade em $cont_i$ sempre que $A[k] = i$. Em seguida, poderíamos escrever $cont_0$ números 0 nas primeiras $cont_0$ posições de A. Depois, escrever $cont_1$ números 1 nas posições seguintes, e assim por diante. Mas lembre-se que apesar de estarmos sempre falando de vetores que armazenam números, esses métodos de ordenação precisam ser gerais o suficiente para funcionar sobre vetores que armazenam qualquer tipo de registro, contanto que cada registro contenha um campo *chave* que possa diferenciá-lo dos outros.

O algoritmo CountingSort usa a contagem, mas de uma forma que os próprios elementos do vetor A cujas chaves são 0 sejam copiados para as primeiras posições, depois todos os elementos de A que têm chave 1 sejam copiados para as posições seguintes, e assim por diante. Para isso, vamos manter um vetor $C[0..k]$ contador que manterá em $C[i]$ a quantidade de elementos cuja chave é *menor ou igual a* i (não apenas os de chave i). A ideia é que o elemento $A[j]$ tem $C[A[j]]$ elementos que devem vir antes dele na ordenação e, por isso, sabemos exatamente em que posição $A[j]$ deve estar ao fim da ordenação. Por causa disso, precisaremos ainda de outro vetor auxiliar B, de tamanho n, que irá receber as cópias dos elementos de A já nas suas posições finais. Devido ao uso desses vetores auxiliares, esse algoritmo não é *in-place*. A ordem relativa de elementos iguais será mantida, de modo que o algoritmo é estável.

Como cada elemento de A é colocado na sua posição final sem que seja feito sua comparação com outro elemento de A, esse algoritmo consegue executar em tempo menor do que $n \log n$. Formalizaremos o tempo a seguir. O CountingSort é formalizado no Algoritmo 19.1 e a Figura 19.1 apresenta um exemplo de execução.

Os quatro laços **para** existentes no CountingSort são executados, respectivamente, k, n, k e n vezes. Portanto, claramente a complexidade do procedimento é $\Theta(n+k)$. Concluímos então que quando $k = O(n)$, o algoritmo CountingSort é executado em tempo $\Theta(n)$, de modo que é assintoticamente mais eficiente que todos os algoritmos de ordenação vistos aqui.

Este algoritmo é comumente utilizado como subrotina de um outro algoritmo de ordenação em tempo linear, chamado *Radix sort*, e é essencial para o funcionamento do *Radix sort* que o *Counting sort* seja estável.

	1	2	3	4	5	6	7	8	9	10
A	3	0	5	4	0	3	1	2	0	4
B	·	·	·	·	·	·	·	·	·	·

	0	1	2	3	4	5
C	0	0	0	0	0	0

(a) Inicialização.

	0	1	2	3	4	5	
C	0	0	0	1	0	0	$A[1] = 3$
C	1	0	0	1	0	0	$A[2] = 0$
C	1	0	0	1	0	1	$A[3] = 5$
C	1	0	0	1	1	1	$A[4] = 4$
C	2	0	0	1	1	1	$A[5] = 0$
C	2	0	0	2	1	1	$A[6] = 3$
C	2	1	0	2	1	1	$A[7] = 1$
C	2	1	1	2	1	1	$A[8] = 2$
C	3	1	1	2	1	1	$A[9] = 0$
C	3	1	1	2	2	1	$A[10] = 4$

(b) Laço **para** da linha 4.

	0	1	2	3	4	5
C	3	1	1	2	2	1
		$\uparrow i$				
C	3	4	1	2	2	1
			$\uparrow i$			
C	3	4	5	2	2	1
				$\uparrow i$		
C	3	4	5	7	2	1
					$\uparrow i$	
C	3	4	5	7	9	1
						$\uparrow i$
C	3	4	5	7	9	10

(c) Laço **para** da linha 6.

	0	1	2	3	4	5
C	3	4	5	7	9	10
$j = 10$					$\uparrow A[j]$	
C	3	4	5	7	8	10
$j = 9$	$\uparrow A[j]$					
C	2	4	5	7	8	10
$j = 8$			$\uparrow A[j]$			
C	2	4	4	7	8	10
$j = 7$		$\uparrow A[j]$				
C	2	3	4	7	8	10
$j = 6$				$\uparrow A[j]$		
C	2	3	4	6	8	10
$j = 5$	$\uparrow A[j]$					
C	1	3	4	6	8	10
$j = 4$					$\uparrow A[j]$	
C	1	3	4	6	7	10
$j = 3$						$\uparrow A[j]$
C	1	3	4	6	7	9
$j = 2$	$\uparrow A[j]$					
C	0	3	4	6	7	9
$j = 1$				$\uparrow A[j]$		
C	0	3	4	5	7	9
$j = 0$						

	1	2	3	4	5	6	7	8	9	10
B	·	·	·	·	·	·	·	·	·	·
									$\uparrow C[A[j]]$	
B	·	·	·	·	·	·	·	·	4	·
			$\uparrow C[A[j]]$							
B	·	·	0	·	·	·	·	·	4	·
					$\uparrow C[A[j]]$					
B	·	·	0	·	2	·	·	·	4	·
				$\uparrow C[A[j]]$						
B	·	·	0	1	2	·	·	·	4	·
						$\uparrow C[A[j]]$				
B	·	·	0	1	2	3	·	·	4	·
		$\uparrow C[A[j]]$								
B	·	0	0	1	2	3	·	·	4	·
								$\uparrow C[A[j]]$		
B	·	0	0	1	2	3	·	4	4	·
										$\uparrow C[A[j]]$
B	·	0	0	1	2	3	·	4	4	5
	$\uparrow C[A[j]]$									
B	0	0	0	1	2	3	·	4	4	5
						$\uparrow C[A[j]]$				
B	0	0	0	1	2	3	3	4	4	5

(d) Laço **para** da linha 8.

Figura 19.1: Execução de CountingSort(A, 10, 5) (Algoritmo 19.1), com $A = (3, 0, 5, 4, 0, 3, 1, 2, 0, 4)$.

Algoritmo 19.1: COUNTINGSORT(A, n, k)

```
/* C é um vetor auxiliar de contadores e B manterá o vetor ordenado */
Sejam B[1..n] e C[0..k] vetores
para i = 0 até k, incrementando faça
    C[i] = 0
/* C[i] inicialmente guarda a quantidade de ocorrências de i em A */
para j = 1 até n, incrementando faça
    C[A[j]] = C[A[j]] + 1
/* C[i] deve guardar a qtd. de ocorrências de elementos ≤ i em A */
para i = 1 até k, incrementando faça
    C[i] = C[i] + C[i − 1]
/* Colocando o resultado da ordenação de A em B: A[j] deve ir para a
   posição C[A[j]] */
para j = n até 1, decrementando faça
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
devolve B
```

Parte

V

Técnicas de construção de algoritmos

"(...) the more comfortable one is with the full array of possible design techniques, the more one starts to recognize the clean formulations that lie within messy problems out in the world."

Jon Kleinberg, Éva Tardos – Algorithm Design, 2005.

Nesta parte

Infelizmente, não existe uma solução única para todos os problemas computacionais. Também não existe fórmula que nos ajude a descobrir qual a solução para um problema. Uma abordagem prática é discutir técnicas que já foram utilizadas antes e que possam ser aplicadas a vários problemas, na esperança de poder reutilizá-las ou adaptá-las aos novos problemas. Veremos os três principais paradigmas de projeto de algoritmos, que são estratégias gerais para solução de problemas.

A maioria dos problemas que consideraremos nesta parte são problemas de *otimização*. Em geral, um problema desses possui um conjunto de restrições que define o que é uma *solução viável* e uma função objetivo que determina o valor de cada solução. O objetivo é encontrar uma *solução ótima*, que é uma solução viável com melhor valor de função objetivo (maximização ou minimização).

Nos próximos capítulos, usaremos os termos "problema" e "subproblema" para nos referenciar igualmente a "uma instância do problema" e "uma instância menor do problema", respectivamente.

Capítulo 20

Divisão e conquista

Divisão e conquista é um paradigma para o desenvolvimento de algoritmos que faz uso da recursividade. Para resolver um problema utilizando esse paradigma, seguimos os três seguintes passos:

- O problema é dividido em pelo menos dois subproblemas menores;
- Os subproblemas menores são resolvidos recursivamente: cada um desses subproblemas menores é dividido em subproblemas ainda menores, a menos que sejam tão pequenos a ponto de ser simples resolvê-los diretamente;
- Soluções dos subproblemas menores são combinadas para formar uma solução do problema inicial.

Os algoritmos *Mergesort* (Capítulo 16) e *Quicksort* (Seção 18.1), para ordenação de vetores, fazem uso desse paradigma. Nesse capítulo veremos outros algoritmos que também são de divisão e conquista.

20.1 Multiplicação de inteiros

Considere o seguinte problema.

Problema 20.1: ***Multiplicação de inteiros***

Dados dois inteiros x e y contendo n dígitos cada, obter o produto xy.

Todos nós conhecemos o algoritmo clássico de multiplicação. Seja $x = 5678$ e $y = 1234$ (ou seja, $n = 4$):

$$
\begin{array}{r}
5\,6\,7\,8 \\
\times\, 1\,2\,3\,4 \\
\hline
2\,2\,7\,1\,2 \\
1\,7\,0\,3\,4\,0 \\
+\; 1\,1\,3\,5\,6\,0\,0 \\
5\,6\,7\,8\,0\,0\,0 \\
\hline
7\,0\,0\,6\,6\,5\,2
\end{array}
$$

Para mostrar que esse algoritmo está de fato correto, precisamos mostrar que para quaisquer dois inteiros x e y, ele devolve xy. Seja $y = y_1 y_2 \ldots y_n$, onde y_i é um dígito de 0 a 9. Note que o algoritmo faz

$$(x \times y_n) + (x \times y_{n-1} \times 10) + \cdots + (x \times y_2 \times 10^{n-2}) + (x \times y_1 \times 10^{n-1}),$$

o que é igual a

$$x\left((y_n) + (y_{n-1} \times 10) + \cdots + (y_2 \times 10^{n-2}) + (y_1 \times 10^{n-1})\right),$$

e isso é exatamente xy.

Com relação ao tempo, observe que, somar ou multiplicar dois dígitos simples é uma operação básica. Note que para obter o primeiro produto parcial $(x \times y_n)$, precisamos de n multiplicações de um dígito e talvez mais $n-1$ somas (para os *carries*), isto é, usamos no máximo $2n$ operações. Similarmente, para obter $x \times y_{n-1}$, outras no máximo $2n$ operações básicas foram necessárias. E isso é verdade para todos os produtos parciais. Veja que a multiplicação por potências de 10 pode ser feita de forma bem simples ao se adicionar zeros à direita. Assim, são no máximo $2n$ operações para cada um dos n dígitos de y, isto é, $2n^2$ operações no máximo. Perceba que cada número obtido nos n produtos parciais tem no máximo $2n+1$ dígitos. Assim, as adições dos produtos parciais leva outras no máximo $2n^2 + n$ operações. Logo, temos que o tempo de execução desse algoritmo é $O(n^2)$, ou seja, quadrático no tamanho da entrada (quantidade de dígitos recebida).

Felizmente, existem algoritmos melhores para resolver o problema da multiplicação. Sim! O algoritmo básico que nós aprendemos na escola não é único.

Vamos escrever $x = 10^{\lceil n/2 \rceil} a + b$ e $y = 10^{\lceil n/2 \rceil} c + d$, onde a e c têm $\lfloor \frac{n}{2} \rfloor$ dígitos cada e b e d têm $\lceil \frac{n}{2} \rceil$ dígitos cada. No exemplo anterior, se $x = 5678$ e $y = 1234$, temos $a = 56$, $b = 78$,

$c = 12$ e $d = 34$. Podemos então escrever

$$xy = (10^{\lceil n/2 \rceil}a + b)(10^{\lceil n/2 \rceil}c + d) = 10^{2\lceil n/2 \rceil}ac + 10^{\lceil n/2 \rceil}(ad + bc) + bd\,. \tag{20.1}$$

Perceba então que reduzimos o problema de multiplicar números de n dígitos para o problema de multiplicar números de $\lceil n/2 \rceil$ ou $\lfloor n/2 \rfloor$ dígitos. Isto é, podemos usar recursão para resolvê-lo. Com $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$ apenas quando $n > 2$, nosso caso base será a multiplicação de números com 1 ou 2 dígitos.

Um algoritmo de divisão e conquista simples para o problema da multiplicação é descrito no Algoritmo 20.1. Ele usa a função IgualaTam(x, y), que deixa os números x e y com o mesmo número de dígitos (igual ao número de dígitos do maior deles) colocando zeros à esquerda se necessário e devolve o número de dígitos (agora igual) desses números.

Algoritmo 20.1: MultiplicaInteiros(x, y)

1 Seja $n =$ IgualaTam(x, y)
2 **se** $n \leq 2$ **então**
3 ⌊ **devolve** xy
4 Seja $x = 10^{\lceil n/2 \rceil}a + b$ e $y = 10^{\lceil n/2 \rceil}c + d$, onde a e c têm $\lfloor \frac{n}{2} \rfloor$ dígitos cada e b e d têm $\lceil \frac{n}{2} \rceil$ dígitos cada
5 $p_1 =$ MultiplicaInteiros(a, c)
6 $p_2 =$ MultiplicaInteiros(a, d)
7 $p_3 =$ MultiplicaInteiros(b, c)
8 $p_4 =$ MultiplicaInteiros(b, d)
9 **devolve** $10^{2\lceil n/2 \rceil}p_1 + 10^{\lceil n/2 \rceil}(p_2 + p_3) + p_4$

É possível provar por indução em n que MultiplicaInteiros corretamente calcula xy, usando a identidade em 20.1. Agora perceba que seu tempo de execução, $T(n)$, pode ser descrito por $T(n) = 4T(n/2) + n$, pois as operações necessárias na linha 9 levam tempo $O(n)$ e IgualaTam(x, y) também é $O(n)$. Pelo Método Mestre (Seção 8.4), temos $T(n) = O(n^2)$, isto é, não houve muita melhora com relação ao algoritmo simples.

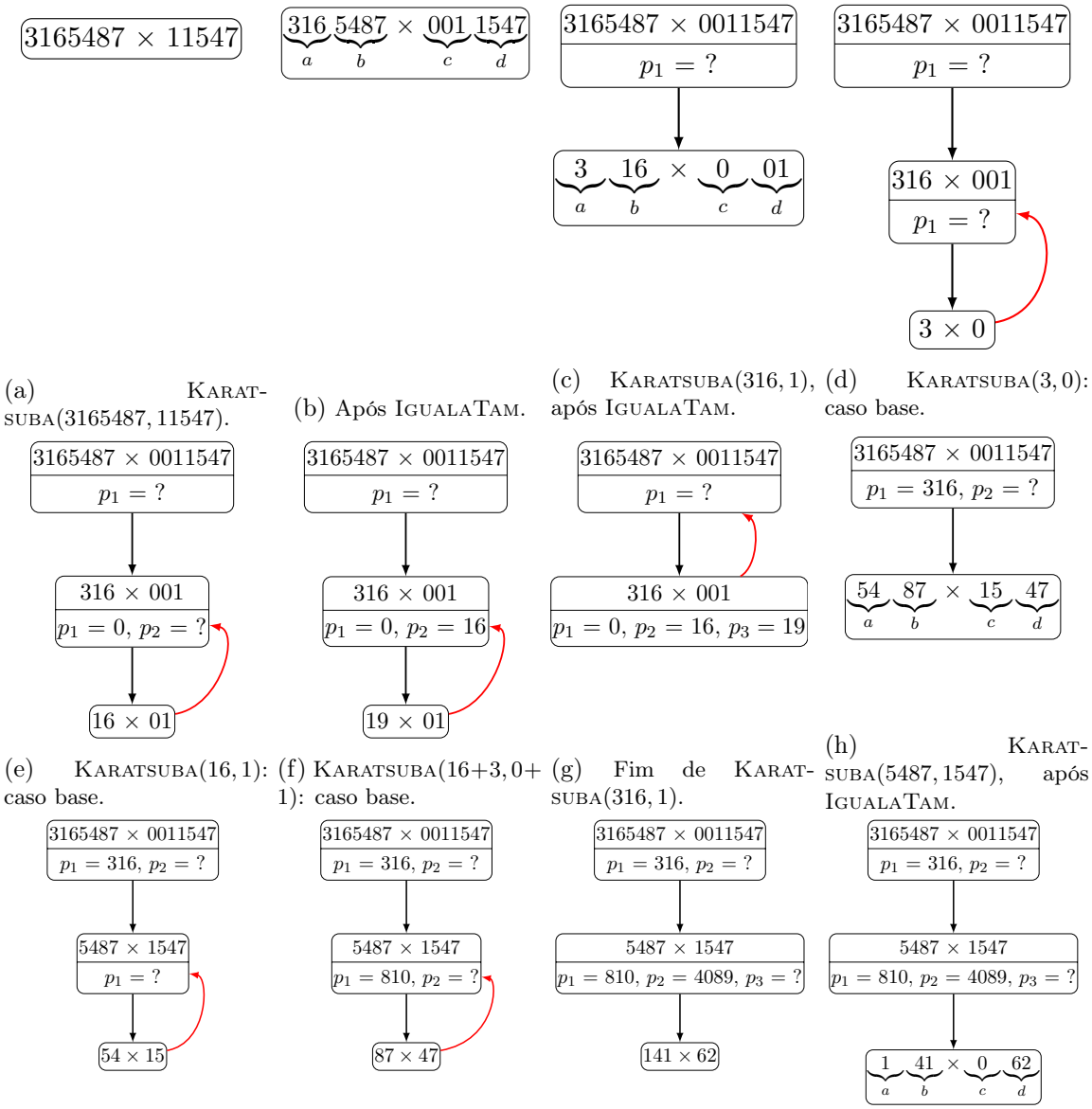
O algoritmo de Karatsuba também usa o paradigma de divisão e conquista, mas ele se aproveita do fato de que $(a + b)(c + d) = ac + ad + bc + bd$ para fazer apenas 3 chamadas recursivas. Calculando apenas os produtos ac, bd e $(a+b)(c+d)$, como $(a+b)(c+d)-ac-bd = ad + bc$, conseguimos calcular (20.1). Veja o procedimento formalizado no Algoritmo 20.2. As Figuras 20.1 e 20.2 mostram um exemplo de execução enquanto a Figura 20.3 mostra a árvore de recursão completa do mesmo exemplo.

Novamente, é possível provar por indução em n que Karatsuba corretamente calcula xy,

Algoritmo 20.2: KARATSUBA(x, y, n)

Seja $n = \text{IGUALATAM}(x, y)$
se $n \leq 2$ **então**
  devolve xy
Seja $x = 10^{\lceil n/2 \rceil} a + b$ e $y = 10^{\lceil n/2 \rceil} c + d$, onde a e c têm $\lfloor \frac{n}{2} \rfloor$ dígitos cada e b e d têm $\lceil \frac{n}{2} \rceil$ dígitos cada
$p_1 = \text{KARATSUBA}(a, c)$
$p_2 = \text{KARATSUBA}(b, d)$
$p_3 = \text{KARATSUBA}(a + b, c + d)$
devolve $10^{2\lceil n/2 \rceil} p_1 + 10^{\lceil n/2 \rceil}(p_3 - p_1 - p_2) + p_2$

usando a identidade em 20.1 e o fato que $(a + b)(c + d) = ac + ad + bc + bd$. Seu tempo de execução, $T(n)$, pode ser descrito por $T(n) = 3T(n/2) + n$, o que é $O(n^{1.59})$. Logo, no pior caso, o algoritmo de Karatsuba é melhor do que o algoritmo básico de multiplicação.



(a) Karatsuba(3165487, 11547).

(b) Após IgualaTam.

(c) Karatsuba(316, 1), após IgualaTam.

(d) Karatsuba(3, 0): caso base.

(e) Karatsuba(16, 1): caso base.

(f) Karatsuba(16+3, 0+1): caso base.

(g) Fim de Karatsuba(316, 1).

(h) Karatsuba(5487, 1547), após IgualaTam.

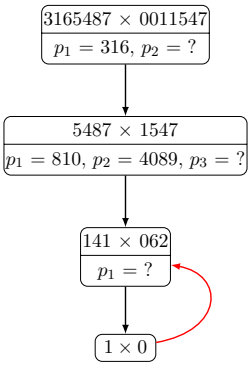
(i) Karatsuba(54, 15): caso base.

(j) Karatsuba(87, 47): caso base.

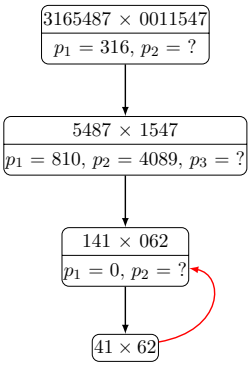
(k) Karatsuba(54 + 87, 15 + 47).

(l) Karatsuba(141, 62) após IgualaTam.

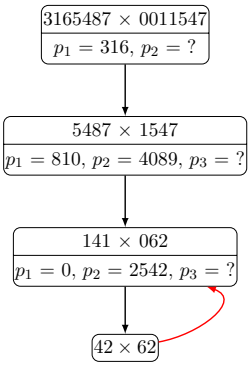
Figura 20.1: Parte 1 da execução de Karatsuba(3165487, 11547) (Algoritmo 20.2).



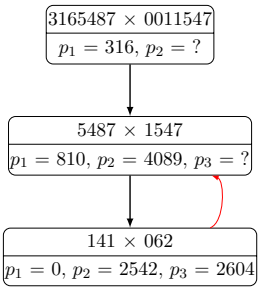
(a) Karatsuba$(87, 47)$: caso base.



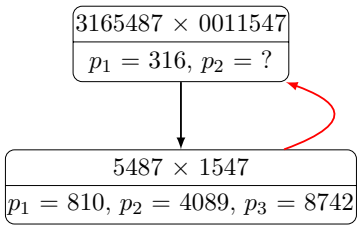
(b) Karatsuba$(54 + 87, 15 + 47)$.



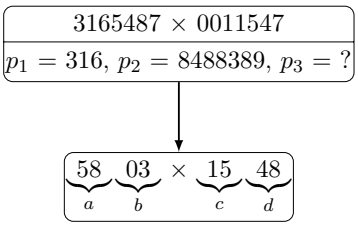
(c) Karatsuba$(141, 62)$ após IgualaTam.



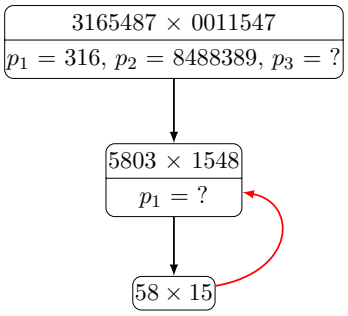
(d) Karatsuba$(141, 62)$ após IgualaTam.



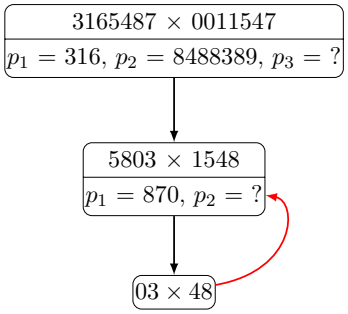
(e) Karatsuba$(141, 62)$ após IgualaTam.



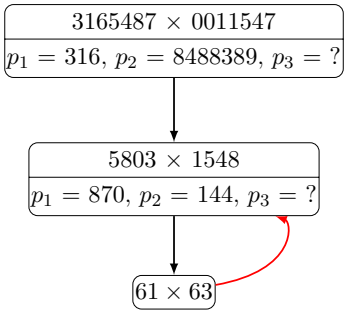
(f) Karatsuba$(141, 62)$ após IgualaTam.



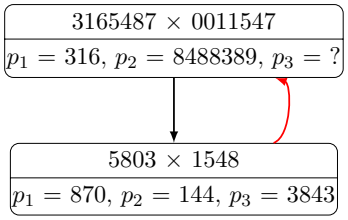
(g) Karatsuba$(141, 62)$ após IgualaTam.



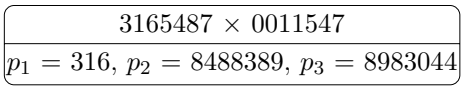
(h) Karatsuba$(141, 62)$ após IgualaTam.



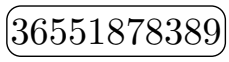
(i) Karatsuba$(141, 62)$ após IgualaTam.



(j) Karatsuba$(141, 62)$ após IgualaTam.



(k) Karatsuba$(141, 62)$ após IgualaTam.



(l) Karatsuba$(141, 62)$ após IgualaTam.

Figura 20.2: Parte 2 da execução de Karatsuba(3165487, 11547) (Algoritmo 20.2).

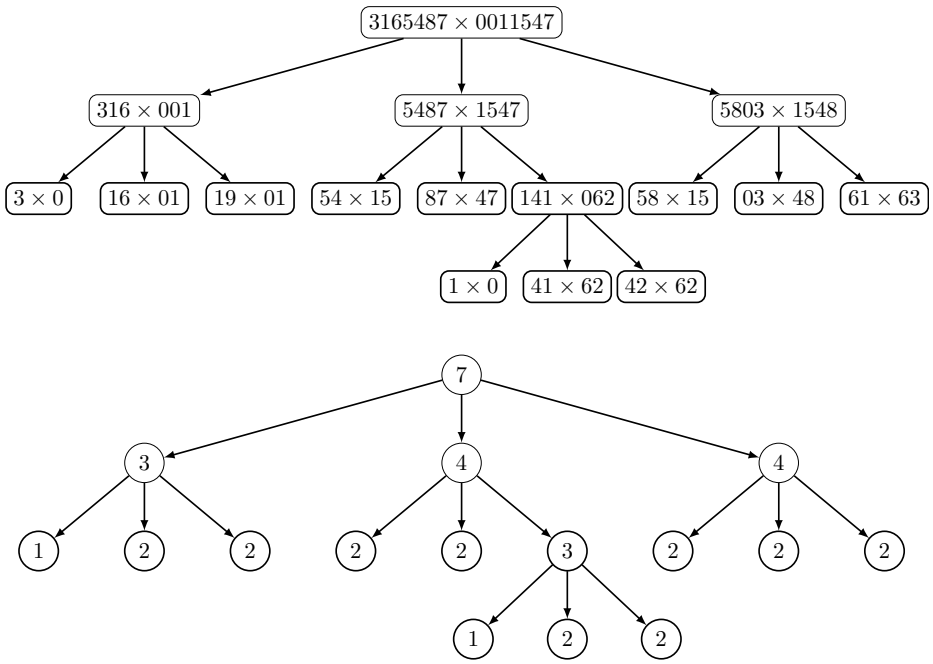


Figura 20.3: Árvore de recursão completa de Karatsuba(3165487, 11547). Na parte superior, cada nó é rotulado com o problema enquanto que na parte inferior cada nó é rotulado com o tamanho do problema.

Capítulo 21

Algoritmos gulosos

Um algoritmo é dito guloso quando constrói uma solução através de uma sequência de decisões que visam o melhor cenário de curto prazo, sem garantia de que isso levará ao melhor resultado global. Algoritmos gulosos são muito usados porque costumam ser rápidos e fáceis de implementar. Em geral, é fácil descrever um algoritmo guloso que forneça uma solução viável e tenha complexidade de tempo fácil de ser analisada. A dificuldade normalmente se encontra em provar se a solução obtida é de fato ótima. Na maioria das vezes, inclusive, elas não são ótimas, mas há casos em que é possível mostrar que elas têm valor próximo ao ótimo.

Neste capítulo veremos diversos algoritmos que utilizam esse paradigma. Também são gulosos alguns algoritmos clássicos em grafos como Kruskal (Seção 25.1), Prim (Seção 25.2) e Dijkstra (Seção 27.1.1).

21.1 Escalonamento de tarefas compatíveis

Uma tarefa t_x tem tempo inicial s_x e tempo final f_x indicando que, se selecionada, acontecerá no intervalo $[s_x, f_x)$. Dizemos que duas tarefas t_i e t_j são *compatíveis* se os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ não se sobrepõem, isto é, $s_i \geq f_j$ ou $s_j \geq f_i$. Considere o seguinte problema.

Problema 21.1: *Escalonamento de tarefas compatíveis*

Dado um conjunto $T = \{t_1, \ldots, t_n\}$ com n tarefas onde cada $t_i \in T$ tem um tempo inicial s_i e um tempo final f_i, encontrar o maior subconjunto de tarefas mutuamente compatíveis.

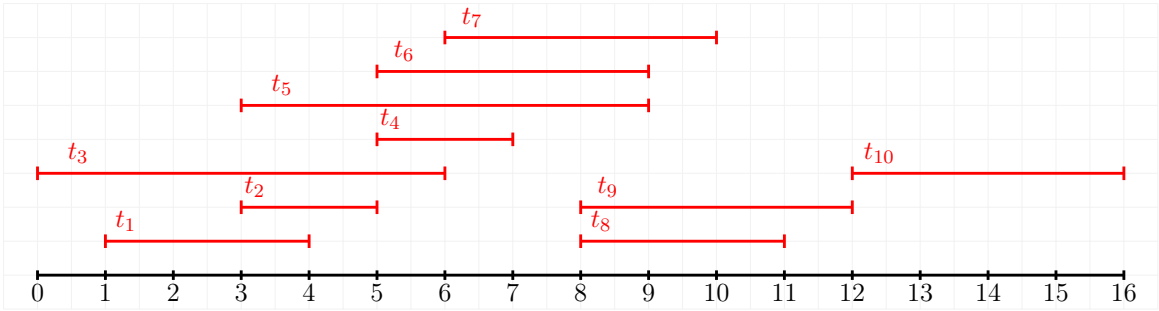


Figura 21.1: Conjunto $T = \{t_1, t_2, \ldots, t_{10}\}$ de tarefas e seus respectivos intervalos. Note que $\{t_3, t_9, t_{10}\}$ é uma solução viável para essa instância. As soluções viáveis $\{t_1, t_4, t_8, t_{10}\}$ e $\{t_2, t_4, t_8, t_{10}\}$, no entanto, são ótimas.

Veja a Figura 21.1 para um exemplo do problema.

Note como temos escolhas a fazer: tarefas que sejam compatíveis com as tarefas já escolhidas. Podemos pensar em vários algoritmos gulosos para esse problema, como um que sempre escolhe as tarefas de menor duração ou outro que sempre escolhe as tarefas que começam primeiro. Note como todos eles têm a boa intenção de escolher o maior número de tarefas. Ademais, todos sempre devolvem soluções viáveis (pois tomam o cuidado de fazer escolhas compatíveis com as tarefas já escolhidas).

Uma vez que temos um algoritmo que devolve soluções viáveis para nosso problema, um primeiro passo é testá-lo, criando instâncias e verificando quais respostas ele dá para elas. Em geral, criamos instâncias que possam fazer o algoritmo falhar, o que no caso de problemas de otimização significa devolver uma solução que não é ótima. As duas estratégias gulosas mencionadas acima para o problema do escalonamento não são ótimas, porque não devolvem soluções ótimas sempre. Para mostrar que um algoritmo não é ótimo, basta encontrar uma instância específica para a qual ele retorna uma solução não ótima. Chamamos essas instâncias de *contraexemplos*.

Uma terceira estratégia para o escalonamento, que parece não possuir contraexemplos, é a de sempre escolher uma tarefa que acabe o quanto antes, ou que termine primeiro (com menor valor f_i). Ela é descrita no Algoritmo 21.1, que mantém em um conjunto S as tarefas já escolhidas. Observe que o fato de ela aparentemente não possuir contraexemplos **não prova** que ela é ótima. Precisamos demonstrar formalmente que *qualquer que seja* o conjunto de tarefas, o algoritmo sempre devolve uma solução ótima.

Note que o primeiro passo do algoritmo é ordenar as tarefas de acordo com o tempo final e renomeá-las, de forma que em t_1 temos a tarefa que termina primeiro. Essa, portanto, é a primeira escolha do algoritmo. Em seguida, dentre as tarefas restantes, são escolhidas apenas aquelas que começam após o término da última tarefa escolhida. Dessa forma, o

Algoritmo 21.1: EscalonaCompativel(T, n)

```
1 Ordene as tarefas em ordem não-decrescente de tempo final
2 Renomeie-as de modo que f_1 ≤ f_2 ≤ ··· ≤ f_n
3 S = {t_1}
4 k = 1 /* k mantém o índice da última tarefa adicionada à S */
5 para i = 2 até n, incrementando faça
6     se s_i ≥ f_k então
7         S = S ∪ {t_i}
8         k = i
9 devolve S
```

algoritmo mantém a invariante de que S é um conjunto de tarefas compatíveis. Assim, o conjunto S devolvido é de fato uma solução viável para o problema. O Lema 21.2 mostra que na verdade S é uma solução ótima.

Lema 21.2

Dado conjunto $T = \{t_1, \ldots, t_n\}$ com n tarefas onde cada $t_i \in T$ tem um tempo inicial s_i e um tempo final f_i, o algoritmo EscalonaCompativel(T, n) devolve uma solução ótima para o problema de Escalonamento de tarefas compatíveis.

Demonstração. Seja $t_k \in T$ uma tarefa qualquer. Denote por $T_k = \{t_i \in T \colon s_i \geq f_k\}$, isto é, o conjunto das tarefas que começam após o fim de t_k. Seja $t_x \in T_k$ uma tarefa que termina primeiro em T_k (com menor valor f_i em T_k). Note que EscalonaCompativel(T_k, $|T_k|$) escolhe t_x primeiro. Vamos supor que essa escolha não está presente em nenhuma solução ótima, isto é, se $S_k \subseteq T_k$ é uma solução ótima para T_k, então $t_x \notin S_k$.

Seja $t_y \in S_k$ uma tarefa que termina primeiro em S_k (com menor valor f_i em S_k). Monte o conjunto $S'_k = (S_k \setminus \{t_y\}) \cup \{t_x\}$. Note que, como ambas t_x e t_y estão em T_k, temos que $f_x \leq f_y$. E como $f_y \leq s_z$ para qualquer $t_z \in S_k$, temos que S'_k é uma solução viável para T_k (é um conjunto de tarefas mutuamente compatíveis). Mas note que $|S_k| = |S'_k|$, de forma que S'_k deve, portanto, ser solução ótima para T_k também, o que é uma contradição, pois contém t_x. Ou seja, a escolha gulosa está de fato presente em uma solução ótima. □

Com relação ao tempo de execução, note que as linhas 1 e 2 levam tempo $\Theta(n \log n)$ para serem executadas (podemos usar, por exemplo, o algoritmo *Mergesort* para ordenar as tarefas). O laço **para** da linha 5 claramente leva tempo total $\Theta(n)$ para executar, pois

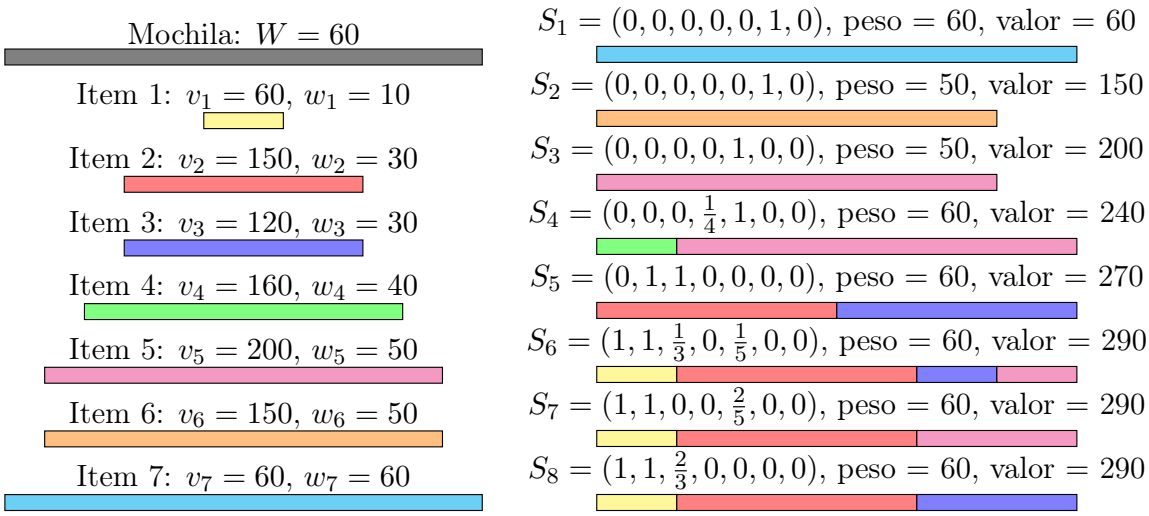


Figura 21.2: Instância do problema da mochila à esquerda com 7 itens e os respectivos pesos e valores. À direita, 8 soluções viáveis, onde as soluções S_6, S_7 e S_8 são ótimas para a instância dada.

analisamos todas as tarefas fazendo operações de tempo constante. Assim, o tempo desse algoritmo é dominado pela ordenação das tarefas, tendo tempo total portanto de $\Theta(n \log n)$.

21.2 Mochila fracionária

O problema da mochila é um dos clássicos em computação. Nessa seção veremos a versão da mochila fracionária. A Seção 22.3 apresenta a versão da mochila inteira.

Problema 21.3: ***Mochila fracionária***

Dado um conjunto $I = \{1, 2, \ldots, n\}$ de n itens onde cada $i \in I$ tem um peso w_i e um valor v_i associados e dada uma mochila com capacidade de peso W, selecionar frações $f_i \in [0, 1]$ dos itens tal que $\sum_{i=1}^{n} f_i w_i \leq W$ e $\sum_{i=1}^{n} f_i v_i$ é máximo.

Veja a Figura 21.2 para um exemplo de instância desse problema e soluções viáveis representadas pelas sequências das frações, isto é, uma solução S é dada por $S = (f_1, \ldots, f_n)$.

Uma estratégia gulosa óbvia é a de sempre escolher o item de maior valor que ainda cabe na mochila. Isso de fato cria soluções viáveis, no entanto não nos dá a garantia de sempre encontrar a solução ótima. No exemplo da Figura 21.2, essa estratégia nos faria escolher inicialmente o item 5, que cabe inteiro, deixando uma capacidade restante de peso 10. O próximo item de maior valor é o 4, que não cabe inteiro. Pegamos então a maior fração possível

sua que caiba, que é $1/4$. Com isso, geramos a solução viável $S_4 = (0, 0, 0, 1/4, 1, 0, 0)$ de custo 240, mas sabemos que existe solução melhor (logo, essa não é ótima).

É importante lembrar que para mostrar que um algoritmo não é ótimo, basta mostrar um exemplo no qual ele devolve uma solução não ótima. E veja que para fazer isso, basta mostrar alguma outra solução que seja melhor do que a devolvida pelo algoritmo, isto é, não é necessário mostrar a solução ótima daquela instância. Isso porque, para uma dada instância específica, pode ser difícil provar que uma solução é ótima. No exemplo da Figura 21.2, dissemos que o valor 290 é o valor de uma solução ótima, mas por que você acreditaria nisso?

Note que a estratégia anterior falha porque a escolha pelo valor ignora totalmente outro aspecto do problema, que é a restrição do peso da mochila. Intuitivamente, o que queremos é escolher itens de maior valor que ao mesmo tempo tenham pouco peso, isto é, que tenham melhor custo-benefício. Assim, uma outra estratégia gulosa é sempre escolher o item com a maior razão v/w (valor/peso). No exemplo da Figura 21.2, temos $v_1/w_1 = 6$, $v_2/w_2 = 5$, $v_3/w_3 = 4$, $v_4/w_4 = 4$, $v_5/w_5 = 4$, $v_6/w_6 = 3$ e $v_7/w_7 = 1$, de forma que essa estratégia funcionaria da seguinte forma. O item com a maior razão valor/peso é o item 1 e ele cabe inteiro na mochila, portanto faça $f_1 = 1$. Temos agora capacidade restante de 50. O próximo item de maior razão valor/peso é o item 2 e ele também cabe inteiro na mochila atual, portanto faça $f_2 = 1$. Temos agora capacidade restante de peso 20. O próximo item de maior razão é o item 3, mas ele não cabe inteiro. Pegamos então a maior fração possível dele que caiba, que é $2/3$, portanto fazendo $f_3 = 2/3$. Veja que essa é a solução S_8, que é de fato uma das soluções ótimas do exemplo dado. Isso **não prova** que a estratégia escolhida é ótima, no entanto. Devemos fazer uma demonstração formal se suspeitarmos que nossa estratégia é ótima. Essa, no caso, de fato é (veja o Lema 21.4). O algoritmo que usa essa estratégia está descrito formalmente no Algoritmo 21.2.

O Algoritmo 21.2 funciona inicialmente ordenando os itens e renomeando-os para ter $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$. Assim, o item 1 tem a maior razão valor/peso. Mantemos uma variável *capacidade* para armazenar a capacidade restante da mochila. No laço **enquanto** da linha 5 o algoritmo seleciona itens inteiros ($f_i = 1$) na ordem da razão valor/peso enquanto eles couberem inteiros na mochila ($w_i \leq$ *capacidade*). Do próximo item, se ele existir, pegamos a maior fração possível que cabe no restante do espaço (linha 10). Nenhum outro item é considerado, tendo $f_i = 0$ (laço da linha 11). Note que a solução gerada é de fato viável, tem custo $\sum_{i=1}^{n} f[i]v_i$ e vale que $\sum_{i=1}^{n} f[i]w_i = W$, pois caso contrário poderíamos pegar uma fração maior de algum item.

Com relação ao tempo de execução, note que a linha 1 leva tempo $\Theta(n \log n)$ (usando, por exemplo, o *Mergesort* para fazer a ordenação). Os dois laços levam tempo total $\Theta(n)$, pois apenas fazemos operações constantes para cada item da entrada. Assim, o tempo desse

Algoritmo 21.2: MochilaFracionaria(I, n, W)

```
1  Ordene os itens pela razão valor/peso e os renomeie de forma que
     v_1/w_1 ≥ v_2/w_2 ≥ ··· ≥ v_n/w_n
2  capacidade = W
3  Seja f[1..n] um vetor
4  i = 1
5  enquanto i ≤ n e capacidade ≥ w_i faça
6      f[i] = 1
7      capacidade = capacidade − w_i
8      i = i + 1
9  se i ≤ n então
10     f[i] = capacidade/w_i
11 para j = i + 1 até n, incrementando faça
12     f[j] = 0
13 devolve f
```

algoritmo é dominado pela ordenação, tendo tempo total portanto de $\Theta(n \log n)$.

Lema 21.4

Dado um conjunto $I = \{1, 2, \ldots, n\}$ de n itens onde cada $i \in I$ tem um peso w_i e um valor v_i associados e dada uma mochila com capacidade de peso W, o algoritmo MochilaFracionaria(I, n, W) devolve uma solução ótima para o problema da Mochila fracionária.

Demonstração. Seja f a solução devolvida por MochilaFracionaria(I, n, W). Seja f^* uma solução ótima para a mesma instância. Se $f = f^*$, então não há o que provar. Então suponha que f difere de f^* em alguns valores. Seja i o menor índice tal que $f[i] > f^*[i]$ (não podemos ter sempre $f[j] \leq f^*[j]$, porque para criar f sempre fazemos a escolha pela maior fração possível e $f[i] \neq 0$). Note que $\sum_{j=1}^{n} f[j]w_j = \sum_{j=1}^{n} f^*[j]w_j = W$, pois caso contrário seria possível melhorar essas soluções, pegando mais frações de itens ainda não escolhidos. Assim, pela escolha de i, vale que $\sum_{j=i}^{n} f[j]w_j = \sum_{j=i}^{n} f^*[j]w_j$, pois $f[i] = f^*[j]$ para $j < i$.

Vamos agora criar uma outra solução f' a partir de f^*. Nossa intenção é que f' não seja tão diferente de f^*, por isso faça inicialmente $f'[j] = f^*[j]$ para todo $j < i$. Com isso, até o momento temos $\sum_{j=1}^{i-1} f'[j]w_j = \sum_{j=1}^{i-1} f^*[j]w_j$. Agora faça $f'[i] = f[i]$, aproximando f' da solução f do algoritmo (lembre-se que, para $j < i$, $f[j] = f^*[j]$).

Como $f'[i] = f[i] > f^*[i]$, estamos pegando uma fração maior de um item, o que causa um desbalanço no peso. Por isso, não podemos simplesmente copiar para o restante de f' os mesmos valores de f^*. Para garantir que f' será uma solução viável, vamos garantir que o restante do peso, $\sum_{j=i}^{n} f'[j]w_j$, seja igual ao restante do peso da solução ótima, $\sum_{j=i}^{n} f^*[j]w_j$. Ademais, cada $f'[j]$, para $i \leq j \leq n$, deve ser tal que $0 \leq f'[j] \leq 1$. Isso forma um sistema de equações lineares que possui solução, então tal f' realmente existe. Reescrevendo a igualdade $\sum_{j=i}^{n} f'[j]w_j = \sum_{j=i}^{n} f^*[j]w_j$ e isolando valores referentes a i, temos então que vale

$$w_i(f'[i] - f^*[i]) = \sum_{j=i+1}^{n} w_j(f^*[j] - f'[j])\,. \tag{21.1}$$

Vamos agora verificar que o valor de f' não difere do valor de f^*. Partimos da definição do valor de f', por construção, e usamos algumas propriedades algébricas junto à propriedades do nosso algoritmo a fim de compará-lo com o valor de f^*:

$$\begin{aligned}
\sum_{j=1}^{n} f'[j]v_j &= \left(\sum_{j=1}^{i-1} f^*[j]v_j\right) + f'[i]v_i + \sum_{j=i+1}^{n} f'[j]v_j \\
&= \left(\sum_{j=1}^{n} f^*[j]v_j - f^*[i]v_i - \sum_{j=i+1}^{n} f^*[j]v_j\right) + f'[i]v_i + \sum_{j=i+1}^{n} f'[j]v_j \\
&= \sum_{j=1}^{n} f^*[j]v_j + v_i\left(f'[i] - f^*[i]\right) - \sum_{j=i+1}^{n} v_j(f^*[j] - f'[j]) \\
&= \sum_{j=1}^{n} f^*[j]v_j + v_i\left(f'[i] - f^*[i]\right)\frac{w_i}{w_i} - \sum_{j=i+1}^{n} v_j(f^*[j] - f'[j])\frac{w_j}{w_j} \\
&\geq \sum_{j=1}^{n} f^*[j]v_j + \frac{v_i}{w_i}\left(f'[i] - f^*[i]\right)w_i - \sum_{j=i+1}^{n} \frac{v_i}{w_i}\left(f^*[j] - f'[j]\right)w_j && (21.2) \\
&= \sum_{j=1}^{n} f^*[j]v_j + \frac{v_i}{w_i}\left((f'[i] - f^*[i])w_i - \sum_{j=i+1}^{n} (f^*[j] - f'[j])w_j\right) \\
&= \sum_{j=1}^{n} f^*[j]v_j\,, && (21.3)
\end{aligned}$$

onde (21.2) vale pois $v_i/w_i \geq v_j/w_j$, e (21.3) vale devido a (21.1), da construção de f'. Com isso, concluímos que f' não é pior do que f^*. De fato, como f^* é ótima, f' também deve ser. Fazendo essa transformação repetidamente chegaremos a f, e, portanto, f também é ótima. □

21.3 Compressão de dados

Considere o seguinte problema.

Problema 21.5: ***Compressão de dados***

Dado um arquivo com caracteres pertencentes a um alfabeto A onde cada $i \in A$ possui uma frequência f_i de aparição, encontrar uma sequência de bits (código) para representar cada caractere de modo que o arquivo binário tenha tamanho mínimo.

Por exemplo, suponha que o alfabeto é $A = \{a, b, c, d\}$. Poderíamos usar um código de *largura fixa*, fazendo $a = 00$, $b = 01$, $c = 10$ e $d = 11$. Assim, a sequência "*acaba*" pode ser representada em binário por "0010000100". Mas note que o símbolo a aparece bastante nessa sequência, de modo que talvez utilizar um código de *largura variável* seja melhor. Poderíamos, por exemplo, fazer $a = 0$, $b = 01$, $c = 10$ e $d = 1$, de forma que a sequência "*acaba*" ficaria representada por "0100010". No entanto, "0100010" poderia ser interpretado também como "*baaac*", ou seja, esse código escolhido possui ambiguidade. Perceba que o problema está no fato de que o bit 0 pode tanto representar o símbolo a quanto um prefixo do código do símbolo b. Podemos nos livrar desse problema utilizando um código de largura variável que seja *livre de prefixo*. Assim, podemos fazer $a = 0$, $b = 10$, $c = 110$ e $d = 111$.

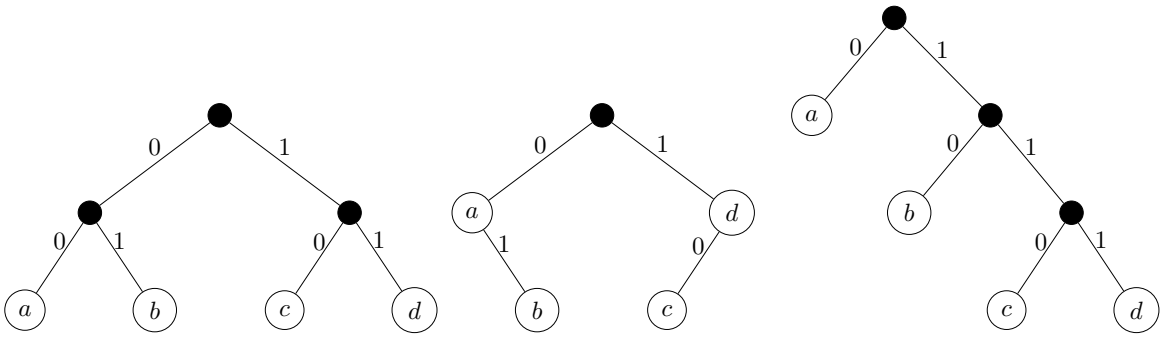
Vamos representar os códigos dos símbolos de um alfabeto A por uma árvore binária onde existe o rótulo 0 nas arestas que levam a filhos da esquerda, rótulo 1 nas arestas que levam a filhos da direita e existem rótulos em alguns nós com os símbolos de A. Assim, o código formado no caminho entre a raiz e o nó rotulado por um símbolo $i \in A$ é o código binário desse símbolo. Note que uma árvore como a descrita acima é livre de prefixo se e somente se os nós rotulados são folhas. Veja a Figura 21.3 para exemplos.

Note que o *comprimento* do código de $i \in A$ é exatamente o nível do nó rotulado com i na árvore T e isso independe da quantidade de 0s e 1s no código. Denotaremos tal valor por $d_T(i)$. Com essa nova representação e notações, podemos redefinir o problema de compressão de dados da seguinte forma.

Problema 21.6: ***Compressão de dados***

Dado um alfabeto A onde cada símbolo $i \in A$ possui uma frequência f_i, encontrar uma árvore binária T cujas folhas são rotuladas com símbolos de A e o custo $c(T) = \sum_{i \in A} f_i d_T(i)$ é mínimo.

No que seque, seja $n = |A|$. Uma forma de construir uma árvore pode ser partir de n



$(a = 00, b = 01, c = 10, d = 11)$ $(a = 0, b = 01, c = 10, d = 1)$ $(a = 0, b = 10, c = 110, d = 111)$

Figura 21.3: Árvores representando três códigos diferentes para o alfabeto $A = \{a, b, c, d\}$.

árvores que contêm um único nó cada, um para cada $i \in A$, e repetitivamente escolher duas árvores e uni-las por um novo nó pai sem rótulo até que se chegue em uma única árvore. Veja na Figura 21.4 três exemplos simples.

Note que independente de como as árvores são escolhidas, são feitas exatamente $n - 1$ uniões para gerar a árvore final. O ponto importante desse algoritmo é decidir quais duas árvores serão escolhidas para serem unidas em um certo momento. Veja que nossa função de custo envolve multiplicar a frequência do símbolo pelo nível em que ele aparece na árvore. Assim, intuitivamente, parece bom manter os símbolos de maior frequência próximos à raiz. Vamos associar a cada árvore um certo peso. Inicialmente, esse peso é a frequência do símbolo que rotula os nós. Quando escolhemos duas árvores e a unimos, associamos à nova árvore a soma dos pesos das duas que a formaram. Assim, uma escolha gulosa bastante intuitiva é selecionar as duas árvores de menor peso sempre. Veja que no início isso equivale aos dois símbolos de menor frequência. Essa ideia encontra-se formalizada no Algoritmo 21.3, conhecido como algoritmo de Huffman. Um exemplo de execução é dado na Figura 21.5.

Algoritmo 21.3: HUFFMAN(A, f)

1 Sejam i e j os símbolos de menor frequência em A
2 **se** $|A| == 2$ **então**
3 ⌊ **devolve** *Árvore com um nó pai não rotulado e i e j como nós filhos*
4 Seja $A' = (A \setminus \{i, j\}) \cup \{ij\}$
5 Defina $f_{ij} = f_i + f_j$
6 $T' =$ HUFFMAN(A', f)
7 Construa T a partir de T' separando a folha rotulada por ij em folhas i e j irmãs
8 **devolve** T

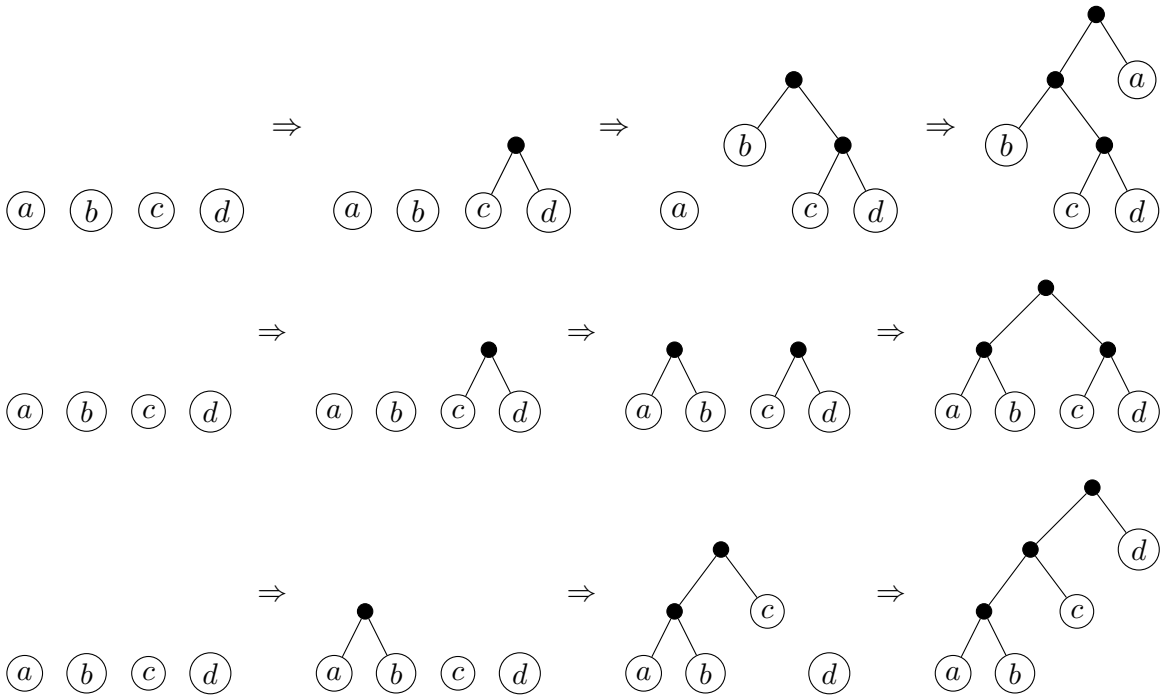
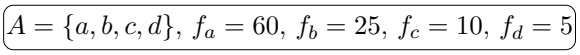


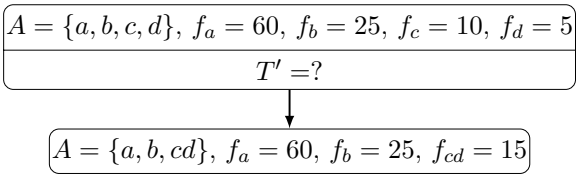
Figura 21.4: Construção de árvores representativas de códigos binários tendo início com $n = |A|$ árvores triviais.

Note que o algoritmo pode ser facilmente implementado em tempo $\Theta(n^2)$ no pior caso: existem $\Theta(n)$ chamadas recursivas pois essa é a quantidade total de uniões que faremos, e uma chamada pode levar tempo $\Theta(n)$ para encontrar os dois símbolos de menor frequência (procurando-os de maneira simples dentre todos os disponíveis). Uma forma de melhorar esse tempo é usando uma estrutura de dados apropriada. Note que a operação que mais leva tempo é a de encontrar os dois símbolos de menor frequência. Assim, podemos usar a estrutura *heap*, que fornece remoção do elemento de maior prioridade (no caso, o de menor frequência) em tempo $O(\log n)$ sobre um conjunto de n elementos. Ela também fornece inserção em tempo $O(\log n)$, o que precisa ser feito quando o novo símbolo é criado e sua frequência definida como a soma das frequências dos símbolos anteriores (linhas 4 e 5). Assim, o tempo total do algoritmo melhora para $\Theta(n \log n)$ no pior caso.

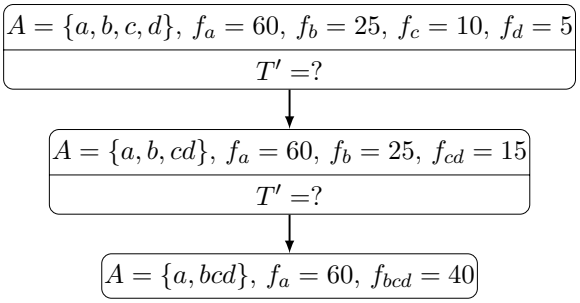
Até agora, o que podemos afirmar é que o algoritmo de Huffman de fato calcula uma árvore binária que representa códigos binários livres de prefixo de um dado alfabeto. Veja que, por construção, os nós rotulados são sempre folhas. O Lema 21.7 mostra que na verdade a estratégia escolhida por Huffman sempre gera uma árvore cujo custo é o menor possível dentre todas as árvores que poderiam ser geradas dado aquele alfabeto.



