

ANÁLISE DE ALGORITMOS E DE 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: 8 de fevereiro de 2025

(Esta versão é um rascunho ainda em elaboração e pode conter erros.)

Sumário

I	Conhecimentos recomendados	1
1	Revisão de conceitos importantes	5
1.1	Potenciação	5
1.2	Logaritmos	6
1.3	Somatórios	9
1.4	Funções	13
1.5	Equações e inequações	15
II	Princípios da análise de algoritmos	17
2	Um problema simples	23
3	Corretude de algoritmos iterativos	31
3.1	Para que serve uma invariante de laço?	34
3.2	Como provar uma invariante de laço?	35
3.3	Outros exemplos usando invariantes de laço	36
4	Tempo de execução	45
4.1	Análise de melhor caso, pior caso e caso médio	54
4.2	Bons algoritmos	57
5	Notação assintótica	59
5.1	Notações O , Ω e Θ	60
5.1.1	Funções de várias variáveis	66
5.1.2	Relações entre as notações O , Ω e Θ	67
5.2	Notações o e ω	68
5.3	Relações entre as notações assintóticas	70
6	Tempo com notação assintótica	71
6.1	Exemplo completo de solução de problema	75

7	Recursividade	79
7.1	Corretude de algoritmos recursivos	81
7.2	Fatorial de um número	82
7.3	Potência de um número	83
7.4	Busca binária	89
7.5	Algoritmos recursivos \times algoritmos iterativos	92
8	Recorrências	97
8.1	Método da substituição	99
8.1.1	Diversas formas de obter o mesmo resultado	102
8.1.2	Ajustando os palpites	103
8.1.3	Desconsiderando pisos e tetos	104
8.1.4	Mais exemplos	105
8.2	Método iterativo	108
8.2.1	Limitantes assintóticos inferiores e superiores	109
8.3	Método da árvore de recursão	110
8.4	Método mestre	113
8.4.1	Versão simplificada do método mestre	116
8.4.2	Resolvendo recorrências com o método mestre	119
8.4.3	Ajustes para aplicar o método mestre	120
III	Estruturas de dados	123
9	Estruturas lineares	129
9.1	Vetor	129
9.2	Lista encadeada	131
10	Pilha e fila	135
10.1	Pilha	135
10.2	Fila	137
11	Árvores	141
11.1	Árvores binárias de busca	142
11.1.1	Busca em árvores binárias de busca	143
11.1.2	Inserção em árvores binárias de busca	145
11.1.3	Remoção em árvores binárias de busca	147
11.1.4	Outras operações sobre árvores binárias de busca	149
11.1.5	Altura de uma árvore binária de busca	151
11.2	Árvores balanceadas	153
12	Fila de prioridades	155
12.1	Heap binário	156

12.1.1	Construção de um heap binário	164
12.1.2	Inserção em um heap binário	167
12.1.3	Remoção em um heap binário	168
12.1.4	Alteração em um heap binário	169
13	Disjoint Set	171
13.1	Union-Find	172
14	Tabelas hash	175
IV	Algoritmos de ordenação	177
15	Ordenação por inserção	183
15.1	Insertion sort	183
15.2	Shellsort	188
16	Ordenação por intercalação	191
17	Ordenação por seleção	201
17.1	Selection sort	201
17.2	Heapsort	205
18	Ordenação por troca	211
18.1	Quicksort	211
18.1.1	Análise do tempo de execução	218
19	Ordenação sem comparação	223
19.1	Counting sort	224
V	Técnicas de construção de algoritmos	227
20	Divisão e conquista	233
20.1	Multiplicação de inteiros	233
21	Algoritmos gulosos	241
21.1	Escalonamento de tarefas compatíveis	241
21.2	Mochila fracionária	246
21.3	Compressão de dados	251
22	Programação dinâmica	259
22.1	Sequência de Fibonacci	260
22.2	Corte de barras de ferro	264
22.3	Mochila inteira	269

22.4 Alinhamento de sequências	276
--	-----

VI Algoritmos em grafos 279

23 Conceitos essenciais em grafos 285

23.1 Relação entre grafos e digrafos	287
23.2 Adjacências e incidências	288
23.3 Grafos e digrafos ponderados	290
23.4 Formas de representação em computadores	291
23.5 Pseudocódigos	293
23.6 Subgrafos	296
23.6.1 Modificando grafos	297
23.7 Passeios, trilhas, caminhos e ciclos	298
23.8 Conexidade	299
23.9 Distância entre vértices	301
23.10 Algumas classes importantes de grafos	302
23.10.1 Árvores	303

24 Buscas em grafos 307

24.1 Busca em largura	312
24.1.1 Distância em grafos não ponderados	315
24.2 Busca em profundidade	318
24.2.1 Ordem de descoberta	322
24.3 Componentes conexas	323
24.4 Busca em digrafos	324
24.4.1 Componentes fortemente conexas	326
24.4.2 Ordenação topológica	329
24.5 Outras aplicações dos algoritmos de busca	331

25 Árvores geradoras mínimas 339

25.1 Algoritmo de Kruskal	343
25.2 Algoritmo de Prim	351

26 Trilhas Eulerianas 359

27 Caminhos mínimos 365

27.1 Única fonte	367
27.1.1 Algoritmo de Dijkstra	369
27.1.2 Algoritmo de Bellman-Ford	377
27.2 Todos os pares	381
27.2.1 Algoritmo de Floyd-Warshall	381
27.2.2 Algoritmo de Johnson	389

VII Teoria da computação	393
28 Redução entre problemas	399
28.1 Problemas de otimização e decisão	403
28.2 Formalizando a redução	406
28.3 O que se ganha com redução?	410
29 Classes de complexidade	411
29.1 Classe NP-completo	415
29.2 Exemplos de problemas NP-completos	417
29.3 Classe NP-difícil	419
30 Abordagens para lidar com problemas NP-difíceis	421

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.

Revisão de conceitos importantes

1.1 Potenciação

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

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

Observe que 0^x está indefinido quando $x < 0$ e que 0^0 também está indefinido.

Se a é um número real e m é um inteiro positivo, não sendo o caso de $a < 0$ enquanto m for par, então $a^{1/m}$ é definido como sendo o número que satisfaz

$$(a^{1/m})^m = a.$$

Portanto, se $a < 0$ e m é par, então $a^{1/m}$ é indefinido.

Se a é um número real e $x = \frac{n}{m}$ é um número racional com n sendo inteiro e m sendo inteiro positivo, então a operação de potenciação é definida por

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

Observe que $a^{n/m}$ está indefinido quando $a < 0$ e m é par e quando $n \leq 0$ e $a = 0$.

É importante 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 para manipulação de potências:

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

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 ($a^x = n$).

Assim, por definição,

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

Algumas bases são tão importantes que o símbolo de logaritmo é modificado ao usá-las. Definimos $\lg n$ como sendo $\log_2 n$, $\ln n$ como sendo $\log_e n$ e $\log n$ como sendo $\log_{10} n$.

Como exemplos, temos que o logaritmo de 32 na base 2 é 5 ($\lg 32 = 5$) 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: $\lg 1024 = 10$, pois 1024 é potência de 2; $\log 100 = 2$, pois 100 é potência de 10; $\log_5 125 = 3$, pois 125 é potência de 5.

Outros precisam de calculadora: $\lg 37$ é aproximadamente 5,20945; $\log 687$ é aproximadamente 2,83696; $\log_5 341$ é aproximadamente 3,62355.

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

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

No fato a seguir listamos as propriedades mais comuns para 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

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

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

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

(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 $\log_c a = \log_c(b^{\log_b a})$. 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,

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

□

1.3 Somatórios

Seja a_1, a_2, \dots, a_n uma sequência de números reais. O **somatório** desses números é denotado por

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

A expressão acima é lida como “a soma de termos a_i , com i variando de 1 até n ”. Chamamos i de *variável do somatório*, cujo valor inicial aparece abaixo do símbolo Σ e à direita do sinal de $=$, e o valor final aparece acima do símbolo Σ .

Não é necessário que a variável do somatório comece em 1, mas ela obrigatoriamente aumenta de uma em uma unidade. Assim, se o valor inicial da variável é maior do que seu valor final, a variável não assume valor nenhum e dizemos que o somatório é vazio, com sua soma sendo igual a 0, como no exemplo a seguir:

$$\sum_{i=67}^{12} i = 0.$$

Veja outros exemplos abaixo:

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

Outras notações comuns são:

- $\sum_{i \leq k \leq j} a_k = a_i + a_{i+1} + \cdots + a_j$, que equivale a $\sum_{k=i}^j a_k$;
 - Qualquer outra notação de intervalo também é permitida, como $k \in [i, j]$.
- $\sum_{x \in S} a_x$, que é a soma de a_x para todo possível x de um conjunto S .
 - Por exemplo, se $S = \{4, 8, 13\}$, então $\sum_{x \in S} x = 4 + 8 + 13$.

No fato abaixo listamos as propriedades para 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:

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

A seguir temos algumas fórmulas para somatórios 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, \dots, 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 pode ser escrita como

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

Definição 1.7: Progressão geométrica

Uma *progressão geométrica com razão q* é uma sequência de números (b_1, b_2, \dots, 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 pode ser escrita como

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

Teorema 1.8

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

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

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

onde na segunda igualdade utilizamos o Teorema 1.5 e na última utilizamos o fato que $a_n = a_1 + r(n-1)$. \square

Teorema 1.9

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

Demonstração. Seja S o valor da soma dessa progressão. Assim, por definição, $S = b_1(1 + q + q^2 + \dots + q^{n-2} + q^{n-1})$.

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 $(c^0, c^1, c^2, \dots, c^n)$ 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

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

□

O resultado do Corolário 1.10 é bastante útil. Quando nos deparamos com o somatório $\sum_{i=0}^n c^i$ e temos $c > 1$, então é mais vantajoso substituir esse somatório por $\frac{c^{n+1}-1}{c-1}$, uma vez que o numerador e o denominador são números positivos. Igualmente, se $c < 1$, então é mais vantajoso substituí-lo por $\frac{1-c^{n+1}}{1-c}$.

1.4 Funções

Uma *função* é uma relação entre dois conjuntos que associa cada elemento de um conjunto, chamado *domínio*, a um elemento do outro conjunto, chamado *contradomínio*.

nio. Se f é uma função e x é um elemento do seu domínio, denotamos o elemento ao qual x é associado no contradomínio por $f(x)$. A *imagem* de uma função f é o conjunto de elementos y tais que $f(x) = y$ para pelo menos um x no domínio de f .

Se f é uma função com domínio X e contradomínio Y , então escrevemos:

$$f: X \rightarrow Y.$$

Assim,

$$f(x) = y \quad \Rightarrow \quad x \in X \text{ e } y \in Y.$$

Seja $f: \{1, 2, 3\} \rightarrow \{2, 4, 6\}$ definida como $f(x) = 2x$, para $x \in \{1, 2, 3\}$. Outra forma de representação para f é tabular:

x	$f(x)$
1	2
2	4
3	6

Sejam a , b e a_i , para $0 \leq i \leq k$, constantes. As funções a seguir são especiais e recebem nomenclatura própria:

- Constante: $f(n) = a$
- Linear: $f(n) = an + b$
- Polinomial: $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$
- Racional: $f(n) = p(n)/q(n)$
- Exponencial: $f(n) = a^n$
- Logarítmica: $f(n) = \log_a n$

Outras duas funções importantes e que usaremos bastante são as funções piso e teto. A função *piso* recebe um número real x e devolve o *maior inteiro que é menor ou igual a x* . Ela é comumente denotada por *piso*(x) ou $\lfloor x \rfloor$. Por exemplo, $\lfloor 3,4 \rfloor = 3$, $\lfloor 3 \rfloor = 3$ e $\lfloor 2,999998 \rfloor = 2$.

A função *teto* recebe um número real x e devolve o *menor inteiro que é maior ou igual a x* . Ela é comumente denotada por *teto*(x) ou $\lceil x \rceil$. Por exemplo, $\lceil 3,4 \rceil = 4$, $\lceil 3 \rceil = 3$ e $\lceil 2,999998 \rceil = 3$.

Algumas propriedades importantes das funções piso e teto são:

- $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
- $\lfloor x \rfloor \leq \lceil x \rceil$
- $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$, se $n \in \mathbb{Z}$

1.5 Equações e inequações

Uma *equação* é uma proposição sobre a igualdade de duas expressões: $a = b$. Se $a = b$ e c é uma outra expressão, então também vale que:

- $a + c = b + c$;
- $ca = cb$, se $c \neq 0$.

Uma *inequação* é uma proposição sobre duas expressões separadas por um símbolo de desigualdade: $a < b$, $a \leq b$, $a > b$ ou $a \geq b$.

Algumas propriedades de inequações também nos serão bastante úteis:

- $a \leq a + b$, com $b \geq 0$.
- $a \geq a - b$, com $b \geq 0$.
- se $a \leq b$, então $a + c \leq b + c$.
- se $a \leq b$ e $b \leq c$, então $a \leq c$.
- se $a \leq b$ e $c \leq d$, então $a + c \leq b + d$.
- se $a \leq b$ e $c \geq 0$, então $ac \leq bc$.
- se $a \leq b$ e $c \leq 0$, então $ac \geq bc$.
- se $a \leq b$, então $-a \geq -b$.
- se $a \leq b$ e a e b têm mesmo sinal, então $\frac{1}{a} \geq \frac{1}{b}$.
- se $a < 0 < b$, então $\frac{1}{a} < \frac{1}{b}$.

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

Um **problema computacional** é um problema que desejamos resolver usando computadores. Cada problema é bem descrito por meio de uma **entrada**, que indica parâmetros e dados importantes para o problema, e uma **saída**, que indica a resolução esperada do problema. Por exemplo, o *problema da multiplicação de inteiros* tem como entrada dois números a e b e tem como saída um único número que seja o resultado da multiplicação dos dois números da entrada ($a \times b$). Uma **instância** de um problema é uma atribuição específica de valores para sua entrada. Toda instância tem uma saída específica esperada, chamada de **solução**. Por exemplo, $a = 5$ e $b = 9$ seria uma instância para o problema da multiplicação de inteiros e 45 seria sua solução.

Para resolver um problema computacional, espera-se que seja desenvolvido um algoritmo. Um **algoritmo** é uma sequência finita de passos descritos de forma não ambígua que recebe um conjunto de dados e produz uma saída. Dizemos que um algoritmo **resolve** um determinado problema, ou que ele está **correto**, se ele devolve uma solução para *qualquer* instância do 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, ainda durante o ensino fundamental aprendemos a resolver o problema da multiplicação de inteiros por um algoritmo bastante conhecido: comece multiplicando o dígito menos significativo do número b pelo número a ; obtenha agora a multiplicação do segundo dígito menos significativo de b por a , multiplique-o por 10 e some com o número obtido no passo anterior; repita esse processo até esgotar os dígitos de b . Mas 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.

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 adequa 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 números¹.

Nesta 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.

¹Estamos falando aqui de números que possam ser representados por uma quantidade constante de bits, como por exemplo 32 ou 64.

Um problema simples

Vetores são estruturas de dados lineares que agrupam elementos, ou variáveis, de 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 $A = (a_1, a_2, \dots, a_n)$ e $A[i] = a_i$ é o elemento que está armazenado na posição i , para todo $1 \leq i \leq n$. Ademais, para quaisquer $1 \leq i < j \leq n$, denotamos por $A[i..j]$ o subvetor de A que contém os elementos $A[i], A[i + 1], \dots, A[j]$ ¹.

Todo elemento $A[i]$ armazenado em um vetor será considerado um **registro** e, assim, terá um campo único **chave**, utilizado para distinguir os elementos armazenados. Assim, $A[i].\text{chave} \neq A[j].\text{chave}$ se e somente se $i \neq j$. Consideraremos, por simplicidade, que tal campo **chave** é numérico.

Problema 2.1: Busca

Entrada: $\langle A, n, k \rangle$, onde $A[1..n]$ é um vetor contendo n elementos e k é um número.

Saída: Posição i do elemento de A cuja chave é igual a k , se esse existe, ou -1 , caso contrário.

O número k da descrição acima é chamado **chave de busca**.

¹Em um dilema estilo “o ovo ou a galinha”, escolhemos por descrever mais detalhes sobre vetores na parte do livro que trata de estruturas de dados, na Seção 9.1.

O algoritmo mais simples para o Problema 2.1 é a **busca linear**, descrita no Algoritmo 2.1. Ele recebe o vetor A , a quantidade n de elementos armazenados em A e uma chave de busca k e devolve um valor i , com $1 \leq i \leq n$, caso encontre um elemento com chave igual a k , ou devolve -1 , caso não encontre tal elemento.

Algoritmo 2.1: BUSCALINEAR(A, n, k)

```

1  $i = 1$ 
2 enquanto  $i \leq n$  e  $A[i].\text{chave} \neq k$  faça
3    $i = i + 1$ 
4 se  $i \leq n$  e  $A[i].\text{chave} == k$  então
5   devolve  $i$ 
6 devolve  $-1$ 

```

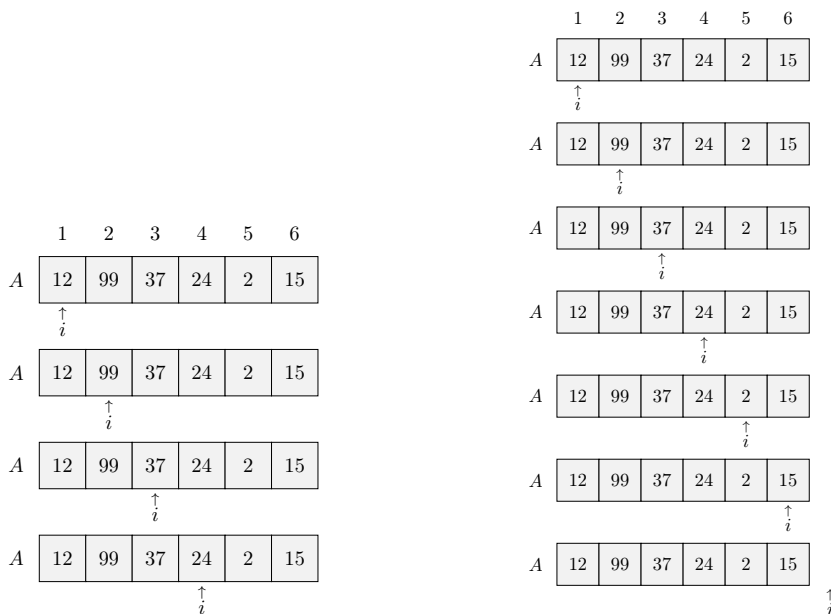
No que segue, seja n a quantidade de elementos armazenados no vetor A (seu tamanho)². O funcionamento do algoritmo BUSCALINEAR é bem simples. A variável i indica qual posição do vetor A estamos analisando e inicialmente fazemos $i = 1$. Incrementamos o valor de i em uma unidade sempre que as duas condições do laço **enquanto** forem satisfeitas, i.e., quando $i \leq n$ e $A[i].\text{chave} \neq k$. Assim, caso k seja encontrada, o laço **enquanto** é encerrado e o algoritmo devolve o índice i tal que $A[i].\text{chave} = k$. Caso contrário, o algoritmo devolve -1 , indicando que um elemento com chave 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 Busca, isto é, que *para quaisquer* vetor A e número k , o algoritmo irá devolver a posição do elemento que tem chave k em A caso ela exista, ou irá devolver -1 caso não exista. Mas, como podemos ter certeza que o comportamento de BUSCALINEAR é sempre como esperamos que seja? Sempre que um elemento com chave k existir no vetor, o algoritmo o encontrará? Sempre que o algoritmo devolver -1 , é verdade que realmente não há elemento com chave igual a k no vetor? Também pela simplicidade desse algoritmo, é fácil verificar que o comportamento é de fato o esperado, porém no Capítulo 3 veremos uma forma de *provar* que um dado algoritmo funciona corretamente, que funciona para algoritmos mais complexos do que este. Antes disso, vejamos outro problema de busca em vetores.

Problema 2.2: Busca em Dados Ordenados

Entrada: $\langle A, n, k \rangle$, onde $A[1..n]$ é um vetor contendo n elementos cujas chaves

²Em 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 $BUSCA LINEAR(A, 6, 24)$. (b) Execução de $BUSCA LINEAR(A, 6, 10)$.

Figura 2.1: Exemplos de execução de $BUSCA LINEAR(A, n, k)$ (Algoritmo 2.1). Por simplicidade, representamos apenas as chaves de busca dentro dos vetores.

estão em ordem crescente, i.e., $A[i].\text{chave} < A[i+1].\text{chave}$ para todo $1 \leq i < n$, e k é um número.

Saída: Posição i do elemento de A cuja chave é igual a k , se esse existe, ou -1 , caso contrário.

Também de forma intuitiva, é fácil acreditar que o algoritmo $BUSCA LINEAR$ resolve o problema 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 cujas chaves são menores do que $A[i].\text{chave}$ e o subvetor $A[i+1..n]$ contém todos os elementos cujas chaves são maiores do que $A[i].\text{chave}$ ³.

Seja i um índice entre 1 e n . Se $k < A[i].\text{chave}$, então pelo fato das chaves do vetor estarem em ordem crescente, certamente não é possível que o elemento com chave k esteja armazenado no subvetor $A[i..n]$. Com isso, podemos ter um terceiro critério de parada para a busca linear: assim que $k > A[i].\text{chave}$, não há o que procurar. Essa nova ideia está formalizada no Algoritmo 2.2. A Figura 2.2 apresenta dois exemplos

³Note 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.

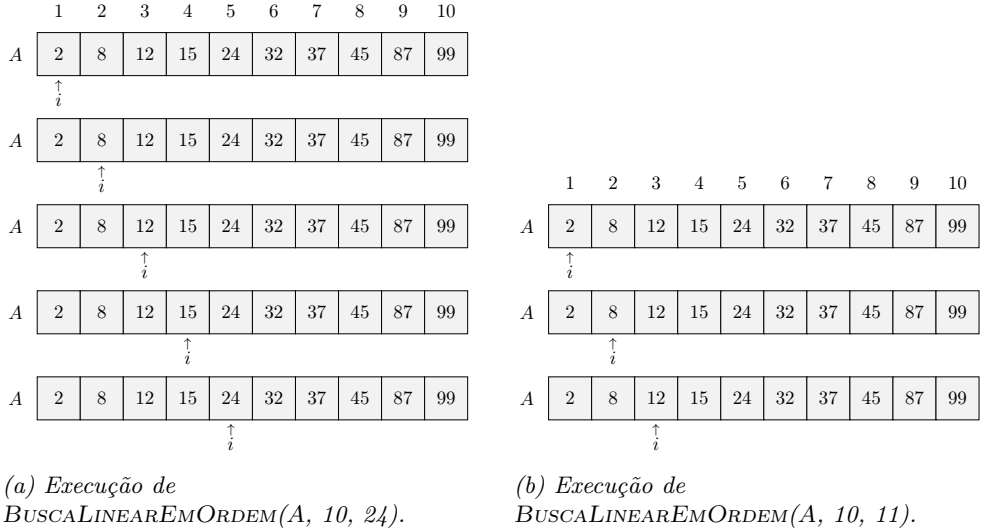


Figura 2.2: Exemplos de execução de $BUSCALINEARORDEN(A, n, k)$ (Algoritmo 2.2).
Por simplicidade, representamos apenas as chaves de busca dentro dos vetores.

de execução de $BUSCALINEARORDEN$.

Algoritmo 2.2: $BUSCALINEARORDEN(A, n, k)$

```

1  $i = 1$ 
2 enquanto  $i \leq n$  e  $A[i].chave < k$  faça
3    $i = i + 1$ 
4 se  $i \leq n$  e  $A[i].chave == k$  então
5   devolve  $i$ 
6 devolve  $-1$ 

```

Para o Problema 2.2, existe ainda um terceiro procedimento, chamado de **busca binária**, que também faz a busca por um elemento de chave k em um vetor com n chaves ordenadas. Na discussão a seguir, por simplicidade, assumimos que n é múltiplo de 2.