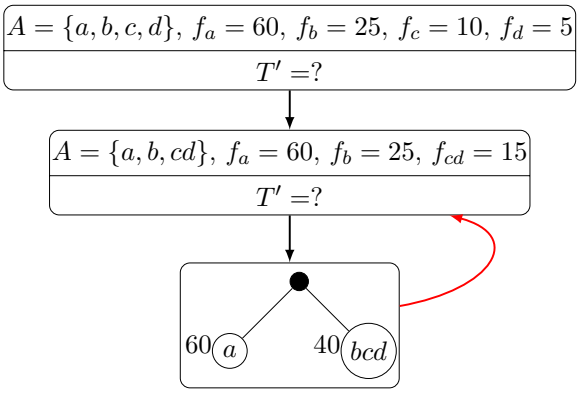
(a) Chamada a HUFFMAN($\{a, b, c, d\}, f$): $|A| > 2$, junta os símbolos c e d.



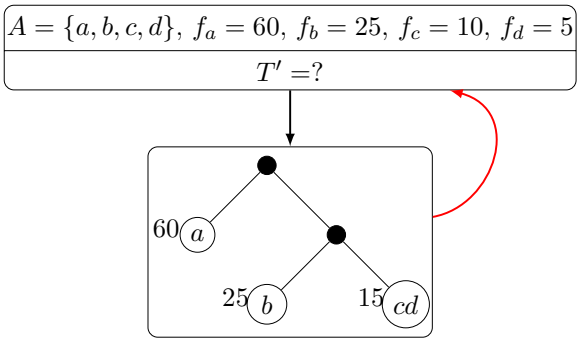
(b) Chamada a HUFFMAN($\{a, b, cd\}, f$): $|A| > 2$, junta os símbolos b e cd.



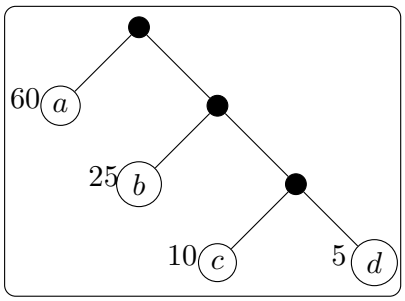
(c) Chamada a HUFFMAN($\{a, bcd\}, f$): $|A| = 2$.



(d) Devolve árvore com dois nós.



(e) Separa a folha rotulada em folhas irmãs.



(f) Separa a folha rotulada em folhas irmãs.

Figura 21.5: Exemplo de execução de HUFFMAN($\{a, b, c, d\}, f$), com $f_a = 60$, $f_b = 25$, $f_c = 10$ e $f_d = 5$. O custo final da árvore é $c(T) = f_a + 2f_b + 3f_c + 3f_d$.

Lema 21.7

Dado um alfabeto A onde cada $i \in A$ possui uma frequência f_i, HUFFMAN(A, f) devolve uma solução ótima para o problema da Compressão de dados.

Demonstração. Perceba que árvore binária T devolvida pelo algoritmo possui apenas folhas rotuladas com símbolos de A. Vamos mostrar por indução em $n = |A|$ que $c(T)$ é mínimo.

Quando $n = 2$, a árvore construída pelo algoritmo é claramente ótima. Suponha que o algoritmo constrói uma árvore ótima para qualquer alfabeto de tamanho menor do que n, dadas as frequências dos símbolos.

Seja $n > 2$ e A um alfabeto com n símbolos. Sejam $i, j \in A$ os dois símbolos de menor frequência em A. Construa A' a partir de A substituindo ambos i e j por um novo símbolo ij e defina a frequência desse novo símbolo como sendo $f_{ij} = f_i + f_j$.

Note que existe uma bijeção entre "árvores cujas folhas são rotuladas com símbolos de A'" e "árvores cujas folhas são rotuladas com símbolos de A onde i e j são irmãos". Vamos chamar o conjunto de árvores desse último tipo de $\mathcal{T}_{i,j}$. Seja $\hat{T}'$ uma árvore cujas folhas são rotuladas com símbolos de A' e seja $\hat{T}$ uma árvore de $\mathcal{T}_{i,j}$. Por definição,

$$c(\hat{T}) = \sum_{k \in A \setminus \{i,j\}} f_k d_{\hat{T}}(k) + f_i d_{\hat{T}}(i) + f_j d_{\hat{T}}(j)\,, \text{ e}$$

$$c(\hat{T}') = \sum_{k \in A' \setminus \{ij\}} f_k d_{\hat{T}'}(k) + f_{ij} d_{\hat{T}'}(ij)\,.$$

Como $A \setminus \{i, j\} = A' \setminus \{ij\}$, temos que

$$c(\hat{T}) - c(\hat{T}') = f_i d_{\hat{T}}(i) + f_j d_{\hat{T}}(j) - f_{ij} d_{\hat{T}'}(ij)\,.$$

Além disso, $d_{\hat{T}}(i) = d_{\hat{T}}(j) = d_{\hat{T}'}(ij) + 1$ e $f_{ij} = f_i + f_j$, por construção. Então temos $c(\hat{T}) - c(\hat{T}') = f_i + f_j$, o que independe do formato das árvores.

Agora note que, por hipótese de indução, o algoritmo encontra uma árvore T' que é ótima para A' (isto é, minimiza $c(T')$ dentre todas as árvores para A'). Então diretamente pela observação acima, a árvore correspondente T construída para A é ótima *dentre as árvores contidas em* $\mathcal{T}_{i,j}$. Com isso, basta mostrar que existe uma árvore ótima para A (dentre todas as árvores para A) que está contida em $\mathcal{T}_{i,j}$ para provar que T é de fato ótima para A.

Seja T^* qualquer árvore ótima para A e sejam x e y nós irmãos no maior nível de T^*. Crie uma árvore $\bar{T}$ a partir de T^* trocando os rótulos de x com i e de y com j. Claramente,

$\bar{T} \in \mathcal{T}_{i,j}$. Seja $B = A \setminus \{x, y, i, j\}$. Temos, por definição,

$$c(T^*) = \sum_{k \in B} f_k d_{T^*}(k) + f_x d_{T^*}(x) + f_y d_{T^*}(y) + f_i d_{T^*}(i) + f_j d_{T^*}(j)\,, \text{ e}$$

$$c(\bar{T}) = \sum_{k \in B} f_k d_{T^*}(k) + f_x d_{T^*}(i) + f_y d_{T^*}(j) + f_i d_{T^*}(x) + f_j d_{T^*}(y)\,.$$

Assim,

$$\begin{aligned} c(T^*) - c(\bar{T}) &= f_x(d_{T^*}(x) - d_{T^*}(i)) + f_y(d_{T^*}(y) - d_{T^*}(j)) \\ &\quad + f_i(d_{T^*}(i) - d_{T^*}(x)) + f_j(d_{T^*}(j) - d_{T^*}(y)) \\ &= (f_x - f_i)(d_{T^*}(x) - d_{T^*}(i)) + (f_y - f_j)(d_{T^*}(y) - d_{T^*}(j))\,. \end{aligned}$$

Pela nossa escolha, $d_{T^*}(x) \geq d_{T^*}(i)$, $d_{T^*}(y) \geq d_{T^*}(j)$, $f_i \leq f_x$ e $f_j \leq f_y$. Então, $c(T^*) - c(\bar{T}) \geq 0$, isto é, $c(T^*) \geq c(\bar{T})$, o que só pode significar que $\bar{T}$ também é ótima. $\square$

Capítulo 22

Programação dinâmica

> "Dynamic programming is a fancy name for divide-and-conquer with a table."
>
> *Ian Parberry* — Problems on Algorithms, 1995.

Programação dinâmica é uma importante técnica de construção de algoritmos, utilizada em problemas cujas soluções podem ser modeladas de forma recursiva. Assim, como na divisão e conquista, um problema gera subproblemas que serão resolvidos recursivamente. Porém, quando a solução de um subproblema precisa ser utilizada várias vezes em um algoritmo de divisão e conquista, a programação dinâmica pode ser uma eficiente alternativa no desenvolvimento de um algoritmo para o problema. Isso porque a característica mais marcante da programação dinâmica é *evitar resolver o mesmo subproblema diversas vezes*. Para isso, os algoritmos fazem uso de memória extra para armazenar as soluções dos subproblemas. Nos referimos genericamente à estrutura utilizada como *tabela* mas, em geral, vetores e matrizes são utilizados.

Algoritmos de programação dinâmica podem ser implementados de duas formas, que são *top-down* (também chamada de memoização) e *bottom-up*.

Na abordagem *top-down*, o algoritmo é desenvolvido de forma recursiva natural, com a diferença que, sempre que um subproblema for resolvido, o resultado é salvo na tabela. Assim, sempre que o algoritmo precisar da solução de um subproblema, ele consulta a tabela antes de fazer a chamada recursiva para resolvê-lo. Em geral, algoritmos top-down são compostos por dois procedimentos, um que faz uma inicialização de variáveis e prepara a tabela, e outro procedimento que compõe o análogo a um algoritmo recursivo natural para o problema.

Na abordagem *bottom-up*, o algoritmo é desenvolvido de forma iterativa, e resolvemos os subproblemas do tamanho menor para o maior, salvando os resultados na tabela. Assim, temos a garantia que ao resolver um problema de determinado tamanho, todos os subproblemas menores necessários já foram resolvidos. Essa abordagem dispensa verificar na tabela se um subproblema já foi resolvido, dado que temos a certeza que isso já aconteceu.

Em geral as duas abordagens fornecem algoritmos com mesmo tempo de execução assintótico. Algoritmos bottom-up são geralmente mais rápidos por conta de sua implementação direta, sem que diversas chamadas recursivas sejam realizadas, como no caso de algoritmos top-down. Por outro lado, é possível que a abordagem top-down seja assintoticamente mais eficiente no caso onde vários subproblemas não precisam ser resolvidos. Um algoritmo bottom-up resolveria todos os subproblemas, mesmo os desnecessários, diferentemente do algoritmo top-down, que resolve somente os subproblemas necessários.

Neste capítulo veremos diversos algoritmos que utilizam a técnica de programação dinâmica e mostraremos as duas implementações para cada um. Também usam programação dinâmica alguns algoritmos clássicos em grafos como Bellman-Ford (Seção 27.1.2) e Floyd-Warshall (Seção 27.2.1).

22.1 Sequência de Fibonacci

A sequência 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ... é conhecida como *sequência de Fibonacci*. Por definição, o n-ésimo número da sequência, escrito como F_n, é dado por

$$F_n = \begin{cases} 1 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ F_{n-1} + F_{n-2} & \text{se } n > 2\,. \end{cases} \tag{22.1}$$

Introduzimos na Seção 7.5 o problema do Número de Fibonacci e apresentamos algoritmos para o mesmo. Repetiremos alguns trechos daquela discussão aqui, por conveniência.

Problema 22.1: *Número de Fibonacci*

Dado um inteiro $n \geq 0$, encontrar F_n.

Pela definição de F_n, o Algoritmo 22.1, recursivo, segue de forma natural.

No entanto, o algoritmo FibonacciRecursivo é extremamente ineficiente. De fato, muito trabalho repetido é feito, pois subproblemas são resolvidos recursivamente diversas vezes. A Figura 22.1 mostra como alguns subproblemas são resolvidos várias vezes em uma

Algoritmo 22.1: FibonacciRecursivo(n)

```
se n ≤ 2 então
    devolve 1
devolve FibonacciRecursivo(n − 1) + FibonacciRecursivo(n − 2)
```

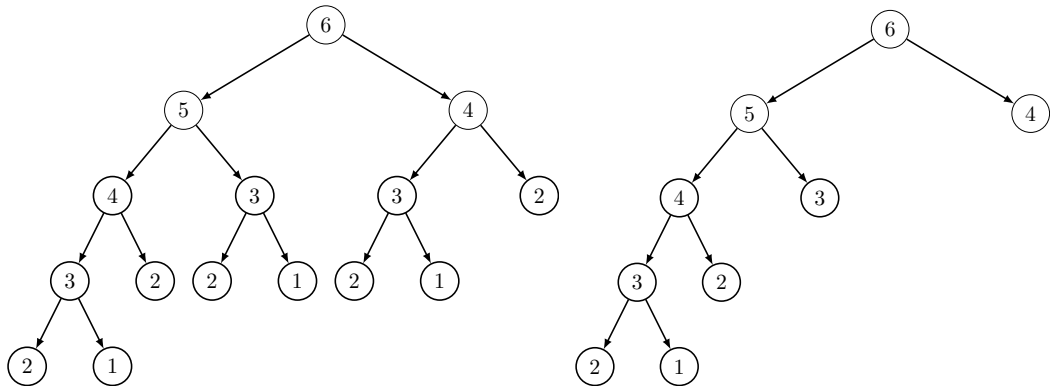


Figura 22.1: Árvore de recursão completa de FibonacciRecursivo(6) à esquerda (Algoritmo 22.1) e de FibonacciRecursivo-TopDown(6) à direita (Algoritmo 22.2). Cada nó representa uma chamada ao algoritmo e é rotulado com o tamanho do problema correspondente.

chamada a FibonacciRecursivo(6).

O tempo de execução $T(n)$ de FibonacciRecursivo(n) pode ser descrito por $T(n-1)+T(n-2)+1 \leq T(n) \leq T(n-1)+T(n-2)+n$, pois uma operação de soma entre dois números grandes leva tempo proporcional à quantidade de bits usados para armazená-los. Podemos usar o método da substituição para mostrar que $T(n)$ é $\Omega\big(\big((1+\sqrt{5})/2\big)^n\big)$. Para ficar claro de onde tiramos o valor $\big((1+\sqrt{5})/2\big)^n$, vamos provar que $T(n) \geq x^n$ para algum $x \geq 1$ de modo que vamos verificar qual o maior valor de x que conseguimos obter. Seja $T(1) = 1$ e $T(2) = 3$. Vamos provar o resultado para todo $n \geq 2$. Assim, temos que $T(2) \geq x^2$ para todo $x \geq \sqrt{3} \approx 1,732$.

Suponha que $T(m) \geq x^m$ para todo $2 \leq m \leq n-1$. Assim, aplicando isso a $T(n)$ temos

$$T(n) \geq T(n-1) + T(n-2) + 1 \geq x^{n-1} + x^{n-2} \geq x^{n-2}(1+x)\,.$$

Note que $1 + x \geq x^2$ sempre que $(1-\sqrt{5})/2 \leq x \leq (1+\sqrt{5})/2$. Portanto, fazendo $x =$

$(1+\sqrt{5})/2$ e substituindo em $T(n)$, obtemos

$$\begin{aligned}
T(n) &\geq \left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\left(1+\left(\frac{1+\sqrt{5}}{2}\right)\right) \\
&\geq \left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\left(\frac{1+\sqrt{5}}{2}\right)^{2} \\
&= \left(\frac{1+\sqrt{5}}{2}\right)^{n} \\
&\approx (1,618)^n\,.
\end{aligned}$$

Portanto, acabamos de provar que o algoritmo FibonacciRecursivo é de fato muito ineficiente, tendo tempo de execução $T(n) = \Omega\big((1,618)^n\big)$.

Mas como podemos evitar que o algoritmo repita trabalho já realizado? Uma forma possível é salvar o valor da solução de um subproblema em uma tabela na primeira vez que ele for calculado. Assim, sempre que precisarmos desse valor, a tabela é consultada antes de resolver o subproblema novamente. Não é difícil perceber que existem apenas n subproblemas diferentes. Pelas árvores de recursão na Figura 22.1, vemos que um subproblema é totalmente descrito por i, onde $1 \leq i \leq n$. O Algoritmo 22.2 é uma variação de FibonacciRecursivo onde, cada vez que um subproblema é resolvido, o valor é salvo no vetor F de tamanho n. Ele foi escrito usando a abordagem top-down.

Algoritmo 22.2: Fibonacci-TopDown(n)

```
1 Cria vetor F[1..n] global
2 F[1] = 1
3 F[2] = 1
4 para i = 3 até n, incrementando faça
5     F[i] = −1
6 devolve FibonacciRecursivo-TopDown(n)
```

Algoritmo 22.3: FibonacciRecursivo-TopDown(n)

```
1 se F[n] == −1 então
2     F[n] = FibonacciRecursivo-TopDown(n − 1) +
        FibonacciRecursivo-TopDown(n − 2)
3 devolve F[n]
```

O algoritmo FIBONACCI-TOPDOWN inicializa o vetor $F[1..n]$ com valores que indicam que ainda não houve cálculo de nenhum subproblema, no caso, com -1. Feito isso, o procedimento FIBONACCIRECURSIVO-TOPDOWN é chamado para calcular $F[n]$. Note que FIBONACCIRECURSIVO-TOPDOWN tem a mesma estrutura do algoritmo recursivo natural FIBONACCIRECURSIVO, com a diferença que em FIBONACCIRECURSIVO-TOPDOWN é realizada uma verificação em F antes de tentar resolver $F[n]$.

Perceba que cada um dos subproblemas é resolvido somente uma vez durante a execução de FIBONACCIRECURSIVO-TOPDOWN, que todas as operações realizadas levam tempo constante, e que existem n subproblemas (calcular F_1, F_2, ..., F_n). Assim, o tempo de execução de FIBONACCI-TOPDOWN é claramente $\Theta(n)$. Isso também pode ser observado pela árvore de recursão na Figura 22.1.

Note que na execução de FIBONACCIRECURSIVO-TOPDOWN(n), várias chamadas recursivas ficam "em espera" até que se chegue ao caso base para que só então os valores comecem a ser devolvidos. Assim, poderíamos escrever um algoritmo não recursivo que já começa calculando o caso base e segue calculando os subproblemas que precisam do caso base, e então os subproblemas que precisam destes, e assim por diante. Dessa forma, não é preciso verificar se os valores necessários já foram calculados, pois temos a certeza que isso já aconteceu. Para isso, podemos inicializar o vetor F nas posições referentes aos casos base do algoritmo recursivo, que nesse caso são as posições 1 e 2. O Algoritmo 22.4 formaliza essa ideia, da abordagem bottom-up.

Algoritmo 22.4: FIBONACCI-BOTTOMUP(n)

```
1 se n ≤ 2 então
2     devolve 1
3 Seja F[1..n] um vetor de tamanho n
4 F[1] = 1
5 F[2] = 1
6 para i = 3 até n, incrementando faça
7     F[i] = F[i − 1] + F[i − 2]
8 devolve F[n]
```

22.2 Corte de barras de ferro

Imagine que uma empresa corta e vende pedaços de barras de ferro. As barras são vendidas em pedaços de tamanho inteiro, onde uma barra de tamanho i tem preço de venda p_i. Por

alguma razão, barras de tamanho menor podem ter um preço maior que barras maiores. A empresa deseja cortar uma grande barra de tamanho inteiro e vender os pedaços de modo a maximizar o lucro obtido.

Problema 22.2: ***Corte de barras de ferro***

Sejam $p_1, \ldots, p_n$ inteiros positivos que correspondem, respectivamente, ao preço de venda de barras de tamanho $1, \ldots, n$. Dado um inteiro positivo n, encontrar o maior lucro obtido com a venda de uma barra de tamanho n, cujos tamanhos dos pedaços vendidos precisam ser inteiros entre 1 e n.

Considere uma barra de tamanho 6 com preços dos pedaços dados por:

p_1	p_2	p_3	p_4	p_5	p_6
1	3	11	16	19	10

Temos várias possibilidades de cortá-la e vender os pedaços. Por exemplo, se a barra for vendida com seis cortes de tamanho 1, então temos lucro $6p_1 = 6$. Caso cortemos três pedaços de tamanho 2, então temos lucro $3p_2 = 9$. Nada disso ainda é melhor do que não cortar a barra, o que nos dá lucro $p_6 = 10$. Outra possibilidade é cortar um pedaço de tamanho 1, outro de tamanho 2 e outro de tamanho 3, e nosso lucro será $p_1 + p_2 + p_3 = 15$. Caso cortemos um pedaço de tamanho 5, o que aparentemente é uma boa opção pois p_5 é o maior valor ali, então a única possibilidade é vender essa parte de tamanho 5 e uma outra de tamanho 1, e isso fornece um lucro de $p_5 + p_1 = 20$. Caso efetuemos um corte de tamanho 4, poderíamos cortar o restante em duas partes de tamanho 1, mas isso seria pior do que vender esse pedaço de tamanho 2, então aqui obteríamos lucro $p_4 + p_2 = 19$. Essa solução certamente não é ótima, pois a anterior já tem lucro maior. Outra opção ainda seria vendermos dois pedaços de tamanho 3, obtendo lucro total de $2p_3 = 22$. De todas as possibilidades, queremos a que permita o maior lucro possível que, nesse caso, é de fato 22.

Veja que é relativamente fácil resolver esse problema: basta enumerar todas as formas possíveis de cortar a barra, calcular o custo de cada forma e guardar o melhor valor possível. No entanto, existem 2^{n-1} formas diferentes de cortar uma barra de tamanho n pois, para cada ponto que está à distância i do extremo da barra, com $1 \leq i \leq n-1$, temos a opção de cortar ali ou não. Além disso, para cada forma diferente de cortar a barra, levamos tempo $O(n)$ para calcular seu custo. Ou seja, esse algoritmo leva tempo $O(n2^n)$ para encontrar uma solução ótima para o problema.

Um algoritmo que enumera todas as possibilidades de solução, testa sua viabilidade e calcula seu custo é chamado de *algoritmo de força bruta*. Eles utilizam muito esforço com-

putacional para encontrar uma solução e ignoram quaisquer estruturas combinatórias do problema.

Certamente é possível criar algoritmos gulosos que nos devolvam soluções viáveis. Por exemplo, podemos fazer uma abordagem gulosa que sempre escolhe cortar em pedaços de maior valor p_i. Uma outra abordagem pode ser uma parecida com a utilizada para resolver o problema da Mochila Fracionária (dada na Seção 21.2), isto é, de sempre escolher cortar pedaços cuja razão p_i/i seja a maior possível. Infelizmente, ambas não rendem algoritmos ótimos. No exemplo dado acima, a primeira abordagem daria a solução de custo $p_5 + p_1$ e a segunda daria a solução de custo $p_4 + p_2$. Dado que a barra tem tamanho 6, não podemos cortá-la em mais de um pedaço de tamanho 4, cuja razão p_i/i é a maior possível. E após cortar um pedaço de tamanho 4, não é possível cortar a barra restante, de tamanho 2, em pedaços de tamanho 5 (cuja razão p_i/i é a segunda maior possível).

Infelizmente, nenhuma abordagem gulosa funcionaria para resolver o problema do corte de barras de forma ótima. Vamos analisar o problema de forma mais estrutural.

Note que ao escolhermos cortar um pedaço de tamanho i da barra, com $1 \leq i \leq n$, então temos uma barra de tamanho $n - i$ restante. Ou seja, reduzimos o tamanho do problema de uma barra de tamanho n para uma de tamanho $n - i$ (quando $i \neq n$). Podemos então utilizar uma abordagem recursiva: se $i \neq n$, então resolva recursivamente a barra de tamanho $n - i$ (essa barra será cortada em pedaços também) e depois combine a solução devolvida com o pedaço i já cortado, para criar uma solução para a barra de tamanho n. Veja que um caso base simples aqui seria quando temos uma barra de tamanho 0 em mãos, da qual não conseguimos obter nenhum lucro.

A questão que fica da abordagem acima é: qual pedaço i escolher? Poderíamos fazer uma escolha gulosa, pelo pedaço de maior p_i, por exemplo. Já vimos anteriormente que essa escolha não levaria à solução ótima sempre. Mas visto que temos apenas n tamanhos diferentes para o pedaço i, podemos simplesmente tentar todos esses tamanhos possíveis e, dentre esses, escolher o que dê o maior lucro quando combinado com a solução recursiva para $n - i$. Esse algoritmo recursivo é mostrado no Algoritmo 22.5.

Pelo funcionamento do algoritmo CorteBarras, podemos ver que ele testa todas as possibilidades de cortes que a barra de tamanho n poderia ter. Veja, por exemplo, sua árvore de execução na Figura 22.2 para uma barra de tamanho 6. Qualquer sequência de cortes possível nessa barra está representada em algum caminho que vai da raiz da árvore até a folha. De fato, perceba que se L_k é o maior lucro obtido ao cortar uma barra de tamanho k, vale que

$$L_k = \max_{1 \leq i \leq k} \{p_i + L_{k-i}\}. \tag{22.2}$$

Algoritmo 22.5: CorteBarras(n, p)

```
se n == 0 então
    devolve 0
lucro = −1
para i = 1 até n, incrementando faça
    valor = p_i + CorteBarras(n − i, p)
    se valor > lucro então
        lucro = valor
devolve lucro
```

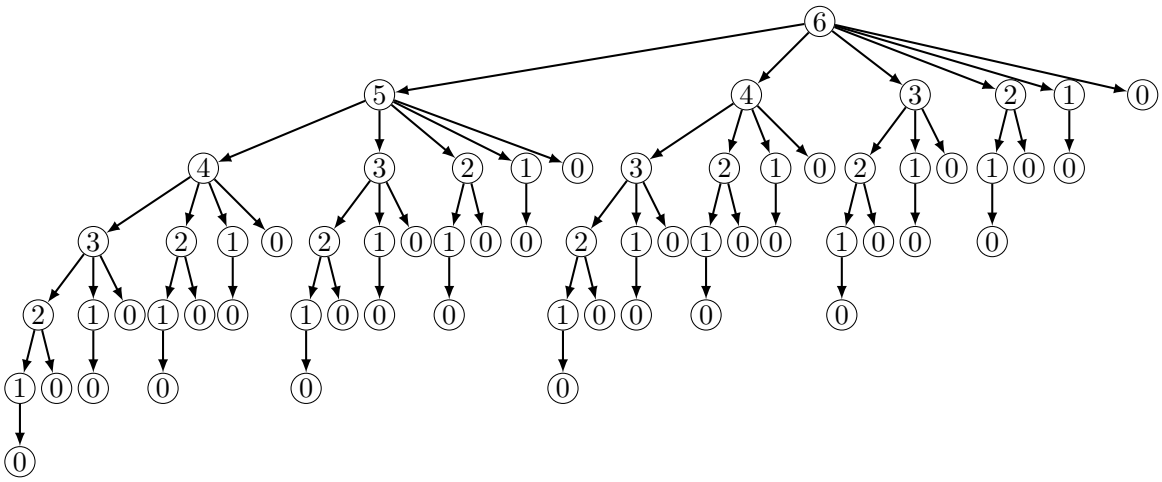


Figura 22.2: Árvore de recursão completa de CorteBarras(6, p) (Algoritmo 22.5). Cada nó representa uma chamada ao algoritmo e é rotulado com o tamanho do problema correspondente.

Isso é verdade porque uma solução ótima para uma barra de tamanho k contém soluções ótimas para barras menores. Considere uma solução S para uma barra de tamanho k e seja $j \in S$ um dos pedaços que existem nessa solução. Perceba que $S \setminus \{j\}$ é uma solução viável para a barra de tamanho $k - j$. Note que $S \setminus \{j\}$ é na verdade ótima para $k - j$, pois se houvesse uma solução melhor S' para $k - j$, poderíamos usar $S' \cup \{j\}$ como solução para k e melhorar nossa solução, o que seria uma contradição com a escolha de S. Isso tudo significa que esse algoritmo de fato devolve uma solução que é ótima.

Outra observação importante que conseguimos fazer nessa árvore é a repetição do cálculo de vários subproblemas. De fato, seja $T(n)$ o tempo de execução de CorteBarras(n, p). Claramente, $T(0) = 1$ e $T(n) = 1 + \sum_{i=1}^{n} T(n - i)$. Vamos utilizar o método da substituição

para provar que $T(n) \geq 2^n$. Claramente temos $T(0) = 1 = 2^0$. Suponha que $T(m) \geq 2^m$ para todo $0 \leq m \leq n-1$. Por definição de $T(n)$,

$$T(n) = 1 + T(0) + T(1) + \cdots + T(n-1) \geq 1 + (2^0 + 2^1 + \cdots + 2^{n-1}) = 2^n\,.$$

Isto é, o tempo de execução continua tão ruim quanto o do algoritmo de força bruta.

Agora note que existem apenas $n+1$ subproblemas diferentes. Isso pode ser visto na árvore de recursão da Figura 22.2, onde cada subproblema é totalmente descrito por i e $0 \leq i \leq n$. Podemos então, com programação dinâmica, utilizar um vetor simples para armazenar os valores de cada um desses subproblemas e acessar o valor diretamente quando necessário. O Algoritmo 22.6 é uma variação de CorteBarras que, cada vez que um subproblema é resolvido, o valor é salvo em um vetor B. Ele foi escrito com a abordagem top-down. O algoritmo também mantém um vetor S tal que $S[j]$ contém o primeiro lugar onde deve-se efetuar um corte em uma barra de tamanho j.

Algoritmo 22.6: CorteBarras-TopDown(n, p)

```
1 Cria vetores B[0..n] e S[0..n] globais
2 B[0] = 0
3 para i = 1 até n, incrementando faça
4     B[i] = −1
5 devolve CorteBarrasRecursivo-TopDown(n, p)
```

Algoritmo 22.7: CorteBarrasRecursivo-TopDown(k, p)

```
1 se B[k] == −1 então
2     lucro = −1
3     para i = 1 até k, incrementando faça
4         valor = p_i + CorteBarrasRecursivo-TopDown(k − i, p)
5         se valor > lucro então
6             lucro = valor
7             S[k] = i
8     B[k] = lucro
9 devolve B[k]
```

O algoritmo CorteBarras-TopDown(n, p) cria os vetores B e S, inicializa $B[0]$ com 0 e as entradas restantes de B com -1, representando que ainda não calculamos esses valores.

Feito isso, CorteBarrasRecursivo-TopDown(n, p) é executado.

O primeiro passo do algoritmo CorteBarrasRecursivo-TopDown(k, p) é verificar se o subproblema em questão já foi resolvido (linha 1). Caso o subproblema não tenha sido resolvido, então o algoritmo vai fazer isso de modo muito semelhante ao Algoritmo 22.5. A diferença é que agora salvamos o melhor local para fazer o primeiro corte em uma barra de tamanho k em $S[k]$ e o maior lucro obtido em $B[k]$. A linha 9 é executada sempre, seja devolvendo o valor que já havia em $B[k]$ (quando o teste da linha 1 falha), ou devolvendo o valor recém calculado (linha 8).

Vamos analisar agora o tempo de execução de CorteBarras-TopDown(n, p) que tem, assintoticamente, o mesmo tempo de execução de CorteBarrasRecursivo-TopDown(n, p). Note que cada chamada recursiva de CorteBarrasRecursivo-TopDown a um subproblema que já foi resolvido retorna imediatamente, e todas as linhas são executadas em tempo constante. Como salvamos o resultado sempre que resolvemos um subproblema, cada subproblema é resolvido somente uma vez. Na chamada recursiva em que resolvemos um subproblema de tamanho k (para $1 \leq k \leq n$), o laço **para** da linha 3 é executado k vezes. Assim, como existem subproblemas de tamanho $0, 1, \ldots, n-1$, o tempo de execução $T(n)$ de CorteBarrasRecursivo-TopDown(n, p) é assintoticamente dado por

$$T(n) = 1 + 2 + \cdots + n = \Theta(n^2)\,.$$

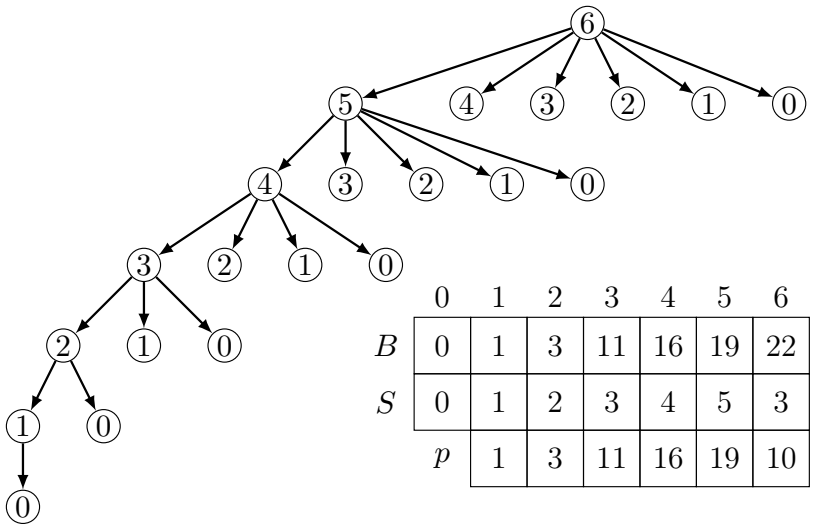
De fato, isso também pode ser observado em sua árvore de recursão, na Figura 22.3.

Acontece que o algoritmo apenas devolve o lucro obtido pelos cortes da barra. Caso precisemos de fato construir uma solução (descobrir o tamanho dos pedaços em que a barra foi cortada), podemos utilizar o vetor S. Veja que para cortar uma barra de tamanho n e obter seu lucro máximo $B[n]$, cortamos um pedaço $S[n]$ da mesma, o que significa que sobrou um pedaço de tamanho $n - S[n]$. Para cortar essa barra de tamanho $n - S[n]$ e obter seu lucro máximo $B[n - S[n]]$, cortamos um pedaço $S[n - S[n]]$ da mesma. Essa ideia é sucessivamente repetida até que tenhamos uma barra de tamanho 0. O procedimento é formalizado no Algoritmo 22.8.

Algoritmo 22.8: ImprimeCortes(n, S)

```
1 enquanto n > 0 faça
2     Imprime S[n]
3     n = n − S[n]
```

Note que na execução de CorteBarrasRecursivo-TopDown(n, p), várias chamadas recursivas ficam "em espera" até que se chegue ao caso base para que só então os valores



	0	1	2	3	4	5	6
B	0	1	3	11	16	19	22
S	0	1	2	3	4	5	3
p	1	3	11	16	19	10	

Figura 22.3: Árvore de execução completa de CorteBarrasRecursivo-TopDown(6, p) (Algoritmo 22.6). Cada nó representa uma chamada ao algoritmo e é rotulado com o tamanho do problema correspondente.

comecem a ser devolvidos. Assim, poderíamos escrever um algoritmo não recursivo que já começa calculando o caso base e segue calculando os subproblemas que precisam do caso base, e então os subproblemas que precisam destes, e assim por diante. Dessa forma, não é preciso verificar se os valores necessários já foram calculados, pois temos a certeza que isso já aconteceu. Para isso, podemos inicializar o vetor B nas posições referentes aos casos base do algoritmo recursivo, que nesse caso é a posição 0. O Algoritmo 22.9 formaliza essa ideia, da abordagem bottom-up.

Veja na Figura 22.4 um exemplo de execução de ambos algoritmos CorteBarras-TopDown e CorteBarras-BottomUp.

22.3 Mochila inteira

O problema da mochila é um dos clássicos em computação. Nessa seção veremos a versão da mochila inteira. A Seção 21.2 apresenta a versão da mochila fracionária.

Problema 22.3: ***Mochila inteira***

Dado um conjunto $I = \{1, 2, \ldots, n\}$ de n itens onde cada $i \in I$ tem um peso w_i e um valor v_i associados e dada uma mochila com capacidade de peso W, selecionar um subconjunto $S \subseteq I$ de itens tal que $\sum_{i \in S} w_i \leq W$ e $\sum_{i \in S} v_i$ é máximo.

Algoritmo 22.9: CorteBarras-BottomUp(n, p)

```
Cria vetores B[0..n] e S[0..n]
B[0] = 0
para k = 1 até n, incrementando faça
    lucro = −1
    para i = 1 até k, incrementando faça
        valor = p_i + B[k − i]
        se valor > lucro então
            lucro = valor
            S[k] = i
    B[k] = lucro
devolve B[n]
```

	0	1	2	3	4	5	6
B	0						
B	0	1					
B	0	1	3				
B	0	1	3	11			
B	0	1	3	11	16		
B	0	1	3	11	16	19	
B	0	1	3	11	16	19	22

p_1	p_2	p_3	p_4	p_5	p_6
1	3	11	16	19	10

$$B[1] = \max\{p_1 + B[0]\} = \max\{1 + 0\}$$

$$B[2] = \max\{p_1 + B[1], p_2 + B[0]\} = \max\{1 + 1, 3 + 0\}$$

$$B[3] = \max\{p_1 + B[2], p_2 + B[1], p_3 + B[0]\} = \max\{1 + 3, 3 + 1, 11 + 0\}$$

$$B[4] = \max\{p_1 + B[3], p_2 + B[2], p_3 + B[1], p_4 + B[0]\} = \max\{1 + 11, 3 + 3, 11 + 1, 16 + 0\}$$

$$B[5] = \max\{p_1 + B[4], p_2 + B[3], p_3 + B[2], p_4 + B[1], p_5 + B[0]\} = \max\{1 + 16, 3 + 11, 11 + 3, 16 + 1, 19 + 0\}$$

$$B[6] = \max\{p_1 + B[5], p_2 + B[4], p_3 + B[3], p_4 + B[2], p_5 + B[1], p_6 + B[0]\} = \max\{1 + 19, 3 + 16, 11 + 11, 16 + 3, 19 + 1, 10 + 0\}$$

Figura 22.4: Exemplo de execução de CorteBarras-TopDown(6, p) e CorteBarras-BottomUp(6, p), com $p_1 = 1$, $p_2 = 3$, $p_3 = 11$, $p_4 = 16$, $p_5 = 19$ e $p_6 = 10$.

Por exemplo, considere $n = 3$, $v_1 = 60$, $w_1 = 10$, $v_2 = 100$, $w_2 = 20$, $v_3 = 120$, $w_3 = 30$ e $W = 50$. Temos várias possibilidades de escolher itens que caibam nessa mochila. Por exemplo, podemos escolher apenas o item 1, o que dá um peso total de $10 \leq W$ e valor total de 60. Outra possibilidade melhor seria escolher apenas o item 3, o que dá um peso total de $30 \leq W$ e valor total melhor, de 120. Uma opção ainda melhor é escolher ambos itens 1 e 2, dando peso total $30 \leq W$ e valor total 160. A melhor opção de todas no entanto, que é a solução ótima, é escolher os itens 2 e 3, cujo peso total é $50 \leq W$ e valor total 220.

Veja que é relativamente fácil resolver o problema da mochila por força bruta: basta enumerar todos os subconjuntos possíveis de itens, verificar se eles cabem na mochila, calcular o valor total e guardar o melhor possível de todos. No entanto, existem 2^n subconjuntos diferentes de itens pois, para cada item, temos a opção de colocá-lo ou não no subconjunto. Para cada subconjunto, levamos tempo $O(n)$ para checar se os itens cabem na mochila e calcular seu valor total. Ou seja, esse algoritmo leva tempo $O(n2^n)$ e, portanto, não é eficiente.

Para facilitar a discussão a seguir, vamos dizer que uma instância da mochila inteira é dada pelo par (n, W), que indica que temos n itens e capacidade W de mochila e deixa os valores e pesos dos itens escondidos. Podemos tentar uma abordagem recursiva para construir uma solução S para (n, W) da seguinte forma. Escolha um item $i \in I$. Você pode utilizá-lo ou não na sua solução. Se você decidir por não utilizá-lo, então a capacidade da mochila não se altera e você pode usar a recursão para encontrar uma solução S' para $(n-1, W)$. Assim, $S = S'$ é uma solução para (n, W). Se você decidir por utilizá-lo, então a capacidade da mochila reduz de w_i unidades, mas você também pode usar a recursão para encontrar uma solução S' para $(n-1, W-w_i)$ e usar $S = S' \cup \{i\}$ como solução para (n, W). Veja que dois casos bases simples aqui seriam um em que não temos nenhum item para escolher, pois independente do tamanho da mochila não é possível obter nenhum valor, ou um em que não temos nenhuma capacidade de mochila, pois independente de quantos itens tenhamos não é possível escolher nenhum.

Mas qual decisão tomar? Escolhemos o item i ou não? Veja que são apenas duas possibilidades: colocamos i na mochila ou não. Considerando o objetivo do problema, podemos tentar ambas e devolver a melhor opção das duas. Mas qual item i escolher dentre os n disponíveis? Mas note que pela recursividade da estratégia, escolher um item i qualquer apenas o remove da instância da chamada atual, deixando qualquer outro item j como possibilidade de escolha para as próximas chamadas. Por isso, podemos escolher qualquer $i \in I$ que quisermos. Por comodidades que facilitam a implementação, vamos escolher $i = n$. O algoritmo recursivo descrito é formalizado no Algoritmo 22.10.

Veja a Figura 22.5 para um exemplo da árvore de recursão da estratégia mencionada.

Algoritmo 22.10: MochilaInteira(n, v, w, W)

```
se n == 0 ou W == 0 então
    devolve 0
se w_n > W então
    devolve MochilaInteira(n − 1, v, w, W)
senão
    usa = v_n + MochilaInteira(n − 1, v, w, W − w_n)
    naousa = MochilaInteira(n − 1, v, w, W)
    devolve max{usa, naousa}
```

Note como qualquer solução está descrita em algum caminho que vai da raiz até uma folha dessa árvore. De fato, se $V_{j,X}$ é o valor de uma solução ótima para (j, X), então

$$V_{j,X} = \begin{cases} \max\{V_{j-1,X}, V_{j-1,X-w_j} + v_j\} & \text{se } w_j \leq X \\ V_{j-1,X} & \text{se } w_j > X \end{cases} . \tag{22.3}$$

Ademais, $V_{0,X} = V_{j,0} = 0$ para qualquer $0 \leq i \leq n$ e $0 \leq X \leq W$. Isso é verdade porque uma solução ótima para uma mochila (j, X) contém soluções ótimas para mochilas menores (com menos capacidade e/ou com menos itens). Seja $S \subseteq I$ uma solução ótima para (j, X), com $|I| = j$. Se $j \in S$, então note que $S \setminus \{j\}$ é uma solução ótima para $(j-1, X-w_j)$. Se não fosse, então haveria S' ótima para $(j-1, X-w_j)$ tal que $S' \cup \{j\}$ teria valor melhor para (j, X), o que seria uma contradição. Se $j \notin S$, então note que S é ótima para $(j-1, X-w_j)$. Se não fosse, então haveria S' ótima para $(j-1, X)$ tal que S' teria valor melhor para (j, X), o que também seria uma contradição. A expressão acima juntamente com uma prova por indução simples mostra que o algoritmo de fato encontra uma solução ótima.

O tempo de execução $T(n, W)$ de MochilaInteira pode ser descrito pela recorrência $T(n, W) \leq T(n-1, W) + T(n-1, W-w_n) + \Theta(1)$. Essa recorrência certamente tem tempo no máximo o tempo da recorrência $S(m) = 2S(m-1) + 1$, que é $\Theta(2^m)$. Assim, o tempo de MochilaInteira é $O(2^n)$. Também não é difícil perceber que o problema desse algoritmo está no fato de ele realizar as mesmas chamadas recursivas diversas vezes. Veja na Figura 22.5, por exemplo, que se $w_n = w_{n-1} = 1$ e $w_{n-2} = 2$, então existem repetições dos problemas $(n-3, W-1)$, $(n-3, W-2)$ e $(n-3, W-3)$. Na verdade, existem no máximo $(n+1) \times (W+1)$ subproblemas diferentes: um subproblema é totalmente descrito por (j, x), onde $0 \leq j \leq n$ e $0 \leq x \leq W$. Assim, podemos usar uma estrutura de dados para manter seus valores e acessá-los diretamente sempre que necessário ao invés de recalculá-los. Poderíamos utilizar um vetor com $(n+1) \times (W+1)$ entradas, uma para cada subproblema,

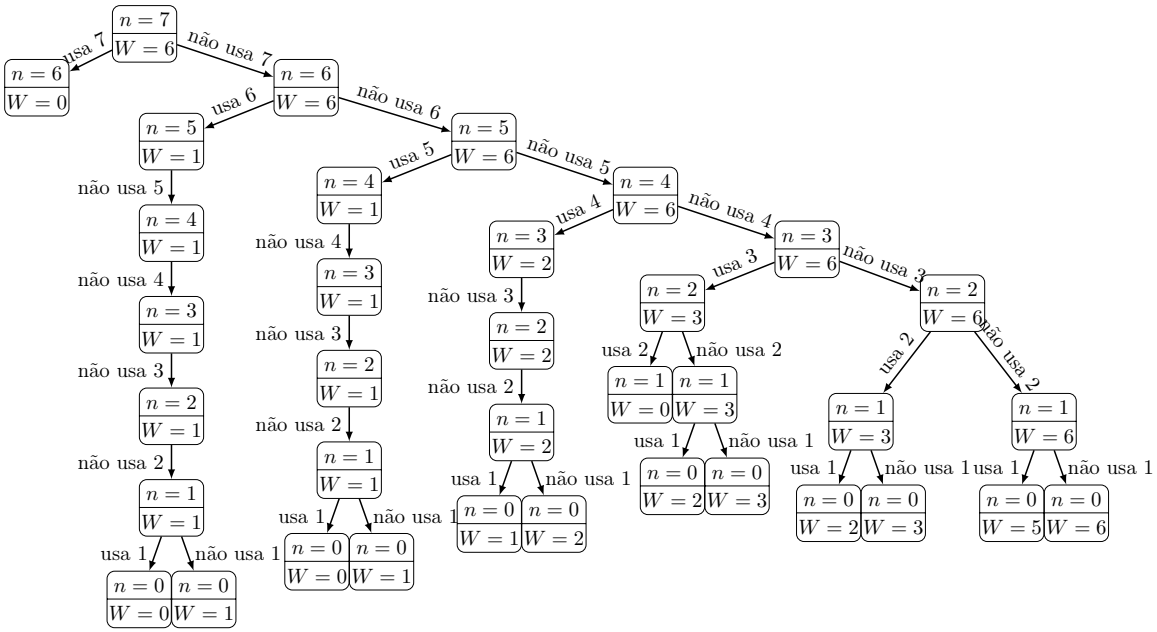


Figura 22.5: Árvore de execução completa de MochilaInteira(7, v, w, 6) (Algoritmo 22.10), com $v_1 = 60$, $w_1 = 1$, $v_2 = 150$, $w_2 = 3$, $v_3 = 120$, $w_3 = 3$, $v_4 = 160$, $w_4 = 4$, $v_5 = 200$, $w_5 = 5$, $v_6 = 150$, $w_6 = 5$, $v_7 = 60$, $w_7 = 6$. Cada nó representa uma chamada ao algoritmo e é rotulado com o tamanho do problema correspondente.

porém utilizar uma matriz de dimensões $(n+1) \times (W+1)$ nos permite um acesso mais intuitivo. Assim, a ideia é armazenar em $M[j][X]$ o valor $V_{j,X}$, de forma que nosso objetivo é calcular $M[n][W]$. O Algoritmo 22.11 formaliza a ideia dessa estratégia de programação dinâmica com a abordagem top-down enquanto que o Algoritmo 22.13 o faz com a abordagem bottom-up.

Algoritmo 22.11: MochilaInteira-TopDown(n, v, w, W)

```
1 Seja M[0..n][0..W] uma matriz global
2 para X = 0 até W, incrementando faça
3     M[0][X] = 0
4     para j = 1 até n, incrementando faça
5         M[j][X] = −1
6         M[j][0] = 0
7 devolve MochilaInteiraRecursivo-TopDown(n, v, w, W)
```

Algoritmo 22.12: MochilaInteiraRecursivo-TopDown(j, v, w, X)

```
1 se M[j][X] == −1 então
2     se w_j > X então
3         M[j][X] = MochilaInteiraRecursivo-TopDown(j − 1, v, w, X)
4     senão
5         usa = v_j + MochilaInteiraRecursivo-TopDown(j − 1, v, w, X − w_j)
6         naousa = MochilaInteiraRecursivo-TopDown(j − 1, v, w, X)
7         M[j][X] = max{usa, naousa}
8 devolve M[j][X]
```

A tabela a seguir mostra o resultado final da matriz M após execução dos algoritmos sobre a instância onde $n = 4$, $W = 7$, $w_1 = 1$, $v_1 = 10$, $w_2 = 3$, $v_2 = 40$, $w_3 = 4$, $v_3 = 50$, $w_4 = 5$ e $v_4 = 70$:

item ↓ \ capacidade →	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1, $v_1 = 10$, $w_1 = 1$	0	10	10	10	10	10	10	10
2, $v_2 = 40$, $w_2 = 3$	0	10	10	40	50	50	50	50
3, $v_3 = 50$, $w_3 = 4$	0	10	10	40	50	60	60	90
4, $v_4 = 70$, $w_4 = 5$	0	10	10	40	50	70	80	90

Algoritmo 22.13: MochilaInteira-BottomUp(n, v, w, W)

```
1  Seja M[0..n][0..W] uma matriz
2  para X = 0 até W, incrementando faça
3      M[0][X] = 0
4  para j = 0 até n, incrementando faça
5      M[j][0] = 0
6  para j = 1 até n, incrementando faça
7      para X = 0 até W, incrementando faça
8          se w_j > X então
9              M[j][X] = M[j − 1][X]
10         senão
11             usa = v_j + M[j − 1][X − w_j]
12             naousa = M[j − 1][X]
13             M[j][X] = max{usa, naousa}
14 devolve M[n][W]
```

Não é difícil perceber que o tempo de execução desses algoritmos de programação dinâmica para o problema da mochila inteira é $\Theta(nW)$. Agora veja que eles não possuem tempo polinomial no tamanho das entradas. O parâmetro W é um número, e seu tamanho é $\log W$, que é a quantidade de bits necessária para armazená-lo. A função nW pode ser escrita como $n2^{\log W}$ e essa sim está em função do tamanho da entrada. Infelizmente, ela é exponencial no tamanho de uma das entradas. Esse algoritmo é o que chamamos de *pseudo-polinomial*. Seu tempo de execução será bom se W for pequeno.

Com relação à solução ótima, sabemos que seu valor é $M[n][W]$, mas não sabemos quais itens a compõem. No entanto, a maneira como cada célula da matriz foi preenchida nos permite descobri-los. Veja o Algoritmo 22.14, que claramente executa em tempo $\Theta(n)$.

22.4 Alinhamento de sequências

Um *alinhamento* de duas sequências de caracteres X e Y é obtido inserindo-se espaços (*gaps*) nas sequências para que elas fiquem com o mesmo tamanho e cada caractere ou *gap* de uma fique emparelhado a um único caractere ou *gap* da outra, contanto que *gaps* não sejam emparelhados com *gaps*. Por exemplo, se $X = AGGGCT$ e $Y = AGGCA$, então dois alinhamentos possíveis são:

Algoritmo 22.14: ConstroiMochila(n, v, w, W, M)

```
1 S = ∅
2 x = W
3 j = n
4 enquanto j ≥ 1 faça
5     se M[j][x] == M[j − 1][x − w_j] + v_j então
6         S = S ∪ {j}
7         x = x − w_j
8     j = j − 1
9 devolve S
```

$$\begin{array}{cccccc} A & G & G & G & C & T \\ A & G & G & - & C & A \end{array} \quad \text{e} \quad \begin{array}{ccccccc} A & G & G & G & C & - & T \\ A & G & G & - & C & A & - \end{array}$$

Dadas duas sequências, várias são as possibilidades de alinhá-las. O primeiro caractere de X pode ser alinhado com um *gap*, ou com o primeiro caractere de Y, ou com o segundo, ou com o sétimo, ou com o último, etc. Assim, é necessário uma forma de comparar os vários alinhamentos e descobrir qual é o melhor deles. Para isso, existe uma função de pontuação α, onde $\alpha(a, b)$ indica a penalidade por alinhar os caracteres a e b e $\alpha(gap)$ indica a penalidade por alinhar um caractere com um *gap*[1]. Assim, se $\alpha(a, b) = -4$ para $a \neq b$, $\alpha(a, a) = 2$ e $\alpha(gap) = -1$, então o alinhamento da esquerda dado acima tem pontuação 3 enquanto que o alinhamento da direita tem pontuação 5.

Problema 22.4: ***Alinhamento de sequências***

Dadas duas sequências X e Y sobre um mesmo alfabeto A, onde $X = x_1x_2 \cdots x_m$, $Y = y_1y_2 \cdots y_n$, $x_i, y_j \in A$ e uma função α de pontuação, encontrar um alinhamento entre X e Y de pontuação máxima.

Uma possível abordagem recursiva para o problema acima é a seguinte. Para reduzir o tamanho da entrada, podemos remover um ou mais caracteres de X e/ou de Y. Escolhamos a opção mais fácil: remover o último caractere delas. Note que x_m pode estar alinhado, ao fim, com um *gap* ou então com qualquer outro caractere de Y. Assim, temos as seguintes possibilidades:

- resolva recursivamente o problema de alinhar $x_1x_2 \cdots x_{m-1}$ com $y_1y_2 \cdots y_{n-1}$ e combine

[1] Existem variações onde caracteres diferentes têm penalidades diferentes ao serem alinhados com *gaps*.

a solução devolvida com o alinhamento de x_m a y_n;

- resolva recursivamente o problema de alinhar $x_1x_2\cdots x_{m-1}$ com $y_1y_2\cdots y_n$ e combine a solução devolvida com o alinhamento de x_m a um *gap*;

- resolva recursivamente o problema de alinhar $x_1x_2\cdots x_m$ com $y_1y_2\cdots y_{n-1}$ e combine a solução devolvida com o alinhamento de y_n a um *gap*.

Veja que essa é uma abordagem recursiva válida, pois estamos sempre reduzindo o tamanho de uma das duas sequências. Note ainda que a chamada recursiva para $x_1x_2\cdots x_m$ e $y_1y_2\cdots y_{n-1}$ tem a possibilidade de alinhar x_m com qualquer outro caractere de Y. Mas qual das três possibilidades escolher? Podemos testar as três e escolher a que dá a melhor pontuação. Nesse caso existem dois casos base, referentes aos casos em que alguma das sequências não possuem nenhum caractere.

Observe que essa abordagem considera todas as possibilidades de alinhamento possíveis entre as duas sequências iniciais X e Y. Veja na Figura 22.6 como qualquer solução possível pode ser descrita por um caminho que vai da raiz a uma folha da árvore de recursão. De fato, se $P_{i,j}$ é a pontuação obtida ao alinhar $x_1x_2\ldots x_i$ com $y_1y_2\ldots y_j$, então

$$P_{i,j} = \max \begin{cases} \alpha(x_i, y_j) + P_{i-1,j-1} \\ \alpha(gap) + P_{i-1,j} \\ \alpha(gap) + P_{i,j-1} \end{cases} . \tag{22.4}$$

Ademais, note que $P_{0,j} = j\alpha(gap)$ e $P_{i,0} = i\alpha(gap)$, pois a única opção é alinhar os caracteres da sequência restante com *gaps*. Isso é verdade pois qualquer solução ótima para alinhar $x_1x_2\ldots x_i$ com $y_1y_2\ldots y_j$ contém alinhamentos ótimos de sequências menores. Seja O um alinhamento ótimo para alinhar $x_1x_2\ldots x_i$ com $y_1y_2\ldots y_j$. A última posição de O tem apenas três possibilidades de preenchimento. Se nela tivermos x_i alinhado com y_j, então O', que é O sem essa posição, deve ser um alinhamento ótimo para $x_1x_2\ldots x_{i-1}$ com $y_1y_2\ldots y_{j-1}$. Se nela tivermos x_i alinhado com *gap*, então O', que é O sem essa posição, deve ser um alinhamento ótimo para $x_1x_2\ldots x_{i-1}$ com $y_1y_2\ldots y_j$. Por fim, se nela tivermos y_j alinhado com *gap*, então O' deve ser um alinhamento ótimo para $x_1x_2\ldots x_i$ com $y_1y_2\ldots y_{j-1}$. A expressão acima juntamente com uma prova por indução simples mostra que esse algoritmo devolve uma solução ótima para o problema.

Também é fácil perceber pela Figura 22.6 que existe muita repetição de subproblemas. De fato, existem no máximo $(m+1)\times(n+1)$ subproblemas diferentes: um subproblema é totalmente descrito por um par (i,j), onde $0 \le i \le m$ e $0 \le j \le n$. Assim, podemos usar uma matriz M de dimensões $(m+1)\times(n+1)$ tal que $M[i][j]$ armazene o valor $P_{i,j}$, de forma que

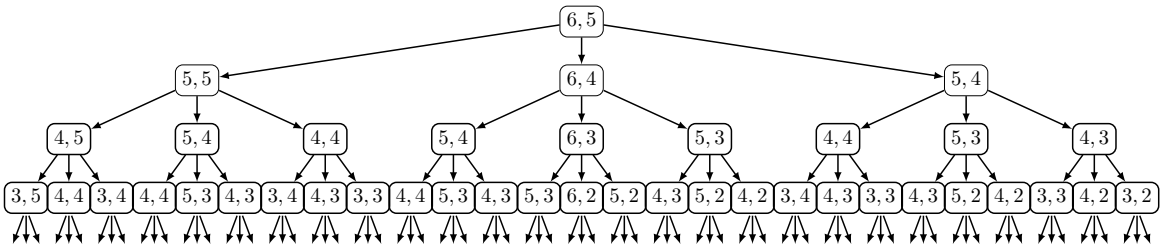


Figura 22.6: Árvore de execução completa do algoritmo recursivo simples para o alinhamento das sequências $X = AGGGCT$ e $Y = AGGCA$ de tamanhos 6 e 5, respectivamente. Cada nó representa uma chamada ao algoritmo e é rotulado com um par m, n, referente ao tamanho do problema correspondente.

nosso objetivo é calcular $M[m][n]$. O Algoritmo 22.15 mostra um algoritmo de programação dinâmica na abordagem bottom-up para o problema do alinhamento de sequências.