A busca binária se aproveita da informação extra de que as chaves do vetor estão ordenadas comparando inicialmente k com a chave do elemento mais ao centro do vetor, $A[n/2]$. Ao realizar essa comparação, existem apenas três possibilidades: $k = A[n/2].chave$, $k < A[n/2].chave$ ou $k > A[n/2].chave$. Se $k = A[n/2].chave$, então encontramos a chave e a busca está encerrada com sucesso. Caso contrário, se $k < A[n/2].chave$, então temos a certeza de que, se um elemento com chave k estiver em A , então ele estará na primeira metade de A , i.e., pode estar somente em $A[1..n/2 - 1]$ (isso segue do fato das chaves em A estarem em ordem). Caso

$k > A[n/2].\text{chave}$, então sabemos que, se um elemento com chave k estiver em A , então ele pode estar somente no vetor $A[n/2 + 1..n]$. Note que se $k \neq A[n/2].\text{chave}$, então essa estratégia elimina metade do espaço de busca, isto é, antes o elemento com chave 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 $k < A[n/2].\text{chave}$. Note que podemos verificar se o elemento com chave k está em $A[1..n/2 - 1]$ utilizando a mesma estratégia, i.e., comparamos k com a chave do elemento que está na metade do vetor $A[1..n/2 - 1]$, isto é, 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. A Figura 2.3 apresenta exemplos de execução de BUSCABINARIA.

O Algoritmo 2.3 formaliza a busca binária, que recebe um vetor $A[1..n]$ cujas chaves estão ordenadas de modo crescente e uma chave de busca k . Ele devolve a posição em que o elemento com chave k está armazenado, se tal elemento existir, ou devolve -1 , caso contrário. As variáveis *esq* e *dir* armazenam, respectivamente, as posições inicial e final do espaço de busca, isto é, o elemento de chave k está, a todo momento, sendo procurado no subvetor $A[\text{esq}..\text{dir}]$. Assim, inicialmente $\text{esq} = 1$ e $\text{dir} = n$.

Algoritmo 2.3: BUSCABINARIA(A, n, k)

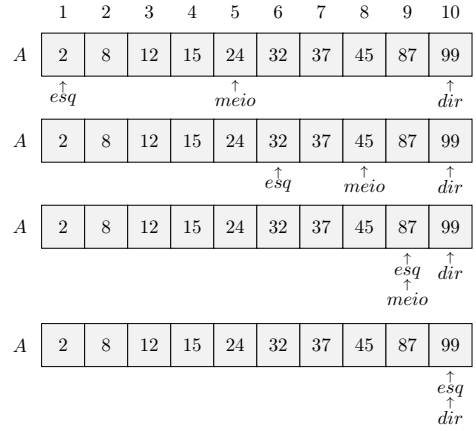
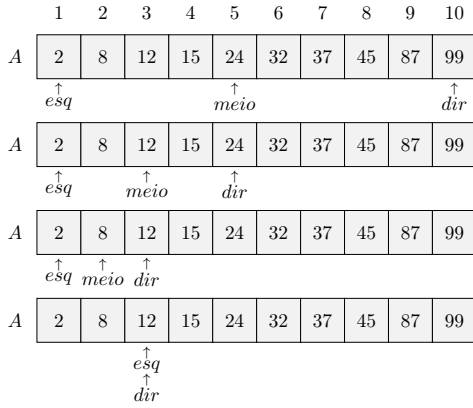
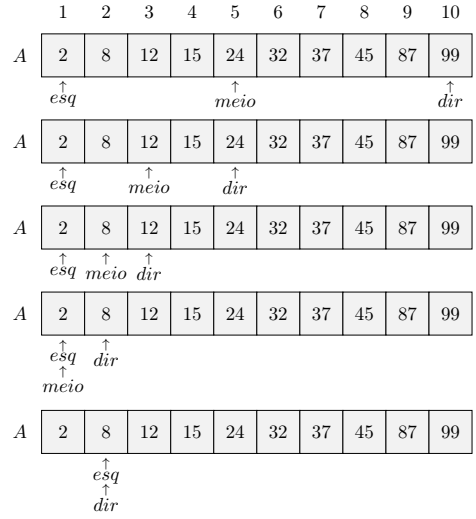
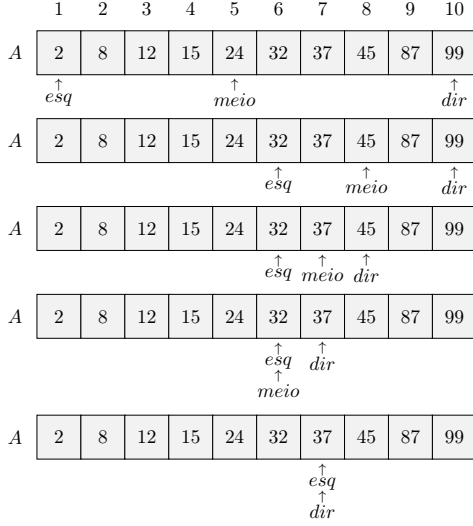
```

1  esq = 1
2  dir = n
3  enquanto esq < dir faça
4      |   meio =  $\lfloor (\text{esq} + \text{dir})/2 \rfloor$ 
5      |   se  $k > A[\text{meio}].\text{chave}$  então
6      |       |   esq = meio + 1
7      |   senão
8      |       |   dir = meio
9  se  $A[\text{esq}].\text{chave} == k$  então
10     |   devolve esq
11 devolve -1
```

Seja *meio* uma variável que vai armazenar o índice mais ao centro do vetor $A[\text{esq}..\text{dir}]$, que é um vetor com $\text{dir} - \text{esq} + 1$ posições. Se $\text{dir} - \text{esq} + 1$ é ímpar, então os vetores $A[\text{esq}..\text{meio} - 1]$ e $A[\text{meio} + 1..\text{dir}]$ têm exatamente o mesmo tamanho, i.e., $\text{meio} - \text{esq} = \text{dir} - \text{meio}$, o que implica $\text{meio} = (\text{esq} + \text{dir})/2$. Caso $\text{dir} - \text{esq} + 1$ seja par, então os tamanhos dos vetores $A[\text{esq}..\text{meio} - 1]$ e $A[\text{meio} + 1..\text{dir}]$ diferem de uma

unidade. Se $A[esq..meio-1]$ é o menor desses vetores, i.e., $meio-esq = dir-meio-1$, então $meio \leq (esq + dir - 1)/2 = \lfloor (esq + dir)/2 \rfloor$. 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 $meio = \lfloor (esq + dir)/2 \rfloor$.

Já não é tão intuitivo o fato de que a busca binária resolve corretamente o problema Busca em Dados Ordenados. Será que para qualquer vetor A recebido, contanto que suas chaves estejam ordenadas de forma crescente, e para qualquer valor k , a busca binária corretamente encontra se há um elemento com chave k armazenado em A ? É possível que o algoritmo devolva -1 mesmo se um elemento com chave 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? Nos próximos capítulos começaremos a responder essas perguntas.



(c) Execução de $\text{BUSCABINARIA}(A, 10, 11)$. (d) Execução de $\text{BUSCABINARIA}(A, 10, 103)$.

Figura 2.3: Exemplos de execução de $\text{BUSCABINARIA}(A, n, k)$ (Algoritmo 2.3). Por simplicidade, representamos apenas as chaves de busca dentro dos vetores.

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 recebida¹.

Para mostrar que um algoritmo está incorreto, portanto, é simples: basta encontrar uma instância específica para a qual o algoritmo devolve uma resposta errada. Mas como mostrar que o algoritmo está correto? Em geral a quantidade de instâncias possíveis é infinita, de forma que não é nem um pouco viável analisar cada uma delas. Assim, é necessário lançar mão de demonstrações matemáticas.

Alguns algoritmos podem ser bastante simples, tanto que alguma demonstração direta pode ser suficiente. No entanto, estamos mais preocupados com algoritmos mais complexos, que provavelmente envolvem laços (**para** ou **enquanto**).

De forma geral, mostraremos que esses algoritmos possuem certas propriedades que continuam verdadeiras após cada iteração de um determinado laço. Ao final do algoritmo essas propriedades devem nos ajudar a fornecer uma prova de que o algoritmo agiu corretamente.

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

Veja que o laço **para** do algoritmo começa fazendo $i = 1$ e, a cada iteração, incrementa

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

Algoritmo 3.1: SOMATORIO(A, n)

```

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

```

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, já acessamos os valores em $A[1..3]$. Durante a quarta iteração, veja que acessamos 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 logo no início de uma t -ésima iteração qualquer, é verdade que o valor da variável i é igual a t e que os elementos em $A[1..i - 1]$ já foram acessados. Nessa iteração acessaremos $A[t]$, mantendo o fato de que os elementos em $A[1..i - 1]$ já foram acessados antes da próxima iteração começar, uma vez que i vale $t + 1$ em seu início. Dizemos então que a afirmação

$$\begin{aligned} \text{“antes da } t\text{-ésima iteração começar, vale que } i = t \text{ e os elementos de } A[1..i - 1] \\ \text{já foram acessados”} \end{aligned} \tag{3.1}$$

é uma *invariante de laço*. Essa afirmação apresenta uma propriedade do algoritmo que é verdadeira no início de qualquer uma das iterações do laço, qualquer que seja o valor da variável i naquele momento.

Mas e antes da primeira iteração, quando $i = 1$, é verdade que os elementos do subvetor $A[1..0]$ já foram acessados? Esse é um caso especial em que a afirmação (3.1) é verdadeira por **vacuidade**. Basicamente, o princípio da vacuidade nos garante que qualquer afirmação sobre um elemento que pertença ao conjunto vazio é verdadeira. Por isso, como o subvetor $A[1..0]$ é vazio, de fato podemos dizer que seus elementos já foram acessados².

Definição 3.1: *Invariante de laço*

É uma afirmação que é verdadeira no início de qualquer iteração do laço.

As afirmações que são invariantes de laço normalmente começam com “Antes da t -

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

ésima iteração começar, vale que ...” e envolvem variáveis importantes para o laço, que provavelmente terão seu valor modificado no decorrer da execução do laço³.

Neste livro, vamos denotar invariantes por alguma letra maiúscula, em geral P , seguida do número que indica a iteração entre parênteses, como por exemplo $P(t)$. Com essa nomenclatura e a partir da definição de invariante, vemos que para provar que uma afirmação P é uma invariante de laço, precisamos provar que cada afirmação $P(1), P(2), \dots, P(T+1)$ é verdadeira, onde T é a quantidade de iterações realizadas.

Por que não provar apenas até $P(T)$? Isso está relacionado com o **objetivo de uma invariante de laço**: queremos provar afirmações que possam ser úteis para, por exemplo, mostrar que um algoritmo está correto. Por isso, o que quer que seja feito durante a execução do laço, terá efeito apenas quando todas as iterações do laço terminarem. Assim, veja que $P(T)$ nos dirá algo sobre variáveis importantes do laço no início da T -ésima (última) iteração. Acontece que durante essa iteração, tais variáveis mudarão de valor, de forma que $P(T)$ não será mais útil ao final da execução do laço. Estamos realmente interessados na afirmação $P(T+1)$: ela é verdadeira antes da $(T+1)$ -ésima iteração, mas como não haverá uma $(T+1)$ -ésima iteração, ela nos dá uma verdade válida **ao final da execução do laço**.

Agora note que verificar se $P(1)$ vale, depois se $P(2)$ vale, depois se $P(3)$ vale, etc., não é algo viável. Primeiro porque se a quantidade total de iterações é, por exemplo, 1000, essa tarefa se torna extremamente tediosa e longa. Mas, pior do que isso, em geral não sabemos o valor de T , de forma que essa tarefa é na verdade impossível!

Felizmente, não precisamos verificar *explicitamente* $P(t)$ para cada valor possível de t . Basta verificar $P(1)$ explicitamente (caso base) e mostrar, supondo que $P(t)$ vale, que $P(t+1)$ também vale, para qualquer $t \geq 1$. Em outras palavras, basta provar que P é verdadeira **por indução**. Lembre-se que uma prova por indução é a melhor estratégia para provar que algo vale para todos os números naturais.

Observe que nessa estratégia da prova por indução, não mencionamos a quantidade de iterações do laço. De fato, precisamos apenas mostrar que algo vale antes da primeira iteração e que, se vale antes de uma iteração qualquer, valerá antes da próxima. A quantidade total de iterações do laço será usada depois que provarmos que a afirmação é de fato uma invariante. Para toda essa conversa fazer mais sentido, veremos em breve alguns exemplos. Antes, porém, entenderemos por quê invariantes são importantes.

³E é por causa dessas modificações que as invariantes não são válidas em qualquer ponto da iteração, apenas no seu início.

3.1 Para que serve uma invariante de laço?

Supondo que $P(t)$ = “antes da t -ésima iteração começar, vale que $i = t$ e os elementos de $A[1..i - 1]$ já foram acessados” é de fato uma invariante para o SOMATORIO (o que precisamos ainda provar por indução), então vale, em particular, $P(n + 1)$, isto é

$$\begin{aligned} \text{“antes da } (n + 1)\text{-ésima iteração começar, temos } i = n + 1 \text{ e os elementos} \\ \text{de } A[1..n] \text{ já foram acessados.”} \end{aligned} \quad (3.2)$$

Agora, como não há uma $(n + 1)$ -ésima iteração (pois o laço não executa e termina quando $i = n + 1$), podemos concluir que ao final da execução do laço, 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 nós possamos concluir que o algoritmo funciona corretamente. Ela precisa, portanto, ter alguma relação com o problema que estamos querendo resolver. Por isso, invariantes em geral nos ajudam inclusive a descrever a ideia dos algoritmos que estamos apresentando.

No caso do algoritmo SOMATORIO, precisamos que, ao fim do laço **para**, o valor $\sum_{j=1}^n A[j]$ seja calculado. Assim, a invariante a seguir será bem mais útil para provar isso.

Invariante: *Laço para* – SOMATORIO

$$P(t) = \text{“Antes da } t\text{-ésima iteração começar, temos } i = t \text{ e } soma = \sum_{j=1}^{i-1} A[j].\text{”}$$

A frase acima é realmente uma invariante? Quer dizer, o fato de ela aparecer em um quadro de destaque não deveria te convencer disso⁴. Precisamos de fato *provar* que ela é uma invariante.

⁴Na verdade, com o tempo, a intenção é que realmente você leia uma frase e se convença facilmente de que ela é uma invariante.

3.2 Como provar uma invariante de laço?

Como observamos anteriormente, as invariantes contêm informações pertinentes ao início da execução de um laço. São afirmações válidas no exato instante entre o teste da condição do laço e a primeira linha do corpo do laço. Se o corpo do laço executou T vezes, veja que o teste da condição de parada foi feito $T + 1$ vezes (uma a mais para permitir o término do laço). Com essa observação, finalmente podemos esclarecer como provar que uma afirmação é uma invariante⁵: fazemos uma indução **no número de vezes que o teste da condição de parada do laço é executado**. Como todo teste de condição de parada de um laço é executado ao menos uma vez, nosso caso base será sempre $t = 1$. Assim, primeiro devemos provar que $P(1)$ é verdadeira diretamente, depois consideramos um $t \geq 1$ qualquer e assumimos que $P(t)$ vale e por fim provamos que $P(t + 1)$ vale.

Voltando à frase anterior sobre o SOMATORIO, vamos mostrar a seguir que ela é uma invariante de laço.

Caso base. Quando $t = 1$, o teste executou uma vez. Logo antes da iteração começar, temos $soma = 0$ e $i = 1$. Como $\sum_{j=1}^{i-1} A[j] = \sum_{j=1}^0 A[j] = 0$ (pois o índice inicial do somatório é maior do que o final), então temos que $soma$ é de fato igual a esse valor. Então $P(1)$ é verdadeira e o caso base vale.

Passo. Considere então que o teste executou t vezes, com $t \geq 1$. Isso significa que a t -ésima iteração vai começar. Suponha que $P(t)$ vale e vamos mostrar que $P(t+1)$ vale. Para isso, argumentamos sobre o que está sendo feito **durante** a t -ésima iteração.

No início da t -ésima iteração, sabemos que t é valor da variável i (porque $P(t)$ vale) e denote por $soma_t$ o valor da variável $soma$ neste momento. Durante a iteração, tudo que é feito é a atualização da variável $soma$ com o valor $soma_t + A[t]$. Como $P(t)$ é verdadeira, então sabemos que $soma_t = \sum_{j=1}^{t-1} A[j]$. Logo, o valor na variável $soma$ antes da próxima iteração começar é $(\sum_{j=1}^{t-1} A[j]) + A[t]$, que é justamente $\sum_{j=1}^{i-1} A[j]$ quando i vale $t + 1$. Como na próxima iteração o valor da variável i será $t + 1$, acabamos de mostrar que $P(t + 1)$ é verdadeira.

Portanto, concluímos que P é invariante de laço. Como esse laço executa n vezes, $P(n + 1)$ é que nos será útil pois nos dá informação sobre o estado dos dados ao fim da execução do laço. Assim, $P(n + 1)$ nos diz que $soma = \sum_{j=1}^{n+1-1} A[j] = \sum_{j=1}^n A[j]$, que é justamente o que precisamos para mostrar que o algoritmo está correto.

⁵Já havíamos mencionado que seria por indução, porém em uma prova por indução é preciso esclarecer: indução sobre o que?

3.3 Outros exemplos usando invariantes de laço

Analisaremos agora o Algoritmo 3.2, que promete resolver outro problema simples. Ele recebe um vetor $A[1..n]$ contendo números e deve devolver o produtório de seus elementos, i.e., $A[1] \times A[2] \times \cdots \times A[n] = \prod_{i=1}^n A[i]$.

Algoritmo 3.2: PRODUTORIO(A, n)

```

1 produto = 1
2 para  $i = 1$  até  $n$ , incrementando faça
3    $\lfloor$  produto = produto  $\times$   $A[i]$ 
4 devolve produto

```

Como podemos definir uma invariante de laço que nos ajude a mostrar a corretude de PRODUTORIO(A, n)? Um primeiro passo é fazer exemplos de execução, observando o que ocorre em cada iteração especificamente. No caso de PRODUTORIO, antes da primeira iteração começar temos *produto* = 1. Ela faz *produto* = *produto* \times $A[1]$ e, portanto, antes da segunda iteração começar temos *produto* = $A[1]$. Na segunda iteração fazemos *produto* = *produto* \times $A[2]$ e, portanto, antes da terceira iteração começar temos *produto* = $A[1] \times A[2]$. Conforme você entende melhor cada iteração, pode tentar explicar genericamente o que ocorre em todas elas. Dos exemplos anteriores, podemos perceber o padrão descrito na seguinte invariante.

Invariante: Laço para – PRODUTORIO

$$P(t) = \text{“Antes da } t\text{-ésima iteração, vale que } i = t \text{ e } \textit{produto} = \prod_{j=1}^{i-1} A[j]\text{.”}$$

Demonstração. Vamos provar que P é uma invariante para PRODUTORIO por indução na quantidade de vezes t que o teste da condição de parada do laço é executado.

Caso base. Quando $t = 1$, o teste executou uma vez. Logo antes da iteração começar, temos *produto* = 1 e $i = 1$. Como $A[1..i - 1]$ é vazio nesse momento, podemos então por vacuidade afirmar que o produto dos elementos em $A[1..0]$ é igual a 1, o que mostra que $P(1)$ é verdadeira.

Passo. Considere então que o teste executou t vezes, com $t \geq 1$. Isso significa que a t -ésima iteração vai começar. Suponha que $P(t)$ vale e vamos mostrar que $P(t+1)$ vale. Para isso, argumentamos sobre o que está sendo feito **durante** a t -ésima iteração.

No início da t -ésima iteração, temos, por $P(t)$, que $i = t$. Denote por $produto_t$ o valor da variável *produto* neste momento. Veja que ao fim dessa iteração, *produto* terá $produto_t \times A[t]$. Como $P(t)$ vale, $produto_t = \prod_{j=1}^{t-1} A[j]$. Logo, o valor de *produto* ao final dessa iteração é $\prod_{j=1}^t A[j]$, que é justamente $\prod_{j=1}^{i-1} A[j]$ quando i vale $t + 1$. Como i vale $t + 1$ no início da próxima iteração, acabamos de mostrar que $P(t + 1)$ é verdade. \square

Agora, como sabemos que P é uma invariante, temos que, em particular, $P(n + 1)$ vale. Isso nos diz que ao final do laço teremos $produto = \prod_{i=1}^n A[i]$, que é de fato o resultado desejado. Portanto, o algoritmo funciona corretamente.

Antes de partir para outros exemplos, é importante mencionar que o valor de t , que é a quantidade de vezes que o teste do laço é executado, nem sempre tem relação com o valor da variável que controla o laço. Nos dois últimos exemplos isso ocorreu pois a variável i é inicializada justamente com o valor 1 e incrementada de uma em uma unidade. Mas você já deve ter visto outros algoritmos em que isso não acontece. O próprio PRODUTORIO poderia ter sido implementado de forma que i começasse em n e fosse decrementando de uma em uma unidade, já que o importante é acessar todos os elementos do vetor para multiplicá-los. Há laços em que a variável controladora é incrementada de duas em duas unidades ou duplicada a cada iteração. E há ainda laços em que há mais de um critério de parada, como o laço da BUSCALINEAR, que veremos a seguir. Veremos ainda outros exemplos de invariantes para laços “diferentes” no decorrer do livro.

Vamos agora analisar a BUSCALINEAR, que promete resolver o problema Busca visto no Capítulo 2. Por conveniência, repetimos o procedimento no Algoritmo 3.3.

Algoritmo 3.3: BUSCALINEAR(A, n, k)

```

1  $i = 1$ 
2 enquanto  $i \leq n$  e  $A[i].chave \neq k$  faça
3    $i = i + 1$ 
4 se  $i \leq n$  e  $A[i].chave == k$  então
5   devolve  $i$ 
6 devolve  $-1$ 
```

Seja A um vetor com n elementos armazenados e k a chave de busca. No caso da busca linear, precisamos mostrar que “existe um elemento com chave k no vetor se e somente se o algoritmo devolve o índice que contém tal elemento”. Veja que essa frase pode ser dividida nas frases “se um elemento com chave k existe, então o algoritmo

devolve seu índice (que é um valor entre 1 e n)” e “se o algoritmo devolve um índice entre 1 e n , então é porque um elemento com chave k existe”, e que essa última frase é equivalente (por contrapositiva) a dizer que se um elemento com aquela chave não existe, então o algoritmo devolve um índice que não está entre 1 e n (no caso, -1). Formalizamos a demonstração de corretude da BUSCALINEAR no Teorema 3.2 a seguir.

Teorema 3.2

Seja $A[1..n]$ um vetor com n elementos e k uma chave de busca. Existe um elemento com chave k no vetor $A[1..n]$ se e somente se $BUSCALINEAR(A, n, k)$ devolve i , com $1 \leq i \leq n$, tal que $A[i].\text{chave} = k$.

Demonstração. Para provar o resultado, precisamos da seguinte invariante de laço.

Invariante: Laço enquanto – BUSCALINEAR

$P(t)$ = “Antes da t -ésima iteração começar, vale que $i = t$ e o vetor $A[1..i-1]$ não contém um elemento com chave k .”

Demonstração. Vamos provar que P é invariante por indução no número t de vezes que o teste da condição de parada é executado.

Caso base. Quando $t = 1$, o teste executou uma única vez. Antes da primeira iteração começar, claramente temos $i = 1$. Como $A[1..0]$ é vazio, certamente não contém um elemento com chave k . Assim, $P(1)$ é verdade e o caso base vale.

Passo. Considere então que o teste executou t vezes, com $t \geq 1$. Isso significa que a t -ésima iteração vai começar. Suponha que $P(t)$ vale e vamos mostrar que $P(t+1)$ vale. Para isso, argumentamos sobre o que está sendo feito **durante** a t -ésima iteração.