Algoritmo 22.15: ALINHAMENTO-BOTTOMUP(X, m, Y, n, α)

1 Seja $M[0..m][0..n]$ uma matriz
2 **para** $i = 0$ *até* m, *incrementando* **faça**
3 $\quad M[i][0] = i \times \alpha(gap)$
4 **para** $j = 0$ *até* n, *incrementando* **faça**
5 $\quad M[0][j] = j \times \alpha(gap)$
6 **para** $i = 1$ *até* m, *incrementando* **faça**
7 $\quad$ **para** $j = 1$ *até* n, *incrementando* **faça**
8 $\quad\quad M[i][j] =$
$\quad\quad \max\{M[i-1][j-1] + \alpha(x_i, y_j), M[i-1][j] + \alpha(gap), M[i][j-1] + \alpha(gap)\}$
9 **devolve** $M[m][n]$

Parte

VI

Algoritmos em grafos

Suponha que haja três casas em um plano (ou superfície de uma esfera) e cada uma precisa ser ligada às empresas de gás, água e eletricidade. O uso de uma terceira dimensão ou o envio de qualquer uma das conexões através de outra empresa ou casa não é permitido. Existe uma maneira de fazer todas os nove ligações sem que qualquer uma das linhas se cruzem?

Não.

Nesta parte

Diversas situações apresentam relacionamentos par-a-par entre objetos, como malhas rodoviárias (duas cidades podem ou não estar ligadas por uma rodovia), redes sociais (duas pessoas podem ou não ser amigas), relações de precedência (uma disciplina pode ou não ser feita antes de outra), hyperlinks na web (um site pode ou não ter *link* para outro), etc. Todas elas podem ser representadas por grafos.

A Teoria de Grafos, que estuda essas estruturas, tem aplicações em diversas áreas do conhecimento, como Bioinformática, Sociologia, Física, Computação e muitas outras, e teve início em 1736 com Leonhard Euler, que resolveu o problema das sete pontes de Königsberg.

Capítulo 23

Conceitos essenciais em grafos

Um *grafo* G é uma tripla (V, E, ψ), onde V é um conjunto de elementos chamados *vértices*[1], E é um conjunto de elementos chamados *arestas*, disjunto de V, e ψ é uma *função de incidência*, que associa uma aresta a um par não ordenado de vértices. Por exemplo, $H = (V, E, \psi)$ em que $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$, $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$, $\psi(e_1) = \{v_5, v_5\}$, $\psi(e_2) = \{v_2, v_3\}$, $\psi(e_3) = \{v_0, v_3\}$, $\psi(e_4) = \{v_4, v_5\}$, $\psi(e_5) = \{v_5, v_1\}$, $\psi(e_6) = \{v_0, v_1\}$, $\psi(e_7) = \{v_0, v_2\}$, $\psi(e_8) = \{v_0, v_3\}$, $\psi(e_9) = \{v_0, v_4\}$ e $\psi(e_{10}) = \{v_3, v_4\}$, é um grafo.

Um *digrafo* D também é uma tripla (V, E, ψ), onde V é um conjunto de vértices, E é um conjunto de *arcos*, disjuntos de V, e ψ é uma *função de incidência*, que associa um arco a um par ordenado de vértices. Por exemplo, $J = (V, E, \psi)$ em que $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$, $E = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}\}$, $\psi(a_1) = (v_5, v_5)$, $\psi(a_2) = (v_2, v_3)$, $\psi(a_3) = (v_0, v_3)$, $\psi(a_4) = (v_4, v_5)$, $\psi(a_5) = (v_5, v_1)$, $\psi(a_6) = (v_0, v_1)$, $\psi(a_7) = (v_0, v_2)$, $\psi(a_8) = (v_0, v_3)$, $\psi(a_9) = (v_0, v_4)$ e $\psi(a_{10}) = (v_3, v_4)$, é um digrafo.

Grafos e digrafos possuem esse nome por permitirem uma representação gráfica. Círculos representam vértices, uma aresta e é representada por uma linha que liga os círculos que representam os vértices x e y se $\psi(e) = \{x, y\}$ e um arco a é representado por uma seta direcionada que liga os círculos que representam os vértices x e y, nessa ordem, se $\psi(a) = (x, y)$. Veja a Figura 23.1, que representa os grafos H e J definidos acima.

Dado um (di)grafo $K = (A, B, \varphi)$, denotamos o conjunto de vértices de K por $V(K)$, o conjunto de arestas ou arcos de K por $E(K)$ e a função de incidência de K por ψ_K, isto é, $V(K) = A$, $E(K) = B$ e $\psi_K = \varphi$. Com essa notação, podemos agora definir um

[1] Alguns materiais também chamam vértices de nós. Evitaremos essa nomenclatura, utilizando o termo nós apenas quando nos referimos a estruturas de dados, como por exemplo listas ligadas ou árvores binárias de busca.

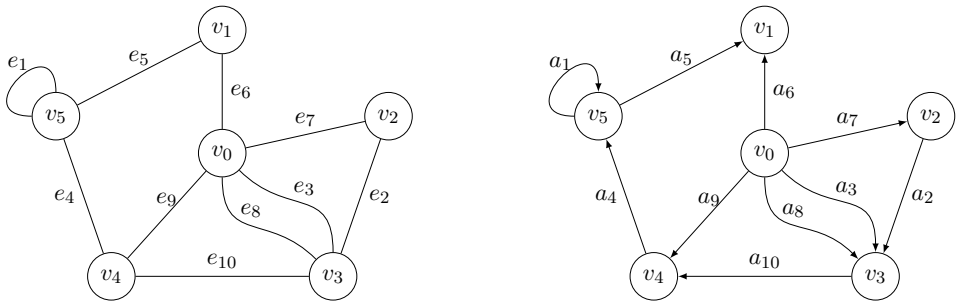


Figura 23.1: Representação gráfica de um grafo à esquerda e um digrafo à direita.

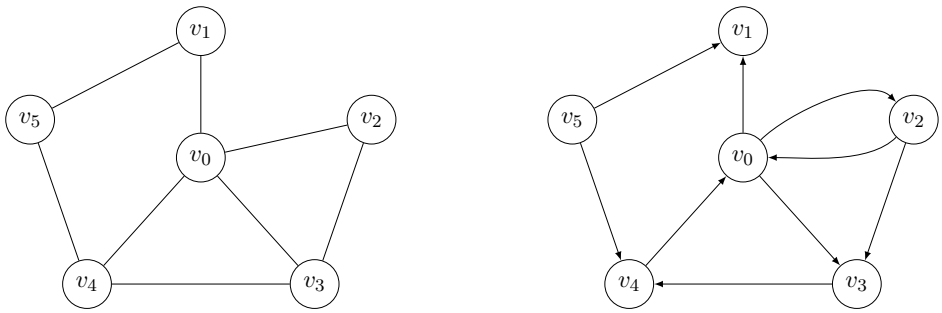


Figura 23.2: Exemplos de grafo e digrafo simples.

(di)grafo sem precisar nomear os elementos da tripla. Por simplicidade, escrevemos $v(G)$ e $e(G)$, respectivamente, no lugar de $|V(G)|$ e $|E(G)|$.

No que segue, seja G um (di)grafo qualquer. A *ordem* de G é a quantidade de vértices de G e o *tamanho* de G é a quantidade total de vértices e arestas (arcos) de G, i.e., é dado por $|V(G)| + |E(G)|$. Duas arestas (arcos) e e f são *paralelas* ou *múltiplas* se $\psi_G(e) = \psi_G(f)$. Uma aresta (arco) e é um *laço* se $\psi_G(e) = \{x, x\}$ ($\psi_G(e) = (x, x)$) para algum $x \in V(G)$. Grafos e digrafos *simples* são aqueles que não possuem laços nem arestas (arcos) paralelas.

A partir de agora, os termos grafo e digrafo se referem exclusivamente a grafo simples e digrafo simples. Os termos multigrafos e multidigrafos serão utilizados caso seja necessário nos referir a estruturas que permitem laços e arestas ou arcos paralelos.

No que segue, seja G um (di)grafo. Note que uma aresta ou arco podem ser unicamente determinados pelos seus extremos. Assim, ψ_G pode ser definida implicitamente fazendo com que $E(G)$ seja um conjunto de pares não ordenados ou pares ordenados de vértices. Por exemplo, H em que $V(H) = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ e $E(H) = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_0, v_3\}, \{v_0, v_4\}, \{v_1, v_5\}, \{v_4, v_5\}, \{v_2, v_3\}, \{v_3, v_4\}\}$ é um grafo simples e J em que $V(J) = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ e $E(J) = \{(v_0, v_1), (v_0, v_2), (v_2, v_0), (v_0, v_3), (v_4, v_0), (v_2, v_3), (v_3, v_4), (v_5, v_4), (v_5, v_1)\}$ é um digrafo simples. Eles são representados na Figura 23.2.

Em geral, vamos indicar uma aresta $e = \{x, y\}$ ou um arco $a = (x, y)$ simplesmente como

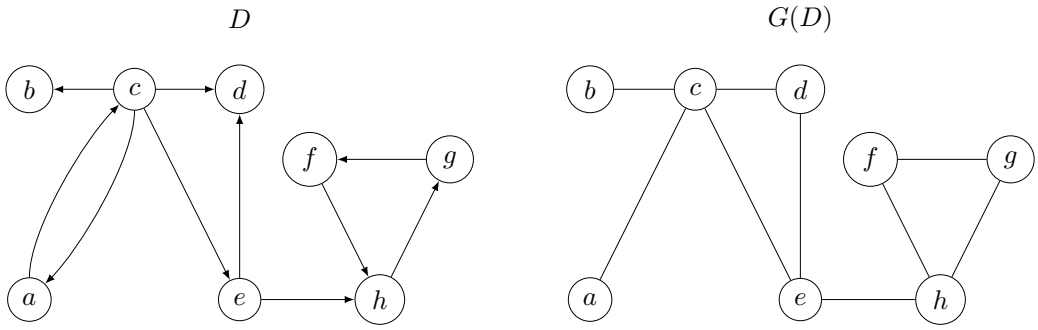


Figura 23.3: Digrafo D e seu grafo subjacente $G(D)$.

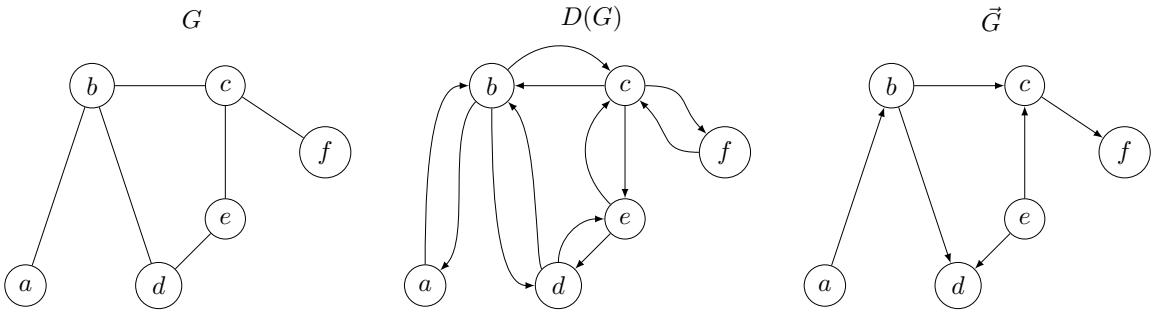


Figura 23.4: Um grafo G, seu digrafo associado $D(G)$ e uma possível orientação de G.

xy. É importante observar que $xy = yx = \{x, y\}$ mas $xy = (x, y) \neq yx = (y, x)$.

23.1 Relação entre grafos e digrafos

Seja D um digrafo qualquer. Podemos associar a D um grafo $G(D)$ tal que $V(G(D)) = V(D)$ e para cada arco xy de D existe uma aresta xy em $G(D)$. Esse grafo é chamado *grafo subjacente de* D. Veja a Figura 23.3 para um exemplo. Muitas definições sobre grafos podem fazer sentido em digrafos se considerarmos seu grafo subjacente.

Seja G um grafo qualquer. Podemos associar a G um digrafo $D(G)$ tal que $V(D(G)) = V(G)$ e para cada aresta xy de G existem dois arcos, xy e yx, em $D(G)$. Esse digrafo é chamado *digrafo associado a* G. Veja a Figura 23.4 para um exemplo.

A partir de um grafo G qualquer, também é possível obter um digrafo $\vec{G}$ fazendo $V(\vec{G}) = V(G)$ e para cada aresta xy de G escolher o arco xy ou o arco yx para existir em $\vec{G}$. Esse grafo $\vec{G}$ é chamado de *orientação de* G. Se um digrafo D qualquer é uma orientação de algum grafo H, então D é chamado de *grafo orientado*. Veja a Figura 23.4 para um exemplo.

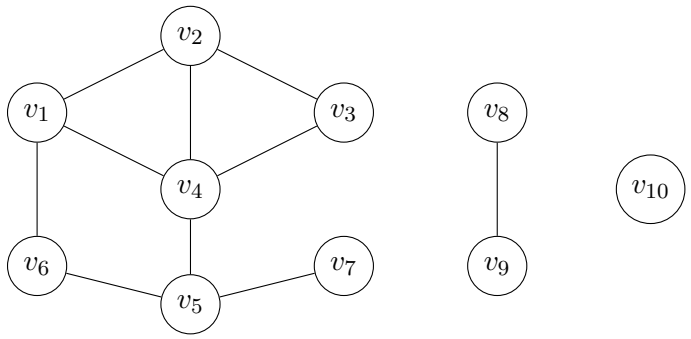


Figura 23.5: Grafo G com $d(v_1) = 3$, $d(v_2) = 3$, $d(v_3) = 2$, $d(v_4) = 4$, $d(v_5) = 3$, $d(v_6) = 2$, $d(v_7) = 1$, $d(v_8) = 1$, $d(v_9) = 1$ e $d(v_{10}) = 0$. Assim, $\delta(G) = 0$, $\Delta(G) = 4$ e $\bar{d}(G) = (3+3+2+4+3+2+1+1+1+0)/10 = 2$. Note que $N(v_1) = \{v_2, v_4, v_6\}$ e $N(v_{10}) = \emptyset$. Ademais, v_3 e v_4 são adjacentes, a aresta v_5v_7 incide em v_5 e v_7 e v_{10} é um vértice isolado.

23.2 Adjacências e incidências

Seja $e = xy$ uma aresta de um grafo G. Dizemos que os vértices x e y são *vizinhos* e que são vértices *adjacentes*. Assim, o vértice x é *adjacente* ao vértice y e vice-versa. Os vértices x e y são chamados de *extremos* da aresta xy. Arestas que possuem um extremo em comum são ditas *adjacentes*. Relacionamos vértices e arestas dizendo que a aresta xy *incide* em x e em y.

O *grau* de um vértice x de um grafo G, denotado por $d_G(x)$, é a quantidade de vizinhos do vértice x. Já o conjunto dos vizinhos de x, a *vizinhança* de x, é denotado por $N_G(x)$. Dado um conjunto $X \subseteq V(G)$, definimos a *vizinhança* de X como $N_G(X) = \bigcup_{x \in X} N_G(x)$. Quando estiver claro a qual grafo estamos nos referindo, utilizamos simplesmente as notações $d(x)$ e $N(x)$, e fazemos o mesmo com todas as notações em que G está subscrito. Um vértice sem vizinhos, isto é, de grau 0, é chamado de *vértice isolado*.

O *grau mínimo* de um grafo G, denotado por $\delta(G)$, é o menor grau dentre todos os vértices de G, i.e., $\delta(G) = \min\{d(x) \colon x \in V(G)\}$. O *grau máximo* de G, denotado por $\Delta(G)$, é o maior grau dentre todos os vértices de G, i.e., $\Delta(G) = \max\{d(x) \colon x \in V(G)\}$. Por fim, o grau médio de G, denotado por $\bar{d}(G)$, é a média de todos os graus de G, i.e., $\bar{d}(G) = \left(\sum_{x \in V(G)} d(x)\right) / v(G)$.

A Figura 23.5 exemplifica os conceitos mencionados acima.

As definições acima se aplicam automaticamente em digrafos. Porém, existem conceitos em que considerar a orientação é essencial.

Seja $a = xy$ um arco de um digrafo D. Também dizemos que os vértices x e y são *vizinhos* e são vértices *adjacentes*. Dizemos ainda que x é a *cabeça* de a e que y é a *cauda*,

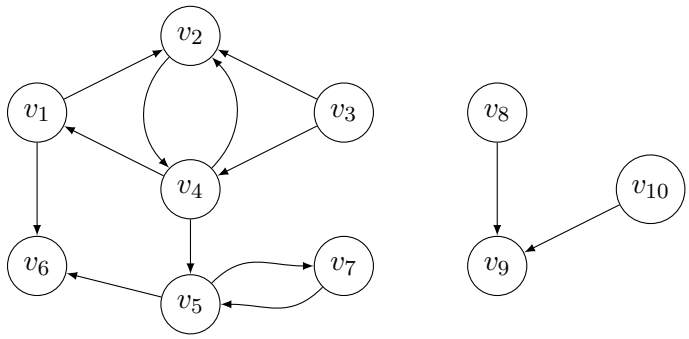


Figura 23.6: Digrafo D com $d^+(v_1) = 2$, $d^-(v_1) = 1$, $d^+(v_2) = 1$, $d^-(v_2) = 3$, $d^+(v_3) = 2$, $d^-(v_3) = 0$, $d^+(v_4) = 3$, $d^-(v_4) = 2$, $d^+(v_5) = 2$, $d^-(v_5) = 2$, $d^+(v_6) = 0$, $d^-(v_6) = 2$, $d^+(v_7) = 1$, $d^-(v_7) = 1$, $d^+(v_8) = 1$, $d^-(v_8) = 0$, $d^+(v_9) = 0$, $d^-(v_9) = 2$, $d^+(v_{10}) = 1$ e $d^-(v_{10}) = 0$. Note que $N^+(v_1) = \{v_2, v_6\}$, $N^-(v_1) = \{v_4\}$, $N^+(v_{10}) = \{v_9\}$ e $N^-(v_{10}) = \emptyset$. Ademais, v_3 domina v_2 e v_4, e v_5 é dominado por v_4 e v_7.

sendo ambos x e y os *extremos* de a. É comum dizer também o arco xy *sai* de x e *entra* em y.

Seja x um vértice de um digrafo D. O *grau de entrada* de x em D, denotado $d_D^-(x)$, é a quantidade de arcos que entram em x. O *grau de saída* de x em D, denotado $d_D^+(x)$, é a quantidade de arcos que saem de x. Os vértices extremos dos arcos que entram em um certo vértice x, exceto o próprio x, são seus *vizinhos de entrada* e formam o conjunto $N_D^-(x)$. Os vértices extremos dos arcos que saem de um vértice x, exceto o próprio x, são seus *vizinhos de saída* e formam o conjunto $N_D^+(x)$.

A Figura 23.6 exemplifica os conceitos definidos acima.

Os Teoremas 23.1 e 23.2 a seguir estabelecem uma relação identidade fundamental que relaciona os graus dos vértices com o número de arestas ou arcos em um grafo ou digrafo.

Teorema 23.1

Para todo grafo G temos que $\sum_{x \in V(G)} d_G(x) = 2e(G)$.

Demonstração. Uma aresta uv é contada duas vezes na soma dos graus, uma em $d_G(u)$ e outra em $d_G(v)$. □

Teorema 23.2

Para todo digrafo D temos que $\sum_{x \in V(G)} d_G^+(x) = \sum_{x \in V(G)} d^-(x) = e(G)$.

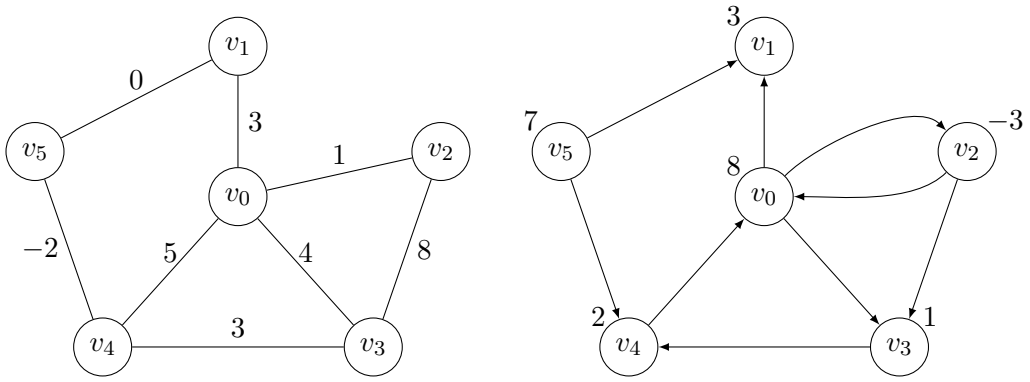


Figura 23.7: Exemplos de ponderação nas arestas e nos vértices.

Demonstração. Um arco uv é contado uma vez no grau de saída de u e uma vez no grau de entrada de v. □

23.3 Grafos e digrafos ponderados

Muitos problemas que são modelados por (di)grafos envolvem atribuir valores às arestas e/ou aos vértices. Suponha que queremos saber a rota mais curta para sair de Fortaleza, CE, e chegar à Maringá, PR. Certamente uma forma natural é representar cada interseção entre estradas como um vértice e cada estrada como uma aresta. É útil para a solução do problema, portanto, indicar qual a quilometragem de cada estrada. De forma geral, se o grafo representa uma malha rodoviária em que vértices são cidades e arestas representam estradas entre cidades, então pode ser útil indicar qual é o comprimento das estradas, ou então quanto tempo leva para percorrê-las, ou mesmo qual é o custo dos pedágios. Se o grafo representa uma rede de distribuição em que vértices são cidades e arestas representam estradas entre cidades, então pode ser útil indicar qual é o custo de abrir uma fábrica em uma cidade e qual seria o custo de transportar bens entre uma fábrica e outras cidades. Se o grafo representa transações financeiras em que vértices são entidades e arestas representam as transações feitas entre as entidades, então pode ser útil indicar qual o valor das transações feitas, sendo que elas terão valor positivo em caso de vendas e negativo em caso de compras.

Por isso, em muitos casos os (di)grafos são *ponderados*, o que indica que, além de G, temos uma função $c\colon V(G) \to \mathcal{N}$ e/ou $w\colon E(G) \to \mathcal{N}$, onde $\mathcal{N}$ em geral é algum conjunto numérico. Graficamente, esses valores são indicados sobre as arestas ou os vértices. Veja a Figura 23.7 para alguns exemplos.

Se um (di)grafo G é ponderado nas arestas por uma função $w\colon E(G) \to \mathcal{N}$, então naturalmente qualquer subconjunto de arestas $F \subseteq E(G)$ tem peso $w(F) = \sum_{e \in F} w(e)$. Se

G é ponderado nos vértices por uma função $c\colon V(G) \to \mathcal{N}$, então naturalmente qualquer subconjunto de vértices $S \subseteq V(G)$ tem peso $c(S) = \sum_{v \in S} c(v)$.

23.4 Formas de representação

Certamente podemos representar (di)grafos simplesmente utilizando conjuntos para vértices e arestas/arcos. Porém, é desejável utilizar alguma estrutura de dados que nos permita ganhar em eficiência dependendo da tarefa que necessitamos. As duas formas mais comuns de representação de (di)grafos são *listas de adjacências* e *matriz de adjacências*. Por simplicidade, vamos assumir que um (di)grafo com n vértices tem conjunto de vértices $\{1, 2, \ldots, n\}$.

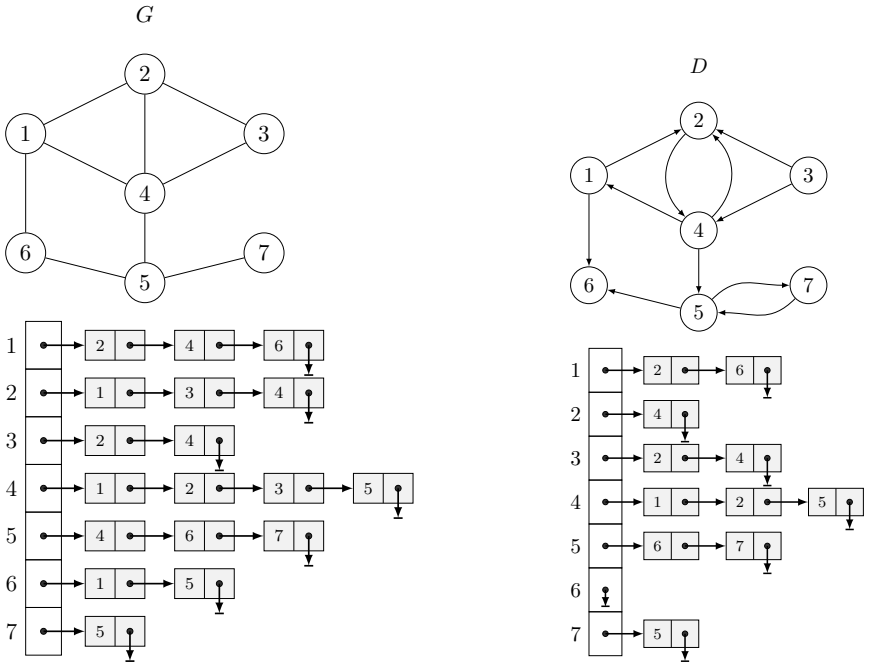
Na representação por *listas de adjacências*, um (di)grafo G é dado por $v(G)$ listas, uma para cada vértice. A lista de um vértice x contém apenas os vizinhos de x, se G é grafo, ou os vizinhos de saída de x, se G é digrafo. Podemos ter um segundo conjunto de listas, para armazenar os vizinhos de entrada de x, caso necessário. Note que são necessários $\Theta(v(G))$ ponteiros para as listas e que a lista de um vértice x tem $d(x)$ nós. Pelo resultado dos Teoremas 23.1 e 23.2, sabemos que a quantidade total de nós é $\Theta(e(G))$. Assim, o espaço necessário para armazenar as listas de adjacências de um (di)grafo é $\Theta(v(G) + e(G))$.

Na representação por *matriz de adjacências*, um (di)grafo G é dado por uma matriz quadrada M de tamanho $v(G) \times v(G)$, em que $M[i][j] = 1$ se $ij \in E(G)$, e $M[i][j] = 0$ caso contrário. Assim, se G é um grafo, então M é simétrica. Note que o espaço necessário para armazenar uma matriz de adjacências de um (di)grafo é $\Theta(v(G)^2)$.

A Figura 23.8 apresenta as duas representações sobre o mesmo (di)grafo.

Em geral, o uso de listas de adjacências é preferido para representar (di)grafos *esparsos*, que são (di)grafos com n vértices e $O(n)$ arestas, pois o espaço $\Theta(n^2)$ necessário pela matriz de adjacências é dispendioso. Já a representação por matriz de adjacências é muito usada para representar (di)grafos *densos*, que são (di)grafos com $\Theta(n^2)$ arestas. Porém, esse não é o único fator importante na escolha da estrutura de dados utilizada para representar um (di)grafo, pois determinados algoritmos precisam de propriedades da representação por listas e outros da representação por matriz para serem eficientes. Sempre que necessário, iremos destacar a diferença de utilizar uma ou outra estrutura específica.

Se o (di)grafo é ponderado, então precisamos adaptar essas representações para armazenar os pesos e isso pode ser feito de várias maneiras diferentes. No caso de pesos nas arestas, os nós das listas de adjacências podem, por exemplo, conter um campo para armazenar os pesos das mesmas. Na matriz de adjacências, o valor em $M[i][j]$ pode ser usado para indicar o peso da aresta ij (nesse caso, deve-se considerar um outro indicador para quando a aresta não existe, uma vez que arestas podem ter peso 0). Outra forma que pode ser utilizada



	1	2	3	4	5	6	7
1	0	1	0	1	0	1	0
2	1	0	1	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	1	0	1	0	0
5	0	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	0	0	0	1	0	0

	1	2	3	4	5	6	7
1	0	1	0	0	0	1	0
2	0	0	0	1	0	0	0
3	0	1	0	1	0	0	0
4	1	1	0	0	1	0	0
5	0	0	0	0	0	1	1
6	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0

Figura 23.8: Representação gráfica de um grafo G e um digrafo D na primeira linha, suas listas de adjacências na segunda linha e suas matrizes de adjacências na última linha.

em ambas representações é manter uma segunda matriz apenas para indicar os pesos das arestas. No caso de pesos nos vértices, ambas representações podem fazer uso de um novo vetor, indexado por vértices, que armazene tais valores.

23.5 Pseudocódigos

O Algoritmo 23.1 recebe um grafo G e devolve o grau máximo de G. Ele não menciona nem acessa uma matriz ou lista, pois isso é muito dependente de detalhes de implementação. É dessa forma que iremos apresentar os pseudocódigos referentes a grafos nesse livro. Iremos, no entanto, analisar o tempo de execução considerando ambas as representações. O algoritmo GrauMaximo(G), por exemplo, leva tempo $\Theta(v(G)^2)$ se implementado com matriz de adjacências e $\Theta(v(G)+e(G))$ se implementado com lista de adjacências. Veremos a seguir detalhes destas análises.

Algoritmo 23.1: GrauMaximo(G)

```
1 max = 0
2 para todo vértice x ∈ V(G) faça
3     grau_x = 0
4     para todo vértice y ∈ N(x) faça
5         grau_x = grau_x + 1
6     se grau_x > max então
7         max = grau_x
8 devolve max
```

Para facilitar a discussão, os Algoritmos 23.2 e 23.3 mostram o cálculo do grau máximo considerando mais detalhes de implementação. No que segue, vamos usar $n = v(G)$ e $m = e(G)$ e vamos denotar o conjunto de vértices por $V(G) = \{v_1, v_2, \ldots, v_n\}$.

No Algoritmo 23.2, o tempo T_M é dado pela seguinte expressão:

$$\begin{aligned}
T_M = {} & \underbrace{\Theta(1)}_{linha\ 1} + \overbrace{\underbrace{\Theta(1) + \cdots + \Theta(1)}_{linha\ 2}}^{n \text{ vezes}} + \overbrace{\underbrace{\Theta(1) + \cdots + \Theta(1)}_{linha\ 3}}^{n \text{ vezes}} + \overbrace{\underbrace{\Theta(n) + \cdots + \Theta(n)}_{linha\ 4}}^{n \text{ vezes}} + \\
& + \overbrace{\underbrace{\Theta(n) + \cdots + \Theta(n)}_{linha\ 5}}^{n \text{ vezes}} + \overbrace{\underbrace{\Theta(1) + \cdots + \Theta(1)}_{linha\ 6}}^{n \text{ vezes}} + \underbrace{x}_{linha\ 7} + \underbrace{\Theta(1)}_{linha\ 8} \\
& = \Theta(1) + \Theta(n) + \Theta(n^2) + \Theta(n) + x + \Theta(1) \\
& = \Theta(n^2) + x\,.
\end{aligned}$$

Note que a linha 7 executa no máximo uma vez por vértice do grafo e leva tempo constante por execução, portanto x é $O(n)$. Logo, $\Theta(n^2)$ domina a expressão acima e, portanto, o tempo do algoritmo quando implementado em matriz de adjacências é $\Theta(n^2)$.

Já para o Algoritmo 23.3 podemos fazer duas análises diferentes (ambas corretas, porém uma menos justa do que a outra). Vejamos primeiro a análise menos justa. Para ela, observe que cada vértice tem no máximo $n-1$ ($O(n)$) outros vértices em sua lista de adjacências. Assim, o tempo T_L é dado por:

$$
\begin{aligned}
T_L = & \underbrace{\Theta(1)}_{linha\ 1} + \overbrace{\underbrace{\Theta(1)+\cdots+\Theta(1)}_{linha\ 2}}^{n \text{ vezes}} + \overbrace{\underbrace{\Theta(1)+\cdots+\Theta(1)}_{linha\ 3}}^{n \text{ vezes}} + \overbrace{\underbrace{\Theta(1)+\cdots+\Theta(1)}_{linha\ 4}}^{n \text{ vezes}} + \\
& + \overbrace{\underbrace{O(n)+\cdots+O(n)}_{linha\ 5}}^{n \text{ vezes}} + \overbrace{\underbrace{O(n)+\cdots+O(n)}_{linha\ 6}}^{\leq n \text{ vezes}} + \overbrace{\underbrace{O(n)+\cdots+O(n)}_{linha\ 7}}^{\leq n \text{ vezes}} + \overbrace{\underbrace{\Theta(1)+\cdots+\Theta(1)}_{linha\ 8}}^{n \text{ vezes}} + \\
& + \underbrace{O(n)}_{linha\ 9} + \underbrace{\Theta(1)}_{linha\ 10} \\
& = \Theta(1) + \Theta(n) + \Theta(n) + \Theta(n) + O(n^2) + O(n^2) + O(n^2) + \Theta(n) + O(n) + \Theta(1) \\
& = O(n^2)\,.
\end{aligned}
$$

Novamente, x é $O(n)$, fazendo com que T_L seja $O(n^2)$. Essa análise nos leva a um pior caso assintoticamente igual ao das matrizes de adjacências.

Vejamos agora a análise mais justa. Para ela, lembra-se que cada vértice v_i tem exatamente $d(v_i)$, seu grau, vértices em sua lista de adjacências. Assim, o tempo T_{L2} é dado por:

$$
\begin{aligned}
T_{L2} = & \underbrace{\Theta(1)}_{linha\ 1} + \overbrace{\underbrace{\Theta(1)+\cdots+\Theta(1)}_{linha\ 2}}^{n \text{ vezes}} + \overbrace{\underbrace{\Theta(1)+\cdots+\Theta(1)}_{linha\ 3}}^{n \text{ vezes}} + \overbrace{\underbrace{\Theta(1)+\cdots+\Theta(1)}_{linha\ 4}}^{n \text{ vezes}} + \\
& + \underbrace{\Theta(d(v_1))+\cdots+\Theta(d(v_n))}_{linha\ 5} + \underbrace{\Theta(d(v_1))+\cdots+\Theta(d(v_n))}_{linha\ 6} + \underbrace{\Theta(d(v_1))+\cdots+\Theta(d(v_n))}_{linha\ 7} + \\
& + \overbrace{\underbrace{\Theta(1)+\cdots+\Theta(1)}_{linha\ 8}}^{n \text{ vezes}} + \underbrace{O(n)}_{linha\ 9} + \underbrace{\Theta(1)}_{linha\ 10} \\
& = \Theta(1) + \Theta(n) + \Theta(n) + \Theta(n) + \sum_{i=1}^{n} \Theta(d(v_i)) + \sum_{i=1}^{n} \Theta(d(v_i)) + \sum_{i=1}^{n} \Theta(d(v_i)) + \Theta(n) + O(n) + \Theta(1) \\
& = \Theta(n+m)\,,
\end{aligned}
$$

pois $\sum_{i=1}^{n} d(v_i) = 2m$. Note como $\Theta(n+m)$ pode ser bem melhor do que $O(n^2)$, dependendo do grafo dado.

Algoritmo 23.2: GRAUMAXIMO(M, n)

```
1 max = 0
2 para x = 1 até n, incrementando faça
3     grau_x = 0
4     para y = 1 até n, incrementando faça
5         se M[x][y] == 1 então
6             grau_x = grau_x + 1
7     se grau_x > max então
8         max = grau_x
9 devolve max
```

Algoritmo 23.3: GRAUMAXIMO(L, n)

```
1 max = 0
2 para x = 1 até n, incrementando faça
3     grau_x = 0
4     atual = L[x]
5     enquanto atual ≠ null faça
6         grau_x = grau_x + 1
7         atual = atual.proximo
8     se grau_x > max então
9         max = grau_x
10 devolve max
```

23.6 Subgrafos

Um (di)grafo H é *sub(di)grafo* de um (di)grafo G se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$, em que os extremos dos elementos de $E(H)$ estão em $V(H)$. Dizemos também que G *contém* H ou G é *supergrafo* de H, e escrevemos $H \subseteq G$ para denotar essa relação.

Um sub(di)grafo H de um (di)grafo G é *gerador* se $V(H) = V(G)$.

Dado um conjunto de vértices $S \subseteq V(G)$ de um (di)grafo G, o sub(di)grafo de G *induzido por* S, denotado por $G[S]$, é o sub(di)grafo H de G tal que $V(H) = S$ e $E(H)$ é o conjunto

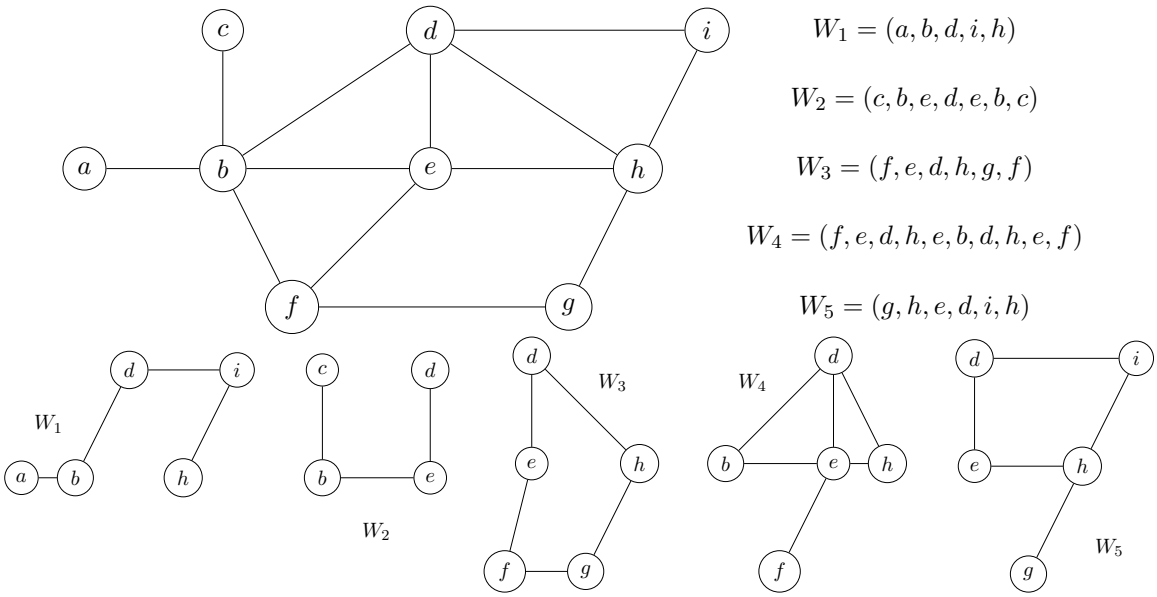


Figura 23.9: Ao topo da figura, um grafo G e passeios $W_1, \ldots, W_5$ sobre ele. Na parte inferior, os subgrafos induzidos pelas arestas desses passeios. Os passeios W_1 e W_5 são abertos enquanto que W_2, W_3 e W_4 são fechados. Veja que W_1 é um caminho e W_3 é um ciclo. Veja que W_2 não é um caminho, apesar do grafo induzido pelas suas arestas ser.

de arestas de G com os dois extremos em S, i.e., $E(H) = \{uv \colon uv \in E(G) \text{ e } u, v \in S\}$. Similarmente, se F é um subconjunto de arestas de G, então o sub(di)grafo de G *induzido por* F, denotado por $G[F]$, é o sub(di)grafo H de G tal que $E(H) = F$ e $V(H)$ é o conjunto de vértices de G que são extremos de alguma aresta de F, i.e., $V(H) = \{v \colon \text{ existe } u \text{ com } uv \in F\}$. Quando conveniente, denotamos por $G - X$ e $G - F$, respectivamente, os (di)grafos obtidos de G pela remoção de X e F, i.e., $G - X = G[V(G) \setminus X]$ e $G - F = G[E(G) \setminus F]$. Ademais, dado um (di)grafo G, um conjunto de vértices S' que não está em $V(G)$ e um conjunto de arestas F' que não está em $E(G)$ (mas é formado por pares de vértices de G), denotamos por $G + S'$ e $G + F'$, respectivamente, os grafos obtidos de G pela adição de S' e F', i.e., $G + S' = (V(G) \cup S', E(G))$ e $G + F' = (V(G), E(G) \cup F')$.

As Figuras 23.9 e 23.10 apresentam exemplos de (di)grafos e sub(di)grafos.

Seja G um (di)grafo e $H \subseteq G$ um sub(di)grafo de G. Se G é ponderado nas arestas por uma função $w \colon E(G) \to \mathbb{R}$, então o peso de H, denotado $w(H)$, é dado pelo peso das arestas de H, isto é, $w(H) = \sum_{e \in E(H)} w(e)$. Se G é ponderado nos vértices por uma função $c \colon V(G) \to \mathbb{R}$, então o peso de H, denotado $c(H)$, é dado pelo peso dos vértices de H, isto é, $w(H) = \sum_{v \in V(H)} c(v)$. Se G é ponderado em ambos os vértices e arestas, o peso de um subgrafo será devidamente definido no problema.

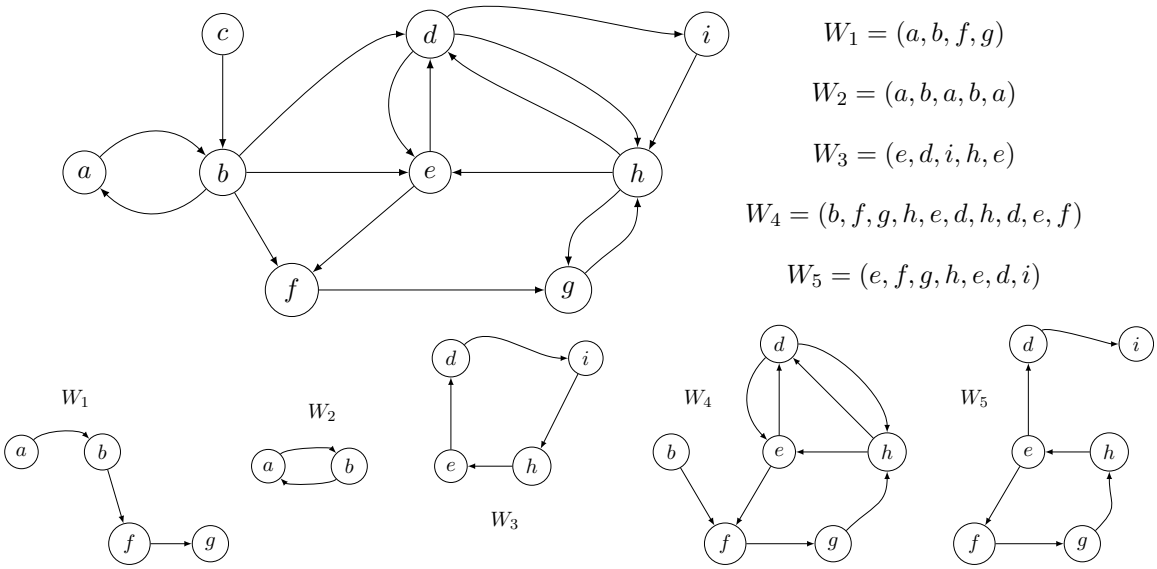


Figura 23.10: Ao topo da figura, um digrafo D e passeios $W_1, \ldots, W_5$ sobre ele. Na parte inferior, os subdigrafos induzidos pelas arestas desses passeios. Os passeios W_1, W_4 e W_5 são abertos enquanto que W_2 e W_3 são fechados. Veja que W_1 é um caminho e W_3 é um ciclo. Veja que W_2 não é um ciclo, apesar do grafo induzido pelas suas arestas ser. Note que (b, f, e, h) não é um passeio em D.

23.6.1 Modificando grafos

Uma forma de gerar um subgrafo de um grafo G é por remoção de elementos de G. Dado $S \subseteq V(G)$, o grafo gerado ao remover S de G é denotado $G - S$. Formalmente, $G - S = G[V(G) \setminus S]$, isto é, é o subgrafo induzido pelos vértices que sobram. Dado $F \subseteq E(G)$, o grafo gerado ao remover F de G é denotado $G - F$. Formalmente, $G - F = G[E(G) \setminus F]$, isto é, é o subgrafo induzido pelas arestas que sobram.

Adotaremos um abuso de notação para quando S e F consistem de um único elemento. Assim, $G - v = G - \{v\}$ para $v \in V(G)$ e $G - e = G - \{e\}$ para $e \in E(G)$.

Uma forma de gerar um supergrafo de um grafo G é por adição de elementos a G. Dado $S \nsubseteq V(G)$, o grafo gerado pela adição de S a G é denotado $G + S$. Formalmente, $G + S$ é tal que $V(G+S) = V(G) \cup S$ e $E(G+S) = E(G)$. Dado $F \nsubseteq E(G)$ cujos elementos são pares de elementos em $V(G)$, o grafo gerado pela adição de F a G é denotado $G + F$. Formalmente, $G + F$ é tal que $V(G + F) = V(G)$ e $E(G + F) = E(G) \cup F$.

Também adotaremos um abuso de notação para quando S e F consistem de um único elemento. Assim, $G + v = G + \{v\}$ para $v \notin V(G)$ e $G + e = G + \{e\}$ para $e \notin E(G)$ com $e = xy$ e $x, y \in V(G)$.

23.7 Passeios, trilhas, caminhos e ciclos

Seja G um (di)grafo. Um *passeio* em G é uma sequência de vértices não necessariamente distintos $W = (v_0, v_1, \ldots, v_k)$ tal que $v_i v_{i+1} \in E(G)$ para todo $0 \leq i < k$. Dizemos que v_0 é a *origem* ou *vértice inicial* do passeio, enquanto v_k é o *término* ou *vértice final*. Ambos são *extremos* do passeio enquanto que $v_1, \ldots, v_{k-1}$ são *vértices internos*. Dizemos ainda que W conecta v_0 a v_k e que ele é um v_0v_k-*passeio*. As arestas $v_0v_1, v_1v_2, \ldots, v_{k-1}v_k$ são chamadas de *arestas do passeio*. O *comprimento* do passeio é dado pelo número de arestas do passeio. Um passeio é dito *fechado* se tem comprimento não nulo e sua origem e término são iguais. Dizemos que um passeio é *aberto* quando queremos enfatizar o fato de ele não ser fechado.

Quando conveniente, é comum tratar um passeio $W = (v_0, v_1, \ldots, v_k)$ como sendo o grafo induzido pelas arestas de W e chamar tal subgrafo também de passeio. Assim, vamos usar a notação $V(W)$ e $E(W)$ para nos referir aos conjuntos $\{v_0, \ldots, v_k\}$ e $\{v_i v_{i+1} \colon 0 \leq i < k\}$, respectivamente. Veja as Figuras 23.9 e 23.10 para exemplos das definições acima.

Seja G um (di)grafo e $W = (v_0, v_1, \ldots, v_k)$ um passeio em G com $v_0 \neq v_k$. Chamamos W de *trilha* se para todo $0 \leq i < j < k$ temos que $v_i v_{i+1} \neq v_j v_{j+1}$, isto é, não há arestas repetidas dentre as arestas de W. Chamamos W de *caminho* se para todo $0 \leq i < j \leq k$ temos que $v_i \neq v_j$, isto é, não há vértices repetidos dentre os vértices de W. Se W é fechado, isto é, se $v_0 = v_k$, então W é chamado de *trilha fechada* se não há arestas repetidas e é chamado de *ciclo* se não há vértices internos repetidos.

Utilizamos o termo *caminho* para nos referirmos a qualquer (di)grafo ou sub(di)grafo H com n vértices e $n-1$ arestas com os vértices do qual é possível escrever uma sequência que é um caminho com n vértices. Denotamos tais (di)grafos por P_n. Por exemplo, os subgrafos W_1 e W_2 da Figura 23.9 são caminhos (P_5 e P_4, respectivamente) e o subgrafo W_5 contém um P_3 (o caminho (d, i, h)). De forma equivalente, utilizamos o termo *ciclo* para nos referirmos a qualquer (di)grafo ou sub(di)grafo H com n vértices e n arestas com os vértices do qual é possível escrever uma sequência que é um ciclo com n vértices. Denotamos tais (di)grafos por C_n. Por exemplo, o subgrafo W_3 da Figura 23.10 é um ciclo C_4 e o subgrafo W_4 contém um C_6 (o ciclo (f, g, h, d, e, f)) e um C_2 (o ciclo (h, d, h)).

23.8 Conexidade

Um (di)grafo G é dito *aresta-maximal* (ou apenas *maximal*) com respeito a uma propriedade $\mathcal{P}$ (por exemplo, uma propriedade de um grafo G pode ser "G não contém C_3", "G tem no máximo k arestas" ou "G é um caminho") se G possui a propriedade $\mathcal{P}$ e qualquer grafo obtido da adição de arestas ou arcos a G não possui a propriedade $\mathcal{P}$.

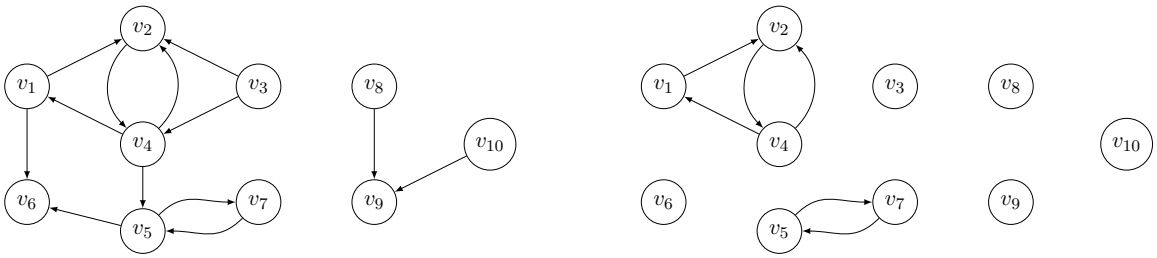


Figura 23.11: Digrafo D à esquerda. Ele é desconexo e possui 2 componentes conexas. Ele não é fortemente conexo (por exemplo, não há caminho entre v_6 e v_3), e possui 7 componentes fortemente conexas, em destaque à direita.

Seja G um grafo. Dizemos que G é *conexo* se existe uv-caminho para todo par de vértices $u, v \in V(G)$. Caso contrário, dizemos que G é *desconexo*. Uma *componente conexa* de G é um subgrafo conexo induzido por um conjunto de vértices S tal que não existem arestas entre S e $V(G) \setminus S$ em G. Em outras palavras, os subgrafos conexos de um grafo G que são maximais com respeito à conexidade são chamados de *componentes conexas*. A quantidade de componentes de um grafo G é denotada por $c(G)$. Por exemplo, o grafo G da Figura 23.5 é desconexo e possui 3 componentes conexas (uma possui os vértices $v_1, \ldots, v_7$, outra possui os vértices v_8, v_9 e a terceira possui o vértice v_{10} apenas). O grafo G da Figura 23.8 é conexo.

Em um grafo, se existe um uv-caminho, então claramente existe um vu-caminho. Em digrafos isso não necessariamente é verdade. Por isso, esse conceito é um pouco diferente.

Seja D um digrafo. Dizemos que D é *conexo* se o grafo subjacente $G(D)$ for conexo e é *desconexo* caso contrário. Dizemos que D é *fortemente conexo* se existe uv-caminho para todo par de vértices $u, v \in V(D)$. Um digrafo que não é fortemente conexo consiste em um conjunto de *componentes fortemente conexas*, que são subgrafos fortemente conexos maximais (com respeito à conexidade em digrafos). Veja a Figura 23.11 para um exemplo.

23.9 Distância entre vértices

Seja G um (di)grafo não ponderado. Denotamos a *distância entre u e v em G* por $\text{dist}_G(u, v)$ e a definimos como o comprimento de um uv-caminho de menor comprimento. Se não existe caminho entre u e v, então convencionamos que $\text{dist}_G(u, v) = \infty$. Assim, $\text{dist}_G(u, u) = 0$. Se um uv-caminho tem comprimento igual à distância entre u e v, então dizemos que ele é um uv-caminho *mínimo*.

Considere o grafo G do canto superior esquerdo da Figura 23.9. Note que (a, b, d, i, h), (a, b, d, h), (a, b, d, e, h), (a, b, e, d, i, h), (a, b, e, d, h), (a, b, e, h), (a, b, e, f, g, h), (a, b, f, g, h), (a, b, f, e, d, i, h), (a, b, f, e, d, h) e (a, b, f, e, h) são todos os ah-caminhos possíveis. Um de me-

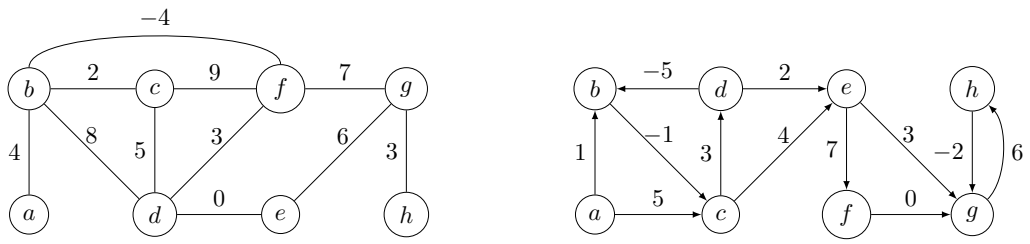


Figura 23.12: O grafo acima tem uma aresta de peso negativo, bf, e digrafo tem um ciclo de peso negativo, (d, b, c, d).

nor comprimento, no entanto, é (a, b, e, h). Como não há duas arestas que ligam a a h, temos $\text{dist}_G(a, h) = 3$. Temos também que (d, b, f, g), (d, e, f, g), (d, e, h, g), (d, i, h, g), (d, h, g), (d, b, f, e, h, g), e alguns outros, são ah-caminhos. O de menor comprimento, no entanto, possui 2 arestas (não há aresta direta entre d e g e existe um caminho com duas arestas). Assim, temos $\text{dist}_G(d, g) = 2$. Como esse grafo é conexo, $\text{dist}_G(u, v) \neq \infty$ para nenhum par $u, v \in V(G)$. Agora considere o grafo D do canto superior esquerdo a Figura 23.10. Note que (b, f, g, h), (b, e, d, h), (b, d, h), (b, e, f, g, h), (b, d, i, h), e alguns outros, são bh-caminhos. O de menor comprimento, no entanto, possui dois arcos, pois não há arco direto entre b e h. Assim, $\text{dist}_D(b, h) = 2$. Note ainda que $\text{dist}_D(a, c) = \infty$ pois não há ac-caminho em D.

Seja G um (di)grafo ponderado nas arestas, com $w\colon E(G) \to \mathbb{R}$ sendo a função de peso. Lembre-se que o peso de um caminho é igual à soma dos pesos das arestas desse caminho. Denotamos a *distância entre u e v em G* por $\text{dist}_G^w(u, v)$ e a definimos como o peso de um uv-caminho de menor peso. Assim, $\text{dist}_G^w(u, u) = 0$. Se não existe caminho entre u e v, então convencionamos que $\text{dist}_G^w(u, v) = \infty$. Se um uv-caminho tem peso igual à distância entre u e v, então dizemos que ele é um uv-caminho *mínimo*.

Infelizmente, os algoritmos que veremos em breve, para resolver o problema de encontrar distância entre vértices de (di)grafos ponderados, não conseguem lidar com duas situações: grafos com arestas de custo negativo e digrafos com ciclos de custo negativo, como os da Figura 23.12. Com o que se sabe até o momento em Ciência da Computação, não é possível existir um algoritmo eficiente que resolva problemas de distância nessas situações[2].

23.10 Algumas classes importantes de grafos

Um (di)grafo G é *nulo* se $V(G) = E(G) = \emptyset$, é *vazio* se $E(G) = \emptyset$ e é *trivial* se $v(G) = 1$ e $E(G) = \emptyset$. As definições a seguir não têm paralelo em digrafos, mas podem ser usadas se estivermos nos referenciando ao grafo subjacente de um digrafo.

[2]Essa afirmação será provada no Capítulo 29.

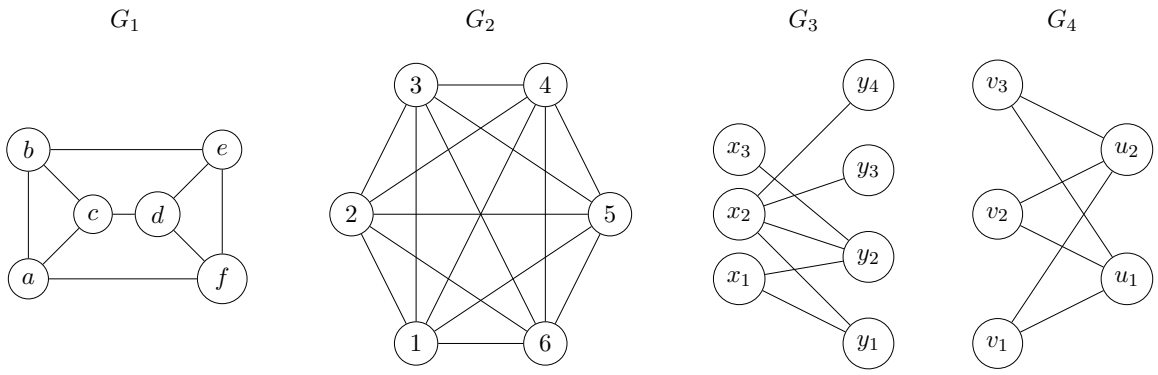


Figura 23.13: O grafo G_1 é 3-regular, G_2 é um K_6 (completo, portanto 5-regular), G_3 é bipartido e G_4 é um $K_{3,2}$ (bipartido completo). Os conjuntos $\{a, b, c\}$, $\{e, f\}$, $\{1, 2, 3, 4\}$, $\{x_2, y_4\}$, $\{v_1, u_1\}$ são alguns exemplos de cliques. Os conjuntos $\{b, f\}$, $\{3\}$, $\{x_1, x_3, y_3, y_4\}$, $\{v_1, v_2, v_3\}$ são alguns exemplos de conjuntos independentes.

Um grafo em que todos os vértices possuem o mesmo grau k é dito *k-regular*. Ademais, um grafo é dito *regular* se é um grafo k-regular para algum inteiro positivo k.

Um grafo com n vértices em que existe uma aresta entre todos os pares de vértices é chamado de *grafo completo* e é denotado por K_n. Um grafo completo com 3 vértices é chamado de *triângulo*. Note que o grafo completo K_n é $(n-1)$-regular e possui $\binom{n}{2}$ arestas, que é a quantidade total de pares de vértices.

Seja G um grafo e $S \subseteq V(G)$ um conjunto qualquer de vértices. Dizemos que S é uma *clique* se todos os pares de vértices em S são adjacentes. Dizemos que S é um *conjunto independente* se não existe nenhuma aresta entre os pares de vértices de S. Assim, clique e conjunto independente são definições complementares.

Note que o conjunto de vértices de um grafo completo é uma clique. Por isso, o maior conjunto independente em um grafo completo contém somente um vértice. Um grafo vazio também pode ser definido como um grafo no qual o conjunto de vértices é independente.

Um grafo G é *bipartido* se $V(G)$ pode ser particionado em dois conjuntos independentes X e Y. Em outras palavras, se existem conjuntos X e Y de vértices tais que $X \cup Y = V(G)$ e $X \cap Y = \emptyset$ e toda aresta de G tem um extremo em X e outro em Y. Usamos a notação (X, Y)-bipartido quando queremos evidenciar os conjuntos que formam a bipartição. Note que todo caminho é bipartido. Um grafo G é *bipartido completo* se G é (X, Y)-bipartido e para todo vértice $u \in X$ temos $N(u) = Y$. Um grafo bipartido completo em que uma parte da bipartição tem p vértices e a outra tem q vértices é denotado por $K_{p,q}$.

A Figura 23.13 exemplifica as terminologias discutidas acima.

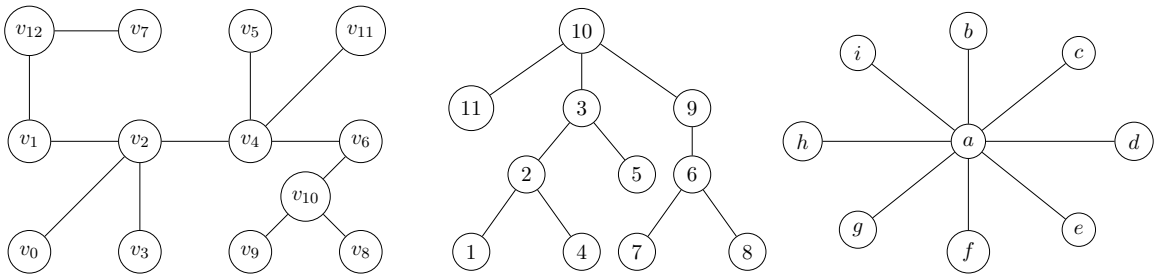


Figura 23.14: Exemplos de árvores.

23.10.1 Árvores

Considere n componentes conexas contendo um único vértice cada. Qual é o menor número de arestas que devem ser acrescentadas para que se tenha uma única componente conexa? Intuitivamente, note que acrescentar uma nova aresta entre duas componentes distintas reduz o número de componentes em uma unidade. Assim, acrescentar $n-1$ arestas (entre componentes distintas) é suficiente. Com uma formalização da discussão anterior é possível mostrar que para qualquer grafo conexo G vale que $e(G) \geq v(G) - 1$. A igualdade dessa inequação vale para uma classe particular de grafos, as *árvores*.

Um grafo T é uma *árvore* se T é conexo e tem $v(T)-1$ arestas ou, alternativamente, se T é conexo e sem ciclos. Note que todo caminho é uma árvore. Também vale que toda árvore é um grafo bipartido. A Figura 23.14 mostra exemplos de árvores.

Vértices de grau 1 são chamados de *folhas* e é possível mostrar que toda árvore tem pelo menos duas folhas. No entanto, não é verdade que qualquer árvore (grafo) possui uma raiz. Qualquer árvore, porém, pode ser desenhada de forma a parecer enraizada. Dizemos que uma árvore T é *enraizada* se há um vértice especial x chamado de raiz. Usamos o termo x-árvore para destacar esse fato.

Se G é um grafo sem ciclos, então note que cada componente conexa de G é uma árvore. Por isso, grafos sem ciclos são chamados de *floresta*. Note que toda árvore é uma floresta.

O Teorema 23.3 a seguir apresenta várias características importantes sobre árvores.

Pelo resultado do Teorema 23.3, note que se T é uma árvore e $e = uv \notin E(T)$ com $u, v \in V(T)$, então $T + e$ contém exatamente um ciclo. Ademais, para qualquer outra aresta $f \neq e$ de tal ciclo vale que $T + e - f$ também é uma árvore.

Seja G um grafo e $T \subseteq G$, isto é, um subgrafo de G. Se T é uma árvore tal que $V(T) = V(G)$, então dizemos que T é uma *árvore geradora* de G. Um fato bem conhecido é que todo grafo conexo contém uma árvore geradora. Isso porque, pelo Teorema 23.3, existe um caminho entre todo par de vértices na árvore, o que define conexidade no grafo. Note que pode haver várias árvores geradoras em um grafo conexo. Também é verdade, pelo

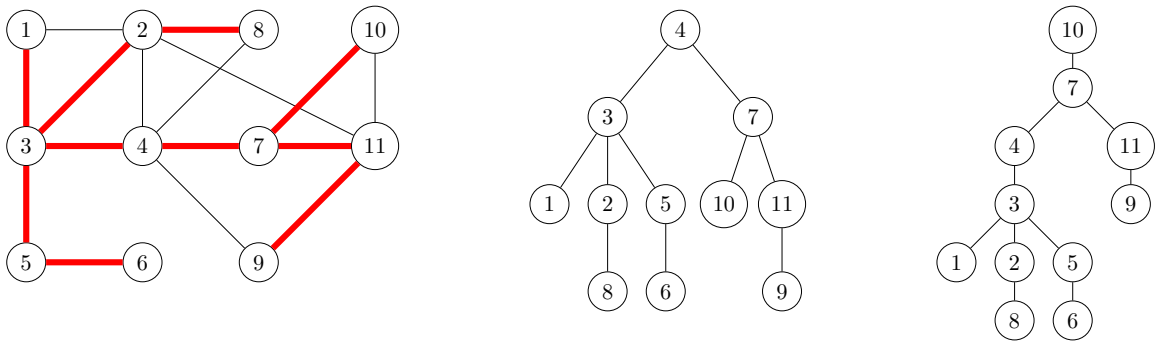


Figura 23.15: Exemplo de uma árvore geradora do grafo G, com destaque em vermelho sobre o próprio G. À direita, a mesma árvore geradora, mas como uma 4-árvore e uma 10-árvore.

teorema, que se T é árvore com $V(T) \neq V(G)$, então para qualquer aresta xy com $x \in V(T)$ e $y \in V(G) \setminus V(T)$ vale que $T + xy$ também é uma árvore.

Teorema 23.3

Seja G um grafo. As seguintes afirmações são equivalentes:

1. G é uma árvore.
2. Existe um único caminho entre quaisquer dois vértices de G.
3. G é conexo e para toda aresta $e \in E(G)$, vale que $G - e$ é desconexo.
4. G é conexo e $e(G) = v(G) - 1$.
5. G não contém ciclos e $e(G) = v(G) - 1$.
6. G não contém ciclos e para todo par de vértices $x, y \in V(G)$ não adjacentes, vale que $G + xy$ tem exatamente um ciclo.

Seja T uma x-árvore (uma árvore enraizada em x). Uma orientação de T na qual todo vértice, exceto x, tem grau de entrada 1 é chamada de *arborescência*. Usamos o termo x-arborescência para destacar o fato da raiz ser x.

Seja D um digrafo e $T \subseteq D$, isto é, um subdigrafo de G. Se T é uma arborescência tal que $V(T) = V(G)$, então dizemos que T é uma *arborescência geradora*. Não é difícil perceber que nem todo digrafo possui uma arborescência geradora.

As Figuras 23.15 e 23.16 mostram exemplos de árvores e arborescências geradoras.

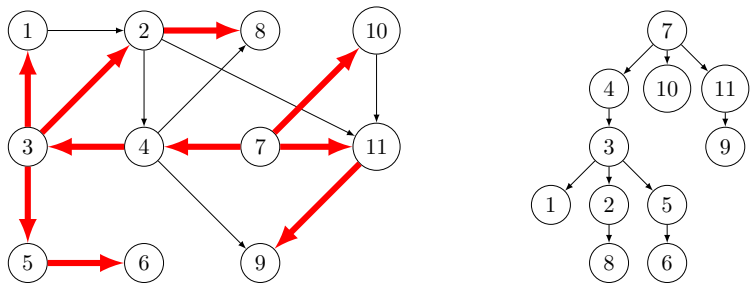


Figura 23.16: Exemplo de uma arborescência geradora do digrafo D, com destaque em vermelho sobre o próprio D. À direita, a mesma arborescência geradora, mas como uma 7-arborescência.

Capítulo 24

Buscas em grafos

Algoritmos de busca são importantíssimos em grafos. Usamos algoritmos de busca para obter mais informações sobre a estrutura do grafo como, por exemplo, para descobrir se a rede representada pelo grafo está totalmente conectada, qual a distância entre dois vértices do grafo, qual o caminho entre dois vértices, se existe um ciclo no grafo ou mesmo para formular um plano (podemos ver um caminho em um grafo como uma sequência de decisões que levam de um estado inicial a um estado final). Ademais, algoritmos de busca servem de "inspiração" para vários algoritmos importantes. Dentre eles, mencionamos o algoritmo de Prim para encontrar árvores geradoras mínimas em grafos e o algoritmo de Dijkstra para encontrar caminhos mais curtos.

Uma forma de descobrir se um dado grafo é conexo é verificando se, para todo par de vértices, existe um caminho entre eles (da definição de conexidade). Mas veja que no caso de grafos grandes essa abordagem pode consumir muito tempo porque o número de caminhos entre os pares pode ser muito grande. Considere uma árvore T que é subgrafo de um grafo G. Se $V(T) = V(G)$, então T é geradora e podemos concluir que G é conexo. Se $V(T) \neq V(G)$, então existem duas possibilidades: não há arestas entre $V(T)$ e $V(G) \setminus V(T)$ em G, caso em que G é desconexo, ou há. Nesse último caso, para qualquer aresta $xy \in E(G)$ com $x \in V(T)$ e $y \in V(G) \setminus V(T)$ vale que $T + xy$ é também uma árvore contida em G.

A discussão acima nos dá uma base para um algoritmo eficiente para testar conexidade de qualquer grafo. Comece com uma árvore trivial (um único vértice s) e aumente-a como descrito acima. Esse procedimento terminará com uma árvore geradora do grafo ou com uma árvore geradora de uma componente conexa do grafo. Procedimentos assim costumam ser chamados de *busca* e a árvore resultante é chamada de *árvore de busca*. A Figura 24.1

exemplifica essa ideia, que é formalizada no Algoritmo 24.1.

Algoritmo 24.1: BUSCA(G, s)

1 Seja T um grafo com $V(T) = \{s\}$ e $E(T) = \emptyset$
2 **enquanto** *há arestas de G entre $V(T)$ e $V(G) \setminus V(T)$* **faça**
3 Seja xy uma aresta com $x \in V(T)$ e $y \in V(G) \setminus V(T)$
4 $V(T) = V(T) \cup \{y\}$
5 $E(T) = E(T) \cup \{xy\}$
6 **devolve** T

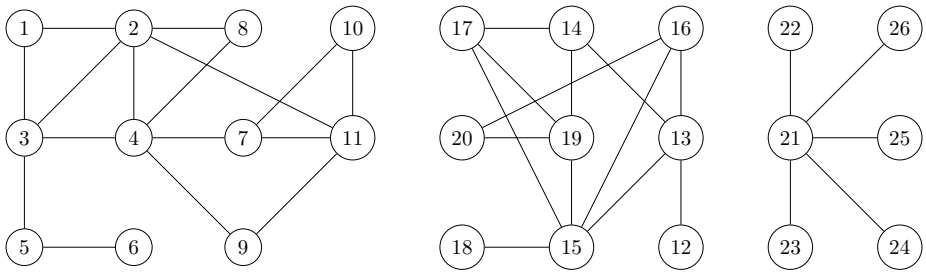
Na prática, nem sempre se constrói uma árvore explicitamente, mas se faz uso de uma terminologia comum. Seja G um grafo e s um vértice qualquer de G. Seja T a árvore devolvida por BUSCA(G, s). Podemos considerar que T está *enraizada* em s. Se $v \in V(G)$, com $v \neq s$, todo vértice no sv-caminho é um *ancestral* de v. O ancestral imediato de $v \neq s$ é seu *predecessor*, que será armazenado em $v.\texttt{predecessor}$. Com isso, temos que $E(T) = \{\{v.\texttt{predecessor}, v\} \colon v \in V(T) \setminus \{s\}\}$, motivo pelo qual não é necessário construir a árvore explicitamente. Cada vértice terá ainda um campo $v.\texttt{visitado}$, cujo valor será 1 se ele já foi adicionado a T, e será 0 caso contrário. O Algoritmo 24.3 formaliza essa ideia. Considere que outro algoritmo, como por exemplo o CHAMABUSCA, apresentado no Algoritmo 24.2, inicializou os campos `visitado` e `predecessor`, já que inicialmente nenhum vértice está visitado e não há informação sobre predecessores. Essa tarefa não faz parte de BUSCA(G, s), pois seu único objetivo é visitar todos os vértices da mesma componente conexa de s.

Algoritmo 24.2: CHAMABUSCA(G)

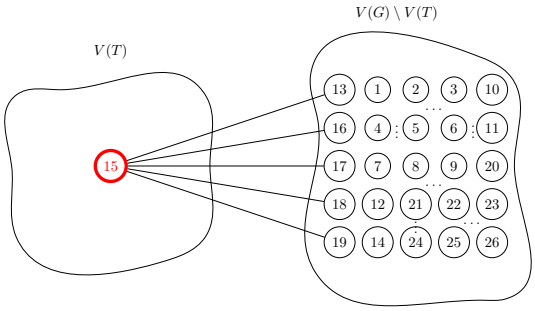
1 **para** *todo vértice $v \in V(G)$* **faça**
2 $v.\texttt{visitado} = 0$
3 $v.\texttt{predecessor} = null$
4 seja $s \in V(G)$ qualquer
5 BUSCA(G, s)

Uma vez executada BUSCA(G, s), pode-se construir o sv-caminho dado pela árvore de busca, mesmo que ela não tenha sido construída explicitamente. Isso porque tal caminho é $(s, \ldots, v.\texttt{predecessor}.\texttt{predecessor}, v.\texttt{predecessor}, v)$. O Algoritmo 24.4, CONSTROICAMINHO(G, s, v), devolve uma lista com um sv-caminho caso exista, ou vazia caso contrário. Ele deve ser executado após a execução de BUSCA(G, s).

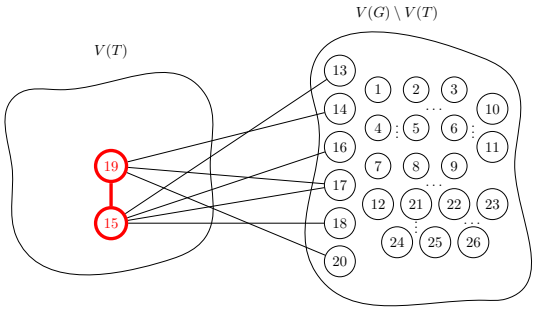
Note que encontrar uma aresta xy na linha 3 do Algoritmo 24.3, BUSCA, envolve percorrer



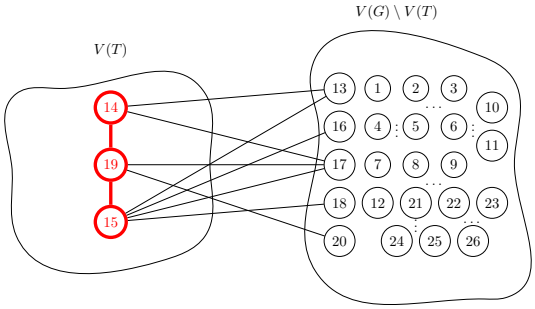
(a) Grafo G de entrada.



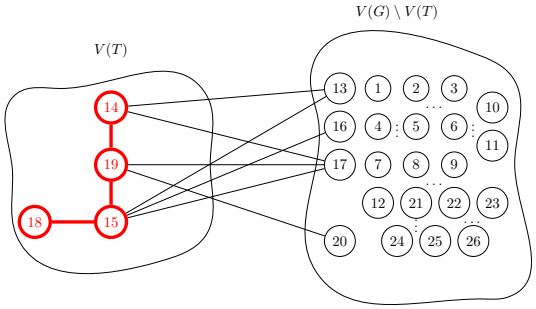
(b) O vértice 15 inicia T. Existem 5 arestas entre $V(T)$ e $V(G) \setminus V(T)$.



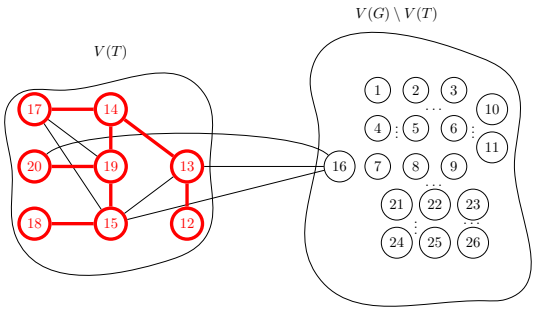
(c) A aresta 15 19 foi escolhida arbitrariamente. Agora existem 7 arestas entre $V(T)$ e $V(G) \setminus V(T)$.



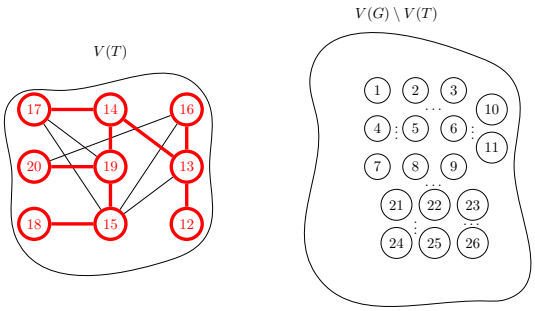
(d) A aresta 19 14 foi escolhida arbitrariamente. Agora existem 8 arestas entre $V(T)$ e $V(G) \setminus V(T)$.



(e) A aresta 15 18 foi escolhida arbitrariamente. Agora existem 7 arestas entre $V(T)$ e $V(G) \setminus V(T)$.



(f) Árvore T após algumas iterações. Agora existem 3 arestas entre $V(T)$ e $V(G) \setminus V(T)$.



(g) A aresta 13 16 foi escolhida arbitrariamente. Não há mais arestas entre $V(T)$ e $V(G) \setminus V(T)$.

Figura 24.1: Ideia da execução de Busca(G, 15). A árvore T está destacada em vermelho. As arestas entre os vértices de $V(G) \setminus V(T)$ estão omitidas.

Algoritmo 24.3: BUSCA(G, s)

```
1 s.visitado = 1
2 enquanto houver aresta com um extremo visitado e outro não faça
3     Seja xy uma aresta com x.visitado == 1 e y.visitado == 0
4     y.visitado = 1
5     y.predecessor = x
```

Algoritmo 24.4: CONSTROICAMINHO(G, s, v)

```
1 seja L uma lista vazia
2 se v.visitado == 0 então
3     devolve L
4 atual = v
5 enquanto atual ≠ s faça
6     InsereNoInicioLista(L, atual)
7     atual = atual.predecessor
8 InsereNoInicioLista(L, s)
9 devolve L
```

a vizinhança dos vértices que já estão visitados, uma a uma, para determinar qual vértice e aresta podem ser adicionados. No exemplo da Figura 24.1 isso foi feito arbitrariamente. Se o seu objetivo é apenas determinar se um grafo é conexo, então qualquer algoritmo de busca serve. Isto é, a ordem em que as vizinhanças dos vértices já visitados são consideradas não importa. No entanto, algoritmos de busca nos quais critérios específicos são utilizados para determinar tal ordem podem prover informação adicional sobre a estrutura do grafo.

Um algoritmo de busca no qual os vértices já visitados são consideradas no estilo "primeiro a entrar, primeiro a sair", ou seja, considera-se primeiro o vértice que foi marcado como visitado há mais tempo, é chamado de *busca em largura* (ou BFS, de *breadth-first search*). A BFS pode ser usada para encontrar as distâncias em um grafo não ponderado, por exemplo.

Já um algoritmo no qual os vértices já visitados são consideradas no estilo "último a entrar, primeiro a sair", ou seja, considera-se o primeiro o vértice que foi marcado como visitado há menos tempo, é chamado de *busca em profundidade* (ou DFS, de *depth-first search*). A DFS pode ser usada para encontrar os vértices e arestas de corte de um grafo, por exemplo, que são arestas e vértices que quando removidos aumentam o número de componentes conexas do grafo.

Nas seções a seguir veremos detalhes dessas duas buscas mais básicas e de outras aplica-

ções de ambas. Na Seção 24.4 discutimos buscas em digrafos.

24.1 Busca em largura

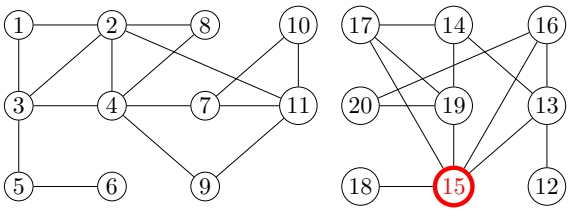
Dado um grafo G e um vértice $s \in V(G)$, o algoritmo de *busca em largura* (BFS, de *breadth-first search*) visita todos os vértices v para os quais existe um sv-caminho. Ele faz isso explorando a vizinhança dos vértices já visitados no estilo "primeiro a entrar, primeiro a sair", o que faz com que os vértices sejam explorados em camadas. O vértice inicial s está na primeira camada, seus vizinhos estão na segunda camada, os vizinhos destes que não foram explorados estão na terceira camada e assim por diante. Como veremos mais adiante, existe uma correspondência direta entre essas camadas e a distância entre s e os vértices do grafo.

Para explorar os vértices de G dessa maneira, vamos utilizar uma *fila* (veja o Capítulo 10 para mais informações sobre filas) para manter os vértices já visitados. Inicialmente, visitamos o vértice s e o enfileiramos. Enquanto a fila não estiver vazia, repetimos o procedimento de visitar e inserir na fila todos os vértices não visitados que são vizinhos do vértice u que está no início da fila. Esse vértice u pode então ser removido da fila. Note que, após s, os próximos vértices inseridos na fila, em ordem, são exatamente os vizinhos de s, em seguida os vizinhos dos vizinhos de s, e assim por diante.

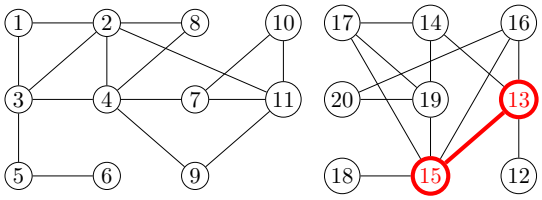
Também utilizaremos, para cada vértice u, os atributos $u.$`predecessor` e $u.$`visitado`. O atributo $u.$`predecessor` indica qual vértice antecede u no su-caminho que está sendo produzido pelo algoritmo. Em particular, ele é o vértice que levou u a ser inserido na fila. Como já vimos, esse atributo nos auxilia a descrever um su-caminho. Já o atributo $u.$`visitado` tem valor 1 se o vértice u já foi visitado pelo algoritmo e 0 caso contrário. O Algoritmo 24.5 mostra o pseudocódigo para a busca em largura. Lembre-se que qualquer função sobre a fila leva tempo $\Theta(1)$ para ser executada. Novamente considere que um outro algoritmo, como por exemplo o ChamaBusca (Algoritmo 24.2), inicializou os campos `visitado` e `predecessor`, uma vez que essa tarefa não faz parte da busca.

Vamos agora explicar o algoritmo BuscaLargura em detalhes. O algoritmo começa marcando o vértice s como visitado, cria uma fila F e enfileira s em F. Enquanto houver vértices na fila, o algoritmo desenfileira um vértice, chamado de u; para todo vizinho v de u que ainda não foi visitado, ele é marcado como visitado, atualiza-se $v.$`predecessor` com u e enfileira-se v. Na Figura 24.2 simulamos uma execução da busca em largura.

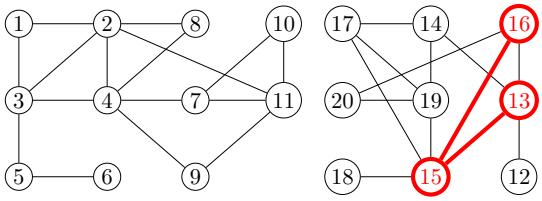
Seja G um grafo e $s \in V(G)$ qualquer. Vamos analisar o tempo de execução de BuscaLargura(G, s). Sejam $V_s(G)$ e $E_s(G)$ os conjuntos de vértices e arestas, respectivamente, que estão na componente que contém s. Sejam $n_s = |V_s(G)|$, $m_s = |E_s(G)|$, $n = v(G)$ e $m = e(G)$. Na inicialização (linhas 1 a 3) é gasto tempo total $\Theta(1)$. Note que antes de



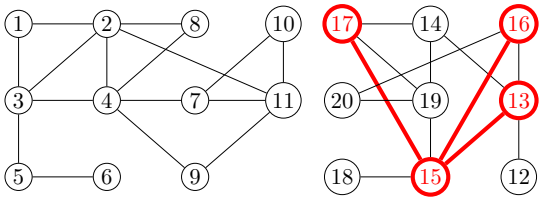
(a) Grafo G de entrada e vértice inicial $s = 15$. Fila inicial: $F = (15)$.



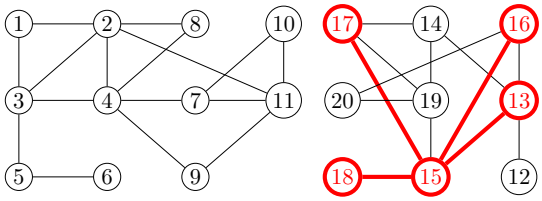
(b) Fila atual: $F = (13)$.



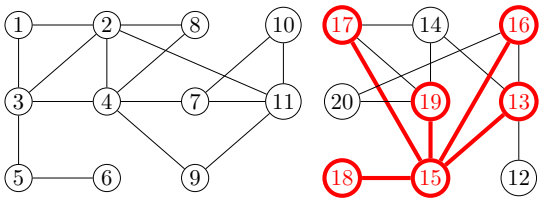
(c) Fila atual: $F = (13, 16)$.



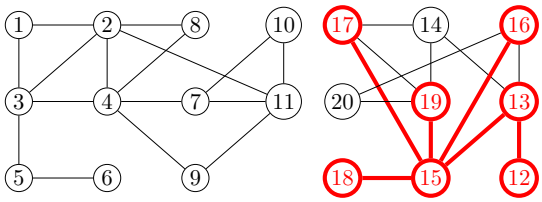
(d) Fila atual: $F = (13, 16, 17)$.



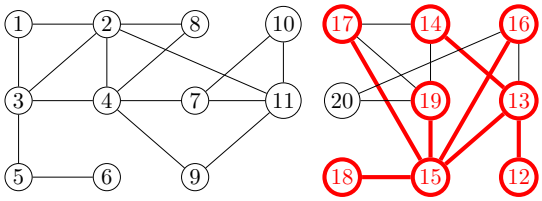
(e) Fila atual: $F = (13, 16, 17, 18)$.



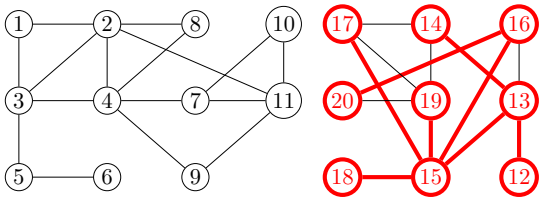
(f) Fila atual: $F = (13, 16, 17, 18, 19)$.



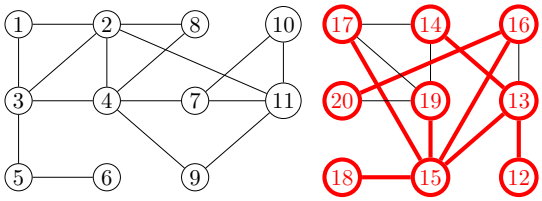
(g) Fila atual: $F = (16, 17, 18, 19, 12)$.



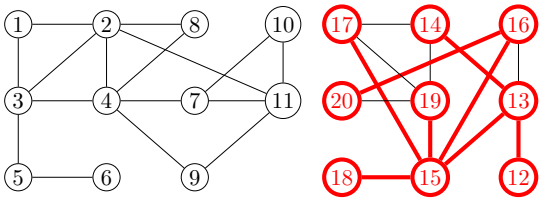
(h) Fila atual: $F = (16, 17, 18, 19, 12, 14)$.



(i) Fila atual: $F = (17, 18, 19, 12, 14, 20)$.



(j) Fila atual: $F = (18, 19, 12, 14, 20)$.



(k) Fila atual: $F = (19, 12, 14, 20)$.

Figura 24.2: Execução de BuscaLargura(G, 15). Visitamos os vizinhos dos vértices ordem numérica crescente. Vértices visitados estão em vermelho. A árvore construída de forma indireta pelos predecessores está em vermelho. Após 24.2k, a fila é esvaziada e nenhum outro vértice é marcado.

Algoritmo 24.5: BUSCALARGURA(G, s)

```
1  s.visitado = 1
2  cria fila vazia F
3  ENFILEIRA(F, s)
4  enquanto F.tamanho > 0 faça
5      u = DESENFILEIRA(F)
6      para todo vértice v ∈ N(u) faça
7          se v.visitado == 0 então
8              v.visitado = 1
9              v.predecessor = u
10             ENFILEIRA(F, v)
```

um vértice v ser enfileirado, atualizamos $v.\texttt{visitado}$ de 0 para 1 (linha 8) e tal atributo não é modificado novamente. Portanto, todo vértice para o qual existe um sv-caminho entra somente uma vez na fila e nunca mais passará no teste da linha 7. Como a linha 5 sempre remove alguém da fila, o teste do laço **enquanto** (linha 4) é executado $n_s + 1$ vezes e a chamada a DESENFILEIRA (linha 5) é executada n_s vezes.

Resta então analisar a quantidade de vezes que o conteúdo do laço **para** da linha 6 é executado. Note que aqui a estrutura utilizada para implementação do grafo pode fazer diferença. Se utilizarmos matriz de adjacências, então o laço **para** (linha 6) é executado $\Theta(n)$ vezes em cada iteração do laço **enquanto**, o que leva a um tempo de execução total de $\Theta(n_s)+\Theta(n_s n) = \Theta(n_s n) = O(n^2)$. Porém, se utilizarmos listas de adjacências, então o laço **para** é executado apenas $|N(u)|$ vezes, de modo que, no total, ele é executado $\sum_{u \in V_s(G)} |N(u)| = 2m_s$ vezes, e então o tempo total de execução do algoritmo é $\Theta(n_s)+\Theta(m_s) = \Theta(n_s+m_s) = O(n+m)$. Aqui vemos que o uso de listas de adjacência fornece uma implementação mais eficiente, pois m pode ser pequeno quando comparado a n.

Por fim, note que a árvore T tal que

$$V(T) = \{v \in V(G) \colon v.\texttt{predecessor} \neq null\} \cup \{s\}$$
$$E(T) = \{\{v.\texttt{predecessor}, v\} \colon v \in V(T) \setminus \{s\}\}$$

é uma árvore geradora de G, contém um único sv-caminho para qualquer $v \in V(T)$ e é chamada de *árvore de busca em largura*.

Lembre-se que tal caminho pode ser construído pelo Algoritmo 24.4, CONSTROICAMINHO.

24.1.1 Distância em grafos não ponderados

Seja G um grafo não ponderado e $s \in V(G)$ um vértice qualquer. Ao executar sobre G a partir de s, o algoritmo de busca em largura visita os vértices seguindo por arestas a partir de s, construindo caminhos de s aos outros vértices. Assim, durante esse processo, o algoritmo pode facilmente calcular a quantidade de arestas que seguiu entre s e v, para todo vértice $v \in V(G)$, mantendo esse valor no atributo $v.\texttt{distancia}$. O Algoritmo 24.6 contém apenas duas diferenças com relação ao algoritmo que havíamos apresentado anteriormente: as linhas 2 e 10. Novamente considere que um outro algoritmo, como por exemplo o ChamaBusca, inicializou os campos `visitado` com 0, `predecessor` com *null*, e `distancia` com ∞.

Algoritmo 24.6: BuscaLarguraDistancia($G = (V, E)$, s)

```
1  s.visitado = 1
2  s.distancia = 0
3  cria fila vazia F
4  Enfileira(F, s)
5  enquanto F.tamanho > 0 faça
6      u = Desenfileira(F)
7      para todo vértice v ∈ N(u) faça
8          se v.visitado == 0 então
9              v.visitado = 1
10             v.distancia = u.distancia + 1
11             v.predecessor = u
12             Enfileira(F, v)
```

Seja T a árvore de busca em largura gerada por BuscaLarguraDistancia(G, s). Em T existe um único sv-caminho, para qualquer $v \in V(T)$, e note que esse caminho contém exatamente $v.\texttt{distancia}$ arestas. A seguir mostramos que, ao fim de BuscaLarguraDistancia(G, s), o atributo $v.\texttt{distancia}$ contém de fato a *distância* entre s e v, para todo vértice $v \in V(G)$ (veja Seção 23.9 sobre distância).

Antes, vamos precisar de um resultado auxiliar, dado pelo Lema 24.1, que garante que os atributos `distancia` de vértices que estão na fila são próximos uns dos outros. Em particular, se um vértice u entra na fila antes de um vértice v, então no momento em que v é adicionado à fila temos $u.\texttt{distancia} \leq v.\texttt{distancia}$.

Lema 24.1

Sejam G um grafo e $s \in V(G)$. Na execução de BuscaLarguraDistancia(G, s), se u e v são dois vértices que estão na fila e u entrou na fila antes de v, então

$$u.\texttt{distancia} \leq v.\texttt{distancia} \leq u.\texttt{distancia} + 1 \ .$$

Demonstração. Vamos mostrar o resultado por indução na quantidade de iterações do laço **enquanto** na execução de BuscaLarguraDistancia(G, s).

Como caso base, considere que houve zero iterações do laço. Nesse caso, a fila possui apenas s e não há o que provar.

Suponha agora que logo após a $(\ell - 1)$-ésima iteração do laço **enquanto** a fila é $F = (x_1, \ldots, x_k)$ e que vale temos $x_i.\texttt{distancia} \leq x_j.\texttt{distancia} \leq x_i.\texttt{distancia} + 1$ para todos os pares x_i e x_j com $i < j$ (isto é, x_i entrou na fila antes de x_j).

Considere agora a ℓ-ésima iteração do laço **enquanto**. Note que $F = (x_1, \ldots, x_k)$ no início dessa iteração. Durante a iteração, o algoritmo remove x_1 de F e adiciona seus vizinhos não visitados, digamos $w_1, \ldots, w_h$ a F, de modo que agora temos $F = (x_2, \ldots, x_k, w_1, \ldots, w_h)$. Ademais, o algoritmo fez $w_j.\texttt{distancia} = x_1.\texttt{distancia} + 1$ para todo vizinho w_j não visitado de x_1. Utilizando a hipótese de indução, sabemos que para todo $1 \leq i \leq k$ temos

$$x_1.\texttt{distancia} \leq x_i.\texttt{distancia} \leq x_1.\texttt{distancia} + 1 \ .$$

Assim, para qualquer vizinho w_j de x_1 temos, pela desigualdade acima, que, para todo $2 \leq i \leq k$,

$$x_i.\texttt{distancia} \leq x_1.\texttt{distancia} + 1 = w_j.\texttt{distancia} = x_1.\texttt{distancia} + 1 \leq x_i.\texttt{distancia} + 1 \ .$$

Portanto, pares de vértices do tipo x_i, w_j satisfazem a conclusão do lema, para quaisquer $2 \leq i \leq k$ e $1 \leq j \leq h$. Já sabíamos, por hipótese de indução, que pares do tipo x_i, x_j também satisfazem a conclusão do lema. Ademais, pares de vizinhos de x_1, do tipo w_i, w_j, também satisfazem pois têm a mesma estimativa de distância ($x_1.\texttt{distancia} + 1$). Portanto, todos os pares de vértices em $\{x_2, \ldots, x_k, w_1, \ldots, w_h\}$ satisfazem a conclusão do lema. $\square$

Note que, como um vértice não tem seu atributo `distancia` alterado mais de uma vez pelo algoritmo, a conclusão do Lemma 24.1 implica que, a qualquer momento, a fila contém zero ou mais vértices à distância k do vértice inicial s, seguidos de zero ou mais vértices à distância $k + 1$ de s. O Teorema 24.2 a seguir prova que BuscaLarguraDistancia(G, s) calcula corretamente os caminhos mais curtos entre s e todos os vértices de G.

Teorema 24.2

Sejam G um grafo e $s \in V(G)$. Ao fim de BuscaLarguraDistancia(G, s), para todo $v \in V(G)$ vale que $v.\texttt{distancia} = \text{dist}_G(s, v)$.

Demonstração. Comecemos mostrando que $v.\texttt{distancia} \geq \text{dist}_G(s, v)$ para todo $v \in V(G)$ por indução na quantidade k de vértices adicionados à fila. Se $k = 1$, então o único vértice adicionado à fila é s, antes do laço **enquanto** começar. Nesse ponto, temos $s.\texttt{distancia} = 0 \geq \text{dist}_G(s, s) = 0$ e $v.\texttt{distancia} = \infty \geq \text{dist}_G(s, v)$ para todo $v \neq s$, e o resultado é válido.

Suponha agora que se x é um dos primeiros $k - 1$ vértices inseridos na fila, então $x.\texttt{distancia} \geq \text{dist}_G(s, x)$. Considere o momento em que o algoritmo realiza a k-ésima inserção na fila, sendo v o vértice que foi adicionado. Note que v foi considerado no laço **para** da linha 7 por ser vizinho de algum vértice u que foi removido da fila. Então u foi um dos $k-1$ primeiros a serem inseridos na fila e, por hipótese de indução, temos que $u.\texttt{distancia} \geq \text{dist}_G(s, u)$. Note que para qualquer aresta uv temos $\text{dist}_G(s, v) \leq \text{dist}_G(s, u) + 1$. Assim, combinando esse fato com o que é feito na linha 10, obtemos

$$v.\texttt{distancia} = u.\texttt{distancia} + 1 \geq \text{dist}_G(s, u) + 1 \geq \text{dist}_G(s, v) \ .$$

Como um vértice entra na fila somente uma vez, o valor em $v.\texttt{distancia}$ não muda mais durante a execução do algoritmo. Logo, $v.\texttt{distancia} \geq \text{dist}_G(s, v)$ para todo $v \in V(G)$.

Agora mostraremos que $v.\texttt{distancia} \leq \text{dist}_G(s, v)$ para todo $v \in V(G)$. Suponha, para fins de contradição, que ao fim da execução de BuscaLarguraDistancia(G, s) existe ao menos um $x \in V(G)$ com $x.\texttt{distancia} > \text{dist}_G(s, x)$. Seja v o vértice com menor valor $\text{dist}_G(s, v)$ para o qual isso acontece, isto é, tal que $v.\texttt{distancia} > \text{dist}_G(s, v)$.

Considere um sv-caminho mínimo $(s, \ldots, u, v)$. Note que $\text{dist}_G(s, v) = \text{dist}_G(s, u) + 1$. Pela escolha de v e como $\text{dist}_G(s, u) < \text{dist}_G(s, v)$, temos $u.\texttt{distancia} = \text{dist}_G(s, u)$. Assim,

$$v.\texttt{distancia} > \text{dist}_G(s, v) = \text{dist}_G(s, u) + 1 = u.\texttt{distancia} + 1 \ . \tag{24.1}$$

Considere o momento em que BuscaLarguraDistancia(G, s) remove u de F. Se nesse momento o vértice v já estava visitado, então algum outro vizinho $w \neq u$ de v já entrou e saiu da fila, visitando v. Nesse caso, fizemos $v.\texttt{distancia} = w.\texttt{distancia} + 1$ e, pelo Lema 24.1, $w.\texttt{distancia} \leq u.\texttt{distancia}$, de forma que $v.\texttt{distancia} \leq u.\texttt{distancia} + 1$, uma contradição com (24.1). Assim, assuma que v não havia sido visitado. Nesse caso, quando v entrar na fila (certamente entra, pois é vizinho de u), teremos $v.\texttt{distancia} = u.\texttt{distancia} + 1$, que é também uma contradição com (24.1). □

24.2 Busca em profundidade

Dado um grafo G e um vértice $s \in V(G)$, o algoritmo de *busca em profundidade* (DFS, de *depth-first search*) visita todos os vértices v para os quais existe um sv-caminho, assim como na Busca em Largura. Ele faz isso explorando a vizinhança dos vértices já visitados no estilo "último a entrar, primeiro a sair", o que faz com que os vértices sejam explorados de forma "agressiva". O vértice inicial s é o primeiro visitado, algum vizinho seu é o segundo, algum vizinho ainda não visitado deste é o terceiro, e assim por diante.

Para explorar os vértices de G dessa maneira, vamos utilizar uma *pilha* (veja o Capítulo 10 para mais informações sobre pilhas) para manter os vértices já visitados. Inicialmente, visitamos o vértice s e o empilhamos. Enquanto a pilha não estiver vazia, repetimos o procedimento de visitar e inserir na pilha apenas um vértice não visitado que é vizinho do vértice u que está no topo da pilha. Esse vértice u ainda não pode, portanto, ser removido da pilha. Somente quando todos os vizinhos de u já tenham sido visitados é que u é desempilhado.

Também utilizaremos, para cada vértice u, os atributos $u.$`predecessor` e $u.$`visitado`. O atributo $u.$`predecessor` indica qual vértice antecede u no su-caminho que está sendo produzido pelo algoritmo. Em particular, ele é o vértice que levou u a ser inserido na pilha. Como já vimos, esse atributo nos auxilia a descrever um su-caminho. Já o atributo $u.$`visitado` tem valor 1 se o vértice u já foi visitado pelo algoritmo e 0 caso contrário. O Algoritmo 24.7 mostra o pseudocódigo para a busca em profundidade. Lembre-se que o procedimento Consulta(P) devolve o último elemento inserido na pilha P mas não o remove da mesma. Qualquer função sobre a pilha leva tempo $\Theta(1)$ para ser executada. Considere que um outro algoritmo, como por exemplo o ChamaBusca (Algoritmo 24.2), inicializou os campos `visitado` e `predecessor`, uma vez que essa tarefa não faz parte da busca.

Note que a única diferença de BuscaLargura para BuscaProfIterativa é a estrutura de dados que está sendo utilizada para manter os vértices visitados. Em BuscaLargura, como inserimos todos os vizinhos não visitados de um vértice u de uma única vez na fila, então u pode ser removido da fila pois não será mais necessário. Em BuscaProfIterativa, inserimos apenas um dos vizinhos não visitados de u na pilha por vez, e por isso ele é mantido na estrutura até que não tenha mais vizinhos não visitados.

Vamos agora explicar o algoritmo BuscaProfIterativa em detalhes. O algoritmo começa marcando o vértice s como visitado, cria uma pilha P e empilha s em P. Enquanto houver vértices na pilha, o algoritmo consulta o topo da pilha, sem removê-lo, chamado de u; se houver algum vizinho v de u que ainda não foi visitado, ele é marcado como visitado, atualiza-se $v.$`predecessor` com u e empilha-se v. Se u não tem vizinhos não visitados, então a exploração de u é encerrada e o mesmo é retirado da pilha. Na Figura 24.3 simulamos uma

Algoritmo 24.7: BUSCAPROFITERATIVA(G, s)

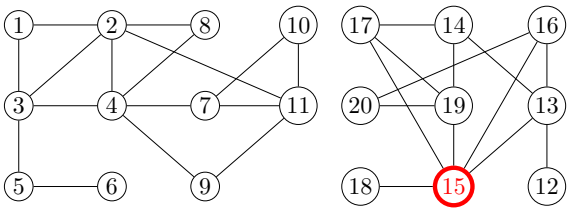
```
1  s.visitado = 1
2  cria pilha vazia P
3  EMPILHA(P, s)
4  enquanto P.tamanho > 0 faça
5      u = CONSULTA(P)
6      se existe uv ∈ E(G) com v.visitado == 0 então
7          v.visitado = 1
8          v.predecessor = u
9          EMPILHA(P, v)
10     senão
11         u = DESEMPILHA(P)
```

execução da busca em profundidade.

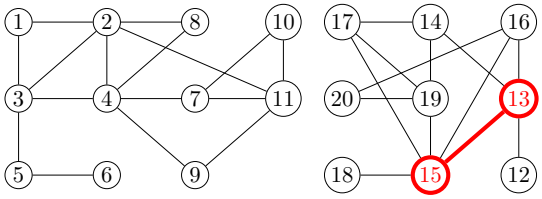
Seja G um grafo e $s \in V(G)$ qualquer. Vamos analisar o tempo de execução de BUSCA-PROFITERATIVA(G, s). Sejam $V_s(G)$ e $E_s(G)$ os conjuntos de vértices e arestas, respectivamente, que estão na componente que contém s. Sejam $n_s = |V_s(G)|$, $m_s = |E_s(G)|$, $n = v(G)$ e $m = e(G)$. Na inicialização (linhas 1 a 3) é gasto tempo total $\Theta(1)$. Note que antes de um vértice v ser empilhado, atualizamos $v.$`visitado` de 0 para 1 (linha 7) e tal atributo não é modificado novamente. Assim, uma vez que um vértice entre na pilha, ele nunca mais passará no teste da linha 6. Portanto, todo vértice para o qual existe um sv-caminho entra somente uma vez na pilha. Assim, as linhas 7, 8, 9 e 11 são executadas n_s vezes cada.

Resta então analisar a quantidade de vezes que as outras linhas do laço **enquanto** são executadas. Note que as linhas 4 e 5 têm, sozinhas, tempo de execução $\Theta(1)$ e que elas são executadas o mesmo número de vezes. Perceba que um vértice u que está na pilha será consultado $|N(u)|$ vezes, até que se visite todos os seus vizinhos. Assim, cada uma dessas linhas executa $\sum_{u \in V_s(G)} |N(u)| = 2m_s$ vezes ao todo.

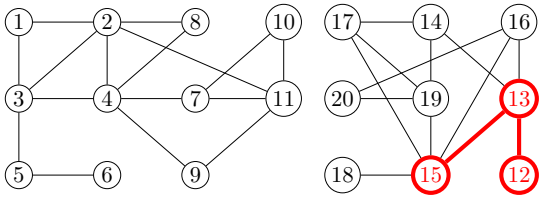
Por fim, a linha 6 esconde um laço, que é necessário para se fazer a busca por algum vértice na vizinhança de u que não esteja visitado. É aqui que a estrutura utilizada para implementação do grafo pode fazer diferença. Se usarmos matriz de adjacências, então esse "laço" é executado $\Theta(n)$ vezes cada vez que um vértice u é consultado da pilha. Se usarmos listas de adjacências, então esse "laço" é executado $\Theta(|N(u)|)$ vezes cada vez que um vértice u é consultado da pilha. Acontece que um mesmo vértice é consultado $|N(u)|$ vezes, de forma que essa implementação é ruim, independente da estrutura. Para implementar isso de maneira



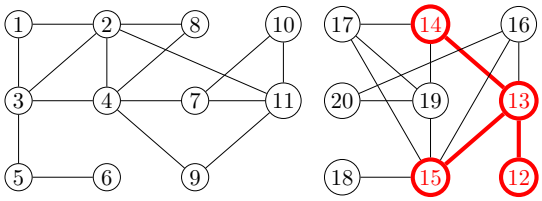
(a) Grafo G de entrada e vértice inicial $s = 15$. Pilha inicial: $P = (15)$.



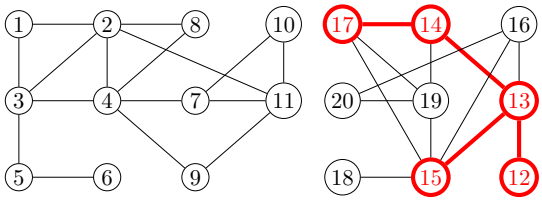
(b) Pilha atual: $P = (15, 13)$.



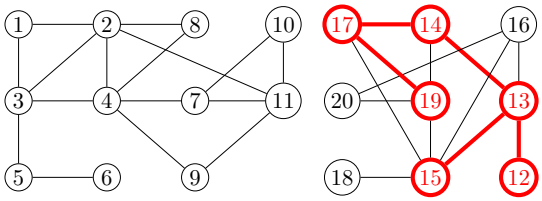
(c) Pilha atual: $P = (15, 13, 12)$.



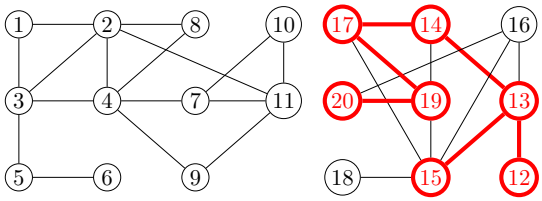
(d) Pilha atual: $P = (15, 13, 14)$.



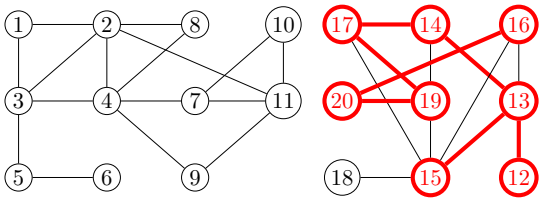
(e) Pilha atual: $P = (15, 13, 14, 17)$.



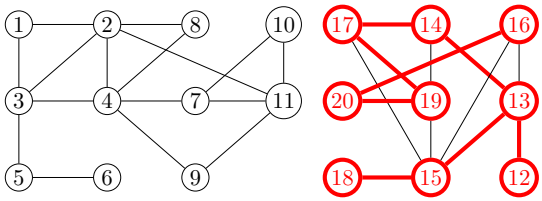
(f) Pilha atual: $P = (15, 13, 14, 17, 19)$.



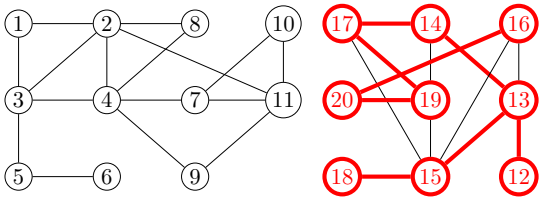
(g) Pilha atual: $P = (15, 13, 14, 17, 19, 20)$.



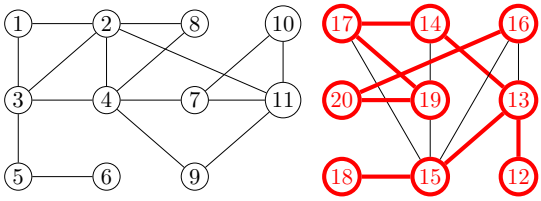
(h) Pilha atual: $P = (15, 13, 14, 17, 19, 20, 16)$.



(i) Pilha atual: $P = (15, 13, 14, 17, 19, 20, 18)$.



(j) Pilha atual: $P = (15, 13, 14, 17, 19, 20)$.



(k) Pilha atual: $P = (15, 13, 14, 17, 19)$.

Figura 24.3: Execução de BuscaProfIterativa(G, 15). Visitamos os vizinhos dos vértices ordem numérica crescente. Vértices visitados estão em vermelho. A árvore construída de forma indireta pelos predecessores está em vermelho. Após 24.3k, a pilha é esvaziada e nenhum outro vértice é marcado.

eficiente, deve-se manter, para cada vértice, a posição onde a última busca em sua vizinhança parou. Com isso, para cada vértice consultado, ao todo sua vizinhança será percorrida uma única vez. Assim, o "laço" é executado $\Theta(n)$ vezes com matriz de adjacências ou $\Theta(|N(u)|)$ vezes com listas de adjacências, mas isso *para cada vértice u da pilha*.

Somando todos os valores mencionados acima, temos o seguinte. Ao usar matriz de adjacências, o tempo é $\Theta(n_s) + \Theta(m_s) + \sum_{u \in V_s(G)} n = \Theta(n_s) + \Theta(m_s) + \Theta(nn_s) = O(n) + O(m) + (n^2) = O(n^2)$, onde a última igualdade vale porque $m = O(n^2)$ em qualquer grafo. Ao usar listas de adjacências, o tempo é $\Theta(n_s)+\Theta(m_s)+\sum_{u \in V_s(G)} |N(u)| = \Theta(n_s)+\Theta(m_s)+\Theta(m_s) = \Theta(n_s + m_s) = O(n + m)$. Aqui novamente vemos que o uso de listas de adjacência fornece uma implementação mais eficiente, pois m pode ser pequeno quando comparado a n.

Note que a árvore T tal que

$$V(T) = \{v \in V(G) \colon v.\texttt{predecessor} \neq null\} \cup \{s\}$$
$$E(T) = \{\{v.\texttt{predecessor}, v\} \colon v \in V(T) \setminus \{s\}\}$$

é uma árvore geradora de G, contém um único sv-caminho para qualquer $v \in V(T)$ e é chamada de *árvore de busca em profundidade*.

Lembre-se que tal caminho pode ser construído pelo Algoritmo 24.4, ConstroiCaminho.

Uma observação interessante é que o uso de uma estrutura *pilha* explicitamente pode ser evitado caso usemos recursão. Nesse caso, a própria pilha de recursão é aproveitada. O Algoritmo 24.8 formaliza a ideia. É importante observar que isso não altera o tempo de execução da busca em profundidade. Perceba que a primeira chamada a esse algoritmo é feita por algum outro, como por exemplo o ChamaBusca.

Algoritmo 24.8: BuscaProfRecursiva(G, s)

```
1 s.visitado = 1
2 para todo vértice v ∈ N(s) faça
3     se v.visitado == 0 então
4         v.predecessor = s
5         BuscaProfRecursiva(G, v)
```

24.2.1 Ordem de descoberta

O algoritmo de busca em profundidade serve como parte essencial de diversos outros algoritmos e tem inúmeras aplicações, práticas e teóricas. Para obter o máximo de propriedades possíveis do grafo em que a busca em profundidade é aplicada, guardaremos algumas infor-

mações ao longo de sua execução. Vamos obter, ao fim da execução da busca em profundidade em um grafo G, três listas ligadas contendo os vértices de G. São elas G.PreOrdem, G.PosOrdem e G.PosOrdemReversa.

Em G.PreOrdem os vértices da lista encontram-se na ordem em que foram visitados pelo algoritmo. Para manter essa lista basta adicionar ao fim da mesma um vértice u no momento em que o algoritmo faz $u.\texttt{visitado} = 1$. Na lista G.PosOrdem os vértices estão ordenados de acordo com o momento em que o algoritmo termina de executar a busca em todos os seus vizinhos. Assim, basta adicionar um vértice u ao fim dessa lista no momento em que o laço que percorre todos os vizinhos de u é terminado. A ordem G.PosOrdemReversa é simplesmente a lista G.PosOrdem em ordem inversa. Assim, basta adicionar um vértice u ao início dessa lista no momento em que o laço que percorre todos os vizinhos de u é terminado.

Manter as informações nas listas G.PreOrdem, G.PosOrdem e G.PosOrdemReversa torna o algoritmo útil para diversas aplicações (veja Seções 24.4.2 e 24.4.1).

O Algoritmo 24.9 apresenta BuscaProfundidade, que inclui as três listas discutidas anteriormente, onde assumimos que inicialmente temos G.PreOrdem $=$ G.PosOrdem $=$ G.PosOrdemReversa $= null$. As inserções em lista são feitas em tempo $\Theta(1)$, e portanto a complexidade de tempo do algoritmo continua a mesma.

Algoritmo 24.9: BuscaProfundidade(G, s)

```
1 InsereNoFimLista(G.PreOrdem, s)
2 s.visitado = 1
3 para todo vértice v ∈ N(s) faça
4     se v.visitado == 0 então
5         v.predecessor = s
6         BuscaProfundidade(G, v)
7 InsereNoFimLista(G.PosOrdem, s)
8 InsereNoInicioLista(G.PosOrdemReversa, s)
```

Observe que essas ordens podem também ser mantidas em vetores indexados por vértices. Assim, a posição v de um vetor conterá um número referente à ordem em que o vértice v começou a ser visitado ou terminou de ser visitado. Essa ordem é relativa aos outros vértices, de forma que esses vetores devem conter valores distintos entre 1 e $v(G)$.

24.3 Componentes conexas

Os algoritmos BuscaLargura(G, s) e BuscaProfundidade(G, s) visitam todos os vértices que estão na mesma componente conexa de s. Se o grafo é conexo, então a busca irá visitar todos os vértices do grafo. No entanto, se o grafo não é conexo, existirão ainda vértices não visitados ao fim de uma execução desses algoritmos. Assim, para encontrar todas as componentes do grafo, podemos fazer com que uma busca se inicie em um vértice de cada uma das componentes. Como não se sabe quais vértices estão em quais componentes, o que fazemos é tentar iniciar a busca a partir de todos os vértices do grafo.

O Algoritmo 24.10 apresenta BuscaComponentes, que executa as buscas em cada componente, garantindo que o algoritmo se encerra somente quando todas as componentes foram visitadas. A chamada a Busca(G, s) pode ser substituída por qualquer uma das buscas vistas, BuscaLargura ou BuscaProfundidade, e por isso foi mantida de forma genérica. Ao fim de sua execução, ele devolve a quantidade de componentes. Cada vértice v terá um atributo $v.$`componente`, que indicará o vértice representante de sua componente (no nosso caso será o vértice no qual a busca se originou). Assim, é fácil testar se dois vértices x e y estão na mesma componente conexa, pois isso ocorrerá se $x.$`componente` $= y.$`componente`.