Como $P(t)$ vale, sabemos que no início da t -ésima iteração, $i = t$ e $A[1..t-1]$ não contém um elemento com chave k . Como o t -ésimo teste foi executado, pois o laço começou, ganhamos também a informação de que $A[t].\text{chave} \neq k$. Assim, ao final dessa iteração temos que $A[1..t]$ não contém um elemento com chave k , o que prova que $P(t+1)$ vale, uma vez que i é atualizado para $t+1$.

Concluimos então que P é uma invariante de laço. \diamond

Vamos agora utilizar a invariante para concluir que o algoritmo funciona corretamente.

Para isso, primeiro note que o laço termina por um dentre dois motivos⁶: (1) $i > n$ ou (2) $i \leq n$ e $A[i].\text{chave} = k$.

Vamos então começar provando que se o elemento não existe no vetor, então o algoritmo devolve -1 . Se o laço terminou porque $i > n$, então ele executou n vezes. Como P é invariante, $P(n+1)$ vale e nos diz que, ao final do laço, $A[1..n]$ não contém um elemento com chave k . Além disso, o teste da linha 4 falha e o algoritmo, portanto, devolve -1 .

Agora vamos provar que se o elemento existe no vetor, então o algoritmo devolve seu índice. A única outra possibilidade do laço ter terminado é porque $i \leq n$ e $A[i].\text{chave} = k$, ou seja, existe um elemento com chave k em A . Neste caso o teste da linha 4 é verdadeiro e o algoritmo devolve i , sendo que $1 \leq i \leq n$ ⁷. \square

Perceba que na prova anterior não fizemos nenhuma suposição sobre os valores das chaves dos elementos de 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). Talvez você mude de ideia com o próximo exemplo.

Considere o seguinte problema.

Problema 3.3: Conversão de Base

Entrada: $\langle n \rangle$, onde n é um inteiro escrito em base decimal.

Saída: Representação binária de n .

Por exemplo, se $n = 2$ a saída deve ser 1; se $n = 10$ a saída deve ser 1010; e se $n = 327$ a saída deve ser 101000111. Esse é um problema clássico que aparece nos primeiros cursos de programação de qualquer estudante.

Para resolvê-lo com um algoritmo, vamos estabelecer primeiro como vamos representar a saída. Armazenaremos a representação binária de um número decimal em um vetor B . Assim, $B[i] = 0$ ou $B[i] = 1$ para qualquer posição válida i . Além

⁶Lembre-se que nas linguagens de programação, na verificação de uma expressão lógica “ A e B ”, se A tem valor *falso*, então o valor de B nem será verificado.

⁷Veja que neste caso também temos a informação dada por $P(i)$, que indica que $A[1..i-1]$ não contém um elemento com chave k , porém ela não nos foi útil.

disso, a representação estará invertida: na posição 1 do vetor teremos o dígito menos significativo da representação binária. Por exemplo, se armazenamos o número 101000111 em um vetor dessa forma, tal vetor será igual a $B = (1, 1, 1, 0, 0, 0, 1, 0, 1)$. Escolhemos essa representação pois ela facilita a conversão do binário para o decimal: um bit que está na posição i do vetor será multiplicado ao número 2^{i-1} no somatório para gerar o número decimal. Formalmente, se $B[1..q]$ é um vetor que armazena q bits na ordem inversa da representação, então essa representação é do número $2^0 B[1] + 2^1 B[2] + \dots + 2^{q-1} B[q] = \sum_{i=1}^q 2^{i-1} B[i]$. Ademais, essa representação facilita o próprio preenchimento do vetor, pois o primeiro número que vamos obter é de fato o último da representação em binário.

O algoritmo que vamos ver formaliza a ideia clássica de conversão de decimal para binário: repetidamente divide o número por 2, sendo que o resto dessa divisão será um dos bits da representação final. Ele encontra-se no Algoritmo 3.4. Definimos o vetor B que armazenará o resultado como tendo capacidade n , uma vez que no máximo n bits serão necessários na representação do número n . Note que ao final apenas um prefixo de B será devolvido com o resultado.

Algoritmo 3.4: CONVERTEBASE(n)

```

1  seja  $B[1..n]$  um vetor  $i = 1$ 
2   $m = n$ 
3  enquanto  $m > 0$  faça
4     $B[i] =$  resto da divisão de  $m$  por 2
5     $i = i + 1$ 
6     $m = \lfloor m/2 \rfloor$ 
7  devolve  $B[1..i - 1]$ 
```

O funcionamento desse algoritmo é bem simples. Para evitar confusão, manteremos o valor da variável n e por isso apresentamos uma nova variável m , que inicialmente é igual a n . Note que, a cada iteração, o algoritmo de fato coloca um valor dentre 0 e 1 na i -ésima posição de B , pois tal valor é o resto de uma divisão por 2. Em particular, é o resto da divisão do valor atual armazenado em m por 2. Em seguida, m é atualizado para aproximadamente metade do valor atual. Assim, de fato B representa algum número inteiro. Considerando $r + 1$ como sendo o valor da variável i após o término do laço **enquanto**, basta provar que $B[1..r]$ terá a representação binária de n . Isso está formalizado no teorema a seguir.

Teorema 3.4

Seja $n \geq 0$ um inteiro qualquer. O algoritmo `CONVERTEBASE(n)` devolve um

vetor $B[1..r]$ com 0s e 1s tal que $n = \sum_{k=1}^r 2^{k-1} B[k]$.

Demonstração. Observe que no início de qualquer iteração, o valor armazenado em m é o resultado de várias divisões de n por 2. Mais precisamente, foram $i - 1$ divisões de n por 2, de forma que o número $m2^{i-1}$ é bem próximo (devido aos pisos) do valor original n . Ao mesmo tempo, o subvetor $B[1..i - 1]$ já foi preenchido com 0s e 1s, e portanto representa algum número. A invariante de laço a seguir mostra que esse número é justamente a diferença entre n e $m2^{i-1}$. Essa invariante será necessária para provar o resultado final.

Invariante: Laço enquanto – CONVERTEBASE

$P(t)$ = “Antes da t -ésima iteração começar, $i = t$ e o vetor $B[1..i-1]$ representa um inteiro z tal que $z = n - m2^{i-1}$.”

Demonstração. Vamos provar que P é invariante por indução no número t de vezes que o teste da condição de parada é executado.

Caso base. Quando $t = 1$, o teste executou uma vez. Antes da primeira iteração começar, claramente temos $i = 1$ e $m = n$. De fato, $B[1..i - 1] = B[1..0]$, que é vazio, representa o inteiro $z = 0$ tal que $0 = n - m2^0 = 0$. Assim, $P(1)$ é verdade e o caso base vale.

Passo. Considere então que o teste executou t vezes, com $t \geq 1$. Isso significa que a t -ésima iteração vai começar. Suponha que $P(t)$ vale e vamos mostrar que $P(t + 1)$ vale. Para isso, argumentamos sobre o que está sendo feito **durante** a t -ésima iteração.

Denote por m_t o valor de m no início da t -ésima iteração. Como $P(t)$ vale, sabemos que $i = t$ e que $B[1..t - 1]$ representa um inteiro z_t tal que $z_t = n - m_t2^{t-1}$. Então, $z_t = \sum_{j=1}^{t-1} 2^{j-1} B[j]$.

Veja que precisamos mostrar que, ao final dessa iteração, $B[1..t]$ vai representar um inteiro z tal que $z = n - \lfloor \frac{m_t}{2} \rfloor 2^t$, uma vez que m será atualizado para $\lfloor \frac{m_t}{2} \rfloor$ e que i será incrementado de uma unidade.

Durante a t -ésima iteração, fazemos $B[t]$ receber o resto da divisão de m_t por 2 e atualizamos m para $\lfloor \frac{m_t}{2} \rfloor$. Agora, como $P(t)$ vale, temos que $B[1..t]$ representa o inteiro $z = z_t + 2^{t-1} B[t]$. Temos então dois casos. Se m_t é par, então $B[t] = 0$ e $B[1..t]$ representa $z = z_t$. Então $z = z_t = n - m_t2^{t-1} = n - (\frac{m_t}{2})2^t = n - m2^{i-1}$ quando $i = t + 1$, o que nos diz que $P(t + 1)$ vale. Se m_t é ímpar, então $B[t] = 1$

e $B[1..t]$ representa $z = z_t + 2^{t-1}$. Então $z = z_t + 2^{t-1} = n - m_t 2^{t-1} + 2^{t-1} = n - (m_t - 1) 2^{t-1} = n - (\frac{m_t - 1}{2}) 2^t = n - \lfloor \frac{m_t}{2} \rfloor 2^t = n - m 2^{i-1}$ quando $i = t + 1$, o que nos diz que $P(t + 1)$ também vale nesse caso.

Logo, P é invariante. ◇

Agora observe que o laço termina porque $m = 0$, uma vez que m só recebe valores inteiros. Observe ainda que, como i começa em 1 e é incrementada de uma em uma unidade, se $r + 1$ é o valor da variável i ao final da execução do laço, então o laço executou r vezes (e, portanto, o teste da condição de parada executou $r + 1$ vezes). Como P é invariante, sabemos que $P(r + 1)$ vale, o que nos diz que $B[1..r]$ representa um inteiro z tal que $z = n - 0 2^r = n$, ou seja, de fato o algoritmo está correto. □

Veremos ainda, no decorrer do livro, 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. Em particular, elas são bastante utilizadas nos algoritmos de ordenação (Parte IV).

Um bom exercício é provar a corretude de BUSCABINARIA utilizando invariante de laço. A proposição $P(t) = \text{“antes da } t\text{-ésima iteração começar, } esq \leq dir \text{ e } A[1..esq-1] \text{ e } A[dir+1..n] \text{ não contêm um elemento com chave } k\text{”}$ pode ser útil.

Tempo de execução

Uma vez que implementamos um algoritmo, é simples verificar seu tempo de execução. Basta fornecer uma instância 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. Um 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 a solução do 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 números pequenos¹: operações aritméticas (soma, subtração,

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

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 (condicional, chamada a função², retorno). Observe que isso também implica que o acesso a índices de um vetor (ou matriz) é uma operação primitiva, bem como o acesso a um campo de um registro.

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**: a quantidade de dados necessária para armazenar qualquer instância do problema. Em geral, o tamanho da entrada será dado por um ou mais números *naturais*. Então se n for o tamanho de uma entrada, usaremos uma função $T(n)$ para indicar quantos passos básicos são realizados pelo algoritmo. Se n e m indicam o tamanho de uma entrada, usaremos uma função $T(n, m)$, e assim por diante.

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 não é formalmente definido porque depende muito do problema. A ideia é que ele possa refletir justamente “o que pode crescer demais” na entrada. 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 (e não o conteúdo de cada elemento ou o tamanho da chave de busca). 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 (até porque a quantidade de números é fixa em 2). Nesses casos, a quantidade de bits necessários para representar os números é o tamanho da entrada. Existem ainda problemas cujo tamanho da entrada pode precisar ser descrito por dois ou mais números. Por exemplo, no problema em que recebemos dois vetores e precisamos verificar se eles possuem elementos em comum, o tamanho da entrada será dado por dois números, um para o tamanho de cada vetor.

Como exemplo concreto de cálculo de tempo, considere novamente o Algoritmo 3.1,

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

SOMATORIO, que, dado um vetor A com n números, 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 simplificar e 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 em 1 unidade 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 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

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

Considere agora o problema de encontrar a soma dos elementos pares de um vetor. Ele é resolvido pelo Algoritmo 4.1, que não é muito 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

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

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

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

Consideramos que verificar se $A[i]$ é par contém 3 operações básicas: 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 um vetor qualquer de tamanho n , certamente vale que

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

Agora 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 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 BUSCA LINEAR, BUSCA LINEAR EM ORDEM e BUSCA BINÁRIA vistos no Capítulo 2. Lembre-se que BUSCA LINEAR resolve o problema da busca em vetores (Problema 2.1) e todos os três, BUSCA LINEAR, BUSCA LINEAR EM ORDEM e BUSCA BINÁRIA, 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 BUSCA LINEAR no Algoritmo 4.2.