Para o bom funcionamento de BuscaComponentes, a única alteração necessária nos algoritmos de busca em largura e profundidade é adicionar um comando que atribua um valor a $v.$`componente` para cada vértice v. Em qualquer caso, se um vértice v foi levado a ser visitado por um vértice u (caso em que $v.$`predecessor` $= u$), faça $v.$`componente` $= u.$`componente`, uma vez que u e v estão na mesma componente.

Algoritmo 24.10: BuscaComponentes(G)

```
1  para todo vértice v ∈ V(G) faça
2      v.visitado = 0
3      v.predecessor = null
4  qtdComponentes = 0
5  para todo vértice s ∈ V(G) faça
6      se s.visitado == 0 então
7          s.visitado = 1
8          s.componente = s
9          qtdComponentes = qtdComponentes + 1
10         Busca(G, s)
11 devolve qtdComponentes
```

Vamos analisar o tempo de execução de BuscaComponentes(G). Seja $n = v(G)$ e $m = e(G)$. Para qualquer vértice $x \in V(G)$, sejam n_x e m_x a quantidade de vértices e arestas, respectivamente, da componente conexa que contém x. Note que as linhas 1, 2, 3, 5 e 6 são executadas $\Theta(n)$ vezes cada, e cada uma leva tempo constante. Já as linhas 7, 8, 9 e 10 são executadas $c(G)$ vezes cada, uma para cada uma das $c(G)$ componentes conexas de G. Dessas, note que apenas a última não leva tempo constante.

Como visto, tanto BuscaLargura(G, s) quanto BuscaProfundidade(G, s) levam tempo $\Theta(n_s s)$, se implementadas com matriz de adjacências, ou $\Theta(n_s+m_s)$, se implementadas com listas de adjacências. Ademais, se X é o conjunto de vértices para o qual houve uma chamada a Busca na linha 10, então note que $\sum_{s\in X} n_s = n$ e $\sum_{s\in X} m_s = m$. Logo, o tempo gasto ao todo por todas as chamadas às buscas é $\sum_{s\in X} \Theta(n_s n) = \Theta(n^2)$, em matriz de adjacências, ou é $\sum_{s\in X} \Theta(n_s + m_s) = \Theta(n + m)$, em listas de adjacência.

Somando todas as linhas, o tempo total de BuscaComponentes(G) é $\Theta(n^2)$ em matrizes de adjacências e $\Theta(n + m)$ em listas de adjacências.

24.4 Busca em digrafos

Como vimos até agora, a busca em largura e em profundidade exploram um grafo por meio do crescimento (implícito) de uma árvore. Essencialmente os mesmos algoritmos vistos podem ser usados em digrafos, especialmente se pensarmos que todo digrafo tem um grafo subjacente. No entanto, é mais coerente explorar um digrafo por meio do crescimento de uma arborescência. Para isso, basta modificar os algoritmos vistos para que eles considerem os vizinhos de saída dos vértices que estão sendo explorados. Os Algoritmos 24.11 e 24.12 formalizam a adaptação das buscas em largura e profundidade, respectivamente, quando um digrafo é recebido. As Figuras 24.4 e 24.5 dão exemplos de execução desses algoritmos.

A busca em largura continua, de fato, encontrando distâncias mínimas, uma vez que a definição de distância considera caminhos (orientados) e as buscas seguem caminhos. No entanto, não necessariamente é possível obter uma arborescência geradora de um digrafo, por exemplo, ou mesmo detectar componentes conexas ou fortemente conexas. Isso fica claro com os exemplos das Figuras 24.4 e 24.5. O digrafo dessas figuras tem apenas uma componente conexa, que certamente não foi descoberta nas buscas. Ademais, ele possui 4 componentes fortemente conexas, sendo uma formada pelos vértices $\{1, 2, 4, 5\}$, porém ambas as buscas visitaram os vértices $\{1, 2, 3, 4, 5\}$.

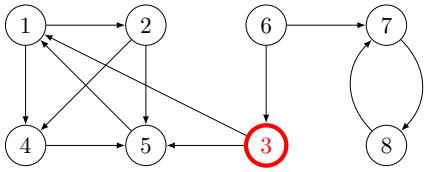
No entanto, a busca em profundidade, pela sua natureza "agressiva" de funcionamento, pode de fato ser utilizada para encontrar componentes fortemente conexas. Ela também pode resolver outros problemas específicos em digrafos. As seções a seguir discutem alguns deles.

Algoritmo 24.11: BuscaLargura(D, s)

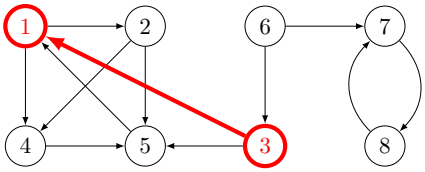
```
1  s.visitado = 1
2  cria fila vazia F
3  Enfileira(F, s)
4  enquanto F.tamanho > 0 faça
5      u = Desenfileira(F)
6      para todo vértice v ∈ N⁺(u) faça
7          se v.visitado == 0 então
8              v.visitado = 1
9              v.predecessor = u
10             Enfileira(F, v)
```

Algoritmo 24.12: BuscaProfundidade(D, s)

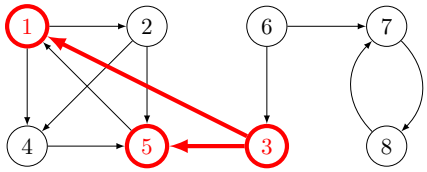
```
1  InsereNoFimLista(D.PreOrdem, s)
2  s.visitado = 1
3  para todo vértice v ∈ N⁺(s) faça
4      se v.visitado == 0 então
5          v.predecessor = s
6          BuscaProfundidade(D, v)
7  InsereNoFimLista(D.PosOrdem, s)
8  InsereNoInicioLista(D.PosOrdemReversa, s)
```



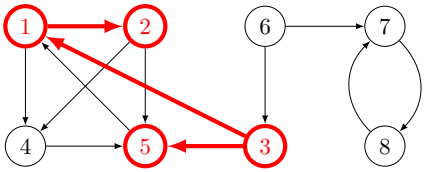
(a) Digrafo D de entrada e vértice inicial $s = 3$. Fila: $F = (3)$.



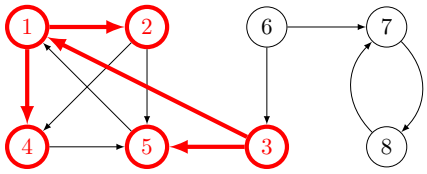
(b) Fila atual: $F = (1)$.



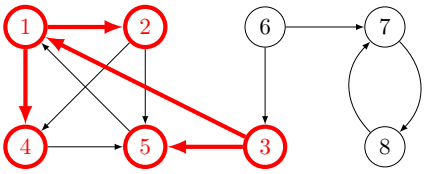
(c) Fila atual: $F = (1, 5)$.



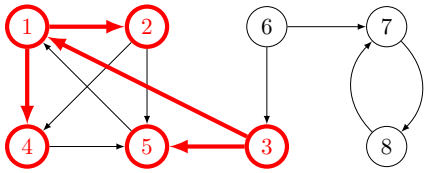
(d) Fila atual: $F = (5, 2)$.



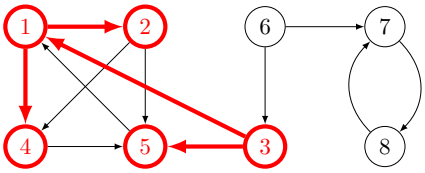
(e) Fila atual: $F = (5, 2, 4)$.



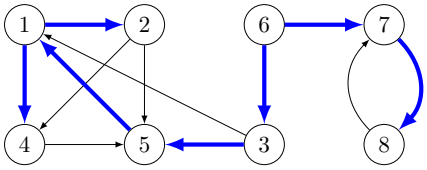
(f) Fila atual: $F = (2, 4)$.



(g) Fila atual: $F = (4)$.

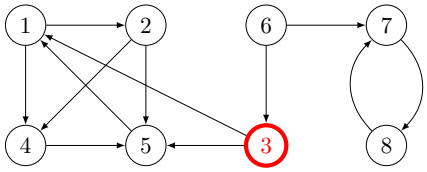


(h) Fila atual: $F = ()$.

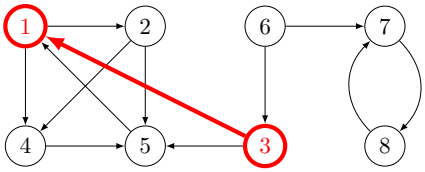


(i) Arborescência geradora de D.

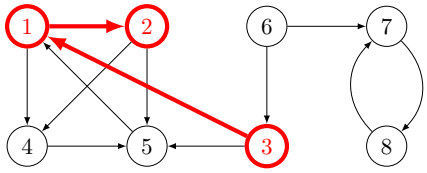
Figura 24.4: Execução de BUSCALARGURA(D, 3). Os vértices visitados estão em vermelho. A arborescência construída de forma indireta pelos predecessores está em vermelho. Note que no fim não temos uma arborescência geradora (mas uma existe, como mostra 24.4i).



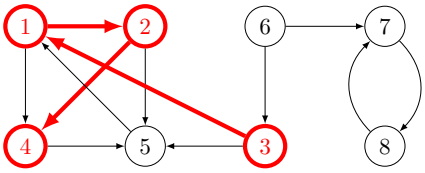
(a) Digrafo D de entrada e vértice inicial $s = 3$.



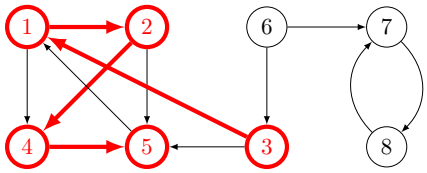
(b) Visita vizinho do 3: 1.



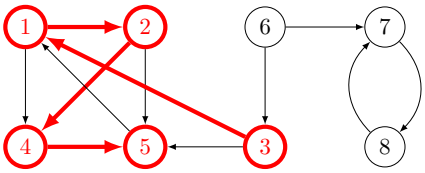
(c) Visita vizinho do 1: 2.



(d) Visita vizinho do 2: 4.



(e) Visita vizinho do 4: 5.



(f) Nenhum vértice é visitado mais.

Figura 24.5: Execução de BuscaProfundidade(D, 3). Os vértices visitados estão em vermelho. A arborescência construída de forma indireta pelos predecessores está em vermelho. Note que no fim não temos uma arborescência geradora.

24.4.1 Componentes fortemente conexas

Considere novamente o digrafo D da Figura 24.5. Note que uma execução da busca em profundidade com início no vértice 5 marcará $\{1, 2, 4, 5\}$ apenas, e eles de fato formam uma componente fortemente conexa. Em seguida, ao executarmos outra busca em profundidade com início no vértice 7, os vértices $\{7, 8\}$ serão marcados, outra componente fortemente conexa. Em seguida, ao executarmos outra busca em profundidade com início no vértice 3, apenas o próprio 3 será marcado, que é outra componente fortemente conexa. Por fim, ao executarmos outra busca em profundidade com início em 6, apenas o próprio 6 será marcado, a quarta componente fortemente conexa de D.

Pela discussão acima, é possível observar que a busca em profundidade é útil para encontrar as componentes fortemente conexas somente quando sabemos a ordem dos vértices iniciais a partir dos quais podemos tentar começar a busca. Felizmente, existe uma forma de descobrir essa ordem utilizando a própria busca em profundidade!

Na discussão a seguir, considere um digrafo D e sejam $D_1, \ldots, D_k$ todas as componentes fortemente conexas de D (cada D_i é um subdigrafo, portanto). Pela maximalidade das componentes, cada vértice pertence somente a uma componente e, mais ainda, entre quaisquer duas componentes D_i e D_j existem arestas apenas em uma direção, pois caso contrário a união de D_i e D_j formaria uma componente maior que D_i e que D_j, contradizendo a maximalidade da definição de componentes fortemente conexas. Por isso, sempre existe pelo menos uma componente D_i que é um *ralo*: não existe aresta saindo de D_i em direção a nenhuma outra componente.

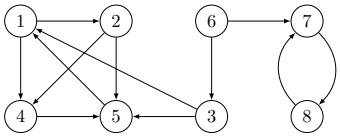
Vamos considerar ainda o digrafo $\overleftarrow{D}$, chamado *digrafo reverso de* D, que é o digrafo obtido de D invertendo a direção de todos os arcos.

O procedimento para encontrar as componentes fortemente conexas de D tem essencialmente dois passos:

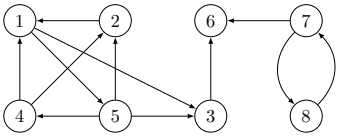
1. Execute BuscaComponentes (Algoritmo 24.10, com Busca substituída por BuscaProfundidade – Algoritmo 24.12) em $\overleftarrow{D}$: esse passo tem o objetivo único de obter a lista ordenada $\overleftarrow{D}$.PosOrdemReversa.

2. Execute BuscaProfundidade, alterada para fazer $v.$`componente` $= s.$`componente` logo após fazer $v.$`predecessor` $= s$, em D visitando os vértices de acordo com a ordem da lista $\overleftarrow{D}$.PosOrdemReversa.

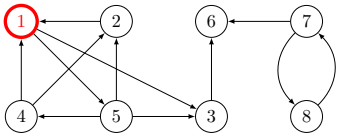
Esse procedimento está descrito formalmente no Algoritmo 24.13. Durante o segundo passo, cada chamada recursiva a BuscaProfundidade(D, u) (linha 9) identifica os vértices de uma das componentes fortemente conexas. As Figuras 24.6 e 24.7 exemplificam uma execução do algoritmo.



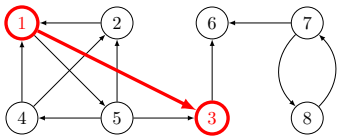
(a) Digrafo D de entrada.



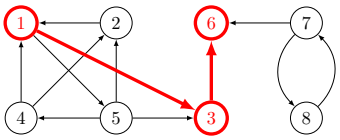
(b) Digrafo $\overleftarrow{D}$. Chama BuscaComponentes($\overleftarrow{D}$).



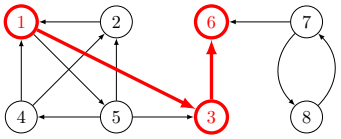
(c) Nenhum visitado. Chama BuscaProfundidade($\overleftarrow{D}$, 1).



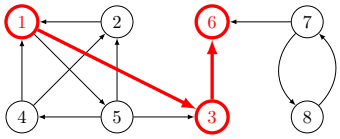
(d) Visita um vizinho do 1, o 3. $\overleftarrow{D}$.PosOrdemReversa: *null*.



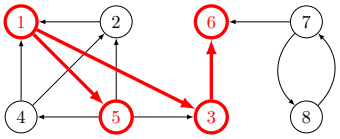
(e) Visita um vizinho do 3, o 6. $\overleftarrow{D}$.PosOrdemReversa: *null*.



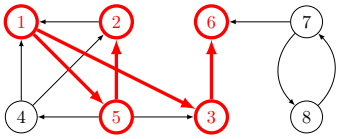
(f) Não há vizinhos do 6. $\overleftarrow{D}$.PosOrdemReversa: (6).



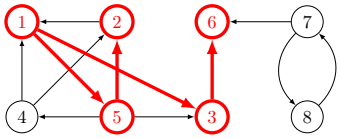
(g) Não há vizinhos do 3. $\overleftarrow{D}$.PosOrdemReversa: $(3, 6)$.



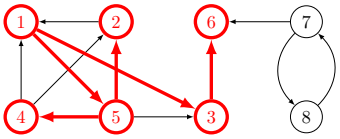
(h) Visita um vizinho do 1, o 5. $\overleftarrow{D}$.PosOrdemReversa: $(3, 6)$.



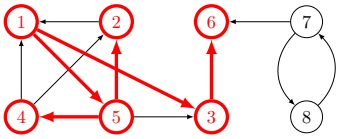
(i) Visita um vizinho do 5, o 2. $\overleftarrow{D}$.PosOrdemReversa: $(3, 6)$.



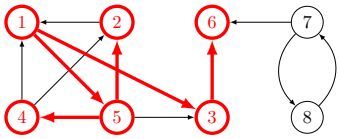
(j) Não há vizinhos do 2. $\overleftarrow{D}$.PosOrdemReversa: $(2, 3, 6)$.



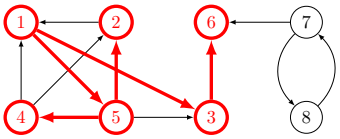
(k) Visita um vizinho do 5, o 4. $\overleftarrow{D}$.PosOrdemReversa: $(2, 3, 6)$.



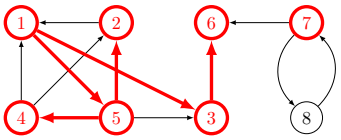
(l) Não há vizinhos do 4. $\overleftarrow{D}$.PosOrdemReversa: $(4, 2, 3, 6)$.



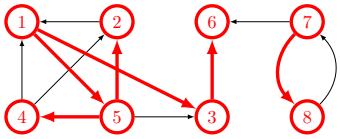
(m) Não há vizinhos do 5. $\overleftarrow{D}$.PosOrdemReversa: $(5, 4, 2, 3, 6)$.



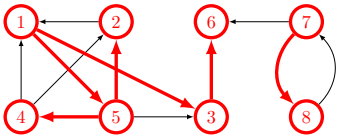
(n) Não há vizinhos do 1. $\overleftarrow{D}$.PosOrdemReversa: $(1, 5, 4, 2, 3, 6)$.



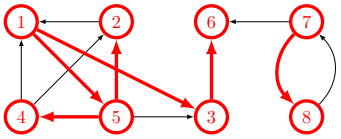
(o) Chama BuscaProfundidade($\overleftarrow{D}$, 7). $\overleftarrow{D}$.PosOrdemReversa: $(1, 5, 4, 2, 3, 6)$.



(p) Visita um vizinho do 7, o 8. $\overleftarrow{D}$.PosOrdemReversa: $(1, 5, 4, 2, 3, 6)$.

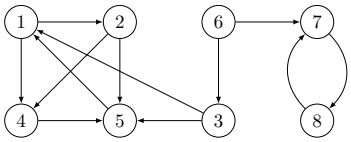


(q) Não há vizinhos do 8. $\overleftarrow{D}$.PosOrdemReversa: $(8, 1, 5, 4, 2, 3, 6)$.

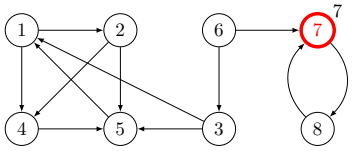


(r) Não há vizinhos do 7. $\overleftarrow{D}$.PosOrdemReversa: $(7, 8, 1, 5, 4, 2, 3, 6)$.

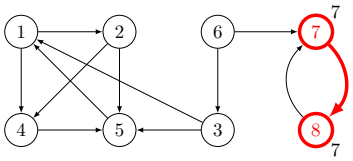
Figura 24.6: Primeira parte da execução de ComponentesFortementeConexas(D): execução de BuscaComponentes($\overleftarrow{D}$.PosOrdemReversa).



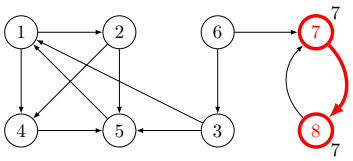
(a) Digrafo D de entrada. $\overleftarrow{D}$.PosOrdemReversa: $(\underline{7}, 8, 1, 5, 4, 2, 3, 6)$.



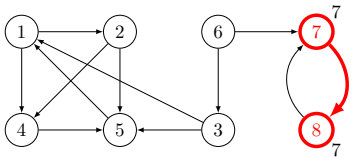
(b) Faz 7.`componente` = 7. Chama BuscaProfundidade(D, 7).



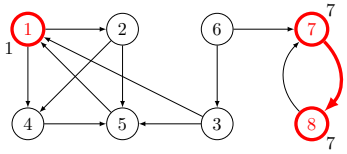
(c) Visita um vizinho do 7, o 8. Faz 8.`componente` = 7.



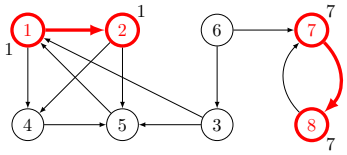
(d) Não há vizinhos do 8.



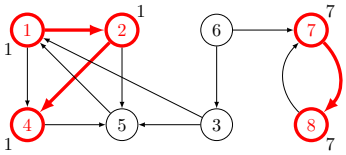
(e) Não há vizinhos do 7. $\overleftarrow{D}$.PosOrdemReversa: $(7, 8, \underline{1}, 5, 4, 2, 3, 6)$.



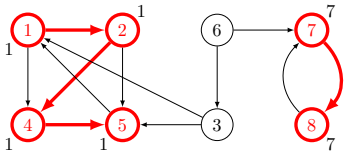
(f) Faz 1.`componente` = 1. Chama BuscaProfundidade(D, 1).



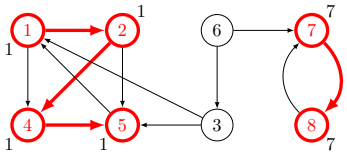
(g) Visita um vizinho do 1, o 2. Faz 2.`componente` = 1.



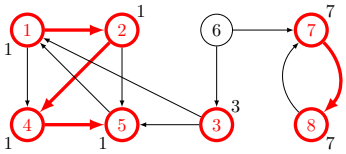
(h) Visita um vizinho do 2, o 4. Faz 4.`componente` = 1.



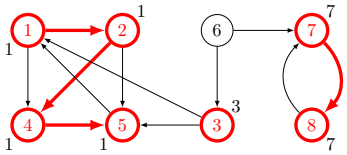
(i) Visita um vizinho do 4, o 5. Faz 5.`componente` = 1.



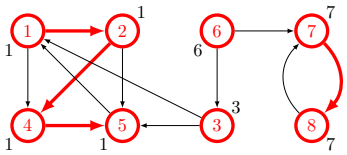
(j) Não há vizinhos do 5. $\overleftarrow{D}$.PosOrdemReversa: $(7, 8, 1, 5, 4, 2, \underline{3}, 6)$.



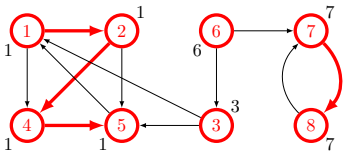
(k) Chama BuscaProfundidade(D, 3). Faz 3.`componente` = 3.



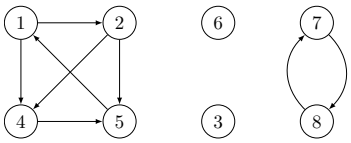
(l) Não há vizinhos do 3. $\overleftarrow{D}$.PosOrdemReversa: $(7, 8, 1, 5, 4, 2, 3, \underline{6})$.



(m) Chama BuscaProfundidade(D, 6). Faz 6.`componente` = 6.



(n) Não há vizinhos do 6.



(o) Componentes.

Figura 24.7: Segunda parte da execução de ComponentesFortementeConexas(D): execução de BuscaProfundidade sobre D na ordem de $\overleftarrow{D}$.PosOrdemReversa, encontrando as componentes. Os números ao redor dos vértices indicam seus atributos `componente`.

A intuição por trás desse procedimento é a seguinte. Após a execução de BuscaComponentes($\overleftarrow{D}$), o primeiro vértice de $\overleftarrow{D}$.PosOrdemReversa pertence a uma componente fortemente conexa D_i que é ralo em D. Logo, a primeira chamada a BuscaProfundidade no laço **enquanto** irá visitar os vértices de D_i apenas. A próxima chamada a BuscaProfundidade vai desconsiderar tal componente (pois seus vértices já foram visitados), como se estivéssemos executando-a no digrafo obtido de D pela remoção dos vértices de D_i (remoção implícita). Assim, sucessivas chamadas vão "removendo" as componentes fortemente conexas uma a uma, de forma que o procedimento encontra todas elas.

Algoritmo 24.13: ComponentesFortementeConexas(D)

```
1  para todo vértice v ∈ V(D) faça
2      v.visitado = 0
3      v.predecessor = null
4  BuscaComponentes(←D) /* Usando BuscaProfundidade no lugar de Busca */
5  u = ←D.PosOrdemReversa.cabeca
6  enquanto u ≠ null faça
7      se u.visitado == 0 então
8          u.componente = u
9          BuscaProfundidade(D, u)
10         u = u.proximo
```

Se o digrafo estiver representado com lista de adjacências, então ComponentesFortementeConexas(D) tem tempo $\Theta(v(D) + e(D))$. No Teorema 24.3 a seguir mostramos que esse algoritmo identifica corretamente as componentes fortemente conexas de D.

Teorema 24.3

Seja D um digrafo. Ao fim da execução de ComponentesFortementeConexas(D) temos que, para quaisquer $u, v \in V(D)$, os vértices u e v estão na mesma componente fortemente conexa se e somente se $u.\texttt{componente} = v.\texttt{componente}$.

Demonstração. Seja u um vértice arbitrário de D para o qual a linha 9 foi executada e seja D_u a componente fortemente conexa de D que contém u. Para provarmos o resultado do teorema, basta mostrarmos que após a chamada a BuscaProfundidade(D, u) (na linha 9), vale o seguinte:

$$v \in V(D) \text{ é visitado durante a chamada } \textsc{BuscaProfundidade}(D,\, u) \\ \text{se e somente se } v \in V(D_u)\,. \tag{24.2}$$

De fato, se (24.2) é válida, então após a execução de $\textsc{BuscaProfundidade}(D,\, u)$ teremos que os únicos vértices com $v.\texttt{componente} = u$ são os vértices que estão em D_u. Assim, para um vértice v ter $v.\texttt{componente} = u$ ele precisa ser visitado durante a chamada $\textsc{BuscaProfundidade}(D,\, u)$. Como o algoritmo $\textsc{ComponentesFortementeConexas}(D)$ só encerra sua execução quando todos os vértices são visitados, provar (24.2) é suficiente para concluir a prova do teorema.

Para provarmos (24.2), vamos primeiro mostrar a seguinte afirmação.

Afirmação 24.4

Se $v \in V(D_u)$, então v é visitado na chamada $\textsc{BuscaProfundidade}(D,\, u)$.

Demonstração. Seja $v \in V(D_u)$. Como v está na mesma componente fortemente conexa de u, então existe um vu-caminho e um uv-caminho em D, por definição. Note que, caso v já tivesse sido visitado no momento em que $\textsc{BuscaProfundidade}(D,\, u)$ é executado, então como existe vu-caminho, certamente o vértice u seria visitado antes de $\textsc{BuscaProfundidade}(D,\, u)$, de modo que a chamada a $\textsc{BuscaProfundidade}(D,\, u)$ nunca seria executada, levando a um absurdo. Portanto, sabemos que no início da execução de $\textsc{BuscaProfundidade}(D,\, u)$, o vértice v ainda não foi visitado. Logo, como existe um uv-caminho, o vértice v é visitado durante essa chamada. □

Para completar a prova, resta mostrar a seguinte afirmação.

Afirmação 24.5

Se $v \in V(D)$ foi visitado na chamada $\textsc{BuscaProfundidade}(D,\, u)$, então $v \in V(D_u)$.

Demonstração. Seja $v \in V(D)$ um vértice que foi visitado na chamada $\textsc{BuscaProfundidade}(D,\, u)$. Então existe um uv-caminho em D, e resta mostrar que existe um vu-caminho em D.

Como o laço **enquanto** visita os vértices na ordem em que eles aparecem na lista $\overleftarrow{D}.\textsc{PosOrdemReversa}$ e v foi visitado durante a chamada $\textsc{BuscaProfundidade}(D,\, u)$, isso significa que u aparece antes de v nessa lista. Portanto, quando executamos $\textsc{BuscaComponentes}(\overleftarrow{D})$ na linha 4, vale que

$$\text{a chamada BuscaProfundidade}(\overleftarrow{D},\, v) \text{ termina antes} \\ \text{do fim da execução de BuscaProfundidade}(\overleftarrow{D},\, u). \tag{24.3}$$

Analisemos agora o momento do *início* da execução de BuscaProfundidade($\overleftarrow{D}$, u). Como existe vu-caminho em $\overleftarrow{D}$ (pois existe uv-caminho em D), sabemos que BuscaProfundidade($\overleftarrow{D}$, u) não pode ter sido iniciada após o término da execução de BuscaProfundidade($\overleftarrow{D}$, v), pois nesse caso u teria sido visitado durante a execução de BuscaProfundidade($\overleftarrow{D}$, v) e, por conseguinte, BuscaProfundidade($\overleftarrow{D}$, v) terminaria depois do fim da execução de BuscaProfundidade($\overleftarrow{D}$, u), contrariando (24.3).

Assim, BuscaProfundidade($\overleftarrow{D}$, u) teve início antes de BuscaProfundidade($\overleftarrow{D}$, v) e, por (24.3), terminou depois do fim de BuscaProfundidade($\overleftarrow{D}$, v), o que significa que existe um uv-caminho em $\overleftarrow{D}$. Portanto, existe um vu-caminho em D. □

As Afirmações 24.4 e 24.5 juntamente com (24.2) provam o resultado do teorema. □

24.4.2 Ordenação topológica

Uma *ordenação topológica* de um digrafo D é uma rotulação $f\colon V(D) \to \{1, 2, \ldots, v(D)\}$ dos vértices de D tal que $f(u) \neq f(v)$ se $u \neq v$, e se $uv \in E(D)$ então $f(u) < f(v)$.

Uma ordenação topológica pode ser visualizada no plano da seguinte forma. Desenha-se os vértices em uma linha horizontal de forma que, para todo arco uv, o vértice u está à esquerda de v. A Figura 24.8 mostra um exemplo de um digrafo e sua ordenação topológica.

Soluções eficientes para diversos problemas fazem uso da ordenação topológica. Isso se dá pelo fato de muitos problemas precisarem lidar com uma certa hierarquia de pré-requisitos ou dependências. Assim, podemos pensar em cada arco uv representando uma relação de dependência, indicando que v depende de u. Por exemplo, em uma universidade, algumas disciplinas precisam que os alunos tenham conhecimento prévio adquirido em outras disciplinas. Isso pode ser modelado por meio de um digrafo no qual os vértices são as disciplinas e os arcos indicam tais pré-requisitos. Para escolher a ordem na qual cursar as disciplinas, o aluno pode fazer uso de uma ordenação topológica de tal grafo.

Comecemos observando que um digrafo admite ordenação topológica se, e somente se, ele não tem ciclos. Isto é, se não existe uma sequência de vértices $(v_1, v_2, \ldots, v_k, v_1)$ tal que $k \geq 2$ e $v_i v_{i+1}$ é arco para todo $1 \leq i < k$, e $v_k v_1$ é arco. Tal digrafo é dito *acíclico*. Note ainda que todo digrafo acíclico possui ao menos um vértice *ralo*, do qual não saem arcos.

Dado um digrafo acíclico D e um vértice $w \in V(D)$ que é ralo, note que fazer $f(w) = v(D)$ é seguro, pois não saem arcos de w, então garantidamente qualquer arco do tipo uw terá $f(u) < f(w)$. Note ainda que $D - w$ também é acíclico (se não fosse, D não seria). Então

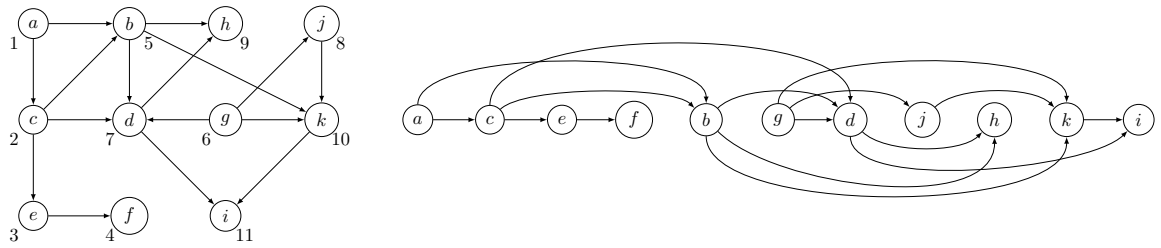


Figura 24.8: Digrafo D à esquerda e uma possível ordenação topológica sua à direita. Os números ao redor dos vértices indicam a rotulação dos mesmos.

tome um vértice $x \in V(D-w)$ que é ralo. Pelo mesmo motivo anterior, fazer $f(x) = v(D)-1$ é seguro. Perceba que esse procedimento recursivo gera uma ordenação topológica de D. Implementá-lo de forma direta pode ser bem custoso, pois a cada passo deve-se procurar por um vértice que é ralo. Felizmente, há uma forma bem eficiente de implementá-lo utilizando busca em profundidade.

O Algoritmo 24.14 promete encontrar uma ordenação topológica de um digrafo acíclico D. Ele simplesmente aplica uma busca em profundidade em D e rotula os vértices de acordo com a ordem em D.PosOrdemReversa. Intuitivamente, note que isso funciona porque um vértice que é ralo, durante uma busca em profundidade, não tem vizinhos de saída. Assim, ele é inserido em D.PosOrdemReversa antes que qualquer vértice de sua vizinhança de entrada seja (e vértices de sua vizinhança de entrada certamente devem aparecer antes dele na ordenação topológica). O Lema 24.6 prova que esse algoritmo de fato está correto.

Algoritmo 24.14: OrdenacaoTopologica(D)

```
BuscaComponentes(D) /* Usando BuscaProfundidade no lugar de Busca */
atual = D.PosOrdemReversa.cabeca
i = 1
enquanto atual ≠ null faça
    f(atual) = i
    i = i + 1
    atual = atual.proximo
devolve f
```

Lema 24.6

Dado um digrafo acíclico D, a rotulação f devolvida OrdenacaoTopologica(D) é uma ordenação topológica.

Demonstração. Por construção, temos $f(u) \neq f(v)$ para todo $u \neq v$ e $f(u) \in \{1, \ldots, v(D)\}$. Resta então provar que para qualquer arco $uv \in E(D)$, temos $f(u) < f(v)$.

Tome um arco uv qualquer e suponha primeiro que u é visitado antes de v pela busca em profundidade. Isso significa que BuscaProfundidade(D, v) termina sua execução antes de BuscaProfundidade(D, u). Dessa forma, v é incluído no início de D.PosOrdemReversa antes de u ser incluído. Portanto, u aparece antes de v nessa lista e $f(u) < f(v)$.

Suponha agora que v é visitado antes de u pela busca em profundidade. Como D é acíclico, não existe vu-caminho. Então BuscaProfundidade(D, v) não visita o vértice u e termina sua execução antes mesmo de BuscaProfundidade(D, u) começar. Dessa forma, v é incluído no início de D.PosOrdemReversa antes de u ser incluído, caso em que $f(u) < f(v)$ também. □

24.5 Outras aplicações dos algoritmos de busca

Tanto a busca em largura como a busca em profundidade podem ser aplicadas em vários problemas além dos já vistos. Alguns exemplos são testar se um dado grafo é bipartido, detectar ciclos em grafos e encontrar vértices ou arestas de corte (vértices ou arestas que quando removidos desconectam o grafo). Ademais, podem ser usados como ferramenta na implementação do método de Ford-Fulkerson, que calcula o fluxo máximo em uma rede de fluxos. Uma outra aplicação interessante da busca em profundidade é resolver de forma eficiente (tempo $O(v(G)+e(G))$) o problema de encontrar uma trilha Euleriana (Capítulo 26). Algoritmos de busca em profundidade também são utilizados para criação de labirintos.

Algoritmos importantes em grafos têm a mesma estrutura dos algoritmos de busca, mudando apenas a ordem na qual os vértices já visitados têm a vizinhança explorada. Esse é o caso do algoritmo de Prim para encontrar uma árvore geradora mínima em grafos ponderados nas arestas, e o algoritmo de Dijkstra, que encontra caminhos mínimos em grafos ponderados nas arestas (pesos não-negativos). Ao invés de fila ou pilha para armazenar os vértices já visitados, eles utilizam uma fila de prioridades.

Além de todas essas aplicações dos algoritmos de busca em problemas clássicos da Teoria de Grafos, eles continuam sendo de extrema importância no desenvolvimentos de novos algoritmos. O algoritmo de busca em profundidade, por exemplo, vem sendo muito utilizado em algoritmos que resolvem problemas em Teoria de Ramsey, uma vertente da Teoria de Grafos e Combinatória.

Capítulo 25

Árvores geradoras mínimas

Uma *árvore geradora* de um grafo G é uma árvore que é um subgrafo gerador de G, i.e., é um subgrafo conexo que não possui ciclos e contém todos os vértices de G. Como visto no Capítulo 24, os algoritmos de busca em largura e busca em profundidade podem ser utilizados para encontrar uma árvore geradora de um grafo. Porém, em muitos casos o grafo é ponderado nas arestas, de forma que diferentes árvores geradoras possuem pesos diferentes.

Dado um grafo G e uma função $w\colon E(G) \to \mathbb{R}$ de pesos nas arestas de G, dizemos que uma árvore geradora T de G tem peso $w(T) = \sum_{e \in E(T)} w(e)$. Diversas aplicações necessitam encontrar uma árvore geradora T de G que tenha peso total $w(T)$ mínimo dentre todas as árvores geradoras de G, i.e., uma árvore T tal que

$$w(T) = \min\{w(T')\colon T' \text{ é uma árvore geradora de } G\} \ .$$

Uma árvore T com essas propriedades é uma *árvore geradora mínima* de G. A Figura 25.1 exemplifica essa discussão.

Problema 25.1: *Árvore geradora mínima*

Dado um grafo G conexo e uma função $w\colon E(G) \to \mathbb{R}$, encontrar uma árvore geradora T de G cujo peso $w(T) = \sum_{e \in E(T)} w(e)$ é mínimo.

Note que podemos considerar que G é um grafo conexo pois, caso não seja, as árvores geradoras mínimas de cada componente conexa de G formam uma floresta geradora mínima para G. Assim, o problema principal ainda é encontrar uma árvore geradora mínima de um

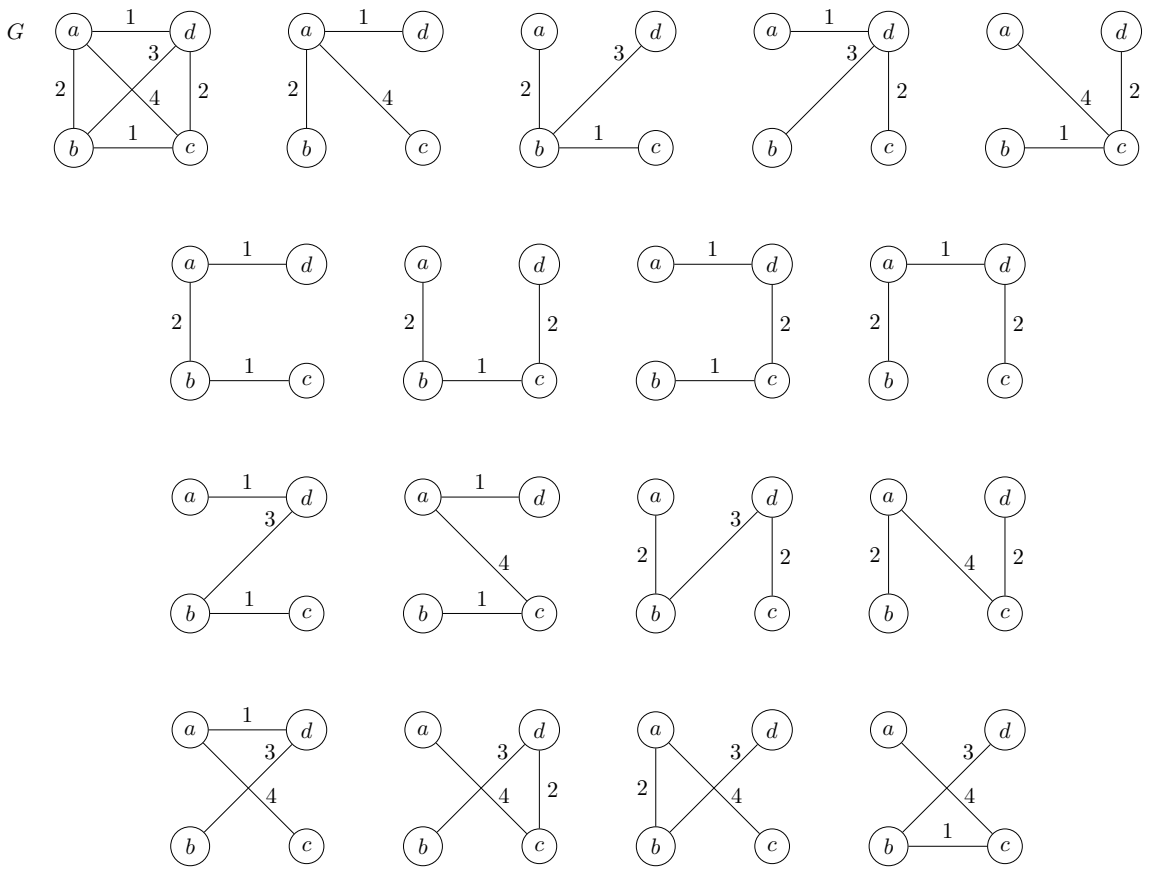


Figura 25.1: Exemplo de um grafo G ao topo e todas as suas 16 árvores geradoras. Cada uma das quatro na segunda linha é uma árvore geradora mínima.

grafo conexo.

Árvores geradoras mínimas podem ser utilizadas, por exemplo, para resolver problema de conexão em redes (de telecomunicação, de computadores, de transporte, de suprimentos). Cada elemento da rede é representado por um vértice e o custo de conectar dois elementos é indicado no peso da aresta que os conecta. Também podem ser utilizadas para resolver problema de análise de clusters, em que objetos similares devem fazem parte do mesmo cluster. Cada objeto é representado por um vértice e a similaridade de dois objetos é indicada no peso da aresta que os conecta. Em ambos os casos, uma árvore geradora mínima é uma solução para os problemas.

Na Seção 23.10.1 já vimos alguns conceitos importantes sobre árvores. Lembre-se, por exemplo, que se T é uma árvore, então $e(T) = v(T) - 1$ e que todo grafo conexo contém uma árvore geradora. Ademais, para qualquer aresta $uv \notin E(T)$ com $u, v \in V(T)$, temos que $T + e$ contém exatamente um ciclo. Os conceitos a seguir são importantes em grafos e

também são bastante úteis em árvores geradoras mínimas.

Dado um grafo G e dois conjuntos de vértices $S, R \subseteq V(G)$, se $S \cap R = \emptyset$ e $S \cup R \neq \emptyset$, então dizemos que (S, R) é um *corte* de G. Uma aresta $uv \in E(G)$ *cruza* o corte (S, R) se $u \in S$ e $v \in R$. Quando $R = V(G) \setminus S$, então denotamos por $\partial_G(S)$ o conjunto de todas as arestas que cruzam o corte $(S, V(G) \setminus S)$.

Dado um conjunto $S \subseteq V(G)$, dizemos que uma aresta de $\partial_G(S)$ é *mínima* para esse corte se ela é uma aresta de menor peso dentre todas as arestas de $\partial_G(S)$. Antes de discutirmos algoritmos para encontrar árvores geradoras mínimas vamos entender algumas características sobre arestas que cruzam cortes para obter uma estratégia gulosa para o problema. O Lema 25.2 a seguir implica que se e é a única aresta que cruza um dado corte, então e não pertence a nenhum ciclo.

Lema 25.2

Sejam H um grafo e C um ciclo de H. Se $e \in E(H)$ pertence a C e $e \in \partial_H(S)$ para algum $S \subseteq V(H)$, então existe outra aresta $f \in E(H)$ que pertence a C tal que $f \in \partial_H(S)$.

Demonstração. Seja $e = uv$ uma aresta pertencente a um ciclo C de H tal que $u \in S$ e $v \in V(G) \setminus S$ para algum $S \subseteq V(G)$. Ou seja, podemos escrever $C = (u, v, x_1, \ldots, x_k, u)$. Note que C pode ser dividido em dois caminhos distintos entre u e v. Um desses caminhos é a própria aresta $e = uv$ e o outro caminho, $(v, x_1, \ldots, x_k, u)$, necessariamente contém uma aresta $f \in \partial_H(S)$, uma vez que u e v estão em lados distintos do corte. $\square$

Considerando o resultado do Lema 25.2, o Teorema 25.3 a seguir fornece uma estratégia para se obter uma árvore geradora mínima de qualquer grafo.

Teorema 25.3

Sejam G um grafo conexo e $w\colon E(G) \to \mathbb{R}$ uma função de pesos. Seja $S \in V(G)$ qualquer. Se $e \in \partial_G(S)$ é uma aresta mínima do corte $(S, V(G) \setminus S)$, então existe uma árvore geradora mínima de G que contém e.

Demonstração. Sejam G um grafo conexo e $w\colon E(G) \to \mathbb{R}$ uma função de pesos. Considere uma árvore geradora mínima T de G e seja $S \subseteq V(G)$ qualquer.

Seja $e = uv \in E(G)$ uma aresta mínima que cruza o corte $(S, V(G) \setminus S)$, isto é, $w(e) = \min_{f \in \partial_G(S)}\{w(f)\}$. Suponha, para fins de contradição, que e não está em nenhuma árvore geradora mínima de G.

Como T é uma árvore geradora, então $e \notin E(T)$ e, portanto, $T + e$ tem exatamente um ciclo. Assim, pelo Lema 25.2, sabemos que existe outra aresta $f \in E(T)$ que está no ciclo e cruza o corte $(S, V(G) \setminus S)$. Portanto, o grafo $T' = T + e - f$ é uma árvore geradora.

Por construção, temos $w(T') = w(T) - w(f) + w(e) \leq w(T)$, pois $w(e) \leq w(f)$, o que vale pela escolha de e. Como T é uma árvore geradora mínima e $w(T') \leq w(T)$, então só podemos ter $w(T') = w(T)$. Assim, concluímos que T' é uma árvore geradora mínima que contém e, uma contradição. □

Como já vimos no início do Capítulo 24, podemos construir uma árvore geradora para qualquer grafo conexo G por meio de um *algoritmo de busca*, que é qualquer algoritmo que parte de um único vértice e cresce uma árvore ao escolher arestas entre vértices já escolhidos e vértices ainda não escolhidos. A seguir, retomamos e generalizamos um pouco essa ideia.

Seja $H \subseteq G$ uma floresta que é subgrafo de um grafo conexo G. Se $V(H) = V(G)$ e H tem uma única componente conexa, então H é uma árvore geradora. Caso contrário, $V(H) \neq V(G)$ ou H tem mais de uma componente conexa. Sejam $H_1, \ldots, H_k$ as componentes conexas de H, com $k \geq 1$, e seja $R = V(G) \setminus \bigcup_{i=1}^{k} V(H_i)$ o conjunto de vértices que não fazem parte das componentes. Nesse caso, qualquer aresta $xy \in E(G)$ tal que $xy \notin E(H)$ será da forma $x \in V(H_i)$ e $y \in V(H_j)$, para $i \neq j$ (se H tem mais de uma componente), ou da forma $x \in V(H_i)$ e $y \in R$, para algum i (se H tem uma única componente). Note que alguma aresta desse tipo deve existir pois G é conexo. Assim, temos que xy é sozinha no corte $\partial_{H+xy}(V(H_i))$ e, pelo Lema 25.2, vale que $H + xy$ também é uma floresta.

Note que qualquer algoritmo que tem como objetivo criar uma árvore geradora pode fazer isso utilizando o processo acima. Ademais, se o objetivo é que a árvore geradora seja mínima, então pelo Teorema 25.3, uma boa escolha para xy é uma aresta de custo mínimo no corte.

Essa é justamente a ideia dos algoritmos de Kruskal e Prim, que resolvem o problema da árvore geradora mínima e serão apresentados nas seções a seguir. Seja G um grafo conexo. O algoritmo de Kruskal inicia H como uma floresta com $v(G)$ componentes triviais, uma para cada vértice, ou seja, uma floresta geradora. A todo momento, ele aumenta o número de arestas em H, mas mantendo sempre uma floresta geradora, até que se chegue em uma única componente. Já o algoritmo de Prim inicia H como um único vértice qualquer. A todo momento, ele aumenta o número de arestas em H, sempre mantendo uma árvore, até que se chegue em $v(G) - 1$ arestas. É interessante notar que o algoritmo de Prim nada mais é do que uma versão de Busca (Algoritmo 24.3) na qual se utiliza uma fila de prioridades como estrutura auxiliar.

25.1 Algoritmo de Kruskal

Dado um grafo conexo G e uma função w de pesos sobre as arestas de G, o algoritmo de Kruskal começa com um conjunto de $v(G)$ componentes triviais, com um vértice em cada, e a cada passo adiciona uma aresta entre duas componentes distintas, garantindo que as componentes são árvores contidas em uma árvore geradora mínima de G. Pelo resultado do Teorema 25.3, a aresta escolhida entre duas componentes deve ser a que tenha menor peso dentre as arestas disponíveis. Ele é considerado um algoritmo guloso (veja Capítulo 21), por ter como escolha gulosa tal aresta de menor peso.

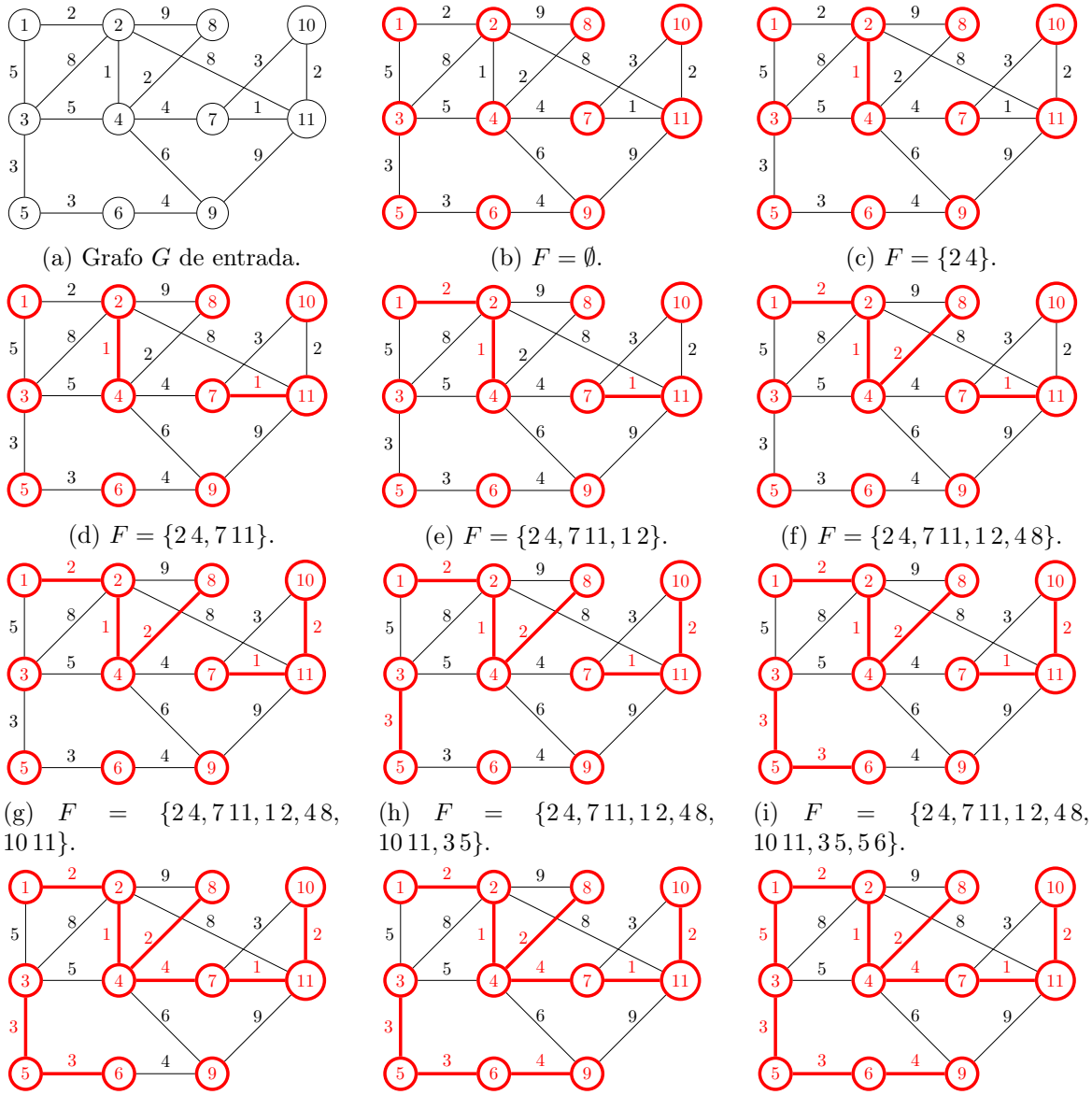
A descrição mais conhecida do algoritmo de Kruskal está formalizada no Algoritmo 25.1. Ele não mantém as componentes conexas de forma explícita, mas apenas um conjunto F de arestas, que inicialmente é vazio e vai sendo aumentado a cada iteração. O primeiro passo do algoritmo é ordenar as arestas de forma não-decrescente nos pesos. Em seguida, percorre todas as arestas por essa ordem, adicionando-as a F caso não formem ciclos com as arestas que já estão em F. Lembre-se que, dado um grafo G e um subconjunto $F \subseteq E(G)$, o grafo $G[F]$ é o subgrafo de G com conjunto de arestas F e com os vértices que são extremos das arestas de F.

Algoritmo 25.1: KRUSKAL(G, w)

1 Crie um vetor $C[1..e(G)]$ e copie as arestas de G para C
2 Ordene C de modo não-decrescente de pesos das arestas
3 Seja $F = \emptyset$
4 **para** $i = 1$ *até* $e(G)$, *incrementando* **faça**
5 **se** $G[F \cup \{C[i]\}]$ *não contém ciclos* **então**
6 $F = F \cup \{C[i]\}$
7 **devolve** F

A Figura 25.2 apresenta um exemplo de execução do algoritmo. Perceba que em alguns momentos é possível fazer mais de uma escolha sobre qual aresta inserir. Por exemplo, em 25.2k tanto a aresta 1 3 quando a aresta 3 4 poderiam ter sido escolhidas. Fizemos a escolha de 1 3 por um critério simples da ordem dos vértices.

No começo do algoritmo, o conjunto de arestas do grafo é ordenado de acordo com seus pesos (linha 2). Assim, para considerar arestas de menor peso, basta percorrer o vetor C em ordem. Na linha 3, criamos o conjunto F que manterá as arestas que compõem uma árvore geradora mínima. Nas linhas 4, 5 e 6, são adicionadas, passo a passo, arestas de peso mínimo que não formam ciclos com as arestas que já estão em F. O Lema 25.4 a seguir mostra que



(a) Grafo G de entrada.

(b) $F = \emptyset$.

(c) $F = \{2\,4\}$.

(d) $F = \{2\,4, 7\,11\}$.

(e) $F = \{2\,4, 7\,11, 1\,2\}$.

(f) $F = \{2\,4, 7\,11, 1\,2, 4\,8\}$.

(g) $F = \{2\,4, 7\,11, 1\,2, 4\,8, 10\,11\}$.

(h) $F = \{2\,4, 7\,11, 1\,2, 4\,8, 10\,11, 3\,5\}$.

(i) $F = \{2\,4, 7\,11, 1\,2, 4\,8, 10\,11, 3\,5, 5\,6\}$.

(j) $F = \{2\,4, 7\,11, 1\,2, 4\,8, 10\,11, 3\,5, 5\,6, 4\,7\}$.

(k) $F = \{2\,4, 7\,11, 1\,2, 4\,8, 10\,11, 3\,5, 5\,6, 4\,7, 6\,9\}$.

(l) $F = \{2\,4, 7\,11, 1\,2, 4\,8, 10\,11, 3\,5, 5\,6, 4\,7, 6\,9, 1\,3\}$.

Figura 25.2: Execução de KRUSKAL(G, w). As componentes estão destacadas em vermelho.

Kruskal(G, w) de fato gera uma árvore geradora mínima para G.

Lema 25.4

Seja G um grafo conexo e w uma função de pesos nas arestas. O conjunto F de arestas devolvido por Kruskal(G, w) é tal que $G[F]$ é árvore geradora mínima de G.

Demonstração. Seja F_i o conjunto de arestas no início da i-ésima iteração do laço **para**, isto é, $F_1 = \emptyset$, e seja F o conjunto devolvido ao fim. Claramente, por construção, $G[F]$ não tem ciclos. Basta mostrar então que $G[F]$ é conexo e que $w(G[F])$ é mínimo.

Seja $S \subseteq V(G)$ qualquer. Considere que $e \in \partial_G(S)$ é a primeira aresta de $\partial_G(S)$ que é considerada por Kruskal e suponha que isso acontece na i-ésima iteração. Sendo ela a primeira desse corte que é considerada, então $\partial_{G[F_i \cup \{e\}]}(S)$ contém apenas essa aresta. Sendo sozinha em um corte, então pelo resultado do Lema 25.2, não existem ciclos em $G[F_i \cup \{e\}]$. Logo, e é de fato escolhida para ser adicionada a F_i. Acabamos de mostrar, portanto, que qualquer corte do grafo possui uma aresta em F que o cruza, de forma que $G[F]$ é conexo.