Considere inicialmente que há um elemento com chave k no vetor $A[1..n]$. Assim, denote por p_k sua posição em A , isto é, a posição tal que $A[p_k].\text{chave} = k$. Note

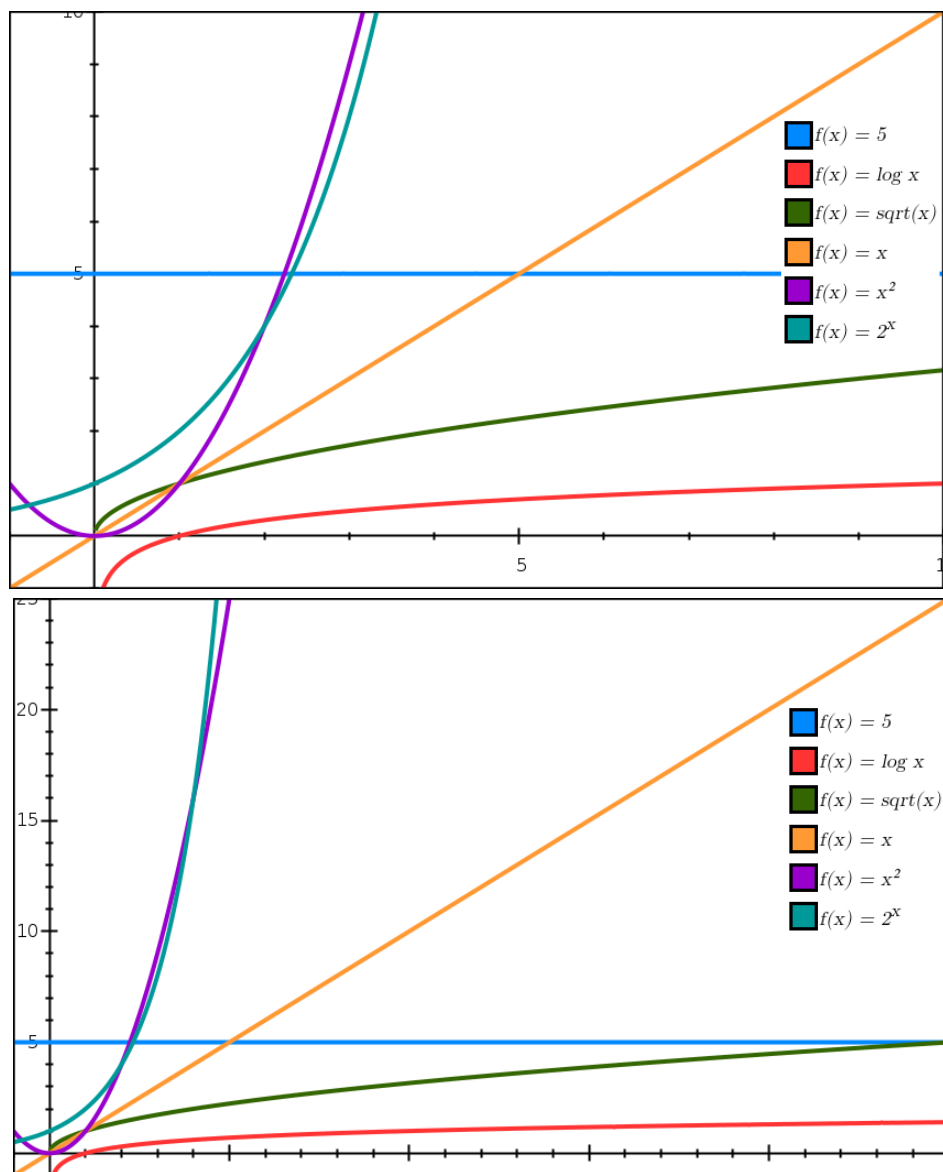


Figura 4.1: Comportamento das funções mais comuns em análise de algoritmos. Observe que na primeira figura, pode parecer que 2^x é menor do que x^2 ou que 5 é maior do que \sqrt{x} .

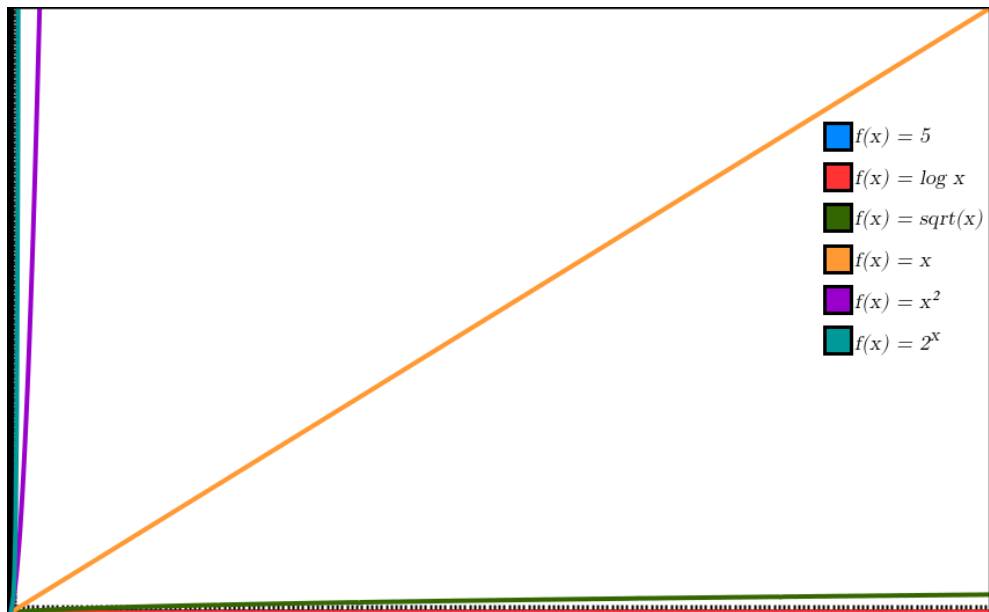
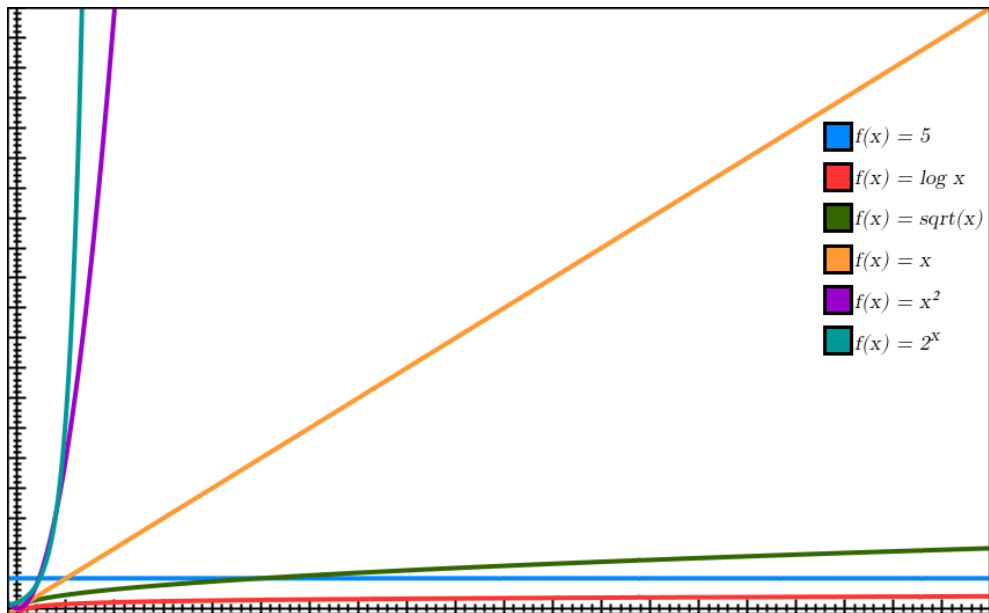


Figura 4.2: Comportamento das funções mais comuns em análise de algoritmos conforme o valor de x cresce. Note como 2^x é claramente a maior das funções observadas enquanto que 5 desaparece completamente.

Algoritmo 4.2: BUSCALINEAR(A, n, k)

```

1  $i = 1$ 
2 enquanto  $i \leq n$  e  $A[i].\text{chave} \neq k$  faça
3    $i = i + 1$ 
4 se  $i \leq n$  e  $A[i].\text{chave} == k$  então
5   devolve  $i$ 
6 devolve  $-1$ 

```

que a linha 1 é executada somente uma vez (com uma operação básica), assim como as linhas 4 (com 5 operações básicas) e 5 (com uma operação básica). Já os testes do laço **enquanto** na linha 2 são executados p_k vezes (e essa linha tem 5 operações básicas) e a linha 3 é executada $p_k - 1$ vezes (e ela tem 2 operações básicas). Assim, o tempo de execução total $T_{BLE}(n)$ de BUSCALINEAR(A, n, k) quando um elemento com chave k está em A é

$$T_{BLE}(n) = t + 5tp_k + 2t(p_k - 1) + 6t = 7tp_k + 5t. \quad (4.1)$$

O tempo de execução, portanto, depende de onde o elemento se encontra. Se o elemento está na última posição de A , então $T_{BLE}(n) = 7tn + 5t$. Se ele está na primeira posição de A , então temos $T_{BLE}(n) = 12t$.

Agora considere que não há um elemento com chave k em A . Nesse caso, a linha 1 é executada somente uma vez, assim como as linhas 4 e 6. A primeira comparação do teste do laço **enquanto** na linha 2 é executada $n + 1$ vezes e a segunda comparação do teste é executada n vezes. Por fim, a linha 3 é executada n vezes. Assim, o tempo de execução total $T_{BLN}(n)$ de BUSCALINEAR(A, n, k) quando não há um elemento com chave k em A é

$$T_{BLN}(n) = t + t(n + 1) + 4tn + 2tn + 6t = 7tn + 8t. \quad (4.2)$$

Note que como $1 \leq p_k \leq n$, temos que $12t \leq T_{BLE}(n) \leq 7tn + 5t$. E como $7tn + 5t \leq 7tn + 8t$, podemos dizer que o tempo de execução $T_{BL}(n)$ de BUSCALINEAR(A, n, k), sobre uma entrada qualquer, é tal que

$$12t \leq T_{BL}(n) \leq 7tn + 8t. \quad (4.3)$$

Para analisarmos a BUSCALINEAREMORDEM, a repetimos novamente por conveniência no Algoritmo 4.3.

Algoritmo 4.3: BUSCALINEAREMORDEM(A, n, k)

```

1  $i = 1$ 
2 enquanto  $i \leq n$  e  $A[i].chave < k$  faça
3    $i = i + 1$ 
4 se  $i \leq n$  e  $A[i].chave == k$  então
5   devolve  $i$ 
6 devolve  $-1$ 

```

Os algoritmos BUSCALINEAR e BUSCALINEAREMORDEM têm uma estrutura muito parecida, com a diferença de que nesse último nós paramos a busca quando ultrapassamos a posição onde o elemento com a chave de busca deveria estar. Assim, se o elemento com chave k existe em A e se p_k é a sua posição, então o tempo de execução total $T_{BOE}(n)$ de BUSCALINEAREMORDEM(A, n, k) é

$$T_{BOE}(n) = t + 5tp_k + 2t(p_k - 1) + 6t = 7tp_k + 5t. \quad (4.4)$$

Agora, se não existe elemento em A com chave k , então vamos definir a_k como sendo a posição do elemento com maior valor de chave x tal que $x < k$, onde fazemos $a_k = 0$ se a chave k for menor do que todas as chaves dos elementos em A . Em outras palavras, se um elemento com chave k estivesse presente em A , ele deveria estar na posição $a_k + 1$. Isso significa que o laço **enquanto** para sua execução quando $i = a_k + 1$.