Por fim, seja $e = uv$ a aresta que é adicionada na i-ésima iteração. Seja $S \subseteq V(G)$ o conjunto de vértices da componente conexa do grafo $G[F_i]$ que contém u. Como e foi escolhida nessa iteração, S não contém v. Note que devido à ordem de escolha do algoritmo, a aresta e é mínima em $\partial_G(S)$. Então, pelo Teorema 25.3, ela deve fazer parte de uma árvore geradora mínima de G. Ou seja, o algoritmo apenas fez escolhas de arestas que estão em uma árvore geradora mínima e, portanto, construiu uma árvore geradora mínima. □

Seja G um grafo conexo com n vértices e m arestas. Se o grafo está representado por listas de adjacências, então executar a linha 1 leva tempo $\Theta(n + m)$. Utilizando *Mergesort* ou *Heapsort*, a linha 2 tem tempo $O(m \log m)$. A linha 3 leva tempo $\Theta(1)$ e o laço **para** (linha 4) é executado m vezes. O tempo gasto na linha 5 depende de como identificamos os ciclos em $F \cup \{C[i]\}$. Podemos utilizar busca em largura ou profundidade, o que leva tempo $O(n + |F|)$ (basta procurar por ciclos em $G[F]$ e não em G). Como F possui no máximo $n - 1$ arestas, a linha 5 é executada em tempo $O(n)$ com busca em largura ou profundidade. Portanto, como o laço é executado m vezes, no total o tempo gasto nas linhas 4 a 6 é $O(mn)$. Se $T(n, m)$ é o tempo de execução de Kruskal(G, w), então vale que

$$\begin{aligned} T(n, m) &= \Theta(n + m) + O(m \log m) + O(mn) \\ &= O(m) + O(m \log n) + O(mn) = O(mn) \ . \end{aligned}$$

Para entender as igualdades acima, note que, como G é conexo, temos $m \geq n - 1$, de modo

que vale $n = O(m)$ e, portanto, $n + m = O(m)$. Também note que, como $m = O(n^2)$ em qualquer grafo simples, temos que $m \log m \leq m \log(n^2) = 2m \log n = O(m \log n) = O(mn)$.

Agora note que a operação mais importante e repetida no algoritmo é a checagem de ciclos. Na análise acima, aplicamos uma busca em largura ou profundidade a cada iteração para verificar isso, gastando tempo $O(n + |F|)$ sempre. Felizmente, é possível melhorar o tempo de execução dessa operação através do uso de uma estrutura de dados apropriada. *Union-find* é um tipo abstrato de dados que mantém uma partição de um conjunto de objetos. Ela oferece as funções FindSet(x), que devolve o representante do conjunto que contém o objeto x, e Union(x, y), que funde os conjuntos que contêm os objetos x e y. Veja mais sobre essa estrutura na Seção 13.1.

Como mencionado no início da seção, o algoritmo de Kruskal no fundo está mantendo um conjunto de componentes conexas de G, isto é, uma partição dos vértices de G. Inicialmente, cada vértice está em um conjunto sozinho. A cada iteração, a aresta escolhida une dois conjuntos. Lembre-se que uma aresta que conecta duas componentes conexas de $G[F]$ não cria ciclos. É suficiente, portanto, adicionar a aresta de menor peso que conecta vértices mantidos em conjuntos diferentes, não sendo necessário procurar explicitamente por ciclos.

O Algoritmo 25.2 reapresenta o algoritmo de Kruskal utilizando explicitamente union-find. O procedimento MakeSet(x) cria um conjunto novo contendo somente o elemento x.

Algoritmo 25.2: KruskalUnionFind(G, w)

```
1  Crie um vetor C[1..e(G)] e copie as arestas de G para C
2  Ordene C de modo não-decrescente de pesos das arestas
3  Seja F = ∅
4  para todo vértice v ∈ V(G) faça
5      MakeSet(v)
6  para i = 1 até e(G), incrementando faça
7      Seja uv a aresta em C[i]
8      se FindSet(u) ≠ FindSet(v) então
9          F = F ∪ {C[i]}
10         Union(u, v)
11 devolve F
```

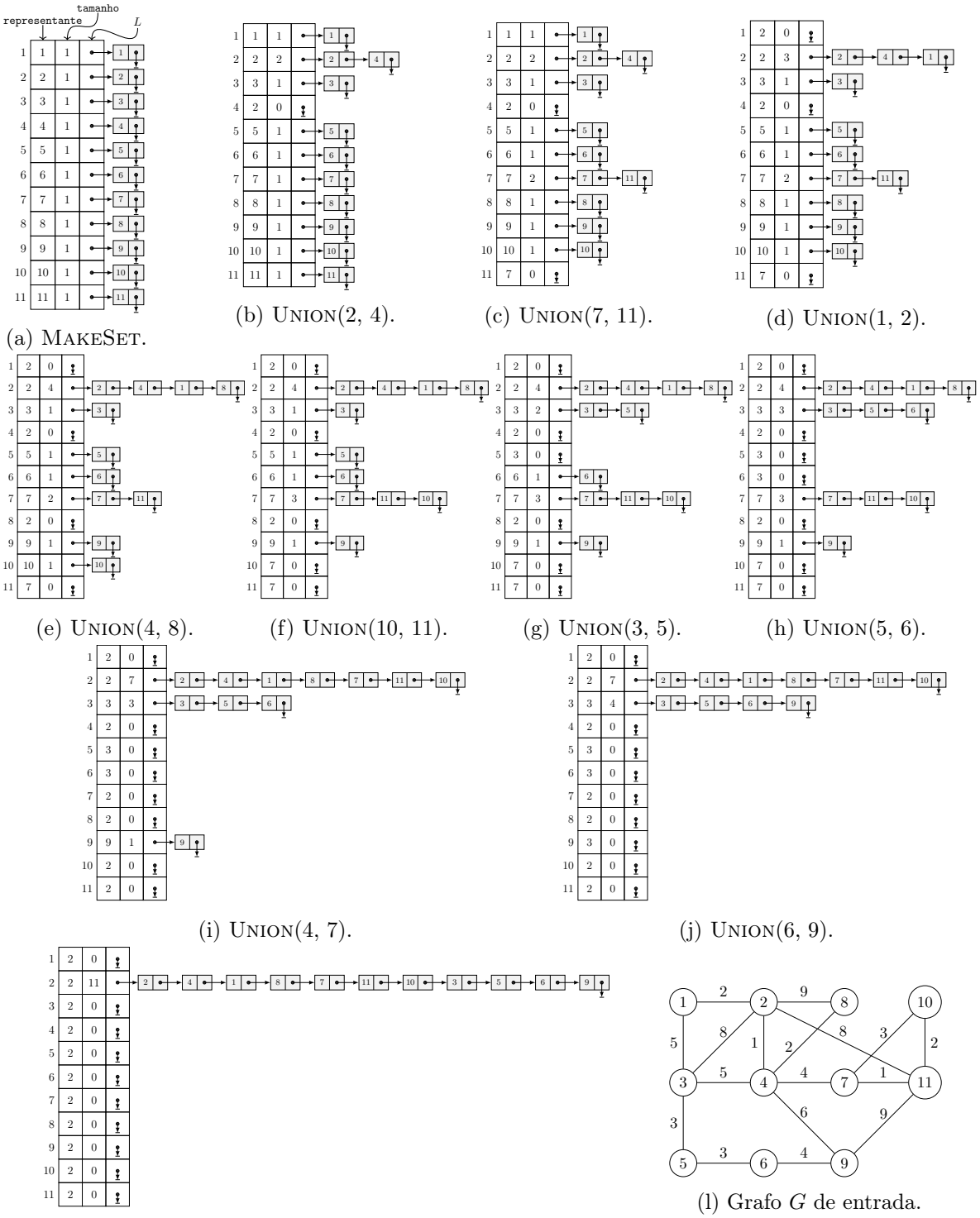
Novamente, nas primeiras linhas as arestas são ordenadas e o conjunto F é criado. No laço **para** da linha 4 criamos um conjunto para cada um dos vértices. Esses conjuntos são nossas componentes conexas iniciais. No laço **para** da linha 6 são adicionadas, passo a passo,

arestas de peso mínimo que conectam duas componentes conexas de $G[F]$. Note que o teste da linha 8 falha para uma aresta cujos extremos estão no mesmo conjunto e, portanto, criariam um ciclo em F. Ao adicionar uma aresta uv ao conjunto F, precisamos unir as componentes que contêm u e v (linha 10).

Aqui vamos considerar uma implementação simples de union-find, como mencionada na Seção 13.1. Cada conjunto tem como representante um vértice que seja membro do mesmo. Cada vértice x tem um atributo $x.\texttt{representante}$, que armazena o vértice representante do seu conjunto. Um vértice x também tem um atributo $x.\texttt{tamanho}$, que armazena o tamanho do conjunto que é representado por x. Manteremos ainda um vetor L de listas tal que $L[x]$ é a lista que armazena os vértices que estão no conjunto representado por x. Assim, FindSet(u) leva tempo $\Theta(1)$. Toda vez que dois conjuntos forem unidos, deve-se atualizar os representantes do menor deles com o representante do maior. Assim, Union(u, v) percorre a lista $L[u.\texttt{representante}]$, se $u.\texttt{tamanho} < v.\texttt{tamanho}$, ou $L[v.\texttt{representante}]$, caso contrário, para atualizar os representantes dos vértices presentes nela. Leva, portanto, tempo $\Theta(\min\{u.\texttt{tamanho}, v.\texttt{tamanho}\})$. A Figura 25.3 mostra a execução de KruskalUnionFind sobre o mesmo grafo da Figura 25.2, porém considerando essa implementação.

Seja G um grafo conexo com n vértices e m arestas. Como na análise do algoritmo Kruskal, executamos a linha 1 em tempo $\Theta(n + m)$ e a linha 2 em tempo $O(m \log m)$. A linha 3 leva tempo $\Theta(1)$ e levamos tempo $\Theta(n)$ no laço da linha 4. O laço **para** da linha 6 ainda é executado m vezes. Como a linha 8 tem somente operações FindSet, ela é executada em tempo $\Theta(1)$ e a linha 9 também, sendo, ao todo, $\Theta(m)$ verificações de ciclos.

Com relação à linha 10, precisamos analisar o tempo que leva para executar todas as chamadas a Union. Uma análise rápida nos diz que isso é $O(mn)$, pois cada conjunto tem $O(n)$ vértices. Acontece que poucos conjuntos terão $\Omega(n)$ vértices. Por exemplo, nas primeiras iterações os conjuntos têm apenas 1 ou 2 vértices cada. Aqui podemos fazer uma análise mais cuidadosa. A operação que mais consome tempo em Union é a atualização de um representante. Assim, contabilizar o tempo que todas as chamadas a Union levam para executar é assintoticamente proporcional a contar quantas vezes cada vértice tem seu representante atualizado. Considere um vértice x qualquer. Como na operação Union somente os elementos do conjunto de menor tamanho têm seus representantes atualizados, então toda vez que o representante de x é atualizado, o seu conjunto pelo menos dobra de tamanho. Assim, como x começa em um conjunto de tamanho 1 e termina em um conjunto de tamanho n, x tem seu representante atualizado no máximo $\log n$ vezes. Logo, o tempo total gasto nas execuções da linha 10 é $O(n \log n)$, que é bem melhor do que $O(mn)$. Se $T(n, m)$ é o



(a) MakeSet.

(b) Union(2, 4).

(c) Union(7, 11).

(d) Union(1, 2).

(e) Union(4, 8).

(f) Union(10, 11).

(g) Union(3, 5).

(h) Union(5, 6).

(i) Union(4, 7).

(j) Union(6, 9).

(k) Union(1, 3).

(l) Grafo G de entrada.

Figura 25.3: Execução de KruskalUnionFind(G, w).

tempo de execução de KruskalUnionFind(G, w), então vale que

$$\begin{aligned} T(n,m) &= \Theta(n+m) + O(m \log m) + \Theta(m) + O(n \log n) \\ &= O(m) + O(m \log n) + \Theta(m) + O(m \log n) \\ &= O(m \log n) \ . \end{aligned}$$

25.2 Algoritmo de Prim

Dado um grafo conexo G e uma função w de pesos nas arestas de G, o algoritmo de Prim começa com um conjunto de uma árvore trivial, de único vértice s, qualquer, e a cada passo aumenta essa árvore com uma nova aresta entre ela e vértices fora dela, garantindo que essa árvore sempre é subárvore de uma árvore geradora mínima de G. Pelo resultado do Teorema 25.3, tal aresta é a de menor peso dentre as disponíveis. Note que esse é o mesmo funcionamento das buscas em largura e profundidade, com a diferença de que agora as arestas têm um valor associado.

Novamente, esse algoritmo não mantém explicitamente a árvore que está sendo construída, mas apenas um conjunto S de vértices já visitados e seus predecessores. Assim, cada vértice u tem atributos $u.$`visitado` e $u.$`predecessor`. O atributo $u.$`predecessor` indica qual vértice levou u a ser visitado. Já o atributo $u.$`visitado` tem valor 1 se o vértice u já foi visitado pelo algoritmo e 0 caso contrário. Ele termina quando não há mais vértices não visitados. Esse é um algoritmo guloso (veja Capítulo 21) e sua característica gulosa é visitar um vértice $y \notin S$ tal que a aresta $xy \in \partial_G(S)$ é mínima.

O algoritmo de Prim está formalizado no Algoritmo 25.3. Note a similaridade do mesmo com o Algoritmo 24.3, Busca. A Figura 25.4 apresenta um exemplo de sua execução.

Algoritmo 25.3: Prim(G, w)

para *todo vértice* $v \in V(G)$ **faça**
  $v.$`visitado` $= 0$
  $v.$`predecessor` $=$ *null*
Seja $s \in V(G)$ qualquer
$s.$`visitado` $= 1$
enquanto *houver vértice não visitado* **faça**
  Seja xy uma aresta de menor peso com $x.$`visitado` $== 1$ e $y.$`visitado` $== 0$
  $y.$`visitado` $= 1$
  $y.$`predecessor` $= x$

O Lema 25.5 a seguir mostra que $\textsc{Prim}(G, w)$ de fato gera uma árvore geradora mínima para G.

Lema 25.5

Seja G um grafo conexo e w uma função de pesos nas arestas. Após a execução de $\textsc{Prim}(G, w)$, sendo $s \in V(G)$ escolhido na linha 4, o subgrafo T com $V(T) = \{v \in V(G) \colon v.\texttt{predecessor} \neq null\} \cup \{s\}$ e $E(T) = \{\{v.\texttt{predecessor}, v\} \colon v \in V(T) \setminus \{s\}\}$ é uma árvore geradora mínima para G.

Demonstração. Primeiro note que o algoritmo termina. Se esse não fosse o caso, haveria alguma iteração onde não haveria escolha para xy, o que significaria que G não é conexo, uma contradição. Então no fim temos de fato todos os vértices em $V(G)$ visitados.

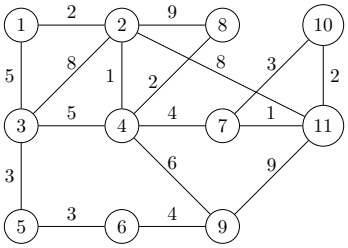
Seja T o subgrafo de G em que $V(T) = \{v \in V(G) \colon v.\texttt{predecessor} \neq null\} \cup \{s\}$ e $E(T) = \{\{v.\texttt{predecessor}, v\} \colon v \in V(T) \setminus \{s\}\}$. Note que T é gerador pois todo vértice $v \in V(T)$ ou é $v = s$, ou tem $v.\texttt{predecessor} \neq null$, e, portanto, está visitado.

Note agora que T não tem ciclos. Considere uma iteração onde xy é escolhida, com $x.\texttt{visitado} = 1$ e $y.\texttt{visitado} = 0$. Seja T_e com $V(T_e) = \{v \in V(G) \colon v.\texttt{predecessor} \neq null\} \cup \{s\}$ e $E(T_e) = \{\{v.\texttt{predecessor}, v\} \colon v \in V(T_e) \setminus \{s\}\}$ o subgrafo de T construído até o início dessa iteração. Assim, $x \in V(T_e)$ e $y \notin V(T_e)$. Note que xy é a única aresta de $\partial_{T_e}(V(T_e))$ e, portanto, pelo Lema 25.2, ela não participa de ciclos em T_e.

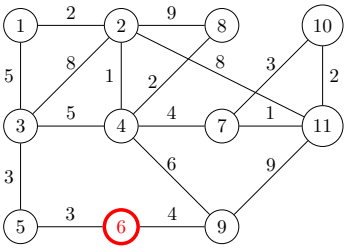
Resta mostrar que $w(T)$ é mínimo. Note que pelo critério de escolha, cada aresta $xy \in E(T)$ é mínima em $\partial_G(V(T_e))$. Então, pelo Teorema 25.3, ela faz parte de uma árvore geradora mínima de G. Assim, T é uma árvore geradora mínima. □

Seja G um grafo conexo com n vértices e m arestas. O laço **para** da linha 1 executa em tempo $\Theta(n)$. A escolha de s e sua visitação levam tempo $\Theta(1)$. A todo momento, um vértice novo é visitado. Assim, as linhas 6, 7, 8 e 9 são executadas $\Theta(n)$ vezes cada. Dessas, apenas a linha 7 não leva tempo constante. Nessa linha, fazemos a escolha de uma aresta xy com $x.\texttt{visitado} = 1$ e $y.\texttt{visitado} = 0$ que tenha menor peso dentre as arestas desse tipo. Uma forma de implementar essa escolha é: percorra todas as arestas do grafo verificando se seus extremos satisfazem a condição e armazenando a de menor custo. Veja que isso leva tempo $\Theta(m)$. Somando todos os tempos, essa implementação leva tempo $\Theta(n) + \Theta(n)\Theta(m) = \Theta(nm)$.

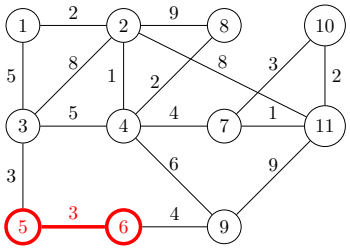
Vemos aqui que a operação mais custosa é a de encontrar a aresta xy a cada iteração e "removê-la" do conjunto de arestas disponíveis. Felizmente, é possível melhorar esse tempo de execução através do uso de uma estrutura de dados apropriada para esse tipo de operação.



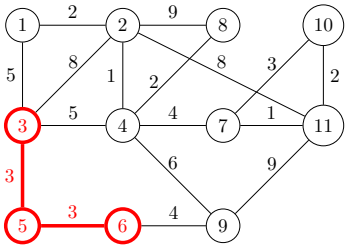
(a) Grafo G de entrada. Vértice inicial arbitrário: $s = 6$.



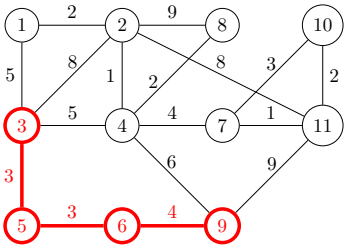
(b) Vértices visitados: $S = \{6\}$. $\partial_G(S) = \{5\,6,\ 6\,9\}$.



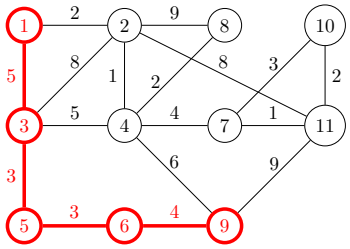
(c) Aresta mínima: $5\,6$. Visitados: $S = \{5, 6\}$. $\partial_G(S) = \{3\,5,\ 6\,9\}$.



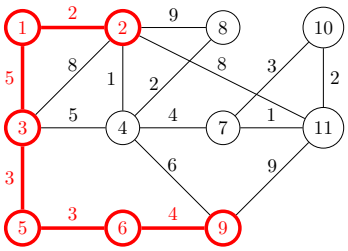
(d) Aresta mínima: $3\,5$. Visitados: $S = \{3, 5, 6\}$. $\partial_G(S) = \{1\,3,\ 2\,3,\ 3\,4,\ 6\,9\}$.



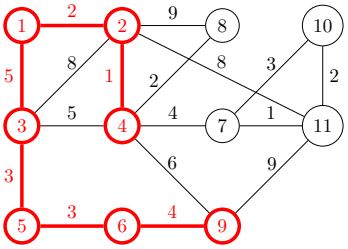
(e) Aresta mínima: $6\,9$. Visitados: $S = \{3, 5, 6, 9\}$. $\partial_G(S) = \{1\,3,\ 2\,3,\ 3\,4,\ 4\,9,\ 9\,11\}$.



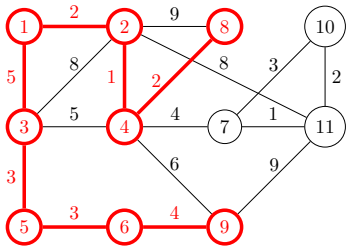
(f) Aresta mínima: $1\,3$. Visitados: $S = \{1, 3, 5, 6, 9\}$. $\partial_G(S) = \{1\,2,\ 2\,3,\ 3\,4,\ 4\,9,\ 9\,11\}$.



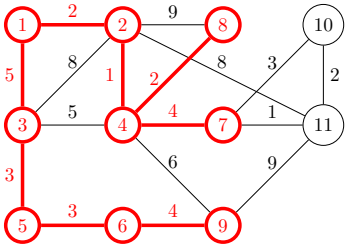
(g) Aresta mínima: $1\,2$. Visitados: $S = \{1, 2, 3, 5, 6, 9\}$. $\partial_G(S) = \{2\,4,\ 2\,8,\ 2\,11,\ 3\,4,\ 4\,9,\ 9\,11\}$.



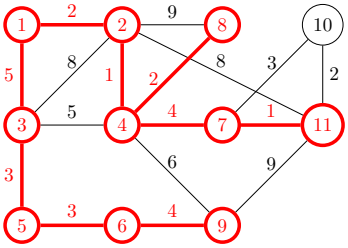
(h) Aresta mínima: $2\,4$. Visitados: $S = \{1, 2, 3, 4, 5, 6, 9\}$. $\partial_G(S) = \{2\,8,\ 2\,11,\ 4\,7,\ 4\,8\ 9\,11\}$.



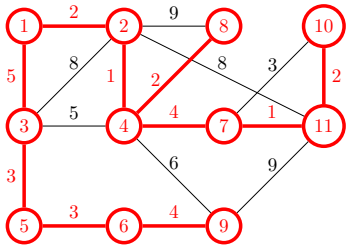
(i) Aresta mínima: $4\,8$. Visitados: $S = \{1, 2, 3, 4, 5, 6, 8, 9\}$. $\partial_G(S) = \{2\,11,\ 4\,7,\ 9\,11\}$.



(j) Aresta mínima: $4\,7$. Visitados: $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. $\partial_G(S) = \{2\,11,\ 7\,10,\ 7\,11,\ 9\,11\}$.



(k) Aresta mínima: $7\,11$. Visitados: $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11\}$. $\partial_G(S) = \{7\,10,\ 10\,11\}$.



(l) Aresta mínima: $10\,11$. Visitados: $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$. $\partial_G(S) = \emptyset$.

Figura 25.4: Execução de Prim(G, w). Os vértices visitados estão em vermelho. A árvore construída de forma indireta pelos predecessores está em vermelho.

Heap é uma estrutura que oferece a operação RemoveDaHeap, que remove o elemento de maior prioridade em tempo $O(\log k)$, onde k é a quantidade de elementos armazenados na estrutura. Veja mais sobre essa estrutura na Seção 12.1.

Lembre-se que o algoritmo de Prim na verdade faz uma escolha por um novo vértice que ainda não foi visitado. Dentre todos os vértices não visitados que possuem uma aresta que os conecta a vértices já visitados, escolhemos o que tenha a aresta de menor custo. Vamos utilizar um heap para armazenar vértices e o valor da prioridade de um vértice x será o custo da aresta de menor custo que conecta x a um vértice que não está mais na heap. Mais especificamente, nossa heap irá manter os vértices de $V(G) \setminus S$ em que $S = \{v \colon v.\texttt{visitado} = 1\}$ e, para cada $x \in V(G) \setminus S$, $x.\texttt{prioridade}$ irá armazenar o custo da aresta de menor custo xv tal que $v \in S$. Se tal aresta não existir, então a prioridade de v será ∞. Note que tem mais prioridade o vértice que tem menor valor de prioridade associado. Assim, o próximo vértice a ser visitado deve ser o vértice removido do heap.

Assuma que $V(G) = \{1, \ldots, v(G)\}$ e que cada vértice x possui os atributos `prioridade`, para armazenar sua prioridade, `indice`, para indicar a posição do heap em que x está armazenado, e `predecessor`, para indicar o vértice v visitado tal que a aresta vx é a de menor custo que conecta x a um elemento já visitado. O Algoritmo 25.4 reapresenta o algoritmo de Prim utilizando explicitamente a estrutura heap e é explicado com detalhes a seguir. Lembre-se que InsereNaHeap(H, v) insere o elemento v em H, RemoveDaHeap(H) remove e devolve o elemento de maior prioridade de H e AlteraHeap(H, $v.\texttt{indice}$, x) atualiza o valor em $v.\texttt{prioridade}$ para x. Todas essas operações mantêm a propriedade de heap em H. As Figuras 25.5 e 25.6 mostram a execução de PrimHeap sobre o mesmo grafo da Figura 25.4, porém considerando essa implementação.

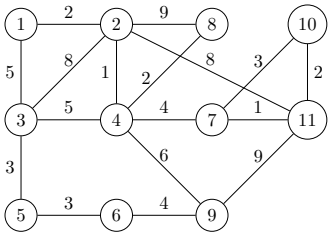
As linhas 1 a 14 fazem apenas a inicialização. Primeiro escolhemos um vértice s qualquer e o inicializamos como único vértice visitado. Em seguida criamos um vetor H que será um heap. Lembre-se que todo vértice que está no heap é não visitado. Assim, um vértice escolhido deve ser sempre um que tenha aresta para um vértice já visitado.

No laço **para** da linha 5 indicamos que os vizinhos de s, ainda não visitados, têm prioridade dada pelo custo da aresta que os conecta a s (lembre-se que o custo é negativo pois maior prioridade é indicada por quem tem menor valor indicador de prioridade). Eles são os únicos que têm aresta para um vértice já visitado. No laço **para** da linha 10 indicamos que todos os outros vértices que não são vizinhos de s têm baixa prioridade e não podem ser escolhidos no início. Todos são inseridos no heap (linhas 9 e 14).

O procedimento de crescimento da árvore encontra-se no laço **enquanto** da linha 15, que a cada vez remove um vértice v do heap e o visita. Note que quando v é visitado, as prioridades de alguns vértices podem mudar, pois os conjuntos de vértices visitados e não

Algoritmo 25.4: PRIMHEAP(G, w)

```
1  Seja s ∈ V(G) qualquer
2  s.visitado = 1
3  s.predecessor = null
4  Seja H[1..v(G) − 1] um vetor vazio
5  para todo vértice v ∈ N(s) faça
6      v.prioridade = −w(sv)
7      v.visitado = 0
8      v.predecessor = s
9      InsereNaHeap(H, v)
10 para todo vértice v ∉ N(s) faça
11     v.prioridade = −∞
12     v.visitado = 0
13     v.predecessor = null
14     InsereNaHeap(H, v)
15 enquanto H.tamanho > 0 faça
16     v = RemoveDaHeap(H)
17     v.visitado = 1
18     para todo vértice x ∈ N(v) faça
19         se x.visitado == 0 e x.prioridade < −w(vx) então
20             x.predecessor = v
21             AlteraHeap(H, x.indice, −w(vx))
```



(a) Grafo G de entrada. Vértice inicial arbitrário: $s = 6$.

H

	1	2	3	4	5	6	7	8	9	10	11
predecessor						*null*					
visitado						1					
indice											
prioridade											

(b) Visita s e atualiza seu `predecessor`.

H

1	2
5	9

	1	2	3	4	5	6	7	8	9	10	11
predecessor					6	*null*			6		
visitado					0	1			0		
indice					1				2		
prioridade					-3				-4		

(c) Atualiza `prioridade`, `visitado` e `predecessor` dos vizinhos de s. Insere-os no heap.

H

1	2	3	4	5	6	7	8	9	10
5	9	1	2	3	4	7	8	10	11

	1	2	3	4	5	6	7	8	9	10
predecessor	*null*	*null*	*null*	*null*	6	*null*	*null*	*null*	6	*null*	*null*
visitado	0	0	0	0	0	1	0	0	0	0	0
indice	3	4	5	6	1		7	8	2	9	10
prioridade	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-3		$-\infty$	$-\infty$	-4	$-\infty$	$-\infty$

(d) Atualiza `prioridade`, `visitado` e `predecessor` dos não vizinhos de s. Insere-os no heap.

H

1	2	3	4	5	6	7	8	9
3	9	1	2	11	4	7	8	10

	1	2	3	4	5	6	7	8	9	10	11
predecessor	*null*	*null*	5	*null*	6	*null*	*null*	*null*	6	*null*	*null*
visitado	0	0	0	0	1	1	0	0	0	0	0
indice	3	4	1	6			7	8	2	9	5
prioridade	$-\infty$	$-\infty$	-3	$-\infty$	-3		$-\infty$	$-\infty$	-4	$-\infty$	$-\infty$

(e) Remove 5 do heap. Atualiza prioridade de 3.

H

1	2	3	4	5	6	7	8
9	2	1	10	11	4	7	8

	1	2	3	4	5	6	7	8	9	10	11
predecessor	3	3	5	3	6	*null*	*null*	*null*	6	*null*	*null*
visitado	0	0	1	0	1	1	0	0	0	0	0
indice	3	2		6			7	8	1	4	5
prioridade	-5	-8	-3	-5	-3		$-\infty$	$-\infty$	-4	$-\infty$	$-\infty$

(f) Remove 3 do heap. Atualiza prioridade de 1, 2, 4.

H

1	2	3	4	5	6	7
1	2	4	10	11	8	7

	1	2	3	4	5	6	7	8	9	10	11
predecessor	3	3	5	3	6	*null*	*null*	*null*	6	*null*	9
visitado	0	0	1	0	1	1	0	0	1	0	0
indice	1	2		3			7	6		4	5
prioridade	-5	-8	-3	-5	-3		$-\infty$	$-\infty$	-4	$-\infty$	-9

(g) Remove 9 do heap. Atualiza prioridade de 11.

H

1	2	3	4	5	6
2	4	7	10	11	8

	1	2	3	4	5	6	7	8	9	10	11
predecessor	3	1	5	3	6	*null*	*null*	*null*	6	*null*	9
visitado	1	0	1	0	1	1	0	0	1	0	0
indice		1		2			3	6		4	5
prioridade	-5	-2	-3	-5	-3		$-\infty$	$-\infty$	-4	$-\infty$	-9

(h) Remove 1 do heap. Atualiza prioridade de 2.

Figura 25.5: Parte 1 da execução de PrimHeap(G, w) em que G e w são dados em 25.5a. Note que em H estamos mostrando os rótulos dos vértices, e não suas prioridades.

H	4	11	7	10	8
	1	2	3	4	5

	1	2	3	4	5	6	7	8	9	10	11
predecessor	3	1	5	2	6	*null*	*null*	2	6	*null*	2
visitado	1	1	1	0	1	1	0	0	1	0	0
indice				1			3	5		4	2
prioridade	-5	-2	-3	-1	-3		$-\infty$	-9	-4	$-\infty$	-8

(a) Remove 2 do heap. Atualiza prioridade de 4, 8, 11.

H	8	7	11	10
	1	2	3	4

	1	2	3	4	5	6	7	8	9	10	11
predecessor	3	1	5	2	6	*null*	4	4	6	*null*	2
visitado	1	1	1	1	1	1	0	0	1	0	0
indice							2	1		4	3
prioridade	-5	-2	-3	-1	-3		-4	-2	-4	$-\infty$	-8

(b) Remove 4 do heap. Atualiza prioridade de 7, 8.

H	7	10	11
	1	2	3

	1	2	3	4	5	6	7	8	9	10	11
predecessor	3	1	5	2	6	*null*	4	4	6	*null*	2
visitado	1	1	1	1	1	1	0	1	1	0	0
indice							1			2	3
prioridade	-5	-2	-3	-1	-3		-4	-2	-4	$-\infty$	-8

(c) Remove 8 do heap. Sem vizinhos não visitados.

H	11	10
	1	2

	1	2	3	4	5	6	7	8	9	10	11
predecessor	3	1	5	2	6	*null*	4	4	6	7	7
visitado	1	1	1	1	1	1	1	1	1	0	0
indice										2	1
prioridade	-5	-2	-3	-1	-3		-4	-2	-4	-3	-1

(d) Remove 7 do heap. Atualiza prioridade de 10, 11.

H	10
	1

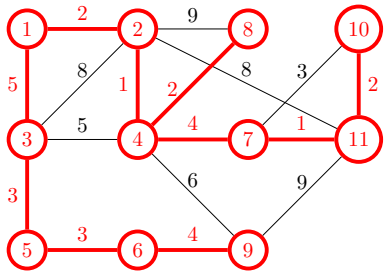
	1	2	3	4	5	6	7	8	9	10	11
predecessor	3	1	5	2	6	*null*	4	4	6	11	7
visitado	1	1	1	1	1	1	1	1	1	0	1
indice										1	
prioridade	-5	-2	-3	-1	-3		-4	-2	-4	-2	-1

(e) Remove 11 do heap. Atualiza prioridade de 10.

H

	1	2	3	4	5	6	7	8	9	10	11
predecessor	3	1	5	2	6	*null*	4	4	6	11	7
visitado	1	1	1	1	1	1	1	1	1	1	1
indice											
prioridade	-5	-2	-3	-1	-3		-4	-2	-4	-2	-1

(f) Remove 10 do heap. Sem vizinhos não visitados.



(g) Heap vazio. Árvore gerada com os predecessores.

Figura 25.6: Parte 2 da execução de PrimHeap(G, w) em que G e w são dados em 25.5a. Note que em H estamos mostrando os rótulos dos vértices, e não suas prioridades.

visitados mudam. No entanto, as únicas novas arestas entre vértices visitados e não visitados são as arestas que saem de v. Por isso, é suficiente recalcular apenas as prioridades dos vértices que são adjacentes a v, o que é feito no laço **para** da linha 18. Note que apenas alteramos a prioridade de um vértice x se a prioridade da aresta vx é maior do que a prioridade que x já tinha (que é a prioridade dada por outra aresta que conecta x a outro vértice já visitado, ou é $-\infty$ se nenhuma aresta ainda conectava x a um vértice já visitado).

Seja G um grafo conexo com n vértices e m arestas. As primeiras linhas da inicialização têm tempo $\Theta(1)$. Observe que as linhas dos dois laços **para** das linhas 5 e 10 executam juntas $\Theta(n)$ vezes. Exceto pelas linhas 9 e 14, que têm tempo $O(\log n)$, as outras linhas levam tempo constante. Assim, nesses dois laços gastamos tempo $O(n \log n)$.

As linhas 15, 16 e 17 executam $\Theta(n)$ vezes cada. Delas, apenas a linha 16 leva tempo não constante. A função RemoveDaHeap leva tempo $O(\log n)$. Todas as linhas do laço **para** que começa na linha 18 executam $\Theta(m)$ vezes ao todo, considerando implementação em listas de adjacência. Delas, apenas a linha 21 não tem tempo constante. A função AlteraHeap leva tempo $O(\log n)$.

Somando todas as linhas, o tempo de execução de PrimHeap(G, w) é $O(n \log n) + \Theta(n) + O(n \log n) + \Theta(m) + O(m \log n) = O(m \log n)$, bem melhor do que $\Theta(mn)$.

Capítulo 26

Trilhas Eulerianas

Lembre-se que uma *trilha* em um (di)grafo G é uma sequência de vértices $(v_0, v_1, \ldots, v_k)$ tal que $v_i v_{i+1} \in E(G)$ para todo $0 \leq i < k$ e todas essas arestas são distintas (mas note pode haver repetição de vértices). Os vértices v_0 e v_k são extremos enquanto que $v_1, \ldots, v_{k-1}$ são internos. O comprimento de uma trilha é o número de arestas na mesma. Uma trilha é dita fechada se tem comprimento não nulo e tem início e término no mesmo vértice. Se a trilha inicia em um vértice e termina em outro vértice, então dizemos que a trilha é aberta. Veja mais sobre trilhas na Seção 23.7.

Uma observação muito importante é que em uma trilha de um grafo, o número de arestas da trilha que incide em um vértice interno é par. Isso também vale para os vértices extremos de trilhas fechadas. Já se a trilha é aberta, o número de arestas da trilha que incide em um vértice extremo é ímpar. Em uma trilha de um digrafo, o número de arcos que entram em um vértice interno é igual ao número de arcos que saem deste mesmo vértice. Isso também vale para os vértices extremos de trilhas fechadas. Se a trilha é aberta, então a diferença entre o número de arcos que saem do vértice inicial e o número de arcos que entram no mesmo é 1. Ademais, a diferença entre o número de arcos que entram no vértice final e o número de arcos que saem do mesmo é 1.

Um clássico problema em Teoria dos Grafos é o de, dado um (di)grafo conexo G, encontrar uma trilha que passa por todas as arestas de G. Uma trilha com essa propriedade é chamada de *trilha Euleriana*, em homenagem a Euler, que observou quais propriedades um grafo deve ter para conter uma trilha Euleriana. O Teorema 26.1 a seguir fornece uma condição necessária e suficiente para que existe uma trilha Euleriana fechada em um grafo conexo e em um digrafo fortemente conexo.

Teorema 26.1

Um grafo conexo G contém uma trilha Euleriana fechada se e somente se todos os vértices de G têm grau par.

Um digrafo fortemente conexo D contém uma trilha Euleriana fechada se e somente se todos os vértices de D têm grau de entrada igual ao grau de saída.

O Teorema 26.2 a seguir trata de trilhas Eulerianas abertas. Note que se um (di)grafo contém uma trilha Euleriana fechada, então ele certamente contém uma trilha Euleriana aberta.

Teorema 26.2

Um grafo conexo G contém uma trilha Euleriana aberta se e somente se G contém exatamente dois vértices de grau ímpar.

Um digrafo fortemente conexo D contém uma trilha Euleriana aberta se e somente se D contém no máximo um vértice com a diferença entre grau de saída e grau de entrada sendo 1, no máximo um vértice com a diferença entre grau de entrada e grau de saída sendo 1 e todo outro vértice tem grau de entrada igual ao grau de saída.

A seguir veremos um algoritmo guloso simples que encontra uma trilha Euleriana em digrafos fortemente conexos que satisfazem as propriedades dos Teoremas 26.1 e 26.2. Se D é um digrafo fortemente conexo para o qual todos os vértices têm grau de entrada igual ao grau de saída, dizemos que ele é *do tipo fechado*. Se isso acontece para todos os vértices exceto por no máximo dois, sendo que em um deles a diferença entre grau de saída e grau de entrada é 1 e no outro a diferença entre grau de entrada e grau de saída é 1, então dizemos que D é *do tipo aberto*.

É importante perceber que não estamos desconsiderando grafos ao fazer isso. Em primeiro lugar, todo grafo conexo pode ser visto como um digrafo fortemente conexo (seu digrafo associado – veja Seção 23.1). Em segundo lugar, os vértices de todo multigrafo subjacente de um digrafo fortemente conexo do tipo fechado têm grau par. Ademais, no máximo dois vértices de todo multigrafo subjacente de um digrafo fortemente conexo do tipo aberto têm grau ímpar. Assim, o algoritmo de fato é válido para grafos conexos com zero ou dois vértices de grau ímpar.

A seguinte definição é importante para esse algoritmo. Um arco é dito *seguro* em um (di)grafo se e somente se ele pertence a um ciclo.

O algoritmo de Fleury, descrito no Algoritmo 26.1, recebe um digrafo fortemente co-

nexo D do tipo fechado ou aberto e um vértice inicial s. Ele devolve um vetor W tal que $(W[1], W[2], \ldots, W[e(G)+1])$ é uma trilha Euleriana fechada ou aberta em D. Se D é do tipo fechado, então s pode ser um vértice qualquer. Caso contrário, espera-se que s seja um dos dois vértices cuja diferença entre os graus de entrada e saída é 1. O algoritmo começa a trilha apenas com o vértice s. A qualquer momento, se x é o último vértice inserido na trilha que está sendo construída, então o próximo vértice a ser inserido é um vértice y tal que o arco xy é seguro, a menos que essa seja a única alternativa. O arco xy é então removido do digrafo. O algoritmo para quando não há mais arcos saindo do último vértice que foi inserido na trilha. A Figura 26.1 contém um exemplo de execução de Fleury.

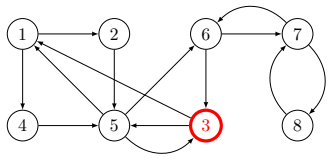
Algoritmo 26.1: Fleury(D, s)

```
Seja W[1..e(D)+1] um vetor vazio
W[1] = s
i = 1
enquanto d+_D(W[i]) ≥ 1 faça
    se existe arco (W[i], y) que é seguro D então
        W[i+1] = y
    senão
        W[i+1] = y, onde (W[i], y) não é seguro em D
    D = D − (W[i], W[i+1]) /* Removendo o arco visitado pela trilha */
    i = i + 1
devolve W
```

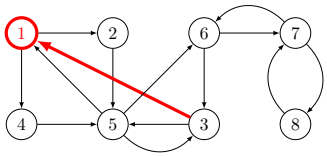
Uma *decomposição* de um (di)grafo G é uma coleção $\mathcal{D} = \{G_1, \ldots, G_k\}$ de subgrafos de G tal que $E(G_i) \cap E(G_j) = \emptyset$ para todo $1 \leq i < j \leq k$ e $\bigcup_{i=1}^{k} E(G_i) = E(G)$. Em geral, fala-se em decomposição em um mesmo tipo de subgrafos. Por exemplo, se todos os subgrafos de uma decomposição $\mathcal{D}$ são ciclos (ou caminhos), então $\mathcal{D}$ é chamada de decomposição em ciclos (ou caminhos). O Teorema 26.3 a seguir implica que se D é fortemente conexo com $d_D^+(v) = d_D^-(v)$ para todo $v \in V(D)$, então D contém apenas arcos seguros. Ele vai ser útil na prova de corretude do algoritmo de Fleury, que é dada no Teorema 26.1.

Teorema 26.3

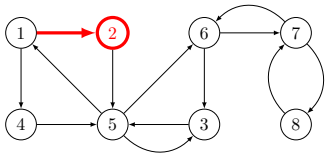
Um digrafo D é fortemente conexo e tem $d_D^+(v) = d_D^-(v)$ para todo $v \in V(D)$ se e somente se ele pode ser decomposto em ciclos.



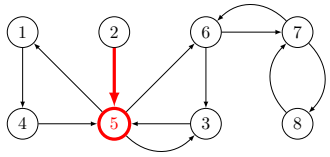
(a) Digrafo D de entrada (do tipo fechado) e vértice inicial $s = 3$. Trilha: (3).



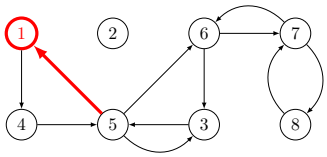
(b) Pode seguir qualquer arco que sai do 3. Trilha: $(3, 1)$.



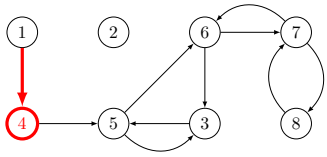
(c) Pode seguir qualquer arco que sai do 1. Trilha: $(3, 1, 2)$.



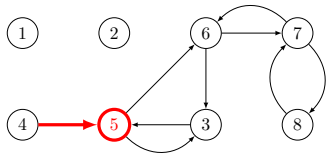
(d) Saem apenas arcos não seguros de 2. Trilha: $(3, 1, 2, 5)$.



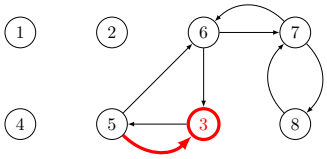
(e) Pode seguir qualquer arco que sai do 5. Trilha: $(3, 1, 2, 5, 1)$.



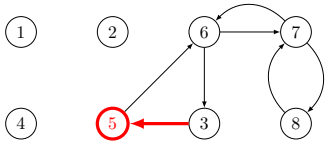
(f) Saem apenas arcos não seguros de 1. Trilha: $(3, 1, 2, 5, 1, 4)$.



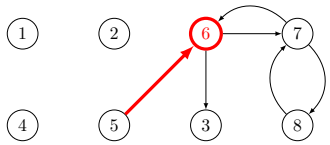
(g) Saem apenas arcos não seguros de 4. Trilha: $(3, 1, 2, 5, 1, 4, 5)$.



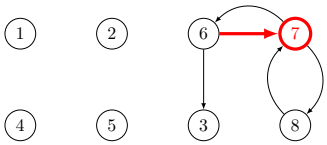
(h) Pode seguir qualquer arco que sai do 5. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3)$.



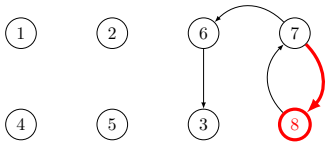
(i) Pode seguir qualquer arco que sai do 3. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3, 5)$.



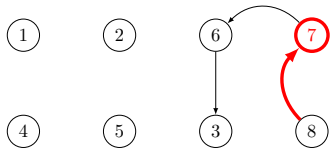
(j) Saem apenas narcos não seguros de 5. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3, 5, 6)$.



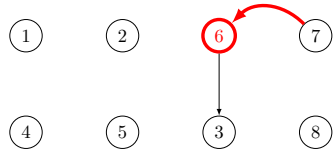
(k) O arco $(6, 3)$ não é seguro mas $(6, 7)$ é. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3, 5, 6, 7)$.



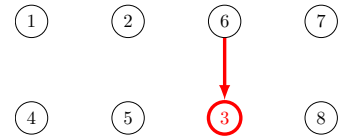
(l) O arco $(7, 6)$ não é seguro mas $(7, 8)$ é. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3, 5, 6, 7, 8)$.



(m) Saem apenas arcos não seguros de 8. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3, 5, 6, 7, 8, 7)$.



(n) Saem apenas arcos não seguros de 7. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3, 5, 6, 7, 8, 7, 6)$.



(o) Saem apenas arcos não seguros de 6. Trilha: (3, 1, 2, 5, 1, 4, 5, 3, 5, 6, 7, 8, 7, 6, 3).

Figura 26.1: Execução de FLEURY(D, 3). O último vértice da trilha em construção está em vermelho.

Teorema 26.4

Seja D um digrafo fortemente conexo onde $d_D^+(v) = d_D^-(v)$ para todo $v \in V(D)$ e seja $s \in V(D)$ qualquer. O algoritmo FLEURY(D, s) devolve uma trilha Euleriana fechada de D.

Demonstração. Seja W o vetor devolvido por FLEURY(D, s). Seja j o último valor assumido pela variável i. Seja $T = (W[1], \ldots, W[j])$ uma sequência de vértices formada pela resposta do algoritmo.

Primeiro note que T é um passeio, por construção, uma vez que um vértice é adicionado em $W[i]$ somente se há um arco saindo de $W[i-1]$ para ele. Além disso, T é uma trilha, pois tal arco é removido imediatamente do digrafo e, portanto, T não possui arcos repetidos.

Agora lembre-se que para qualquer trilha fechada, o número de arcos da trilha que entram em um vértice qualquer é igual ao número de arcos que saem deste mesmo vértice. Assim, T é uma trilha fechada, pois o algoritmo só termina quando atinge um vértice com grau de saída igual a zero e sabemos que inicialmente todos os vértices tinham grau de entrada igual ao de saída.

Resta mostrar que T é euleriana. Seja H o digrafo restante ao final da execução do algoritmo, isto é, $V(H) = V(D)$ e $E(H) = E(D) \setminus \{W[i]W[i+1] \colon 1 \leq i < j\}$. Suponha, para fins de contradição, que T não é Euleriana. Assim, existem arestas em H. Ademais, como T é fechada, perceba que $d_H^+(v) = d_H^-(v)$ para todo $v \in V(D)$.

Seja X o conjunto de vértices de H que têm grau de saída e de entrada não nulos. Note que $V(D) \setminus X \neq \emptyset$, isto é, há vértices com grau de entrada e saída iguais a zero em H (pois ao menos $s \in V(D) \setminus X$). Note ainda que $\partial_H(X) = \emptyset$, pela construção de X, mas $\partial_D(X) \neq \emptyset$ pois D é fortemente conexo. Por isso, existem arcos da trilha T que possuem cauda em X e cabeça em $V(D) \setminus X$ e existem arcos que possuem cauda em $V(D) \setminus X$ e cabeça em X. Seja $W[k]W[k+1] = uv$ o arco de T com $u \in X$ e $v \in V(D) \setminus X$ tal que $k < j$ é o maior possível. Note que no momento em que esse arco foi escolhido pelo algoritmo, ele era o único no corte $(X, V(D) \setminus X)$. Pelo Lema 25.2, ele não pertence a nenhum ciclo, o que significa que, naquele momento, ele era um arco não seguro no digrafo. Porém, como $u \in X$, então $d_H^+(v) > 0$ e, pelo Teorema 26.3, H contém apenas arcos seguros. Logo, havia outro arco que era seguro disponível no momento em que o algoritmo escolheu uv, uma contradição com a escolha do algoritmo. □

Seja D um digrafo fortemente conexo do tipo fechado com n vértices e m arcos. Com relação ao tempo de execução, perceba que o teste laço **enquanto** e cada uma de suas linhas executam $\Theta(m)$ vezes. Desse laço, apenas as linhas 5 e 9 podem não levar tempo constante.

Considerando implementação em matriz de adjacências, remover um arco uv qualquer de D envolve acessar a posição $M[u][v]$ e alterá-la, assim levando tempo $\Theta(1)$. Usando listas de adjacências, isso envolve percorrer a lista de u para remover o nó v, assim levando tempo $O(d^+(W[i]))$. A linha 9, portanto, executa em tempo $\Theta(m)$ em matriz de adjacências ou $O(d^+(W[i])m)$ em listas de adjacências.

Na linha 5 precisamos descobrir se um arco $(W[i], y)$ é seguro ou não para todo $y \in N^+(W[i])$. Dado qualquer arco uv, uma maneira simples de testar se ele é seguro é removendo uv do digrafo e executando uma busca em profundidade ou largura começando em u no digrafo restante. Se ao término tivermos $v.\texttt{visitado} == 0$, então uv não fazia parte de nenhum ciclo (pois caso contrário um vu-caminho teria sido seguido pela busca). Assim, essa linha leva tempo $O(d^+(W[i])mn^2)$ para ser executada usando matriz de adjacências ou $O((d^+(W[i]))^2m(n+m))$ usando listas de adjacências.

Capítulo 27

Caminhos mínimos

Na Seção 23.9, definimos formalmente o termo *distância* em (di)grafos. Por comodidade, repetimos a seguir as partes daquele texto mais importantes para este capítulo. Seja G um (di)grafo ponderado nas arestas, com $w\colon E(G) \to \mathbb{R}$ sendo a função de peso. Denotamos a *distância entre u e v em G* por $\mathrm{dist}_G^w(u,v)$ e a definimos como o peso de um uv-caminho de menor peso. Lembre-se que o peso de um caminho $P = (v_0, v_1, \ldots, v_k)$ é igual à soma dos pesos das arestas ou arcos desse caminho, isto é, $w(P) = \sum_{i=0}^{k-1} w(v_i v_{i+1})$. Se não existe caminho entre u e v, então convencionamos que $\mathrm{dist}_G^w(u,v) = \infty$. Também convencionamos que $\mathrm{dist}_G^w(u,u) = 0$. Se um uv-caminho tem peso igual à distância entre u e v, então dizemos que ele é um uv-caminho *mínimo*.

Os algoritmos que lidam com problemas de encontrar distâncias em (di)grafos não funcionam corretamente quando o grafo possui arestas com pesos negativos e o digrafo possui ciclos com pesos negativos. Com o que se sabe até o momento em Ciência da Computação, não é possível existir um algoritmo eficiente que resolva problemas de distância nessas situações [1]

Uma tecnicalidade que precisa ser discutida é sobre considerar grafos ou digrafos. Lembre-se que dado um grafo G, seu digrafo associado é o digrafo $D(G)$ com conjunto de vértices $V(D(G)) = V(G)$ e $\{u,v\} \in E(G)$ se e somente se $(u,v) \in E(D(G))$ e $(v,u) \in E(D(G))$ (Seção 23.1). Ademais, se G é ponderado nas arestas por uma função w, então podemos ponderar $D(G)$ nos arcos com uma função w' fazendo $w'(uv) = w'(vu) = w(uv)$. Por fim, lembre-se da discussão acima que em G o problema só será resolvido se G não contém arestas com peso negativo. Como $w(uv) < 0$ implica que (u,v,u) é um ciclo com peso negativo em $D(G)$, então em $D(G)$ o problema também não será resolvido. Assim vemos que se

[1] Essa afirmação será provada no Capítulo 29.

conseguirmos resolver o problema em qualquer tipo de digrafo (que não tenha ciclo negativo), então conseguiremos resolvê-lo, em particular, para digrafos que são digrafos associados de grafos. Dessa forma, podemos encontrar caminhos mínimos em grafos se conseguirmos fazê-lo em digrafos. Por isso, consideraremos apenas digrafos a partir de agora, pois dessa forma ganhamos em generalidade.

Finalmente, vamos considerar duas variações do problema de calcular caminhos mínimos, definidas a seguir.

Problema 27.1: ***Caminhos mínimos de única fonte***

Dados um digrafo D, uma função w de peso nos arcos e um vértice $s \in V(D)$, calcular $\text{dist}_D^w(s, v)$ para todo $v \in V(D)$.

Problema 27.2: ***Caminhos mínimos entre todos os pares***

Dados um digrafo D e uma função w de peso nos arcos, calcular $\text{dist}_D^w(u, v)$ para todo par $u, v \in V(D)$.

Nas seções a seguir apresentaremos algoritmos clássicos que resolvem os problemas quando pesos estão envolvidos.

Antes disso, é importante observar que caso o digrafo não tenha pesos nos arcos (ou todos os arcos tenham peso idêntico), então o algoritmo de busca em largura resolve muito bem o problema de caminhos mínimos de única fonte (Seção 24.1.1). Mas ele também pode ser utilizado caso existam pesos inteiros positivos nos arcos. Seja D um digrafo e $w\colon E(D) \to \mathbb{Z}_+$ uma função de custo nos arcos de D. Construa o digrafo H tal que cada arco $e \in E(D)$ é substituído por um caminho com $w(e)$ arcos em H. Assim, H possui os mesmos vértices de G e alguns vértices extras. É fácil mostrar que para quaisquer $u, v \in V(D)$ (e, portanto, $u, v \in V(H)$), um uv-caminho em D tem peso mínimo se e somente se o uv-caminho correspondente em H tem o menor número de arcos. Assim, uma busca em largura sobre H resolve o problema em D. Qual é o problema dessa abordagem? Deve haver um, pois caso contrário não precisaríamos de todos os algoritmos que serão vistos nas próximas seções.

Também é importante para as discussões a seguir perceber que se $P = (v_1, \ldots, v_k)$ é um v_1v_k-caminho mínimo, então qualquer subcaminho $(v_i, \ldots, v_j)$ de P também é mínimo. Suponha que existam pares $1 \leq i < j \leq k$ tais que $(v_i, v_{i+1} \ldots, v_{j-1}, v_j)$ não é mínimo. Seja então $(v_i, u_1, \ldots, u_q, v_j)$ um v_iv_j-caminho mínimo. Ao substituir esse trecho em P, temos que $(v_1, \ldots, v_i, u_1, \ldots, u_q, v_j, v_{j+1}, \ldots, v_k)$ é um v_1v_k-caminho de peso menor do que o peso

de P, o que é uma contradição.

27.1 Única fonte

Nesta seção apresentaremos dois algoritmos clássicos que resolvem o Problema 27.1, dos caminhos mínimos de única fonte, que são Dijkstra e Bellman-Ford. Considere um digrafo com n vértices e m arcos. O algoritmo de Bellman-Ford é executado em tempo $\Theta(mn)$. Já o algoritmo de Dijkstra é executado em tempo $O((m+n)\log n)$. Assim, se m for assintoticamente menor que $\log n$, então o algoritmo de Dijkstra é mais eficiente. Representamos essa relação entre m e $\log n$ como $m = \omega(\log n)$ (para detalhes dessa notação, veja a Seção 5.2). De fato, como $m = \omega(\log n)$, temos que $mn = \omega(n \log n)$. Como sabemos que $n = O(n - \log n)$, obtemos que $m(n - \log n) = \omega(n \log n)$, de onde concluímos que $(m+n)\log n = \omega(mn)$. Portanto, o tempo de execução de Dijkstra é, nesse caso, assintoticamente menor que o tempo de execução de Belmann-Ford.

Apesar do algoritmo de Dijkstra em geral ser mais eficiente, o algoritmo de Bellman-Ford tem a vantagem de funcionar em digrafos que contêm arcos de peso negativo, diferentemente do algoritmo de Dijkstra. Por fim, observamos que o algoritmo de Bellman-Ford também tem a capacidade de identificar a existência de ciclos negativos no digrafo. Nas próximas seções apresentamos esses algoritmos em detalhes. A seguir definimos alguns conceitos importantes em ambas.

Dado um digrafo D, uma função w de pesos nos arcos e um vértice $s \in V(D)$ qualquer, queremos calcular os pesos de caminhos mínimos de s para todos os outros vértices do digrafo. Para resolver esse problema, todo vértice $v \in V(D)$ terá um atributo $v.\texttt{distancia}$, que manterá o peso do caminho que o algoritmo calculou de s até v. Chamaremos esse valor também de *estimativa* (da distância). A ideia é que ao fim da execução dos algoritmos todo vértice v tenha $v.\texttt{distancia} = \text{dist}_D^w(s, v)$. Cada vértice possui também um atributo $v.\texttt{predecessor}$, que contém o predecessor de v no sv-caminho que foi encontrado pelo algoritmo, e um atributo $v.\texttt{visitado}$, que indica se v já foi alcançado a partir de s ou não. Já vimos que com o atributo $v.\texttt{predecessor}$ é possível construir todo o sv-caminho encontrado pelo algoritmo, utilizando o Algoritmo 24.4, ConstroiCaminho.

Uma peça chave em algoritmos que resolvem esse problema é um procedimento chamado de *relaxação*. Os algoritmos que vamos considerar modificam os atributos `distancia` dos vértices por meio de relaxações. Dizemos que um arco uv é *relaxada* quando verificamos se $v.\texttt{distancia} > u.\texttt{distancia} + w(uv)$, e atualizamos, em caso positivo, o valor de $v.\texttt{distancia}$ para $u.\texttt{distancia} + w(uv)$. Em outras palavras, se o peso do sv-caminho calculado até o momento é maior do que o peso do sv-caminho construído utilizando o su-caminho

calculado até o momento seguido do arco uv, então atualizamos a estimativa da distância até v com o valor da estimativa até u somada ao custo desse arco.

Dado um caminho $P = (v_0, v_1, v_2, \ldots, v_k)$ em um digrafo, dizemos que uma sequência de relaxações de arcos é *P-ordenada* se os arcos v_0v_1, v_1v_2, ..., $v_{k-1}v_k$ forem relaxadas nesta ordem, não importando se outros arcos forem relaxadas entre quaisquer das relaxações v_0v_1, v_1v_2, ..., $v_{k-1}v_k$. O lema abaixo é a peça chave para os algoritmos que veremos.

Lema 27.3

Considere um digrafo D, uma função de pesos w em seus arcos e $s \in V(D)$. Façamos $s.\texttt{distancia} = 0$ e $v.\texttt{distancia} = \infty$ para todo vértice $v \in V(D) \setminus \{s\}$. Seja A um algoritmo que modifica as estimativas de distância somente através de relaxações.

1. Para todo $v \in V(D)$, em qualquer momento da execução de A temos $v.\texttt{distancia} \geq \text{dist}_D^w(s, v)$.

2. Se $P = (s, v_1, v_2, \ldots, v_k)$ é um sv_k-caminho mínimo e A realiza uma sequência P-ordenada de relaxações, então teremos $v_k.\texttt{distancia} = \text{dist}_D^w(s, v_k)$ ao fim da execução de A;

Demonstração. Comecemos mostrando o item 1. Suponha, para fins de contradição, que em algum momento da execução de A temos um vértice v com $v.\texttt{distancia} < \text{dist}_D^w(s, v)$. Seja v o *primeiro* vértice a ficar com $v.\texttt{distancia} < \text{dist}_D^w(s, v)$ durante a sequência de relaxações. Pela natureza do algoritmo A, o vértice v teve $v.\texttt{distancia}$ alterado quando algum arco uv foi relaxado. Pela escolha de v, sabemos que $u.\texttt{distancia} \geq \text{dist}_D^w(s, u)$. Portanto, ao relaxar o arco uv, obtivemos $v.\texttt{distancia} = u.\texttt{distancia} + w(uv) \geq \text{dist}_D^w(s, u) + w(uv) \geq \text{dist}_D^w(s, v)$, uma contradição. A última desigualdade segue do fato de $\text{dist}_D^w(s, u) + w(uv)$ ser o comprimento de um possível sv-caminho.

O resultado do item 2 será por indução na quantidade de arcos de um caminho mínimo $P = (s, v_1, v_2, \ldots, v_k)$. Se o comprimento do caminho é 0, i.e., não há arcos, então o caminho é formado somente pelo vértice s. Logo, tem distância 0. Para esse caso, o teorema é válido, dado que temos $s.\texttt{distancia} = 0 = \text{dist}_D^w(s, s)$.

Seja então $k \geq 1$ e suponha que para todo caminho mínimo com menos de k arcos o teorema é válido.

Primeiro considere um caminho mínimo $P = (s, v_1, v_2, \ldots, v_k)$ de s a v_k com k arcos e suponha que os arcos sv_1, v_1v_2, ..., $v_{k-1}v_k$ foram relaxados nessa ordem. Note que como $P' = (s, v_1, v_2, \ldots, v_{k-1})$ é um caminho dentro de um caminho mínimo, então P' também é um caminho mínimo. Assim, $\text{dist}_D^w(s, v_k) = \text{dist}_D^w(s, v_{k-1}) + w(v_{k-1}v_k)$.

Note que como os arcos de P', a saber sv_1, v_1v_2, ..., $v_{k-2}v_{k-1}$, foram relaxadas nessa ordem e P' tem $k-1$ arcos, concluímos por hipótese de indução que $v_{k-1}.\texttt{distancia} = \text{dist}_D^w(s, v_{k-1})$. Caso $v_k.\texttt{distancia} = \text{dist}_D^w(s, v_k)$, então a prova está concluída. Assim, podemos assumir que

$$v_k.\texttt{distancia} \neq \text{dist}_D^w(s, v_k) \ . \tag{27.1}$$

Não é o caso de $v_k.\texttt{distancia} < \text{dist}_D^w(s, v_k)$, como já vimos no item 1. Assim, $v_k.\texttt{distancia} > \text{dist}_D^w(s, v_k)$. Ao relaxar o arco $v_{k-1}v_k$, o algoritmo vai verificar que $v_k.\texttt{distancia} > \text{dist}_D^w(s, v_k) = \text{dist}_D^w(s, v_{k-1}) + w(v_{k-1}v_k) = v_{k-1}.\texttt{distancia} + w(v_{k-1}v_k)$, atualizando o valor de $v_k.\texttt{distancia}$, portanto, para $v_{k-1}.\texttt{distancia} + w(v_{k-1}v_k)$, que é igual a $\text{dist}_D^w(s, v_k)$, uma contradição com (27.1). Logo, $v_k.\texttt{distancia} = \text{dist}_D^w(s, v_k)$. □

27.1.1 Algoritmo de Dijkstra

Um dos algoritmos mais clássicos na Ciência da Computação é o *algoritmo de Dijkstra*, que resolve o problema de caminhos mínimos de única fonte (Problema 27.1). Esse algoritmo é muito eficiente, mas tem um ponto fraco, que é o fato de não funcionar quando os pesos nos arcos são negativos. Assim, seja D um digrafo, w uma função de peso sobre os arcos de D e $s \in V(D)$ um vértice qualquer. Nesta seção, vamos considerar que $w(e) \geq 0$ para todo $e \in E(D)$. Nosso objetivo é calcular sv-caminhos mínimos para todo $v \in V(D)$.

A ideia do algoritmo de Dijkstra é similar à dos algoritmos de busca vistos no Capítulo 24. De forma geral, teremos um conjunto de vértices visitados e um conjunto de vértices não visitados. Dizer que um vértice foi visitado significa que o algoritmo foi capaz de encontrar um caminho de s até ele e que sua estimativa de distância não mudará mais até o fim da execução. Inicialmente, nenhum vértice está visitado, s tem estimativa de distância $s.\texttt{distancia} = 0$, e todos os outros vértices têm estimativa de distância $v.\texttt{distancia} = \infty$. A cada iteração, um vértice não visitado x é escolhido para ser visitado (e, portanto, o valor em $x.\texttt{distancia}$ não mudará). Neste momento, todos os arcos que saem de x são relaxados. Isso acontece porque o peso do sx-caminho calculado (que está em $x.\texttt{distancia}$) seguido do peso do arco xy pode ser melhor do que a estimativa atual de y (que está em $y.\texttt{distancia}$). A escolha que Dijkstra faz para escolher o próximo vértice x a ser visitado é gulosa, pelo vértice que, naquele momento, tem a menor estimativa de distância dentre os não visitados. Esse procedimento é repetido enquanto houver vértices não visitados com atributo $\texttt{distancia} \neq \infty$, pois isso significa que em algum momento houve uma relaxação de um arco que chega nesses vértices, isto é, há caminho a partir de s até tal vértice.

Consideraremos que todo vértice $v \in V(D)$ possui atributos $v.\texttt{predecessor}$ e $v.\texttt{visitado}$, além do atributo $v.\texttt{distancia}$ já mencionado. O atributo $v.\texttt{predecessor}$ deve conter o

predecessor de v no sv-caminho que está sendo construído pelo algoritmo. Já o atributo $v.\texttt{visitado}$ deve ter valor 1 se v foi visitado e 0 caso contrário.

O Algoritmo 27.1 formaliza o algoritmo de Dijkstra. Note que o digrafo T tal que

$$V(T) = \{v \in V(D) \colon v.\texttt{predecessor} \neq null\} \cup \{s\}$$
$$E(T) = \{(v.\texttt{predecessor}, v) \colon v \in V(T) \setminus \{s\}\}$$

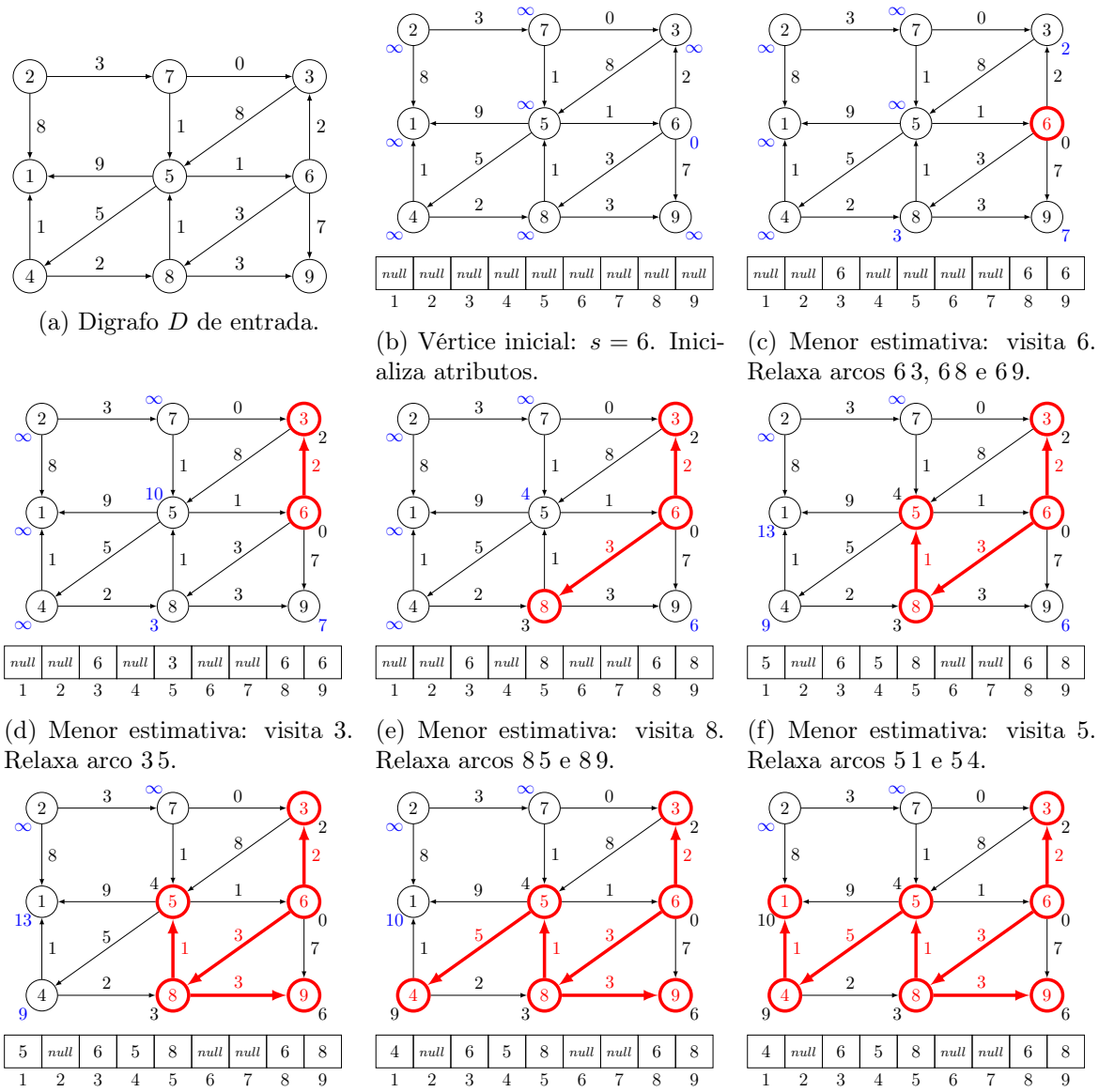
é um subdigrafo que é arborescência em D (não necessariamente geradora) e contém um único sv-caminho para qualquer $v \in V(T)$. Tal caminho, cujo peso é $v.\texttt{distancia}$, pode ser construído pelo Algoritmo 24.4, ConstroiCaminho. A Figura 27.1 mostra um exemplo de execução.

Algoritmo 27.1: Dijkstra(D, w, s)

1 **para** *todo vértice* $v \in V(D)$ **faça**
2 $\quad v.\texttt{predecessor} = null$
3 $\quad v.\texttt{distancia} = \infty$
4 $\quad v.\texttt{visitado} = 0$
5 $s.\texttt{distancia} = 0$
6 **enquanto** *houver vértice* u *com* $u.\texttt{visitado} == 0$ *e* $u.\texttt{distancia} \neq \infty$ **faça**
7 $\quad$ seja x um vértice não visitado com menor valor $x.\texttt{distancia}$
8 $\quad x.\texttt{visitado} = 1$
9 $\quad$ **para** *todo vértice* $y \in N^+(x)$ **faça**
10 $\quad\quad$ **se** $y.\texttt{visitado} == 0$ **então**
11 $\quad\quad\quad$ **se** $x.\texttt{distancia} + w(xy) < y.\texttt{distancia}$ **então**
12 $\quad\quad\quad\quad y.\texttt{distancia} = x.\texttt{distancia} + w(xy)$
13 $\quad\quad\quad\quad y.\texttt{predecessor} = x$

Note que as linhas 11, 12 e 13 realizam a relaxação do arco xy. Perceba que em nenhum momento o algoritmo de Dijkstra verifica se o digrafo de entrada possui arcos de peso negativo. De fato, ele encerra sua execução normalmente. Acontece, porém, que ele não calcula os pesos dos caminhos mínimos *corretamente*. O Teorema 27.4 a seguir mostra que o algoritmo de Dijkstra funciona corretamente quando os pesos dos arcos são não negativos.

Teorema 27.4



(a) Digrafo D de entrada.

(b) Vértice inicial: $s = 6$. Inicializa atributos.

(c) Menor estimativa: visita 6. Relaxa arcos 6 3, 6 8 e 6 9.

(d) Menor estimativa: visita 3. Relaxa arco 3 5.

(e) Menor estimativa: visita 8. Relaxa arcos 8 5 e 8 9.

(f) Menor estimativa: visita 5. Relaxa arcos 5 1 e 5 4.

(g) Menor estimativa: visita 9. Sem arcos para relaxar.

(h) Menor estimativa: visita 4. Relaxa arco 4 1.

(i) Menor estimativa: visita 1. Sem arcos para relaxar.

Figura 27.1: Execução de Dijkstra(D, w, 6). Vértices visitados estão em vermelho. Estimativas de distância estão em azul ao lado dos vértices. Predecessores são indicados no vetor. A árvore construída indiretamente pelos predecessores está em vermelho. Depois de 27.1i, não há vértices não visitados com `distancia` $\neq \infty$.

Seja D um digrafo, w uma função de peso nos arcos de D com $w(e) \geq 0$ para todo $e \in E(D)$, e $s \in V(D)$ um vértice qualquer. Ao final da execução de DIJKSTRA(G, w, s) temos $v.\texttt{distancia} = \text{dist}_D^w(s, v)$ para todo $v \in V(D)$.

Demonstração. Considere uma execução de DIJKSTRA(G, w, s). Inicialmente perceba que a cada iteração do laço **enquanto**, um vértice que ainda não foi visitado pelo algoritmo é visitado e uma vez que um vértice é visitado essa condição não muda mais. Assim, o algoritmo certamente encerra sua execução após $O(v(D))$ iterações do laço **enquanto**.

Precisamos mostrar que ao fim da execução temos $v.\texttt{distancia} = \text{dist}_D^w(s, v)$ para todo $v \in V(D)$. Inicialmente, fazemos $v.\texttt{distancia} = \infty$ e $s.\texttt{distancia} = 0$. A partir daí, o algoritmo só modifica as estimativas $v.\texttt{distancia}$ através de relaxações. Assim, pelo Lema 27.3, sabemos que $v.\texttt{distancia} \geq \text{dist}_D^w(s, v)$. Uma vez que o algoritmo nunca modifica o atributo $v.\texttt{distancia}$ depois que v é visitado, basta provarmos que $v.\texttt{distancia} \leq \text{dist}_D^w(s, v)$ *no momento em que v é visitado.*

Mostraremos, por indução na quantidade i de iterações, que para todo $v \in V(D)$ temos $v.\texttt{distancia} \leq \text{dist}_D^w(s, v)$ no momento em que v é visitado.

Quando $i = 1$, note que s é escolhido na linha 7 para ser visitado. Isso porque antes da primeira iteração começar, $s.\texttt{distancia} = 0$ e todo outro vértice tem $\texttt{distancia} = \infty$, de forma que s tem o menor valor em $\texttt{distancia}$. Assim, neste momento, $0 = s.\texttt{distancia} \leq \text{dist}_D^w(s, s) = 0$ e o resultado vale.

Considere então uma iteração $i > 1$ qualquer e seja x o vértice visitado durante ela. Claramente, $x \neq s$. Suponha que todos os vértices u visitados nas iterações anteriores têm $u.\texttt{distancia} < \text{dist}_D^w(s, u)$.

Seja P um sx-caminho mínimo qualquer e seja t o primeiro vértice não visitado de P, dentre os vértices que não estão visitados na iteração i atual. Ademais, seja z o vértice que precede t no caminho P. Pela escolha de t, sabemos que z é visitado. Assim, por hipótese de indução, sabemos que $z.\texttt{distancia} \leq \text{dist}_D^w(s, z)$, o que significa que $z.\texttt{distancia} = \text{dist}_D^w(s, z)$. Note que $w(P) = \text{dist}_D^w(s, x)$.

Note que podemos descrever o peso de P como $\text{dist}_D^w(s, x) = w(P) = w(P') + w(zt) + w(P'')$, em que P' é o subcaminho que vai de s a z e P'' é o subcaminho que vai de t a x; Além disso, P' é um sz-caminho mínimo e certamente $w(P'') \geq 0$, pois não há arcos com peso negativo em D. Logo, $\text{dist}_D^w(s, x) \geq w(P') + w(zt) = \text{dist}_D^w(s, z) + w(zt) = z.\texttt{distancia} + w(zt)$. No momento em que z foi visitado, os arcos que saem de z para vértices não visitados foram relaxados, incluindo o arco zt. Portanto, $t.\texttt{distancia} \leq z.\texttt{distancia} + w(zt)$. Juntando com o resultado anterior, temos $t.\texttt{distancia} \leq \text{dist}_D^w(s, x)$.

Por fim, note que no momento em que x é escolhido para ser visitado, o vértice t

ainda não tinha sido visitado, de forma que ele era uma possível escolha para o algoritmo. Como o algoritmo faz uma escolha pelo vértice de menor estimativa de distância, temos $x.\texttt{distancia} \leq t.\texttt{distancia}$, o que junto com o resultado anterior nos diz que $x.\texttt{distancia} \leq \text{dist}_D^w(s, x)$, concluindo a prova do teorema. □

Seja D um digrafo, w uma função de peso nos arcos e $s \in V(D)$ qualquer. Voltemos nossa atenção para o tempo de execução de Dijkstra(D, w, s). No que segue, considere $n = v(D)$ e $m = e(D)$. A inicialização dos vértices, no laço **para** da linha 1, claramente leva tempo $\Theta(n)$. Agora note a cada iteração do laço **enquanto** da linha 6, um vértice que ainda não foi visitado pelo algoritmo é visitado e uma vez que um vértice é visitado essa condição não muda mais. Assim, existem $O(n)$ iterações desse laço, uma vez que nem todos os vértices possuem caminho a partir de s. Com isso, a linha 8, de tempo constante, leva tempo total $O(n)$ para ser executada. Todos os comandos internos ao laço **para** da linha 9 são de tempo constante e o comando de teste da linha 10 é sempre executado. Assim, se o digrafo foi dado em matriz de adjacências, uma única execução do laço leva tempo $O(n)$ e ele leva tempo $O(n^2)$ ao todo. Se foi dado em listas de adjacências, então uma execução dele leva tempo $\Theta(|N^+(x)|)$ e ele leva tempo $\sum_x \Theta(|N^+(x)|) = O(m)$ ao todo.

Resta analisar o tempo gasto para executar as linhas 6 e 7, que não executam em tempo constante. Note que ambas podem ser executadas de forma ingênua em tempo $\Theta(n)$. Lembre-se que ambas são executadas $O(n)$ vezes ao todo.

Assim, essa implementação simples leva tempo $\Theta(n) + O(n) + O(n^2) + O(n)\Theta(n) + O(n)\Theta(n) = O(n^2)$, em matriz de adjacências, ou $\Theta(n) + O(n) + O(m) + O(n)\Theta(n) + O(n)\Theta(n) = O(n^2 + m)$.

De fato, a operação mais custosa é a procura por um vértice cujo atributo `distancia` é o menor possível (e diferente de ∞) e sua "remoção" do conjunto de vértices não visitados. Felizmente, é possível melhorar esses tempos de execução através do uso de uma estrutura de dados apropriada para esse tipo de operação. Heap é uma estrutura que oferece a operação RemoveDaHeap, que remove o elemento de maior prioridade em tempo $O(\log k)$, onde k é a quantidade de elementos armazenados na estrutura. Veja mais sobre heaps na Seção 12.1.

Como o algoritmo de Dijkstra faz uma escolha por um vértice que ainda não foi visitado e que tenha a menor estimativa de distância, vamos utilizar um heap para armazenar os vértices não visitados e o valor da prioridade de um vértice v será justamente $-v.\texttt{distancia}$. Note que vamos manter um valor negativo pois tem maior prioridade o vértice que tiver menor valor em $v.\texttt{distancia}$. Assim, o próximo vértice a ser visitado deve ser o vértice removido do heap.

Assuma que $V(D) = \{1, \ldots, v(D)\}$ e que cada vértice v ainda possui os atributos

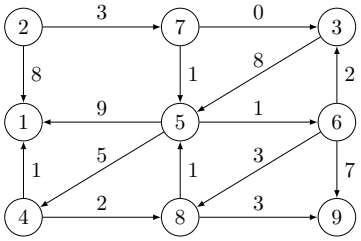
`prioridade`, para armazenar seu valor de prioridade, e `indice`, para indicar a posição do heap em que v está armazenado. O Algoritmo 27.2 reapresenta o algoritmo de Dijkstra utilizando explicitamente a estrutura heap. Lembre-se que InsereNaHeap(H, v) insere o elemento v em H, RemoveDaHeap(H) remove e devolve o elemento de maior prioridade de H e AlteraHeap(H, v.`indice`, x) atualiza o valor em v.`prioridade` para x. Todas essas operações mantêm a propriedade de heap em H. A Figura 27.2 mostra a execução de Dijkstra-Heap sobre o mesmo grafo da Figura 27.1, porém considerando essa implementação.

Algoritmo 27.2: Dijkstra-Heap(D, w, s)

```
1  Seja H[1..v(D)] um vetor vazio
2  para todo vértice v ∈ V(D) faça
3      v.predecessor = null
4      v.prioridade = −∞
5      v.visitado = 0
6      InsereNaHeap(H, v)
7  AlteraHeap(H, s.indice, 0)
8  enquanto u = ConsultaHeap(H) e u.prioridade ≠ −∞ faça
9      x = RemoveDaHeap(H)
10     x.visitado = 1
11     para todo vértice y ∈ N⁺(x) faça
12         se y.visitado == 0 então
13             se x.prioridade +(−w(xy)) < y.prioridade então
14                 AlteraHeap(H, y.indice, x.prioridade +(−w(xy)))
15                 y.predecessor = x
```

Seja D um digrafo, w uma função de peso nos arcos e $s \in V(D)$ qualquer. Vamos analisar o tempo de execução de Dijkstra-Heap(D, w, s). No que segue, considere $n = v(D)$ e $m = e(D)$. A inicialização dos vértices, no laço **para** da linha 2, agora leva tempo $O(n \log n)$, pois o laço executa $\Theta(n)$ vezes mas uma chamada a InsereNaHeap leva tempo $O(\log n)$. A chamada a AlteraHeap na linha 7 leva tempo $O(\log n)$.

Sobre o laço **enquanto**, na linha 8, veja que seu teste agora é feito em tempo $\Theta(1)$. Como o laço executa $O(n)$ vezes, essa linha e a linha 10 levam tempo total $O(n)$. A linha 9, que chama RemoveDaHeap, leva tempo $O(\log n)$ e, portanto, ao todo executa em tempo $O(n \log n)$. Os comandos internos ao laço **para** da linha 11 levam tempo constante para serem executados, exceto pela chamada a AlteraHeap na linha 14, que leva tempo $O(\log n)$. No



(a) Digrafo D de entrada. Vértice inicial: $s = 6$.

H	6	2	1	4	5	3	7	8	9
	1	2	3	4	5	6	7	8	9

	1	2	3	4	5	6	7	8	9
predecessor	*null*	*null*	*null*	*null*	*null*	*null*	*null*	*null*	*null*
visitado	0	0	0	0	0	0	0	0	0
indice	3	2	6	4	5	1	7	8	9
prioridade	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$

(b) Inicializa heap. Apenas 6. `prioridade` $= 0$.

H	3	8	9	2	5	1	7	4
	1	2	3	4	5	6	7	8

	1	2	3	4	5	6	7	8	9
predecessor	*null*	*null*	6	*null*	*null*	*null*	*null*	6	6
visitado	0	0	0	0	0	1	0	0	0
indice	6	4	1	8	5		7	2	3
prioridade	$-\infty$	$-\infty$	-2	$-\infty$	$-\infty$	0	$-\infty$	-3	-7

(c) RemoveDaHeap(H) $= 6$. Relaxa arcos 6 3, 6 8 e 6 9.

H	8	5	9	2	4	1	7
	1	2	3	4	5	6	7

	1	2	3	4	5	6	7	8	9
predecessor	*null*	*null*	6	*null*	3	*null*	*null*	6	6
visitado	0	0	1	0	0	1	0	0	0
indice	6	4		5	2		7	1	3
prioridade	$-\infty$	$-\infty$	-2	$-\infty$	-10	0	$-\infty$	-3	-7

(d) RemoveDaHeap(H) $= 3$. Relaxa arco 3 5.

H	5	9	7	2	4	1
	1	2	3	4	5	6

	1	2	3	4	5	6	7	8	9
predecessor	*null*	*null*	6	*null*	8	*null*	*null*	6	8
visitado	0	0	1	0	0	1	0	1	0
indice	6	4		5	1		3		2
prioridade	$-\infty$	$-\infty$	-2	$-\infty$	-4	0	$-\infty$	-3	-6

(e) RemoveDaHeap(H) $= 8$. Relaxa arcos 8 5 e 8 9.

H	9	4	7	2	1
	1	2	3	4	5

	1	2	3	4	5	6	7	8	9
predecessor	5	*null*	6	5	8	*null*	*null*	6	8
visitado	0	0	1	0	1	1	0	1	0
indice	5	4		2			3		1
prioridade	-13	$-\infty$	-2	-9	-4	0	$-\infty$	-3	-6

(f) RemoveDaHeap(H) $= 5$. Relaxa arcos 5 1 e 5 4.

H	4	1	7	2
	1	2	3	4

	1	2	3	4	5	6	7	8	9
predecessor	5	*null*	6	5	8	*null*	*null*	6	8
visitado	0	0	1	0	1	1	0	1	1
indice	2	4		1			3		
prioridade	-13	$-\infty$	-2	-9	-4	0	$-\infty$	-3	-6

(g) RemoveDaHeap(H) $= 9$. Sem arcos para relaxar.

H	1	2	7
	1	2	3

	1	2	3	4	5	6	7	8	9
predecessor	4	*null*	6	5	8	*null*	*null*	6	8
visitado	0	0	1	1	1	1	0	1	1
indice	1	2					3		
prioridade	-10	$-\infty$	-2	-9	-4	0	$-\infty$	-3	-6

(h) RemoveDaHeap(H) $= 4$. Relaxa arco 4 1.

H	7	2
	1	2

	1	2	3	4	5	6	7	8	9
predecessor	4	*null*	6	5	8	*null*	*null*	6	8
visitado	1	0	1	1	1	1	0	1	1
indice		2					1		
prioridade	-10	$-\infty$	-2	-9	-4	0	$-\infty$	-3	-6

(i) RemoveDaHeap(H) $= 1$. Sem arcos para relaxar.

Figura 27.2: Execução de Dijkstra-Heap(D, w, 6), que considera a implementação de Dijkstra com heap. Note que em H estamos mostrando os rótulos dos vértices, e não suas prioridades.

entanto, apenas o comando condicional da linha 12 executa sempre. Assim, se o digrafo foi dado em matriz de adjacências, essa linha executa $O(n)$ vezes, levando tempo total $O(n^2)$. Se foi dado em listas de adjacências, então ela executa $\Theta(|N^+(x)|)$ vezes, levando tempo total $\sum_x \Theta(|N^+(x)|) = O(m)$.

Já a chamada a AlteraHeap na linha 14 executa sempre que um arco é relaxado. Note que isso ocorre no máximo uma vez para cada arco xy, que é no momento em que x é visitado. Assim, são no máximo $O(m)$ execuções dessa linha, o que dá um tempo total de $O(m \log n)$.

Portanto, o tempo total de execução de Dijkstra-Heap(D, w, s) é $O(n \log n)+O(\log n)+O(n)+O(n \log n)+O(n^2)+O(m \log n) = O(m \log n+n^2)$ em matriz de adjacências. Já em listas de adjacências, o tempo é $O(n \log n)+O(\log n)+O(n)+O(n \log n)+O(m)+O(m \log n) = O((m+n) \log n)$.

27.1.2 Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford resolve o problema de caminhos mínimos de única fonte mesmo quando há arcos de peso negativo no digrafo em questão. Mais ainda, quando existe um ciclo de peso total negativo, o algoritmo identifica a existência de tal ciclo. Assim, seja D um digrafo, w uma função de peso sobre os arcos de D e $s \in V(D)$. Nosso objetivo é calcular sv-caminhos mínimos para todo $v \in V(D)$. Consideraremos que $V(D) = \{1, \ldots, v(D)\}$.

A ideia do algoritmo de Bellman-Ford é tentar, em $v(D) - 1$ iterações, melhorar a estimativa de distância conhecida a partir de s para todos os vértices v analisando todos os $e(D)$ arcos de D em cada iteração. A intuição por trás dessa ideia é garantir que, dado um sv-caminho mínimo, o algoritmo relaxe os arcos desse caminho em ordem. Deste modo, a corretude do algoritmo é garantida pelo Lema 27.3.

Consideraremos que todo vértice $v \in V(D)$ possui um atributo $v.\texttt{predecessor}$, além do atributo $v.\texttt{distancia}$ já mencionado. O atributo $v.\texttt{predecessor}$ deve contar o predecessor de v no sv-caminho que está sendo construído pelo algoritmo.

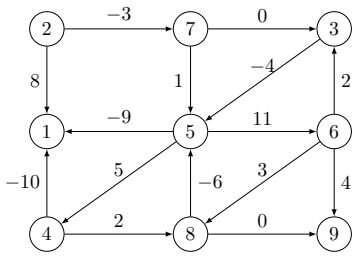
O Algoritmo 27.3 formaliza o algoritmo de Bellman-Ford. Consideramos que o digrafo D tem um atributo $D.\texttt{cicloNegativo}$, que ao fim de uma execução do algoritmo terá valor 1 se D tem ciclos negativos e 0 caso contrário. Note que a arborescência T tal que

$$V(T) = \{v \in V(D) \colon v.\texttt{predecessor} \neq null\} \cup \{s\}$$
$$E(T) = \{(v.\texttt{predecessor}, v) \colon v \in V(T) \setminus \{s\}\}$$

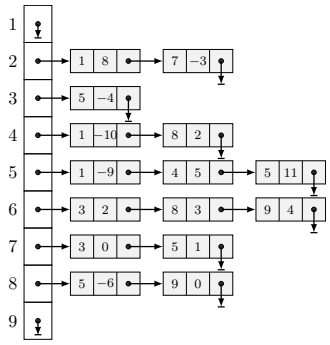
é uma arborescência de D (não necessariamente geradora) e contém um único sv-caminho para qualquer $v \in V(T)$. Tal caminho pode ser construído pelo Algoritmo 24.4, Constroi-Caminho. As Figuras 27.3 e 27.4 mostram exemplos de execução.

Algoritmo 27.3: BELLMAN-FORD(D, w, s)

```
D.cicloNegativo = 0
para todo vértice v ∈ V(D) faça
    v.distancia = ∞
    v.predecessor = null
s.distancia = 0
para i = 1 até v(D) − 1, incrementando faça
    para todo arco xy ∈ E(D) faça
        se y.distancia > x.distancia + w(xy) então
            y.distancia = x.distancia + w(xy)
            y.predecessor = x
para todo arco xy ∈ E(D) faça
    se y.distancia > x.distancia + w(xy) então
        D.cicloNegativo = 1
```



(a) Digrafo D de entrada. Vértice inicial: $s = 6$.



(b) Lista de adjacências de D.

	1	2	3	4	5	6	7	8	9
1									
2	8						−3		
3					−4				
4	−10							2	
5	−9			5		11			
6			2					3	4
7			0		1				
8					−6				0
9									

(c) Matriz de adjacências de D.

$$((2,1),(2,7),(3,5),(4,1),(4,8),(5,1),(5,4),(5,6),(6,3),(6,8),(6,9),(7,3),(7,5),(8,5),(8,9))$$

(d) Sequência dos arcos seguida pelo algoritmo.

	1	2	3	4	5	6	7	8	9
distancia	∞	∞	∞	∞	∞	0	∞	∞	∞
predecessor	*null*	*null*	*null*	*null*	*null*	*null*	*null*	*null*	*null*

(e) Inicializa atributos.

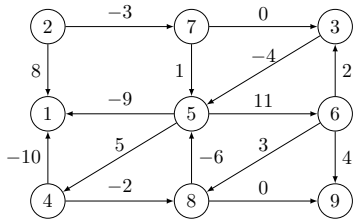
	1	2	3	4	5	6	7	8	9
distancia	∞	∞	2	∞	−3	0	∞	3	3
predecessor	*null*	*null*	6	*null*	8	*null*	*null*	6	8

(f) Após iteração $i = 1$.

	1	2	3	4	5	6	7	8	9
distancia	−12	∞	2	2	−3	0	∞	3	3
predecessor	5	*null*	6	5	8	*null*	*null*	6	8

(g) Após iteração $i = 2$.

Figura 27.3: Execução de BELLMAN-FORD(D, w, 6). Após a iteração $i = 2$, não há mudanças.



(a) Digrafo D de entrada. Vértice inicial: $s = 6$.

$$((2,1),(2,7),(3,5),(4,1),(4,8),(5,1),(5,4),(5,6),$$
$$(6,3),(6,8),(6,9),(7,3),(7,5),(8,5),(8,9))$$

(b) Sequência dos arcos seguida pelo algoritmo.

	1	2	3	4	5	6	7	8	9
distancia	∞	∞	∞	∞	∞	0	∞	∞	∞
predecessor	null	null	null	null	null	null	null	null	null

(c) Inicializa atributos.

	1	2	3	4	5	6	7	8	9
distancia	∞	∞	2	∞	−3	0	∞	3	3
predecessor	null	null	6	null	8	null	null	6	8

(d) Após iteração $i = 1$.

	1	2	3	4	5	6	7	8	9
distancia	−12	∞	2	2	−3	0	∞	3	3
predecessor	5	null	6	5	8	null	null	6	8

(e) Após iteração $i = 2$.

	1	2	3	4	5	6	7	8	9
distancia	−12	∞	2	2	−6	0	∞	0	0
predecessor	5	null	6	5	8	null	null	4	8

(f) Após iteração $i = 3$.

	1	2	3	4	5	6	7	8	9
distancia	−15	∞	2	−1	−6	0	∞	0	0
predecessor	5	null	6	5	8	null	null	4	8

(g) Após iteração $i = 4$.

	1	2	3	4	5	6	7	8	9
distancia	−15	∞	2	−1	−9	0	∞	−3	−3
predecessor	5	null	6	5	8	null	null	4	8

(h) Após iteração $i = 5$.

	1	2	3	4	5	6	7	8	9
distancia	−18	∞	2	−4	−9	0	∞	−3	−3
predecessor	5	null	6	5	8	null	null	4	8

(i) Após iteração $i = 6$.

	1	2	3	4	5	6	7	8	9
distancia	−18	∞	2	−4	−12	0	∞	−6	−6
predecessor	5	null	6	5	8	null	null	4	8

(j) Após iteração $i = 7$.

	1	2	3	4	5	6	7	8	9
distancia	−21	∞	1	−7	−12	−1	∞	−6	−6
predecessor	5	null	6	5	8	5	null	4	8

(k) Após iteração $i = 8$.

Figura 27.4: Execução de Bellman-Ford(D, w, 6). O algoritmo detecta ciclo negativo.

Note que as linhas 8, 9 e 10 realizam a relaxação do arco xy. O Lema 27.3 garante que se os arcos de um sv-caminho mínimo forem relaxadas na ordem do caminho, então o algoritmo de Bellman-Ford calcula corretamente o peso de um sv-caminho mínimo. Mas como o algoritmo de Bellman-Ford garante isso para todo vértice $v \in V(D)$? A chave é notar que todo caminho tem no máximo $v(D) - 1$ arcos, de modo que relaxando todos os arcos $v(D) - 1$ vezes, é garantido que qualquer que seja o sv-caminho mínimo $P = (s, v_1, v_2, \ldots, v_k, v)$, os arcos desse caminho vão ser relaxados na ordem correta. Por exemplo, no digrafo da Figura 27.3, um 6 5-caminho mínimo é $P = (6, 8, 5)$, de peso -3. Note que como todos os arcos são visitados em cada iteração, temos a relaxação do arco 6 8 em uma iteração e a relaxação do arco 8 5 em uma iteração posterior. O Lema 27.5 a seguir torna a discussão acima precisa, mostrando que o algoritmo Bellman-Ford calcula corretamente os sv-caminhos mínimos, se não houver ciclo de peso negativo no digrafo.

Lema 27.5

Seja D um digrafo, w uma função de pesos em seus arcos e seja $s \in V(D)$. Se D não contém ciclos de peso negativo, então após a execução de Bellman-Ford(D, w, s) temos $v.\texttt{distancia} = \text{dist}_D^w(s, v)$ para todo vértice $v \in V(D)$. Ademais, temos

$D.\texttt{cicloNegativo} = 0$.

Demonstração. Suponha que D não tem ciclos de peso negativo, e considere o momento após o término da execução do laço **para** que começa na linha 6.

Seja $v \in V(D)$ um vértice para o qual não existe sv-caminho algum. Não é difícil verificar que o algoritmo nunca vai modificar o valor de $v.\texttt{distancia}$. Assim, para esse tipo de vértice vale que $v.\texttt{distancia} = \infty = \text{dist}_D^w(s, v)$.

Agora seja $v \in V(D)$ tal que existe algum sv-caminho. Ademais, como não existem ciclos de peso negativo, sabemos que existe algum sv-caminho mínimo. Assim, seja $P = (s, v_1, v_2, \ldots, v_k, v)$ um sv-caminho mínimo. Note que como P é mínimo, então P tem no máximo $v(D) - 1$ arcos.

Para facilitar a discussão a seguir, denote $v_0 = s$ e $v_{k+1} = v$. Veja que em cada uma das $v(D) - 1$ iterações do laço **para** na linha 6 todos os arcos do digrafo são relaxados. Assim, em particular, o arco $v_{i-1}v_i$ é relaxada na iteração i, para $1 \leq i \leq k + 1$. Com isso, os arcos v_0v_1, v_1v_2, $\ldots$, v_kv_{k+1} são relaxados nessa ordem pelo algoritmo. Pelo Lema 27.3, temos $v_{k+1}.\texttt{distancia} = \text{dist}_D^w(s, v_{k+1})$.

Uma vez que $y.\texttt{distancia} = \text{dist}_D^w(s, y)$ para qualquer $y \in V(D)$ e $\text{dist}_D^w(s, y) = \text{dist}_D^w(s, x) + w(xy)$ para qualquer $xy \in E(D)$, a linha 13 nunca é executada. Assim, a prova do lema está concluída. □

Usando o Lema 27.5, podemos facilmente notar que o algoritmo identifica um ciclo de peso negativo.

Corolário 27.6

Seja D um digrafo, w uma função de pesos em seus arcos e seja $s \in V(D)$. Se D contém ciclos de peso negativo, então após a execução de Bellman-Ford(D, w, s) temos $D.\texttt{cicloNegativo} = 1$.

Demonstração. Seja D um digrafo que contém um ciclo C de peso negativo. Não importa quantas vezes relaxemos os arcos de C, sempre será possível relaxar novamente algum deles, melhorando a estimativa de distância de algum vértice do ciclo. Portanto, sempre existirá um arco uv tal que $v.\texttt{distancia} > u.\texttt{distancia} + w(uv)$, de modo que a linha 13 é executada, fazendo $D.\texttt{cicloNegativo} = 1$. □

Seja D um digrafo, w uma função de pesos nos arcos de D e $s \in V(D)$. Vamos analisar o tempo de execução de Bellman-Ford(D, w, s). Considere $n = v(D)$ e $m = e(D)$. Claramente, a inicialização, que inclui o laço **para** da linha 2, leva tempo $\Theta(n)$. A linha 6

leva tempo $\Theta(n)$ também. Já o tempo gasto no laço na linha 7, cujas linhas internas levam tempo constante, depende da implementação. Em matriz de adjacências, ele leva tempo $\sum_{v \in V(D)} \Theta(n) = \Theta(n^2)$ e em lista de adjacências leva tempo $\sum_{v \in V(D)} |N^+(v)| = \Theta(m)$. Como esse laço é executado $\Theta(n)$ vezes, ele pode levar ao todo tempo $\Theta(n^3)$ ou $\Theta(nm)$. Por fim, o laço **para** da linha 11 também é executado $\Theta(n^2)$ ou $\Theta(m)$ vezes, dependendo da implementação. Portanto, Bellman-Ford(D, w, s) leva tempo $\Theta(n) + \Theta(n) + \Theta(n^3) + \Theta(n^2) = \Theta(n^3)$ em matriz de adjacências ou $\Theta(n) + \Theta(n) + \Theta(nm) + \Theta(m) = \Theta(nm)$ em listas de adjacências.

27.2 Todos os pares

Considere agora o problema de encontrar caminhos mínimos entre todos os pares de vértices de um digrafo D com pesos nos arcos dados por uma função w (Problema 27.2). Uma primeira ideia que podemos ter para resolver esse problema é utilizar soluções para o problema de caminhos mínimos de única fonte. Seja $n = v(D)$ e $m = e(D)$. Podemos executar Dijkstra ou Bellman-Ford n vezes, passando cada um dos vértices $s \in V(D)$ como vértice inicial para esses algoritmos. Dessa forma, em cada uma das n execuções encontramos caminhos mínimos do vértice s a todos os outros vértices de D. Note que, como o tempo de execução de Dijkstra(G, w, s) é $O((m + n) \log n)$, então n execuções levam tempo total $O((mn + n^2) \log n)$. Para digrafos densos (i.e., digrafos com $\Theta(n^2)$ arcos), esse valor representa um tempo de execução da ordem de $O(n^3 \log n)$. O tempo de execução de Bellman-Ford(D, w, s) é $\Theta(nm)$, então n execuções dele levam $\Theta(n^2 m)$. Assim, no caso de grafos densos esse valor representa um tempo de execução da ordem de $\Theta(n^4)$. Lembre-se ainda que, se existirem arcos de peso negativo em D, então o algoritmo de Dijkstra nem funciona.

Nas seções a seguir veremos algoritmos específicos para o problema de caminhos mínimos entre todos os pares. Um deles é o algoritmo de Floyd-Warshall, que é executado em tempo $\Theta(n^3)$ independente do digrafo ser denso ou não, e funciona mesmo que o digrafo tenha arcos com pesos negativos. Outro algoritmo é o de Johnson, que também funciona em digrafos com arcos de pesos negativos e combina execuções de Bellman-Ford e Dijkstra, executando em tempo $\Theta(nm \log n)$.

27.2.1 Algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall é um famoso algoritmo de programação dinâmica (veja Capítulo 22) que encontra caminhos mínimos entre todos os pares de vértices de um digrafo em tempo $\Theta(n^3)$.

Nesta seção, considere um digrafo D um digrafo, uma função w de pesos nos arcos de D e que $V(D) = \{1, \ldots, n\}$, em que $n = v(D)$.

Sejam $i, j \in V(D)$ dois vértices quaisquer de D. Para construir um ij-caminho, uma abordagem possível é a seguinte. Escolha um outro vértice $k \in V(D)$ e decida usar k como vértice interno desse caminho ou não. Se decidirmos por usar k, então podemos construir recursivamente um ik-caminho e um kj-caminho, que juntos formarão um ij-caminho. Se decidirmos por não usar k, então podemos construir recursivamente um ij-caminho no digrafo sem o vértice k. Uma questão que fica é: usamos k ou não? Considerando nosso objetivo de criar caminhos mínimos, então podemos simplesmente testar ambas as opções e escolher a que gere um caminho de menor peso dentre as duas.

Note que nas duas opções acima, o vértice k não é interno de nenhum dos caminhos que são construídos recursivamente. Assim, efetivamente estamos, a cada chamada recursiva, desconsiderando algum vértice do digrafo, de forma que a recursão eventualmente para. Quando queremos construir um ij-caminho e não há outros vértices disponíveis, então a única possibilidade de construir um ij-caminho é se o arco ij existir (caso base). O Algoritmo 27.4 formaliza essa ideia. Ele recebe o digrafo D, a função de custo nos arcos, os vértices i e j, e um conjunto X de vértices disponíveis para serem internos no ij-caminho. Ele devolve apenas o custo do ij-caminho construído.

Algoritmo 27.4: CAMINHO(D, w, i, j, X)

```
1 se |X| == 0 então
2     se ij ∈ E(D) então
3         devolve w(ij)
4     devolve ∞
5 Seja k ∈ X
6 nao_usa_k = CAMINHO(D, w, i, j, X \ {k})
7 usa_k = CAMINHO(D, w, i, k, X \ {k}) + CAMINHO(D, w, k, j, X \ {k})
8 devolve min{nao_usa_k, usa_k}
```

Note que o Algoritmo 27.4 na verdade devolve o custo de um ij-caminho mínimo. Isso porque todos os vértices disponíveis para serem internos do ij-caminho estão, em alguma chamada recursiva, sendo testados para fazer parte do mesmo. Assim, efetivamente todas as possibilidades de ij-caminho estão sendo verificadas e apenas a de menor peso está sendo mantida. Veja isso exemplificado na Figura 27.5. Note ainda que ele leva tempo $T(n) = 3T(n-1) + \Theta(1)$, se implementado com matriz de adjacências ou $T(n) = 3T(n-1) + O(n)$ se implementado com listas de adjacências. Em ambos os casos, $T(n) = O(3^n)$.

A primeira pergunta que deve aparecer é: na linha 7, o ik-caminho obtido pela recursão não pode ter vértices em comum com o kj-caminho obtido pela outra recursão? Afinal, o conjunto de vértices disponíveis para serem usados internamente em ambos é o mesmo. O problema aqui é que se isso for verdade, então a junção deles não dará um ij-caminho, mas sim um ij-passeio. Veja que se isso for verdade, então esse ij-passeio é da forma $(i, \ldots, x, \ldots, k, \ldots, x, \ldots, j)$, de modo que a remoção do trecho entre os vértices x é na verdade um ij-passeio que não usa k. Se ele não usa k, foi considerado na chamada que tenta construir um ij-caminho que não usa k e o teste $\min\{nao_usa_k, usa_k\}$ o eliminou.

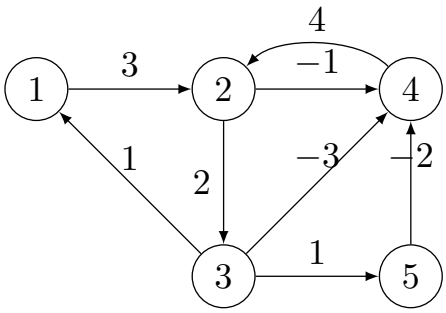
A última questão que fica é: mas qual k escolher? Podemos pensar em várias estratégias, como escolher um vértice k que seja vizinho de saída do i e para o qual $w(ik)$ é mínimo, ou então um vizinho de entrada do j para o qual $w(kj)$ seja mínimo. Mas veja que isso não importa muito, pois todos os vértices disponíveis serão igualmente testados eventualmente. Por comodidades que facilitam a implementação, escolhemos k como sendo o vértice com maior rótulo disponível. Assim, inicialmente $k = n$.

Finalmente, para resolver o problema de caminhos mínimos entre todos os pares, basta executar o Algoritmo 27.4 para todo possível valor de i e j. Assim temos n^2 execuções desse algoritmo, que leva tempo $O(3^n)$, de foram que o tempo total ficaria $O(n^2 3^n)$. Ou seja, esse algoritmo resolve nosso problema, mas em tempo exponencial!

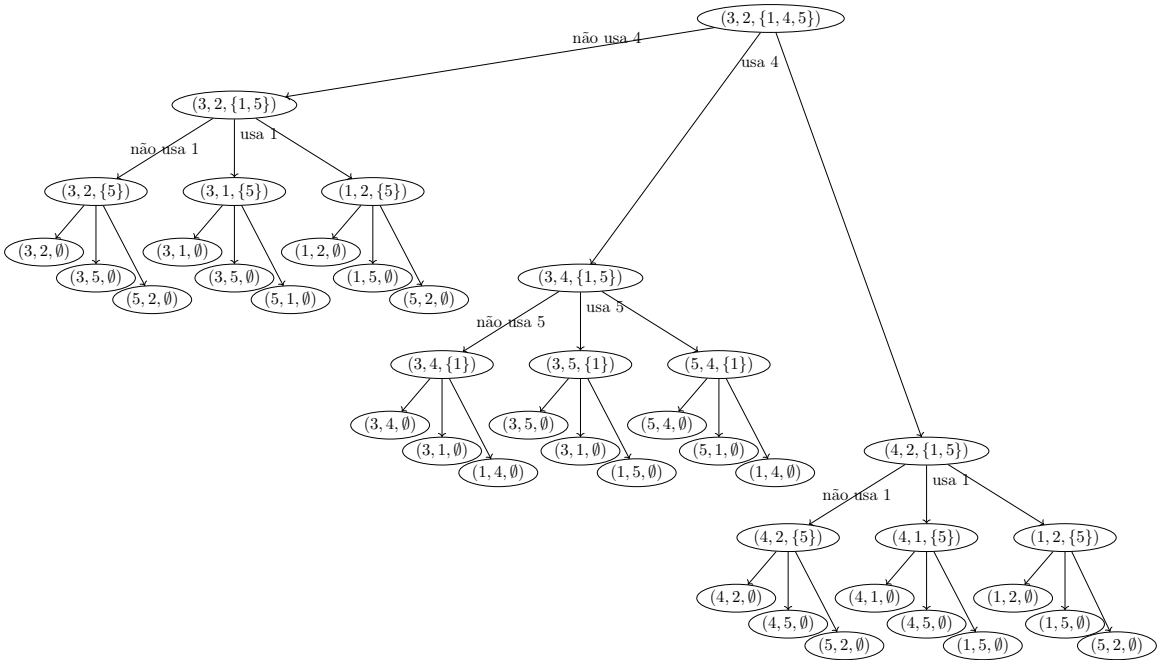
Agora observe que podemos descrever um subproblema por meio de uma tripla (i, j, k), que indica que desejamos encontrar um ij-caminho e temos k vértices disponíveis. Como D tem n vértices, o número total de triplas diferentes que temos é $n \times n \times (n + 1) = n^3 + n^2$, pois i e j são vértices do grafo e k pode variar entre 0 (nenhum vértice está disponível para construir o caminho) e n (todos os vértices estão disponíveis). Podemos observar então que a abordagem descrita acima realmente é muito ineficiente, pois recalcula vários desses subproblemas. O algoritmo de Floyd-Warshall utiliza uma estrutura de dados para guardar o valor da solução ótima para cada um desses subproblemas, assim não sendo necessário recalculá-los. Vamos formalizá-lo a seguir.

Definimos $P^k_{i,j}$ como o peso de um ij-caminho mínimo que contém vértices internos pertencentes ao conjunto $V_k = \{1, 2, \ldots, k\}$, para $0 \leq k \leq n$. Lembre-se que um ij-caminho cujos vértices internos pertençam ao conjunto $V_0 = \emptyset$ é, caso exista, o caminho (i, j), que contém somente o arco ij. Assim, para $k = 0$, que será nosso caso base, temos que

$$P^0_{i,j} = \begin{cases} 0 & \text{se } i = j \\ w(ij) & \text{se } ij \in E(D) \text{ e } i \neq j \\ \infty & \text{se } i \neq j \text{ e } ij \notin E(D) \end{cases} . \tag{27.2}$$



(a) Digrafo D de entrada.



(b) Árvore de recursão para Caminho(D, w, 3, 2, $\{1, 4, 5\}$).

Figura 27.5: Construindo 3 2-caminho no digrafo D. Note como todos os caminhos entre 3 e 2 estão sendo testadas em algum ramo da árvore. Por exemplo, o caminho $(3, 5, 1, 2)$ é testado quando seguimos "não usa 4" e "usa 1" a partir do vértice raiz. O caminho $(3, 4, 2)$ é testado quando seguimos "usa 4", "não usa 5" e "não usa 1" combinado com "usa 4", "não usa 1" e "não usa 5".

E, de modo geral, para $1 \leq k \leq n$, temos

$$P_{i,j}^{k} = \min\left\{P_{i,j}^{k-1}, P_{i,k}^{k-1} + P_{k,j}^{k-1}\right\} \quad . \tag{27.3}$$

Veja que essa análise vale porque caminhos mínimos contêm caminhos mínimos. Por isso, esse algoritmo é de fato ótimo.

Com essa nomenclatura, nosso objetivo é, portanto, calcular $P_{i,j}^{n}$ para todo par $i, j \in V(D)$. Como nossos subproblemas são descritos por uma tripla, será conveniente utilizar uma matriz W de dimensões $n \times n \times (n+1)$ para representar os valores de $P_{i,j}^{k}$, para $1 \leq i, j, k \leq n$. O objetivo do algoritmo de Floyd-Warshall é manter a relação $W[i][j][k] = P_{i,j}^{k}$.

Observe que cada vértice pode participar de vários caminhos. Assim, cada vértice j terá um atributo $j.$`predecessor` que será um vetor de tamanho n tal que $j.$`predecessor`$[i]$ contém o vértice predecessor de j em um ij-caminho mínimo. O Algoritmo 27.5 (que faz uso ainda do Algoritmo 27.6) e o Algoritmo 27.7 formalizam essas ideias em estilo top-down e bottom-up, respectivamente.

Algoritmo 27.5: FLOYD-WARSHALL-TOPDOWN(D, w)

```
1 para i = 1 até n, incrementando faça
2     para j = 1 até n, incrementando faça
3         para k = 0 até n, incrementando faça
4             W[i][j][k] = ∞
5 para i = 1 até n, incrementando faça
6     para j = 1 até n, incrementando faça
7         W[i][j][n] = FLOYD-WARSHALLREC-TOPDOWN(D, w, n, i, j)
8 devolve W
```

Agora note que devido à ordem em que os laços são executados, a terceira dimensão da matriz W é um tanto desperdiçada: para atualizar a k-ésima posição, utilizamos apenas a informação armazenada na $(k-1)$-ésima posição. Assim, é possível utilizar apenas uma matriz bidimensional para obter o mesmo resultado. O Algoritmo 27.8 formaliza essa ideia.

Por causa dos três laços aninhados, independente da economia de espaço ou não, claramente o tempo de execução de FLOYD-WARSHALL-BOTTOMUP(D, w) é $\Theta(n^3)$.

Por fim, perceba que em nenhum momento o algoritmo de Floyd-Warshall verifica se o digrafo de entrada possui um ciclo de peso negativo. De fato, em digrafos com ciclos de peso negativo, o algoritmo encerra sua execução normalmente. Acontece, porém, que ele não calcula os pesos dos caminhos mínimos *corretamente*. Felizmente, podemos verificar isso com

Algoritmo 27.6: FLOYD-WARSHALLREC-TOPDOWN(D, w, k, i, j)

```
1  se W[i][j][k] == ∞ então
2      se k == 0 então
3          se i == j então
4              W[i][j][0] = 0
5              j.predecessor[i] = i
6          senão se ij ∈ E(D) então
7              W[i][j][0] = w(ij)
8              j.predecessor[i] = i
9          senão
10             W[i][j][0] = ∞
11             j.predecessor[i] = null
12     senão
13         nao_usa_k = FLOYD-WARSHALLREC-TOPDOWN(D, w, k − 1, i, j)
14         usa_k = FLOYD-WARSHALLREC-TOPDOWN(D, w, k − 1, i, k) +
             FLOYD-WARSHALLREC-TOPDOWN(D, w, k − 1, k, j)
15         se nao_usa_k < usa_k então
16             W[i][j][k] = nao_usa_k
17         senão
18             W[i][j][k] = usa_k
19             j.predecessor[i] = j.predecessor[k]
20 devolve W[i][j][k]
```

Algoritmo 27.7: FLOYD-WARSHALL-BOTTOMUP(D, w)

```
Seja W[1..n][1..n][0..n] uma matriz
para i = 1 até n, incrementando faça
    para j = 1 até n, incrementando faça
        se i == j então
            W[i][j][0] = 0
            j.predecessor[i] = i
        senão se ij ∈ E(D) então
            W[i][j][0] = w(ij)
            j.predecessor[i] = i
        senão
            W[i][j][0] = ∞
            j.predecessor[i] = null
para k = 1 até n, incrementando faça
    para i = 1 até n, incrementando faça
        para j = 1 até n, incrementando faça
            nao_usa_k = W[i][j][k − 1]
            usa_k = W[i][k][k − 1] + W[k][j][k − 1]
            se nao_usa_k < usa_k então
                W[i][j][k] = nao_usa_k
            senão
                W[i][j][k] = usa_k
                j.predecessor[i] = j.predecessor[k]
devolve W
```

Algoritmo 27.8: FLOYD-WARSHALL-MELHORADO(D, w)

```
Seja W[1..n][1..n] uma matriz
para i = 1 até n, incrementando faça
    para j = 1 até n, incrementando faça
        se i == j então
            W[i][j] = 0
            j.predecessor[i] = i
        senão se ij ∈ E(D) então
            W[i][j] = w(ij)
            j.predecessor[i] = i
        senão
            W[i][j] = ∞
            j.predecessor[i] = null
para k = 1 até n, incrementando faça
    para i = 1 até n, incrementando faça
        para j = 1 até n, incrementando faça
            se W[i][j] > W[i][k] + W[k][j] então
                W[i][j] = W[i][k] + W[k][j]
                j.predecessor[i] = j.predecessor[k]
devolve W
```

o resultado do próprio FLOYD-WARSHALL-BOTTOMUP. Caso a matriz W contenha alguma posição $W[i][i]$ com valor negativo, então existe um ciclo de peso total negativo no digrafo. Veja o Algoritmo 27.9.