Temos então dois motivos para o término do laço: i atinge o valor $n + 1$, de forma que $a_k = n$, e a primeira condição do teste falha sem que a segunda seja testada, ou $a_k + 1 \leq n$ e o segundo teste falha. No segundo caso, podemos perceber que os dois testes do laço **enquanto** são executados $a_k + 1$ vezes cada, enquanto a linha 3 é executada a_k vezes. A linha 4 será sempre executada, bem como o comando **devolve** -1 . Já no primeiro caso, quando $a_k = n$, o que muda é que o primeiro teste do laço **enquanto** é executado $n + 1$ vezes e o segundo é executado apenas n vezes. Assim, o tempo total de execução $T_{BON}(n)$ de BUSCALINEAREMORDEM(A, n, k) é

$$T_{BON}(n) = \begin{cases} t + 5t(a_k + 1) + 2ta_k + 6t = 7ta_k + 12t & \text{se } 0 \leq a_k \leq n - 1 \\ 7tn + 8t & \text{se } a_k = n. \end{cases} \quad (4.5)$$

Note que como $1 \leq p_k \leq n$, temos que $12t \leq T_{BOE}(n) \leq 7tn + 5t$. Para $0 \leq a_k \leq n - 1$, temos que $12t \leq T_{BON}(n) \leq 7tn + 5t$. Assim, podemos dizer que o tempo de execução

$T_{BO}(n)$ de $\text{BUSCALINEAREMORDEM}(A, n, k)$, sobre uma entrada qualquer, é tal que

$$12t \leq T_{BO}(n) \leq 7tn + 8t. \quad (4.6)$$

Para a busca binária, a análise é levemente diferente. Novamente por comodidade, repetimos o algoritmo BUSCABINARIA no Algoritmo 4.4.

Algoritmo 4.4: $\text{BUSCABINARIA}(A, n, k)$

```

1  esq = 1
2  dir = n
3  enquanto esq < dir faça
4    |  meio =  $\lfloor (\textit{esq} + \textit{dir})/2 \rfloor$ 
5    |  se  $k > A[\textit{meio}].\textit{chave}$  então
6    |    |  esq = meio + 1
7    |  senão
8    |    |  dir = meio
9  se  $A[\textit{esq}].\textit{chave} == k$  então
10 |  devolve esq
11 devolve -1
```

Observe que a condição de parada do laço **enquanto** na linha 3 não depende dos valores armazenados no vetor. Por isso, vamos denotar apenas por r a quantidade de vezes que o teste do laço é executado. As linhas 1 e 2 são executadas uma vez cada. A linha 4 é executada $r - 1$ vezes (e ela tem 4 operações básicas), a linha 5 é executada $r - 1$ vezes, e as linhas 6 e 8 são executadas ao todo $r - 1$ vezes também (certamente uma das duas é executada sempre). Por fim, a linha 9 sempre será executada e apenas uma dentre as linhas 10 e 11 será, a depender de haver um elemento com chave k em A ou não. Por isso, em qualquer caso, o tempo de execução $T_{BB}(n)$ de $\text{BUSCABINARIA}(A, n, k)$ é

$$T_{BB}(n) = 2t + tr + 4t(r - 1) + 3t(r - 1) + 2t(r - 1) + 3t + t = 10tr - 3t. \quad (4.7)$$

Pelo algoritmo, percebemos que o laço **enquanto** termina quando $\textit{esq} \geq \textit{dir}$. Pela invariante de laço, sabemos que ao final do laço vale que $\textit{esq} \leq \textit{dir}$. Portanto, conseguimos concluir que ao final da execução do laço temos $\textit{esq} = \textit{dir}$. Isso nos permite concluir também qual o valor r , que é a quantidade de vezes que o teste da linha 3 é executado. Note que o algoritmo sempre descarta aproximadamente metade do vetor que está sendo considerado, até chegar em um vetor de tamanho 1 (quando

$esq = dir$). Assim, o algoritmo analisa um vetor de tamanho n na primeira iteração, um vetor de tamanho $\lceil n/2 \rceil$ ou $\lfloor n/2 \rfloor$ na segunda, e assim por diante. Por isso, podemos dizer que $\lceil \lg n \rceil \leq r \leq \lfloor \lg n \rfloor + 2$, de forma que o tempo de execução $T_{BB}(n)$ de $BUSCABINARIA(A, n, k)$ é tal que

$$10t\lceil \lg n \rceil - 3t \leq T_{BB}(n) \leq 10t\lfloor \lg n \rfloor + 17t. \quad (4.8)$$

Agora temos o tempo de execução de três algoritmos que resolvem o Problema 2.2, da busca por um elemento com chave k em um vetor A cujas chaves estão em ordem crescente. Resta descobrir então qual deles é melhor.

4.1 Análise de melhor caso, pior caso e caso médio

Perceba que, na análise de tempo que fizemos para os algoritmos de busca, mesmo considerando instâncias de um mesmo tamanho n , o tempo de execução dependia de *qual* instância era dada.

Definição 4.2: Tempo de melhor caso

O tempo de execução de *melhor caso* de um algoritmo é o menor tempo de execução do algoritmo dentre os tempos de execução de todas as instâncias de um dado tamanho n .

No caso da $BUSCALINEAR$, já vimos, na Equação (4.3), que o tempo de execução de qualquer entrada está entre $12t$ e $7tn + 8t$. Por isso, dizemos que o tempo de execução $T_{BL}(n)$ da busca linear no melhor caso é $T_{BL}(n) = 12t$.

Ademais, lembre-se que o valor $12t$ foi obtido ao considerar que o elemento com chave k está no vetor e que $p_k = 1$. Por isso, conseguimos também *caracterizar* o melhor caso: ele ocorre quando o elemento com a chave de busca está na primeira posição do vetor.

No caso da $BUSCALINEAREMORDEM$, vimos, na Equação (4.6), que o tempo de execução de qualquer entrada está entre $12t$ e $7tn + 8t$. Por isso, dizemos que o tempo de execução $T_{BO}(n)$ da busca linear no melhor caso é $T_{BO}(n) = 12t$. E também conseguimos, além disso, caracterizar o melhor caso: ele ocorre quando o elemento com a chave de busca não está no vetor e tem chave menor do que todos os elementos armazenados.

Já no caso da BUSCABINARIA, vimos que o tempo de execução não varia muito, ficando entre $10t\lceil\lg n\rceil - 3t$ e $10t\lceil\lg n\rceil + 17t$. Assim, podemos dizer que no melhor caso a busca binária tem tempo de execução $T_{BB}(n) = 10t\lceil\lg n\rceil - 3t$.

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 $T_{BL}(n) = 12t$, podemos dizer que o tempo de execução de BUSCALINEAR é $T_{BL}(n) \geq 12t$. Perceba aqui a diferença no uso da igualdade e da desigualdade de acordo com as palavras que as precedem.

Uma análise mais comum, no entanto, é a 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 os tempos de execução de todas as instâncias de um dado tamanho n .

Como vimos acima, no pior caso a BUSCALINEAR tem tempo $T_{BL}(n) = 7tn + 8t$, no pior caso a BUSCALINEAREMORDEM tem tempo $T_{BO}(n) = 7tn + 8t$ e no pior caso a BUSCABINARIA tem tempo $T_{BB}(n) = 10t\lceil\lg n\rceil + 17t$.

Além disso, também conseguimos caracterizar alguns desses casos. Tanto na BUSCALINEAR quanto na BUSCALINEAREMORDEM, o pior caso ocorre quando não há elemento com chave k no vetor, o que exige que os laços **enquanto** executem o máximo de vezes que puderem.

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, como o tempo de execução $T_{BL}(n)$ de BUSCALINEAR no pior caso é $T_{BL}(n) = 7tn + 8t$, podemos dizer o tempo de execução de BUSCALINEAR sobre qualquer entrada é $T_{BL}(n) \leq 7tn + 8t$. 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 (inferior e superior, respectivamente). Eles são calculados pensando-se em como o algoritmo pode executar o mínimo e o máximo de instruções possíveis. Acontece que essas duas expressões podem ter taxas de crescimento muito diferentes. Por exemplo, considere novamente a BUSCALINEAR,

cujo tempo de execução $T_{BL}(n)$ é tal que $12t \leq T_{BL}(n) \leq 7tn + 8t$. Veja que a diferença entre esses dois limitantes é da ordem de n , de forma que quando $n = 2$ o tempo de execução pode ser algo entre $12t$ e $22t$, mas quando $n = 50$ o tempo pode ser qualquer coisa entre $12t$ e $358t$. 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 de todas as instâncias de um dado tamanho n .

Para analisar o melhor (ou pior) caso, não tivemos que analisar todos os tempos de todas as instâncias possíveis para calcular o menor (ou o maior) deles. Para analisar o caso médio não será diferente. Veja que é impossível encontrar esses tempos considerando a definição de forma literal. Para o caso médio, o que costumamos fazer é considerar algo sobre a distribuição das entradas e fazer uma análise probabilística.

Por exemplo, para os algoritmos de busca, podemos supor que a entrada tem a mesma probabilidade de ser qualquer uma das $n!$ permutações dos n elementos de A . A partir dessa suposição, observe que a probabilidade do elemento com chave k estar em uma certa posição i é $1/n$, para qualquer i (vamos assumir que existe um elemento com a chave procurada). Assim, se I_i é uma variável aleatória indicadora para o evento “ $A[i].\text{chave} = k$ ” e X é uma variável aleatória cujo valor é igual ao tempo de execução do algoritmo, então

$$X = \sum_{i=1}^n I_i(7ti + 5t),$$

onde, lembrando, $7ti + 5t$ é o tempo da busca quando a chave encontra-se no elemento da posição i . Tirando a esperança dos dois lados dessa equação, temos

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E}\left[\sum_{i=1}^n I_i(7ti + 5t)\right] = \sum_{i=1}^n \mathbb{E}[I_i](7ti + 5t) \\ &= \sum_{i=1}^n \mathbb{P}(A[i].\text{chave} = k)(7ti + 5t) = \sum_{i=1}^n \frac{1}{n}(7ti + 5t) \\ &= \frac{1}{n} \left(7t \sum_{i=1}^n i + 5t \sum_{i=1}^n 1\right) = \frac{1}{n} \left(7t \frac{n(n+1)}{2} + 5tn\right) \\ &= \frac{7t(n+1)}{2} + 5t = \frac{7tn + 17t}{2}, \end{aligned}$$

o que nos dá o tempo de execução da busca no caso médio.

Muitas vezes o tempo de execução no caso médio é quase tão ruim quanto no pior caso. No caso da busca linear, que tem tempo de execução $7tn + 8t$ no pior caso e $(7tn + 17t)/2$ no caso médio, veja que ambas são expressões da forma $an + b$, para constantes a e b , isto é, funções lineares em n . Assim, em ambos os casos temos um 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 $n \lg n$, 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 ou mesmo assumir alguma coisa sobre a distribuição das entradas. Por mais que na análise acima supomos que cada elemento poderia estar em qualquer posição com a mesma probabilidade, na prática esse tipo de coisa pode ser difícil de acontecer, ainda mais se tivermos usuários malignos que sempre tentam oferecer o pior caso para o nosso algoritmo. Uma das formas de lidar com isso pode ser criando **algoritmos aleatorizados**, que são algoritmos que em algum momento fazem operações baseadas em um resultado dado por um gerador de números aleatórios. Nesses algoritmos, a análise de tempo é similar à feita acima para o caso médio, porém eles nos permitem ter mais certeza sobre a distribuição da entrada. Com isso, usamos o termo **tempo esperado** ao invés de pior, melhor ou médio.

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* pode ser descrito por uma função que é limitada superiormente 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 essa pergunta é *depende*. Depende se temos mais informação sobre a entrada, sobre os dados que estão sendo recebidos, ou mesmo se sabemos em que aplicações precisamos desses 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, os três algoritmos que temos para resolver o problema da busca em vetor ordenado são eficientes, porém fazemos uma escolha pela 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 podem e 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 $n < 5$, por exemplo, a BUSCALINEAR executa menos operações do que a BUSCABINARIA).

Notação assintótica

Uma abstração que ajuda bastante na análise do tempo de execução de algoritmos é o estudo da **taxa de crescimento** de funções. É isso que nos permite comparar os tempos de execução independentemente da plataforma utilizada, da linguagem, etc¹. Estamos particularmente interessados no que acontece com as funções quando n tende a infinito, o que chamamos de **análise assintótica**.