Algoritmo 27.9: RESOLVECAMINHOSENTRETODOSPARES(D, w)

```
1 W = Floyd-Warshall-BottomUp(D, w)
2 para i = 1 até v(G), incrementando faça
3     se W[i][i] < 0 então
4         devolve null
5 devolve W
```

O Algoritmo 27.10 mostra como construir um caminho mínimo entre dois vértices i e j quaisquer após a execução correta de RESOLVECAMINHOSENTRETODOSPARES. A ideia é que se ℓ é o predecessor de j em um ij-caminho mínimo, então basta construir o $i\ell$-caminho mínimo e depois acrescentar o arco ℓj.

Algoritmo 27.10: CONSTROICAMINHO(D, i, j)

```
1 Seja L uma lista vazia
2 atual = j
3 enquanto atual ≠ i faça
4     InsereNoInicioLista(L, atual)
5     atual = atual.predecessor[i]
6 InsereNoInicioLista(L, i)
7 devolve L
```

27.2.2 Algoritmo de Johnson

O algoritmo de Johnson também é um algoritmo para tratar do problema de caminhos mínimos entre todos os pares de vértices (Problema 27.2). Assim como Floyd-Warshall, Johnson também permite que os arcos tenham pesos negativos.

Uma observação importante quando se tem arcos com pesos negativos é a de que somar um valor constante a todos os arcos para deixá-los com pesos positivos e então usar o algoritmo de Dijkstra, por exemplo, não resolve o problema. Caminhos mínimos no digrafo original deixam de ser mínimos com essa modificação e vice-versa. O algoritmo de Johnson, no entanto, de fato reescreve os pesos dos arcos para deixá-los positivos e então utilizar Dijkstra, porém faz isso de uma maneira que garante a correspondência entre os digrafos envolvidos.

Formalmente, seja D um digrafo com função de pesos $w\colon E(D) \to \mathbb{R}$ nos arcos sobre o qual queremos resolver o problema de caminhos mínimos entre todos os pares de vértices. Crie um novo digrafo $\ddot{D}$ com $V(\ddot{D}) = V(D) \cup \{s\}$ e $E(\ddot{D}) = E(D) \cup \{sv\colon v \in V(D)\}$, isto é, existe um novo vértice s que se conecta a todos os outros vértices. Estenda a função w para uma função $\ddot{w}$ tal que $\ddot{w}(sv) = 0$ para todo $v \in V(D)$ e $\ddot{w}(uv) = w(uv)$ para todo $uv \in E(D)$. Note que esse novo vértice s não interfere nos caminhos e ciclos já existentes em D e também não cria novos caminhos e ciclos entre os vértices de D, uma vez que arcos apenas saem de s.

Crie agora um novo digrafo $\bar{D}$ com $V(\bar{D}) = V(D)$ e $E(\bar{D}) = E(D)$. Crie uma função $\bar{w}$ de pesos nos arcos de $\bar{D}$ definida como

$$\bar{w}(xy) = \left(\mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,x) + \ddot{w}(xy)\right) - \mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,y)$$

para todo $xy \in E(\bar{D})$. Observe que $\bar{w}(xy) \geq 0$ para todo $xy \in E(\bar{D})$, pois $\mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,x) + \ddot{w}(xy) \geq \mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,y)$, uma vez que um sy-caminho pode ser construído a partir de qualquer sx-caminho seguido do arco xy. Com isso, o digrafo $\bar{D}$ com função de peso $\bar{w}$ é uma entrada válida para o algoritmo de Dijkstra.

O próximo passo, portanto, é aplicar Dijkstra($\bar{D}$, $\bar{w}$, u) para cada $u \in V(\bar{D})$, calculando os caminhos mínimos de u a v, para todo $v \in V(\bar{D})$.

Resta mostrar então que um uv-caminho mínimo calculado por Dijkstra($\bar{D}$, $\bar{w}$) equivale a um uv-caminho mínimo em D com função w, para todo $u, v \in V(D)$. Seja $P = (u, x_1, \ldots, x_k, v)$ um uv-caminho qualquer em $\bar{D}$ com função $\bar{w}$. A expressão que calcula o peso de P com a função $\bar{w}$ pode ser convertida, pelas definições de $\bar{w}$ e $\ddot{w}$, em uma expressão que dependa apenas de w da seguinte forma:

$$\begin{aligned}
\bar{w}(P) &= \bar{w}(ux_1) + \bar{w}(x_1x_2) + \cdots + \bar{w}(x_{k-1}x_k) + \bar{w}(x_kv) \\
&= \left(\mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,u) + \ddot{w}(ux_1) - \mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,x_1)\right) + \\
&\qquad \left(\mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,x_1) + \ddot{w}(x_1x_2) - \mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,x_2)\right) + \cdots + \\
&\qquad \left(\mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,x_{k-1}) + \ddot{w}(x_{k-1}x_k) - \mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,x_k)\right) + \\
&\qquad \left(\mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,v) + \ddot{w}(x_kv) - \mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,v)\right) \\
&= \ddot{w}(ux_1) + \ddot{w}(x_1x_2) + \cdots + \ddot{w}(x_{k-1}x_k) + \ddot{w}(x_kv) + \left(\mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,u) - \mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,v)\right) \\
&= w(ux_1) + w(x_1x_2) + \cdots + w(x_{k-1}x_k) + w(x_kv) + \left(\mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,u) - \mathrm{dist}_{\ddot{D}}^{\ddot{w}}(s,v)\right) \quad .
\end{aligned}$$

E veja essa expressão independe dos vértices internos do caminho. Isso significa que o peso de qualquer uv-caminho em $\bar{D}$ com função $\bar{w}$ tem valor igual ao peso deo mesmo uv-caminho em

D c com função w somado ao mesmo valor fixo $\mathrm{dist}^{\ddot{w}}_{\ddot{D}}(s,u) - \mathrm{dist}^{\ddot{w}}_{\ddot{D}}(s,v)$. Essa correspondência garante que qualquer caminho mínimo em D com função w será um caminho mínimo em $\bar{D}$ com função $\bar{w}$.

Uma questão que fica é como calcular $\bar{w}$, que precisa de $\mathrm{dist}_{\ddot{D}}(s,v)$ para todo $v \in V(D)$? Essa é justamente a definição do problema de caminhos mínimos de única fonte. Note que $\ddot{D}$ com função de peso $\ddot{w}$ pode conter arcos de pesos negativos, de forma que calcular distâncias em $\ddot{D}$ é algo que não pode ser feito por Dijkstra, por exemplo. Porém, aqui podemos utilizar o algoritmo de Bellman-Ford. Inclusive, se em $\ddot{D}$ houver um ciclo com peso negativo, Bellman-Ford irá reconhecer esse fato, que também implica que D possui ciclo com peso negativo.

O Algoritmo 27.11 formaliza o algoritmo de Johnson, que, caso não exista ciclo de peso negativo em D, calcula o peso de um caminho mínimo de u a v, para todos os pares $u, v \in V(D)$. Cada vértice u possui um campo $u.$`distancias`, que é um vetor com $v(D)$ posições e deverá armazenar em $u.$`distancias`$[v]$ o peso de um uv-caminho mínimo. Para facilitar o entendimento do algoritmo, cada vértice u possui ainda um campo $u.$`distanciaOrig` numérico, que armazenará a distância de s a u calculada por Bellman-Ford sobre $\ddot{D}$ com função $\ddot{w}$, e um campo $u.$`distancia` numérico, que armazenará a su-distância calculada pelo algoritmo de Dijkstra sobre $\bar{D}$, $\bar{w}$ e s.

Algoritmo 27.11: JOHNSON(D, w)

Seja $\ddot{D}$ um digrafo com $V(\ddot{D}) = V(D) \cup \{s\}$ e $E(\ddot{D}) = E(D) \cup \{sv \colon v \in V(D)\}$
Seja $\ddot{w}$ função nos arcos de $\ddot{D}$ com $\ddot{w}(sv) = 0$, para todo $v \in V(D)$, e
  $\ddot{w}(uv) = w(uv)$, para todo $uv \in E(D)$
BELLMAN-FORD($\ddot{D}$, $\ddot{w}$, s)
se $\ddot{D}.$`cicloNegativo` $== 1$ **então**
  devolve *"O digrafo D contém ciclo de peso negativo"*
Seja $\bar{D}$ um digrafo com $V(\bar{D}) = V(D)$ e $E(\bar{D}) = E(D)$
Seja $\bar{w}$ função nos arcos de $\bar{D}$ com
  $\bar{w}(uv) = u.$`distanciaOrig` $+ \ddot{w}(uv) - v.$`distanciaOrig`, para todo $uv \in E(D)$
para *todo vértice* $u \in V(D)$ **faça**
  DIJKSTRA($\bar{D}$, $\bar{w}$, u) `/* Assim,` $v.$`distancia` $= \mathrm{dist}^{\bar{w}}_{\bar{D}}(u,v)\ \forall v \in V(D)$ `*/`
  para *todo vértice* $v \in V(D)$ **faça**
    $u.$`distancias`$[v] = v.$`distancia` $+ (v.$`distanciaOrig` $- u.$`distanciaOrig`$)$

Note que o tempo de execução de JOHNSON(D, w) é o mesmo de n execuções de DIJKSTRA somada a uma execução de BELLMAN-FORD e a duas construções de digrafos, que é dominado pelas execuções de DIJKSTRA, sendo $O((mn + n^2)\log n)$.

Parte

VII

Teoria da computação

"Os problemas computacionais vêm em diferentes variedades: alguns são fáceis e outros, difíceis. Por exemplo, o problema da ordenação é fácil. (...) Digamos que você tenha que encontrar um escalonamento de aulas para a universidade inteira que satisfaça algumas restrições razoáveis (...). Se você tem somente mil aulas, encontrar o melhor escalonamento pode requerer séculos (...).
O que faz alguns problemas computacionalmente difíceis e outros fáceis?"

Michael Sipser – Introdução à Teoria da Computação, 2006.

Nesta parte

A maioria dos problemas que vimos até aqui neste livro são ditos *tratáveis*. São problemas para os quais existem algoritmos eficientes para resolvê-los.

Definição 27.7

Um algoritmo é dito *eficiente* se seu tempo de execução no pior caso é $O(n^k)$, onde n é o tamanho da entrada do algoritmo e k é um inteiro positivo que não depende de n.

Busca (2.1), Ordenação (14.1), Mochila fracionária (21.3), Corte de barras (22.2), Árvore geradora mínima (25.1), Caminhos mínimos em grafos (27.1 e 27.2) são alguns exemplos de problemas tratáveis. No entanto, muitos problemas, até onde se sabe, não possuem algoritmos eficientes que os resolvam, como é o caso do problema da Mochila inteira (22.3), por exemplo. Estes são ditos *intratáveis*.

Na verdade, muitos problemas interessantes e com fortes motivações e aplicações práticas são intratáveis, como por exemplo escalonar um conjunto de tarefas a processadores, interligar de forma barata computadores específicos em uma rede com diversos outros computadores que podem ser usados como intermediários, cortar placas de vidros em pedaços de tamanhos específicos desperdiçando pouco material, ou decompor um número em fatores primos. Para esses problemas, não se tem muita esperança de encontrar algoritmos eficientes que os resolvam, porém felizmente existem vários algoritmos eficientes que encontram boas soluções.

Nos capítulos a seguir veremos mais sobre a teoria envolvendo esses tipos de problemas e formas de lidar com os mesmos.

Capítulo 28

Redução entre problemas

Redução entre problemas é uma técnica muito importante de projeto de algoritmos. A ideia intuitiva é utilizar um algoritmo que já existe para um certo problema, ou qualquer algoritmo que venha a ser criado para ele, para resolver outro problema[1].

Comecemos com um exemplo. Considere o seguinte problema, da Seleção.

Problema 28.1: *Seleção*

Dados um vetor $V[1..n]$ de tamanho n e um inteiro $k \in \{1, \ldots, n\}$, obter o k-ésimo menor elemento que está armazenado em V.

Por exemplo, se $V = (3, 7, 12, 6, 8, 234, 9, 78, 45)$ e $k = 5$, então a resposta é 9. Se $k = 8$, então a resposta é 78. Uma forma bem simples de resolver esse problema é ordenando V. Com isso, teremos o vetor$(3, 6, 7, 8, 9, 12, 45, 78, 234)$ e fica bem fácil ver quem é o quinto ou o oitavo menor elemento, pois basta acessar as posições 5 ou 8 diretamente. Nós acabamos de reduzir o problema da Seleção para o problema da Ordenação! Com isso, temos agora um algoritmo para o problema da Seleção, mostrado no Algoritmo 28.1.

Algoritmo 28.1: ALG_SELECAO(V, n, k)

```
1 ALG_ORDENACAO(V, n)
2 devolve V[k]
```

[1]Não confundir com a palavra redução usada em algoritmos recursivos. Lá, estamos tentando diminuir o tamanho de uma entrada para se manter no mesmo problema e poder resolvê-lo recursivamente. Aqui, estamos falando sobre conversão entre dois problemas diferentes.

Vamos observar o que está acontecendo nessa redução com detalhes. Nós recebemos uma instância $\langle V, n, k\rangle$ para o problema da Seleção. Então nós a transformamos em uma instância válida para o problema da Ordenação. No caso desses problemas, não há modificação necessária e a instância nova criada é $\langle V, n\rangle$. Tendo uma instância válida para a Ordenação, qualquer algoritmo de ordenação pode ser usado no lugar de ALG_ORDENACAO, como por exemplo MergeSort ou InsertionSort. Dado o resultado do problema de Ordenação, que é o vetor modificado, nós o transformamos em um resultado para o problema da Seleção, que é apenas um elemento do vetor.

Observe que ALG_SELECAO leva tempo $\Theta(n \log n)$ se usarmos MergeSort no lugar de ALG_ORDENACAO, não importando qual o valor de k. Uma vez que temos um algoritmo para um problema e sabemos que ele funciona e qual seu tempo de execução, a próxima pergunta costuma ser "será que dá para fazer melhor?". De fato, existe um algoritmo que resolve o problema da Seleção (sem uso de redução) em tempo $\Theta(n)$, qualquer que seja o valor de k também. Isso nos mostra que a redução pode nos dar uma forma para resolver um problema, mas não podemos nos esquecer de que outras formas ainda podem existir.

Vejamos outro exemplo, que envolve os dois problemas a seguir.

Problema 28.2: *Quadrado*

Dado um inteiro x, obter o valor x^2.

Problema 28.3: *Multiplicação de inteiros*

Dados dois inteiros x e y contendo n dígitos cada, obter o produto $x \times y$.

Não é difícil notar que o problema do Quadrado se reduz ao problema da Multiplicação. Dada uma instância $\langle a\rangle$ para o problema do Quadrado, podemos transformá-la na instância $\langle a, a\rangle$ para o problema da Multiplicação e, agora, qualquer algoritmo de multiplicação pode ser utilizado. Como $a \times a = a^2$, temos diretamente a solução para o problema original.

Talvez mais interessante seja a redução na direção inversa: do problema da Multiplicação para o problema do Quadrado. Para fazer essa redução, queremos utilizar um algoritmo que resolva quadrados para resolver a multiplicação. Especificamente, dados dois inteiros x e y quaisquer, qual deve ser o valor a para que a^2 seja útil no cálculo de $x \times y$? Veja que $(x + y)^2 = x^2 + 2xy + y^2$, o que significa que $xy = ((x + y)^2 - x^2 - y^2)/2$. O Algoritmo 28.2 mostra essa redução.

Considere agora os problemas de caminhos mínimos, reescritos a seguir, e que já haviam

Algoritmo 28.2: ALG_MULTIPLICACAO(x, y)

```
1 a ← ALG_QUADRADO(x + y)
2 b ← ALG_QUADRADO(x)
3 c ← ALG_QUADRADO(y)
4 devolve (a − b − c)/2
```

sido definidos no Capítulo 27.

Problema 28.4: ***Caminhos mínimos de única fonte***

Dados um digrafo D, uma função w de peso nos arcos e um vértice $s \in V(D)$, calcular $\text{dist}_D^w(s, v)$ para todo $v \in V(D)$.

Problema 28.5: ***Caminhos mínimos entre todos os pares***

Dados um digrafo D e uma função w de peso nos arcos, calcular $\text{dist}_D^w(u, v)$ para todo par $u, v \in V(D)$.

Observe que é possível reduzir do problema de caminhos mínimos entre todos os pares para o problema de caminhos mínimos de única fonte: tendo um algoritmo que resolve o segundo, conseguimos criar um algoritmo para o primeiro, pois basta calcular caminhos mínimos de s para os outros vértices, para cada vértice s do digrafo.

Esses exemplos talvez possam nos levar a (erroneamente) achar que só é possível reduzir problemas que tenham entrada parecida (números com números, vetores com vetores ou grafos com grafos). O exemplo a seguir nos mostrará que é possível fazer redução entre problemas que aparentemente não teriam relação. O problema do escalonamento de tarefas compatíveis já foi visto na Seção 21.1 e encontra-se replicado abaixo.

Problema 28.6: ***Escalonamento de tarefas compatíveis***

Dado conjunto $T = \{t_1, \ldots, t_n\}$ com n tarefas onde cada $t_i \in T$ tem um tempo inicial s_i e um tempo final f_i, encontrar o maior subconjunto de tarefas mutuamente compatíveis.

Vamos mostrar que ele pode ser reduzido ao problema do conjunto independente máximo, definido a seguir.

Problema 28.7: ***Conjunto independente máximo***

Dado um grafo G, encontrar um conjunto $S \subseteq V(G)$ tal que para todo par $u, v \in S$ vale que $uv \notin E(G)$ e S tem tamanho máximo.

Queremos então reduzir do problema de escalonamento de tarefas compatíveis para o problema do conjunto independente máximo. Isso significa que queremos usar um algoritmo que resolve o segundo para criar um algoritmo que resolve o primeiro. Especificamente, dada uma entrada $\langle T, n, s, f \rangle$ para o problema das tarefas, precisamos criar algum grafo G específico e relacionado com tal entrada de tal forma que ao encontrar um conjunto independente máximo em G poderemos o mais diretamente possível encontrar um conjunto de tarefas mutuamente compatíveis de tamanho máximo em T.

Vamos criar um grafo G em que $V(G) = T$. Adicionaremos uma aresta entre dois vértices u e v de G se as tarefas u e v forem incompatíveis. Com isso, seja $S \subseteq V(G)$ um conjunto independente em G, isto é, não há arestas entre nenhum par de vértices de S. Mas então, por construção do grafo G, S é um conjunto de tarefas mutuamente compatíveis. Como S foi escolhido de forma arbitrária, note que qualquer conjunto independente em G representa um conjunto de tarefas mutuamente compatíveis. Em particular, isso vale para as soluções ótimas. Essa redução está descrita no Algoritmo 28.3.

Algoritmo 28.3: ALG_TAREFAS(T, n, s, f)

```
1 Crie um grafo G
2 Faça V(G) = T
3 Faça E(G) = {t_i t_j : s_i < f_j e s_j < f_i}
4 S ← ALG_CONJINDEPENDENTE(G)
5 devolve S
```

Em resumo, reduzir de um problema A para um problema B significa que dado um algoritmo ALG_B qualquer para o problema B, que recebe qualquer instância para B e devolve uma solução para B, o que pode ser representado graficamente da seguinte forma

$$I_B \longrightarrow \boxed{ALG_B} \longrightarrow S_B\,,$$

nós queremos transformar qualquer entrada I_A do problema A, por meio de alguma transformação f, para uma entrada específica para B, usar o algoritmo para B e depois transformar

a solução do B em uma solução S_A válida para o A, por alguma outra transformação g:

$$I_A \xrightarrow{f} I_B \longrightarrow \boxed{ALG_B} \longrightarrow S_B \xrightarrow{g} S_A\,.$$

Isso nos dá um algoritmo para o problema A:

$$I_A \longrightarrow \boxed{\xrightarrow{f} I_B \longrightarrow \boxed{ALG_B} \longrightarrow S_B \xrightarrow{g}} \longrightarrow S_A\,.$$

Perceba que isso não impede que outros algoritmos existam para o problema A!

28.1 Redução entre problemas de otimização e decisão

Definição 28.8

Um *problema de decisão* é um problema cuja solução é uma resposta `sim` ou `não`.

Por exemplo, o problema "dado um número, ele é par?" é um problema de decisão. Outro problema de decisão é "dados um grafo G e dois vértices $u, v \in V(G)$, existe uv-caminho?".

Problema 28.9: *Caminho*

Dados um grafo G, uma função w de pesos nas arestas, dois vértices $u, v \in V(G)$ e um valor k, existe uv-caminho de peso no máximo k?

Como toda instância válida para um problema de decisão só pode ter resposta `sim` ou `não`, é possível dividir o conjunto de instâncias de um problema em dois: aquele que contém instâncias `sim` e aquele que contém instâncias `não`. Por isso, é comum nos referirmos a essas instâncias como *instâncias* ***`sim`*** e *instâncias* ***`não`***.

Note que para convencer alguém de que uma instância é `sim`, basta mostrar algum *certificado*. Por exemplo, considerando o grafo G com pesos da Figura 28.1, temos que $\langle G, w, 3, 10, 20\rangle$ é uma instância `sim` para o problema do caminho. Para se convencer disso, observe o caminho $(3, 1, 2, 11, 10)$: é um caminho do vértice 3 ao vértice 10 de peso 17, o que é menor ou igual a 20. Dizemos que isso é um *certificado* de que a instância é `sim`. Veja que existem outros 3 10-caminhos de peso menor ou igual a 20 existem naquele grafo e também caminhos de peso maior do que 20, mas basta que haja um com peso menor ou igual a 20 para que a instância $\langle G, w, 3, 10, 20\rangle$ seja `sim`.

Já a instância $\langle G, w, 3, 10, 9\rangle$ é uma instância `não`, pois não há caminho entre 3 e 10 de peso menor ou igual à 9. E note como isso é mais difícil de ser verificado: precisamos observar

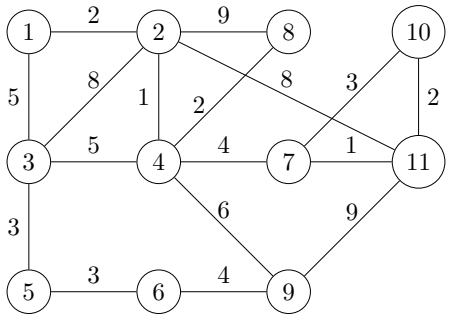


Figura 28.1: Parte de uma instância válida para o problema Caminho: um grafo com pesos nas arestas.

que qualquer caminho entre 3 e 10 tem peso maior do que isso para nos convencermos disso.

Por essa diferença, os certificados para instâncias `sim` são chamados também de *certificados positivos* e os certificados para instâncias `não` são chamados de *certificados negativos*.

Os problemas anteriores têm objetivos diferentes do problema a seguir.

Problema 28.10: *Caminho mínimo*

Dados um grafo G com pesos nas arestas e dois vértices $u, v \in V(G)$, encontrar um uv-caminho de peso mínimo.

O problema do caminho mínimo descrito acima é um *problema de otimização*.

Definição 28.11

Um *problema de otimização* é um problema cuja solução deve ser a de melhor valor dentre todas as soluções possíveis.

Observe que é mais difícil convencer alguém de que o grafo G com pesos da Figura 28.1 tem como um 3 10-caminho mínimo o caminho $(3, 4, 7, 10)$. Precisaríamos listar todos os outros 3 10-caminhos para ter essa garantia, e conforme o tamanho do grafo aumenta isso se complica.

Mesmo com essas diferenças, existe uma relação muito importante entre o Problema 28.9 e o Problema 28.10: se resolvermos um deles, então resolvemos o outro, conforme a discussão a seguir. Seja G um grafo com função w de pesos nas arestas e sejam $u, v \in V(G)$ dois vértices quaisquer. Suponha primeiro que sabemos resolver o problema do caminho mínimo e que z é o custo do menor uv-caminho. Para um k qualquer, se $z \leq k$, então a resposta para o problema de decisão certamente é `sim`, isto é, existe um uv-caminho com custo menor

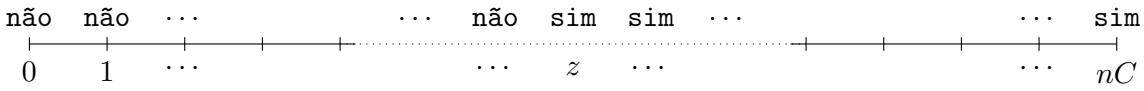


Figura 28.2: Exemplificação da discussão sobre a relação entre problemas de decisão e problemas de otimização.

que k (tome, por exemplo, o próprio uv-caminho mínimo). Por outro lado, se $z > k$, então a resposta para o problema de decisão certamente é `não`, pois se o menor uv-caminho tem custo maior do que k e qualquer outro uv-caminho tem custo maior que z, então não é possível existir um uv-caminho com custo no máximo k.

Agora suponha que sabemos resolver o problema do caminho (sabemos dizer `sim` ou `não` para qualquer valor de k). Seja C o custo da aresta de maior custo do grafo e seja $n = |V(G)|$. Note que qualquer uv-caminho terá custo no máximo nC pois ele pode no máximo usar $n-1$ arestas. Assim, podemos testar todos os valores de $k \in \{0, 1, 2, \ldots, nC\}$ e, para o menor valor cuja solução for `sim`, temos a resposta para o caminho mínimo. Veja a Figura 28.2 para um esquema dessa discussão.

Em outras palavras, um é redutível ao outro! Por esse motivo, a partir de agora vamos apenas considerar problemas de decisão, que são bem mais simples.

28.2 Formalizando a redução

No que segue, se A é o nome de um problema de decisão, chamaremos de I_A uma instância (entrada) para A. Lembre-se que A é uma instância `sim` ou uma instância `não` apenas.

Definição 28.12: *Redução polinomial*

Sejam A e B problemas de decisão. O problema A é *redutível polinomialmente* para B se existe algoritmo eficiente f tal que $f(I_A) = I_B$ e

$$I_A \text{ é } \texttt{sim} \text{ se e somente se } I_B \text{ é } \texttt{sim}.$$

Ou seja, uma redução é uma função f que *mapeia* instâncias `sim` de A para instâncias `sim` de B e, por consequência, instâncias `não` de A para instâncias `não` de B. Fazer reduções com essa garantia nos permite usar um algoritmo para o problema B sobre $f(I_A)$ de tal forma que se tal algoritmo responder `sim`, teremos certeza de que I_A é `sim`, e se ele responder `não`, teremos certeza de que I_A é `não`.

Uma observação importante antes de continuarmos é que se conseguimos reduzir de um

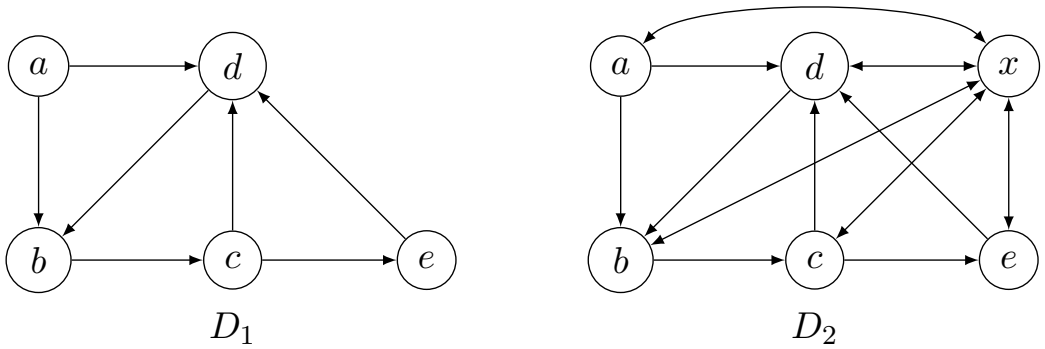


Figura 28.3: Digrafo D_1 entrada para o problema do Caminho hamiltoniano e digrafo D_2 criado a partir de D_1 via redução para o problema do Ciclo hamiltoniano.

problema A para outro problema B não é necessariamente verdade que será garantido reduzir de B para A (ao menos não em tempo polinomial). A seguir veremos três exemplos para que esse formalismo fique claro.

Exemplo 1. Um *ciclo hamiltoniano* é um ciclo que passa por todos os vértices de um (di)grafo. De forma equivalente, um *caminho hamiltoniano* é um caminho que passa por todos os vértices. Considere os dois problemas a seguir, clássicos em computação.

Problema 28.13: *Caminho hamiltoniano*

Dado um digrafo D, existe caminho hamiltoniano em D?

Problema 28.14: *Ciclo hamiltoniano*

Dado um digrafo D, existe ciclo hamiltoniano em D?

Para um exemplo, veja a Figura 28.3. Observe que o digrafo D_1 dessa figura é uma instância `sim` para o problema do caminho hamiltoniano, o que pode ser certificado pelo caminho (a, d, b, c, e). Veja que ele é uma instância `não` para o problema do ciclo hamiltoniano, o que não é possível de ser certificado de forma tão simples como no caso anterior. Já digrafo D_2 dessa figura é uma instância `sim` para o ciclo hamiltoniano, e isso pode ser certificado pelo ciclo (x, a, d, b, c, e, x). Certamente, D_2 também é `sim` para caminho hamiltoniano, já que ao remover qualquer arco de um ciclo hamiltoniano temos um caminho hamiltoniano.

Vamos mostrar que o problema do caminho hamiltoniano é redutível em tempo polinomial ao problema do ciclo hamiltoniano. Assim, dado qualquer digrafo D, precisamos criar um

algoritmo que decida se D possui caminho hamiltoniano por meio de um algoritmo que decida se um digrafo possui ciclo hamiltoniano. Por definição, para fazer a redução, precisamos criar um digrafo $D' = f(D)$ tal que se D possui caminho hamiltoniano, então D' possui ciclo hamiltoniano, e se D' possui ciclo hamiltoniano, então D possui caminho hamiltoniano.

Veja que a entrada para ciclo hamiltoniano é um digrafo. Assim, será que se fizermos $f(D) = D$ a redução funcionaria? O digrafo D_1 da Figura 28.3 prova que não.

Nossa transformação será a seguinte. Dado um digrafo D qualquer (instância para caminho hamiltoniano), crie um digrafo D' tal que $V(D') = V(D) \cup \{x\}$ e $E(D') = E(D) \cup \{xu, ux \colon u \in V(D)\}$. Note que isso leva tempo polinomial no tamanho de D. Essa é a transformação mostrada na Figura 28.3. Resta provar que D é instância `sim` para caminho hamiltoniano se e somente se D' é instância `sim` para ciclo hamiltoniano.

Suponha primeiro que D é `sim` para caminho hamiltoniano. Isso significa que existe um caminho $C = (v_1, v_2, \ldots, v_n)$ que é hamiltoniano em D. Mas então $C' = (v_1, v_2, \ldots, v_n, x, v_1)$ é um ciclo hamiltoniano em D'. Logo, D' é `sim` para ciclo hamiltoniano.

Suponha agora que D' é `sim` para ciclo hamiltoniano. Isso significa que existe um ciclo $C = (v_1, \ldots, v_i, x, v_{i+1}, \ldots, v_n, v_1)$ que é hamiltoniano em D'. Mas então $C' = (v_{i+1}, \ldots, v_n, v_1, \ldots, v_i)$ é um caminho hamiltoniano em D. Logo, D é `sim` para caminho hamiltoniano.

Exemplo 2. Considere os dois problemas a seguir, que são as versões de decisão dos problemas do corte de barras e da mochila inteira.

Problema 28.15: *BARRA*

Dados inteiros positivos $p_1, \ldots, p_n$ que correspondem, respectivamente, ao preço de venda de barras de tamanho $1, \ldots, n$ e dado um inteiro positivo n, é possível cortar uma barra de tamanho n e vender os pedaços obtendo lucro pelo menos k?

Problema 28.16: *MOCHILA*

Dado um conjunto $I = \{1, 2, \ldots, n\}$ de n itens onde cada $i \in I$ tem um peso w_i e um valor v_i associados, dada uma mochila com capacidade de peso W e dado um valor V, é possível selecionar um subconjunto $S \subseteq I$ de itens tal que $\sum_{i \in S} w_i \leq W$ e $\sum_{i \in S} v_i \geq V$?

Reduziremos BARRA para MOCHILA da seguinte forma. Seja $\langle n, p, k \rangle$ uma entrada para BARRA. Construa $\langle I, m, v, w, W, V \rangle$ para MOCHILA da seguinte forma:

- crie $\lfloor n/i \rfloor$ itens de peso i cada e valor p_i cada, que são correspondentes a um pedaço

de tamanho i da barra, para $1 \leq i \leq n$;

- $m = \sum_{i=1}^{n} \lfloor n/i \rfloor$ e $I = \{1, \ldots, m\}$;
- $W = n$;
- $V = k$.

Note que isso é feito em tempo polinomial no tamanho de $\langle n, p, k \rangle$. Resta mostrar que $\langle n, p, k \rangle$ é `sim` para BARRA se e somente se $\langle I, m, v, w, W, V \rangle$ é `sim` para MOCHILA.

Suponha que $\langle n, p, k \rangle$ é `sim` para BARRA. Então existem pedaços $(c_1, c_2, \ldots, c_x)$ tais que $\sum_{i=1}^{x} c_i = n$ e $\sum_{i=1}^{x} p_{c_i} \geq k$. Para cada pedaço c_i, coloque um item j correspondente ao mesmo em um conjunto S. Note que $\sum_{j \in S} w_j = \sum_{i=1}^{x} c_i = n = W$ e $\sum_{j \in S} v_j = \sum_{i=1}^{x} p_{c_i} \geq k = V$. Então $\langle I, m, v, w, W, V \rangle$ é `sim` para MOCHILA.

Suponha agora que $\langle I, m, v, w, W, V \rangle$ é `sim` para MOCHILA. Então existe conjunto $S \subseteq I$ de itens tais que $\sum_{j \in S} w_j \leq W$ e $\sum_{j \in S} v_j \geq V$. Para cada item $j \in S$, corte a barra em um tamanho i correspondente ao mesmo. Sejam $(c_1, c_2, \ldots, c_{|S|})$ os pedaços cortados da barra. Note que $\sum_{i=1}^{|S|} c_i = \sum_{j \in S} w_j \leq W = n$ e $\sum_{i=1}^{|S|} p_{c_i} = \sum_{j \in S} v_j \geq V = k$. Então $\langle n, p, k \rangle$ é `sim` para BARRA.

Exemplo 3. Considere o problema de decisão da mochila inteira novamente e o novo problema dado a seguir.

Problema 28.17: *SUBSETSUM*

Dado um conjunto $A = \{s_1, \ldots, s_n\}$ que contém n inteiros e dado um inteiro B, existe $A' \subseteq A$ tal que $\sum_{s \in A'} s = B$?

Reduziremos SUBSETSUM para MOCHILA da seguinte forma. Seja $\langle A, n, B \rangle$ uma entrada para SUBSETSUM. Construa $\langle I, m, v, w, W, V \rangle$ para MOCHILA da seguinte forma:

- crie um item de peso $w = s_i$ e valor $v = s_i$ para cada $s_i \in A$;
- $m = n$ e $I = \{1, \ldots, n\}$;
- $W = B$;
- $V = B$.

Note que isso é feito em tempo polinomial no tamanho de $\langle A, n, B \rangle$. Resta mostrar que $\langle A, n, B \rangle$ é `sim` para SUBSETSUM se e somente se $\langle I, m, v, w, W, V \rangle$ é `sim` para MOCHILA.

Suponha que $\langle A, n, B\rangle$ é `sim` para SUBSETSUM. Então existe $A' \subseteq A$ com $\sum_{s\in A'} s = B$. Para cada $s \in A'$, coloque o item j correspondente em um conjunto S. Note que $\sum_{j\in S} w_j = \sum_{s\in A'} s = B = W$ e $\sum_{j\in S} v_j = \sum_{s\in A'} s = B = V$. Então $\langle I, m, v, w, W, V\rangle$ é `sim` para MOCHILA.

Agora suponha que $\langle I, m, v, w, W, V\rangle$ é `sim` para MOCHILA. Então existe $S \subseteq I$ com $\sum_{j\in S} w_j \leq W$ e $\sum_{j\in S} v_j \geq V$. Para cada $j \in S$, coloque o valor s_i correspondente em um conjunto A'. Note que $\sum_{s\in A'} s = \sum_{j\in S} w_j \leq W = B$ e $\sum_{s\in A'} s = \sum_{j\in S} v_j \geq V = B$. Mas então só pode ser que $\sum_{s\in A'} s = B$. Então $\langle A, n, B\rangle$ é `sim` para SUBSETSUM.

28.3 O que se ganha com redução?

A definição de redução nos permite obter dois tipos de resultados importantes. Suponha que conseguimos reduzir do problema A para o problema B em tempo polinomial:

- De forma bem direta, se temos um algoritmo eficiente que resolve B, então automaticamente temos um algoritmo eficiente que resolve A, a saber, o algoritmo obtido pela redução.

- Por contrapositiva, isso significa que se não houver algoritmo eficiente que resolva A, então não há algoritmo eficiente que resolva B. Em outras palavras, se não há algoritmo eficiente que resolve A, não pode ser o algoritmo obtido pela redução que será eficiente. Como esse algoritmo utiliza um algoritmo para B, então B não pode ter algoritmo eficiente que o resolva.

Em resumo, se A é redutível para B, então B é tão difícil quanto A. O conceito de redução portanto nos permite tanto aumentar o conjunto de problemas tratáveis quanto o dos intratáveis.

Capítulo 29

Classes de complexidade

Definição 29.1: *Classe P*

P é o conjunto de todos os problemas de decisão que podem ser resolvidos por um algoritmo eficiente.

Sabemos que o Problema 28.9, de determinar se existe um caminho entre dois vértices de um grafo, está na classe **P**, pois, por exemplo, os algoritmos de busca em largura e profundidade são algoritmos eficientes que o resolvem.

Outro exemplo de problema na classe **P** é o problema de decidir se um grafo possui uma árvore geradora de peso total menor do que um valor k. Isso porque se executarmos o algoritmo de Prim, por exemplo, e verificarmos se a árvore geradora mínima devolvida tem peso menor do que k, então sabemos que a resposta para o problema de decisão é `sim`, caso contrário a resposta é `não`. Ademais, a maioria dos problemas vistos anteriormente nesse livro, portanto, possuem uma versão de decisão correspondente que está em **P**. Dizemos "a maioria", pois nem todos os problemas do universo estão em **P** ainda: existem problemas para os quais não se conhece algoritmos eficientes que os resolvam.

Considere agora o problema a seguir.

Problema 29.2: *TSP*

Dado um digrafo D completo, $w\colon E(D) \to \mathbb{R}$ e um valor k, existe um ciclo hamiltoniano em D de custo no máximo k?

TSP é uma sigla para *Travelling Salesman Problem*, nome em inglês de um famoso pro-

blema em computação (o Problema do Caixeiro Viajante). Na versão de otimização, mais famosa, o objetivo é encontrar um ciclo hamiltoniano de custo mínimo no digrafo. Veja que não é difícil pensar em um algoritmo simples de força bruta para resolvê-los: podemos enumerar todas as $n!$ permutações dos n vértices do digrafo, calcular seu custo e manter a menor delas. Claramente, esse algoritmo simples não é nem um pouco eficiente.

Na verdade, o TSP é um problema que acredita-se não estar na classe **P**. Desde sua origem, em torno de 1800, ninguém conseguiu encontrar um algoritmo eficiente que o resolva.

Acontece que o fato de ninguém ter conseguido encontrar um algoritmo para um problema não implica diretamente que ele não está em **P**; apenas significa que ninguém ainda foi capaz de encontrá-lo. A área de projeto de algoritmos é muito rica e, apesar de já existirem várias técnicas como de algoritmos gulosos ou divisão e conquista, novas técnicas são criadas a todo momento. Será que em algum momento futuro alguém conseguirá descobrir uma técnica diferente que resolva o TSP, por exemplo?

A afirmação "acredito que o TSP não está em **P**" não é feita apenas porque ninguém conseguiu um algoritmo eficiente que resolva o TSP. Ela é feita porque ninguém conseguiu um algoritmo eficiente que resolve muitos outros problemas que são tão difíceis quanto o TSP! E para evidenciar essa intratabilidade do TSP, podendo dizer que ele é tão difícil quanto muitos outros problemas, precisamos da ideia de *completude*.

Se $\mathcal{X}$ é um conjunto qualquer de problemas, dizemos que um problema A é $\mathcal{X}$-*completo* se $A \in \mathcal{X}$ e se todos os outros problemas de $\mathcal{X}$ são redutíveis a A. Quer dizer, A é tão difícil quanto todos os outros problemas em $\mathcal{X}$. Se tivermos TSP pertencente a $\mathcal{X}$ e dissermos que todos os problemas de $\mathcal{X}$ são intratáveis, então nossa afirmação terá mais impacto quanto maior for $\mathcal{X}$.

Poderíamos talvez pensar em $\mathcal{X}$ contendo todos os problemas conhecidos? Infelizmente, alguns problemas conhecidos sequer podem possuir algoritmos que os resolvam, sendo portanto estritamente mais difíceis do que o TSP (mesmo ruim, o algoritmo de força bruta que descrevemos anteriormente o resolve). Esses problemas são chamados *indecidíveis*, sendo o mais famoso deles o *problema da parada*.

Problema 29.3: *Parada*

Dados um algoritmo e uma instância, a execução desse algoritmo sobre essa instância termina?

E se pensarmos em $\mathcal{X}$ contendo os problemas que podem ser resolvidos por força bruta? Note que todos os problemas desse tipo possuem algo em comum: uma solução para eles pode ser facilmente reconhecida. Por exemplo, dada uma sequência de vértices de um grafo,

é fácil decidir se ela é um ciclo que contém todos os vértices do mesmo em tempo polinomial. Ou então, dada uma sequência de vértices de um grafo, é fácil decidir se ela é um caminho que tem custo menor do que um dado k. Um algoritmo que toma esse tipo de decisão é chamado de *algoritmo verificador*.

Definição 29.4: *Algoritmo verificador*

Seja T um problema qualquer. Um algoritmo $\mathcal{A}$ é dito *verificador* se:

1. para toda instância I_T que é `sim`, existe um conjunto de dados D tal que $\mathcal{A}(I_T, D)$ devolve `sim`; e
2. para toda instância I_T que é `não`, qualquer conjunto de dados D faz $\mathcal{A}(I_T, D)$ devolver `não`.

O conjunto de dados D acima é um *certificado positivo*.

Encontramos o conjunto $\mathcal{X}$ desejado acima!

Definição 29.5: *Classe NP*

NP é o conjunto de todos os problemas de decisão para os quais existe um algoritmo verificador que aceita um certificado positivo.

Muitos problemas de decisão estão na classe **NP**. O problema do Ciclo hamiltoniano está, pois um certificado positivo para este problema é qualquer sequência de vértices: um algoritmo verificador pode simplesmente percorrer essa sequência e verificar se ela contém todos os vértices do grafo e se há aresta entre vértices adjacentes na sequência. O problema MOCHILA está, pois um certificado positivo para este problema é qualquer subconjunto de itens: um algoritmo verificador pode somar os pesos e valores desses itens para verificar se eles cabem na mochila e se têm valor pelo menos k. O TSP está em **NP**, pois um certificado para ele também é qualquer sequência de vértices: um algoritmo verificador pode verificar se essa sequência é um ciclo hamiltoniano e somar os custos das arestas presentes nele, para testar se tem custo no máximo k.

Vejamos a seguir outros problemas que pertencem à classe **NP**.

Problema 29.6: *CLIQUE*

Dados um grafo G e um inteiro positivo k, existe conjunto $S \subseteq V(G)$ de vértices tais que para todo par $u, v \in S$ existe uma aresta $uv \in E(G)$ (S é clique) e $|S| \geq k$?

O problema CLIQUE está em **NP** pois, dados G, k e um conjunto S qualquer de vértices, é fácil escrever um algoritmo eficiente que verifica se S é uma clique de tamanho pelo menos k: basta verificar se todos os pares de vértices em S têm aresta entre si e contar a quantidade de vértices de S.

Problema 29.7: *BIPARTIDO*

Dado um grafo G, é possível particionar $V(G)$ em dois conjuntos S e $V(G) \setminus S$ tal que para toda aresta $uv \in E(G)$, $u \in S$ e $v \in V(G) \setminus S$?

O problema BIPARTIDO está em **NP** pois, dados G e um conjunto S qualquer de vértices, é fácil escrever um algoritmo eficiente que verifica se todas as arestas do grafo possuem um extremo em S e outro não.

Note que todos os problemas em **P** também estão em **NP**, pois um algoritmo que resolve o problema pode ser usado diretamente como verificador para o mesmo. Ou seja, claramente temos $\mathbf{P} \subseteq \mathbf{NP}$. A grande questão é, será que $\mathbf{NP} \subseteq \mathbf{P}$?

Problema 29.8: *P vs. NP*

P é igual a **NP**?

Esse problema, porém, continua em aberto até os dias atuais. Dada sua importância, ele é um dos *Problemas do Milênio* e o *Clay Institute* oferece um prêmio monetário de $U\$1.000.000,00$ para quem conseguir resolvê-lo[1].

[1] https://www.claymath.org/millennium-problems

29.1 Classe NP-completo

> "Even without a proof that NP-completeness implies intractability, the knowledge that a problem is NP-complete suggests, at the very least, that a major breakthrough will be needed to solve it with a polynomial time algorithm."
>
> *Michael R. Garey, David S. Johnson* – Computers and Intractability, 1979.

Definição 29.9: *Classe NP-completo*

NP-completo é o conjunto que contém problemas A tais que $A \in \mathbf{NP}$ e todo outro problema de **NP** é redutível a A.

Pela definição acima e pela definição de redução, podemos concluir que se um único algoritmo eficiente para resolver um problema **NP**-completo for encontrado, então teremos um algoritmo eficiente para resolver **todos** os problemas em **NP**.

Teorema 29.10

Seja A um problema **NP**-completo. $\mathbf{P} = \mathbf{NP}$ se e somente se A pertence a **P**.

Por isso, se quisermos dar uma forte razão da intratabilidade de um problema, basta mostrarmos que ele é **NP**-completo.

Mas como mostramos que um problema é **NP**-completo? Pela definição, precisamos mostrar primeiro que o novo problema está em **NP** e depois precisaríamos enumerar todos os problemas em **NP** e fazer uma redução deles para o nosso problema. Essa segunda parte não parece nada simples. Acontece que a redução de problemas é uma operação que pode ser composta. Isto é, se A reduz para B e B reduz para C, então diretamente temos que A reduz para C. Por isso, basta escolher algum problema que já é **NP**-completo e reduzir dele para o nosso. Porém, para que essa estratégia funcione, ainda é necessário um ponto de partida, i.e., é necessário que exista uma prova de que algum problema é **NP**-completo que não necessite de outro problema **NP**-completo para funcionar. Esse ponto de partida é o problema 3-SAT.

Considere um conjunto de *variáveis booleanas* $x_1, \ldots, x_n$, i.e., que só recebem valores 0 ou 1, e uma *fórmula* composta por conjunções (operadores **e**) de conjuntos de disjunções

(operadores **ou**) das variáveis dadas e suas negações. Exemplos dessas fórmulas são

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_2) \quad \text{e} \quad (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_4 \vee x_5) \wedge (\overline{x_4} \vee x_5 \vee \overline{x_6}) \ .$$

Cada conjunto de disjunções é chamado de *cláusula* e um *literal* é uma variável x ou sua negação $\overline{x}$. Uma fórmula booleana composta por conjunções de cláusulas que contêm exatamente 3 literais é chamada de 3-CNF. Por exemplo, as fórmulas abaixo são 3-CNF.

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_4) \quad \text{e} \quad (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_4) \wedge (\overline{x_4} \vee x_5 \vee \overline{x_6}) \ .$$

Problema 29.11: *3-SAT*

Dada uma fórmula 3-CNF ϕ contendo literais de variáveis booleanas $x_1, \ldots, x_n$, existe uma atribuição de valores a $x_1, \ldots, x_n$ tal que ϕ é *satisfatível*, i.e., ϕ tem valor 1?

Note que o 3-SAT está em **NP** pois, dada uma fórmula ϕ e uma atribuição das variáveis, é fácil verificar se essa atribuição satisfaz a fórmula. Em 1971, os pesquisadores Stephen Cook e Leonid Levin provaram que o 3-SAT é **NP**-completo.

Teorema 29.12: *Cook-Levin*

3-SAT é NP-completo.

Em 1972, Richard Karp apresentou um artigo com uma lista de 21 outros problemas em **NP**-completo, criando de fato, na época, um conjunto desses problemas. Hoje em dia temos milhares de problemas **NP**-completos.

29.2 Exemplos de problemas NP-completos

Nessa seção mostraremos vários exemplos de reduções para mostrar que um problema novo é **NP**-completo. Partiremos apenas do fato que o 3-SAT é **NP**-completo.

Nosso primeiro resultado é sobre o problema CLIQUE (29.6).

Teorema 29.13

3-SAT é redutível para CLIQUE.

Demonstração. Precisamos exibir um algoritmo eficiente que converte uma entrada do 3-SAT,

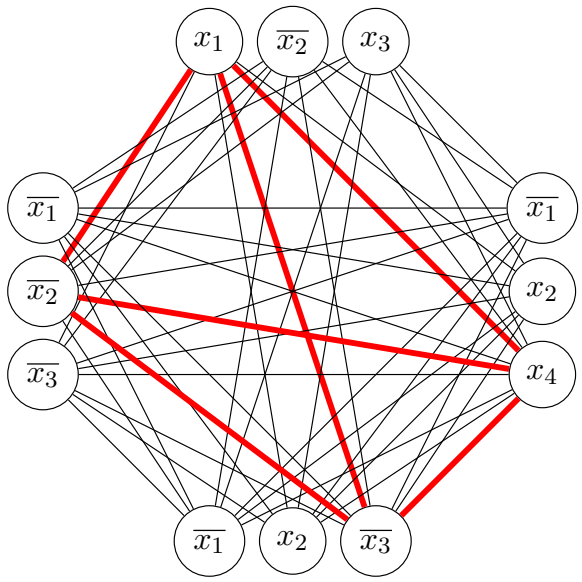


Figura 29.1: Grafo G gerado a partir da instância $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$ do 3-SAT. A clique em destaque corresponde à solução $x_1 = 1$, $x_2 = 0$, $x_3 = 0$ e $x_4 = 1$.

isto é, uma fórmula 3-CNF ϕ, em um grafo G e um número k de forma que ϕ é satisfatível se e somente se G contém uma clique com pelo menos k vértices.

Seja então ϕ uma fórmula com m cláusulas sobre as variáveis $x_1, \ldots, x_n$. O grafo G que construiremos possui $3m$ vértices, de modo que cada uma das m cláusulas tem 3 vértices representando cada um de seus literais. Um par de vértices v e w tem uma aresta entre eles se e somente se v e w estão em cláusulas diferentes, v corresponde a um literal x, e w não corresponde ao literal $\overline{x}$. Veja a Figura 29.1 para um exemplo de construção de G.

Tomando $k = m$, temos uma instância para o CLIQUE. O próximo passo é verificar que ϕ é satisfatível se e somente se G contém um grafo completo com $k = m$ vértices.

Para mostrar um lado dessa implicação note que se ϕ é satisfatível, então em cada uma das $k = m$ cláusulas existe ao menos um literal com valor 1. Como um literal e sua negação não podem ter ambos valor 1, sabemos que em todo par $\{x, y\}$ desses ao menos k literais temos $x \neq \overline{y}$. Portanto, existe uma aresta entre quaisquer dois vértices representando esses literais em G, de modo que elas formam uma clique com pelo menos k vértices dentro de G.

Para verificar a volta da implicação, suponha existe subconjunto S dos vértices de G que é uma clique com pelo menos k vértices. Como existe uma aresta entre quaisquer dois vértices de S, sabemos que qualquer par de vértices de S representa dois literais que não são a negação um do outro e estão em diferentes cláusulas. Dando valor 1 aos literais representados pelos vértices de S, portanto, satisfaz ϕ. $\square$

Já havíamos mostrado anteriormente que CLIQUE está em **NP**. Isso juntamente com o Teorema 29.13 que acabamos de ver prova o seguinte resultado.

Teorema 29.14

CLIQUE-k é **NP**-completo.

Considere agora o seguinte problema.

Problema 29.15: *VERTEXCOVER*

Dado um grafo G e um inteiro k, existe conjunto $S \subseteq V(G)$ tal que, para toda aresta $uv \in E(G)$, $u \in S$ ou $v \in S$ e $|S| \leq k$?

Primeiro note que esse problema está em **NP**, pois dados G, k e algum conjunto de vértices, é fácil em tempo polinomial verificar se tal conjunto tem tamanho no máximo k e se todas as arestas do grafo têm ao menos um extremo nesse conjunto. O teorema a seguir mostra uma redução de CLIQUE para VERTEXCOVER.

Teorema 29.16

CLIQUE é redutível para VERTEXCOVER.

Demonstração. Precisamos exibir um algoritmo eficiente que converte uma entrada de CLIQUE, isto é, um grafo G e um inteiro k, em um grafo G' e um inteiro k' de forma que G tem uma clique de tamanho pelo menos k se e somente se G' tem uma cobertura por vértices de tamanho no máximo k'. Não é difícil perceber que fazer $G' = G$ e $k' = k$ não nos ajudará.

Faremos $G' = \overline{G}$, o grafo complemento de G, e $k'v(G) - k$. Assim, temos então uma instância VERTEXCOVER construída em tempo polinomial. Resta verificar se G contém uma clique de tamanho pelo menos k se e somente se $\overline{G}$ contém uma cobertura por vértices de tamanho no máximo $k' = v(G) - k$.

Suponha que G contém uma clique S de tamanho pelo menos k. Isso significa que para todo par $u, v \in S$ temos $uv \in E(G)$, o que implica em $uv \notin E(\overline{G})$. Então para toda aresta $xy \in E(\overline{G})$, devemos ter que $x \notin S$ ou $y \notin S$. Logo, $V(G) \setminus S$ é uma cobertura por vértices de $\overline{G}$. Como $|S| \geq k$, temos $|V(G) \setminus S| = |V(G)| - |S| \leq |V(G)| - k = k'$.

Agora suponha que $\overline{G}$ contém uma cobertura por vértices S de tamanho no máximo k'. Isso significa que para toda aresta $uv \in E(\overline{G})$, temos $u \in S$ ou $v \in S$. De forma equivalente, para qualquer par de vértices x, y tais que $x \notin S$ e $y \notin S$, devemos ter $xy \notin E(\overline{G})$, o que

implica em $xy \in E(G)$. Logo, $V(G) \setminus S$ é uma clique em G. Como $S \leq k' = |V(G)| - k$, temos $|V(G) \setminus S| = |V(G)| - |S| \geq |V(G)| - (|V(G)| - k) = k$. $\square$

O Teorema 29.16 que acabamos de ver juntamente com o fato de VERTEXCOVER estar em **NP** demonstra diretamente o seguinte resultado.

Teorema 29.17

VERTEXCOVER é **NP**-completo.

29.3 Classe NP-difícil

Definição 29.18: ***Classe NP-difícil***

NP-difícil é o conjunto que contém problemas A tais que todo outro problema de **NP** é redutível a A.

Pela definição acima, vemos que outra definição para a classe **NP**-completo pode ser: o conjunto de problemas que estão em **NP** e são **NP**-difíceis.

Mas por que precisamos de duas classes de problemas tão parecidas? Essa distinção se dá basicamente porque problemas de otimização não estão em **NP**. Veja por exemplo o problema da mochila inteira. É fácil verificar se um dado conjunto de itens cabe na mochila (basta somar seus pesos e comparar com a capacidade máxima), porém não é fácil saber se o conjunto dá o melhor valor possível. Ao menos não sem de fato resolver o problema de fato. Assim, **NP**-completo $\subset$ **NP**-difícil.

Para mostrar que um problema novo é **NP**-difícil, basta tomarmos um problema que já é **NP**-difícil ou já é **NP**-completo e reduzi-lo para o novo problema. Pela composição da redução, isso mostraria que todos os problemas em **NP** também se reduzem ao novo problema. Por exemplo, o Teorema 29.13 prova diretamente o seguinte resultado.

Teorema 29.19

CLIQUE é **NP**-difícil.

Lembre-se que o fato de CLIQUE ser **NP** finalizou a prova de que ele é **NP**-completo.

Capítulo 30

*Abordagens para lidar com problemas **NP**-difíceis*

> "Discovering that a problem is NP-complete is usually just the beginning of work on that problem. (...) In short, the primary application of the theory of NP-completeness is to assist algorithm designers in directing their problem-solving efforts toward those approaches that have the greatest likelihood of leading to useful algorithms."
>
> *Michael R. Garey, David S. Johnson* – Computers and Intractability, 1979.

Em breve.