Assim, 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, nós não queremos estar preocupados com os valores de a , b ou c , mas sim com o fato de que a função é quadrática. Por isso, o termo que realmente importa em $f(n)$ para grandes valores de n é n^2 .

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 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, sempre teremos $f(n) \leq g(n)$. Também é possível observar que $f(n) \leq 475g(n)$ para qualquer $n \geq 1$, uma vez que $f(n) = 475n + 34 \leq 475n^2 + 1425 = 475g(n)$. Essas constantes que multiplicam as funções e os valores iniciais de n ficam escondidos na **notação assintótica**.

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

5.1 Notações O , Ω e Θ

As notações O e Ω servem para limitar funções superiormente e inferiormente, respectivamente.

Definição 5.1: Notações O e Ω

Sejam n um inteiro positivo, e $f(n)$ 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.

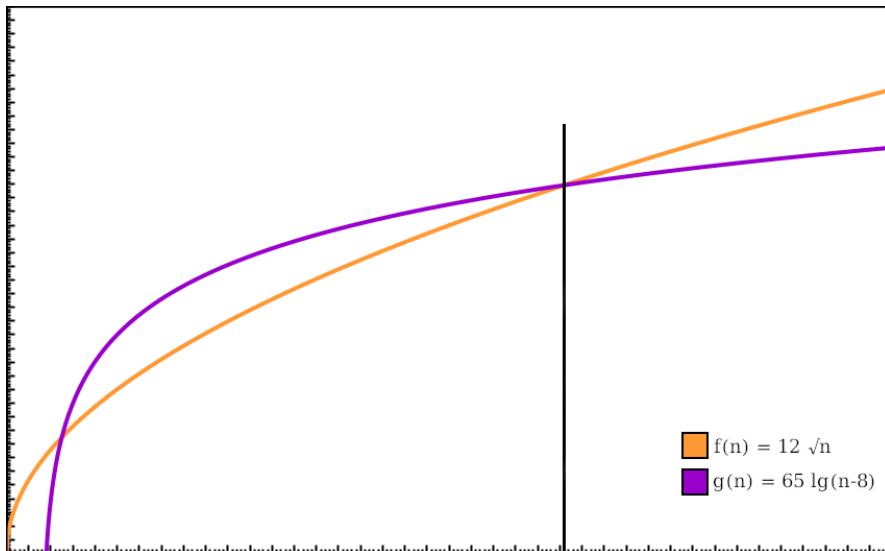


Figura 5.1: A função $g(n) = 65 \lg(n-8)$ é $O(f(n))$, e isso pode ser visualizado no gráfico, pois a partir de um certo ponto (a barra vertical) temos $f(n)$ sempre acima de $g(n)$. Também veja que $f(n) = 12\sqrt{n}$ é $\Omega(g(n))$, pois a partir de um certo ponto (no caso, o mesmo) temos $g(n)$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

Considere as funções $f(n) = 12\sqrt{n}$ e $g(n) = 65 \lg(n-8)$ da Figura 5.1. Intuitivamente,

sabemos que a função $\lg n$ cresce mais devagar do que a função \sqrt{n} (a figura inclusive 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

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

onde só podemos usar o fato que $\lg n \leq \sqrt{n}$ 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 $65 \lg(n-8) \geq c \cdot 12\sqrt{n}$ sempre que $n \geq n_0$. Se a expressão anterior vale, também vale que $c \leq \frac{65 \lg(n-8)}{12\sqrt{n}}$. Acontece que $\frac{65 \lg(n-8)}{12\sqrt{n}}$ 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. 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 $n \geq n_0$.

A partir das definições, é possível provar que $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$ se e somente se $f(n) = \Theta(g(n))$.

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

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

vale sempre que $\lg n > 0$, e por isso basta tomar $n \geq 2$. Assim, tomando $C = 2$ e

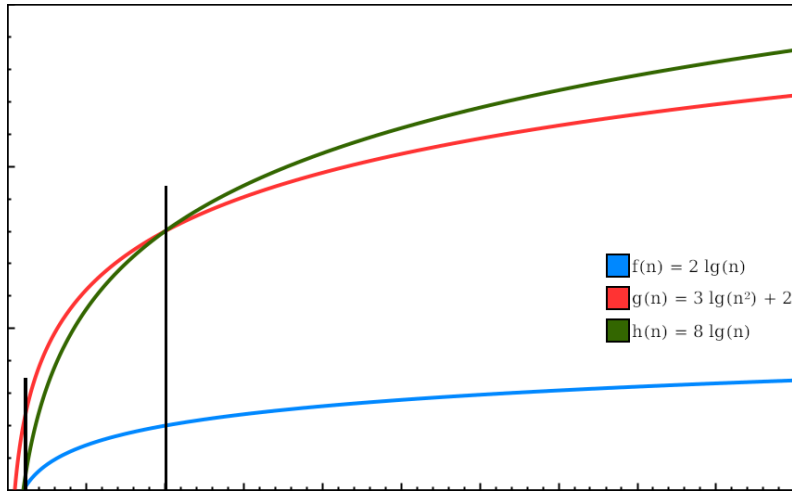


Figura 5.2: A função $g(n) = 3 \lg n^2 + 2$ é $\Theta(\lg n)$, e isso pode ser visualizado no gráfico, pois a partir de um certo ponto (a barra vertical mais à direita), temos $g(n)$ sempre entre $2 \lg n$ e $8 \lg n$.

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

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

que vale para qualquer valor $n \geq 1$. Então, tomando $c = 3$ e $n_0 = 1$, podemos afirmar que, por definição, $g(n)$ é $\Omega(f(n))$. Também podemos ver que $f(n)$ é $\Omega(\lg n)$, o que implica em $g(n)$ ser $\Omega(\lg n)$. Assim, acabamos de mostrar que $g(n)$ é $\Theta(\lg n)$.

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 apresentar mais exemplos para entender melhor o formalismo das notações. É importante, no entanto, começar a entendê-las também de forma intuitiva. Por exemplo, no Fato 5.3 a seguir, $10n^2 + 5n + 2$ é intuitivamente $\Theta(n^2)$, uma vez que n^2 é o termo dominante.

Fato 5.3

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

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

então 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 $f(n) = O(n^2)$.

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

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

então 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 $f(n) = O(n^2)$ 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 escolhidos 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 $f(n) = 5 \lg n + \sqrt{n}$, então $f(n) = O(\sqrt{n})$.

Demonstração. Comece percebendo que $f(n) = O(n)$, pois sabemos que $\lg n$ 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 $5 \lg n + \sqrt{n} \leq C\sqrt{n}$. Logo, queremos que

$$C \geq \frac{5 \lg n}{\sqrt{n}} + 1. \quad (5.1)$$

Mas nesse caso precisamos ter cuidado ao escolher n_0 , pois com $n_0 = 1$, temos $5(\lg 1)/\sqrt{1} + 1 = 1$, 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 $5(\lg 2)/\sqrt{2} + 1 > 1$.

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

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

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

$$5 \lg n + \sqrt{n} \leq 5\sqrt{n} + \sqrt{n} = 6\sqrt{n}, \quad (5.2)$$

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 $f(n) = O(\sqrt{n})$. \square

Uma terceira estratégia ainda pode ser vista no próximo exemplo.

Fato 5.6

Se $f(n) = 5 \lg n + \sqrt{n}$, 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 $5 \lg n + \sqrt{n} \leq C\sqrt{n}$. Logo, queremos que $C \geq \frac{5 \lg n}{\sqrt{n}} + 1$.

Note que

$$\lim_{n \rightarrow \infty} \left(\frac{5 \lg n}{\sqrt{n}} + 1 \right) = \lim_{n \rightarrow \infty} \left(\frac{5 \lg n}{\sqrt{n}} \right) + \lim_{n \rightarrow \infty} 1 \quad (5.3)$$

$$= \lim_{n \rightarrow \infty} \left(\frac{5 \frac{1}{n \ln 2}}{\frac{1}{2\sqrt{n}}} \right) + 1 \quad (5.4)$$

$$= \lim_{n \rightarrow \infty} \left(\frac{10/\ln 2}{\sqrt{n}} \right) + 1 = 0 + 1 = 1, \quad (5.5)$$

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

Perceba que podem existir diversas possibilidades de escolha para n_0 e C mas, 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 $5 \lg n + \sqrt{n} = O(\sqrt{n})$. Outra escolha possível seria $n_0 = 1$ e $C = 11$. Não é difícil mostrar que $5 \lg n + \sqrt{n} = \Omega(\sqrt{n})$.

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

Por fim, vamos utilizar a definição da notação assintótica para mostrar que $7n^2 \neq O(n)$. Veja que isso também é intuitivamente verdade: $7n^2$ tem como termo dominante o n^2 , que é muito maior do que n conforme n cresce e, portanto, não poderia ser limitada superiormente por tal função.

Fato 5.7

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

Demonstração. Suponha, para fins de 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 nessa equação, 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 $n \geq n_0$. \square

5.1.1 Funções de várias variáveis

As notações O , Ω e Θ podem ser definidas também para funções que dependam de mais de uma variável.

Definição 5.8: Notações O e Ω

Sejam n_1, \dots, n_k inteiros positivos, e $f(n_1, \dots, n_k)$ e $g(n_1, \dots, n_k)$ funções positivas. Dizemos que

- $f(n_1, \dots, n_k) = O(g(n_1, \dots, n_k))$ se existem constantes positivas C e n_0 tais que, para algum i , $f(n_1, \dots, n_k) \leq Cg(n_1, \dots, n_k)$ para todo $n_i \geq n_0$;
- $f(n_1, \dots, n_k) = \Omega(g(n_1, \dots, n_k))$ se existem constantes positivas c e n_0 tais que, para algum i , $cg(n_1, \dots, n_k) \leq f(n_1, \dots, n_k)$ para todo $n_i \geq n_0$.

Definição 5.9: Notação Θ

Sejam n_1, \dots, n_k inteiros positivos e sejam $f(n_1, \dots, n_k)$ e $g(n_1, \dots, n_k)$ funções positivas. Dizemos que $f(n_1, \dots, n_k) = \Theta(g(n_1, \dots, n_k))$ se existem constantes positivas c , C e n_0 tais que, para algum i , $cg(n_1, \dots, n_k) \leq f(n_1, \dots, n_k) \leq Cg(n_1, \dots, n_k)$ para todo $n_i \geq n_0$.

Como exemplo, podemos dizer que $f(n, m) = 45n^2 + 2n\frac{m}{3} + 678m$ é $O(n^2 + nm)$, pois

$$45n^2 + 2n\frac{m}{3} + 678m \leq 678n^2 + 678nm + 678nm < 1356n^2 + 1356nm = 1356(n^2 + nm),$$

onde a primeira desigualdade vale porque $45 < 678$, $\frac{2}{3} < 678$ e n é positivo, e a segunda desigualdade vale porque $678 < 1356$.

Observe que na expressão $45n^2 + 2n\frac{m}{3} + 678m$ o termo $2n\frac{m}{3}$ domina o termo $678m$ e por isso um termo com apenas m não aparecerá na notação O . Além disso, não se sabe qual dentre n^2 ou nm é maior, uma vez que ambos n e m crescem indefinidamente. Essa é a intuição de por que tentamos mostrar que $f(n, m)$ é $O(n^2 + nm)$.

Formalmente, $f(n, m)$ não é $O(n^2)$ pois, se fosse, deveriam existir constantes c e n_0 tais que $45n^2 + 2n\frac{m}{3} + 678m \leq cn^2$ sempre que $n > n_0$ ou $m > n_0$. Ou então, deveria valer que $c \geq 45 + \frac{2m}{3n} + \frac{678m}{n^2}$ sempre que $n > n_0$ ou $m > n_0$. Mas note que, por mais que n e m ambos cresçam indefinidamente, eles crescem independentemente (isto é, não sabemos se m é sempre maior ou menor do que n). Assim, a expressão $45 + \frac{2m}{3n} + \frac{678m}{n^2}$ não consegue ser limitada superiormente por uma constante sempre.

5.1.2 Relações entre as notações O , Ω e Θ

O teorema a seguir descreve propriedades importantes acerca das relações entre as três notações assintóticas.

Teorema 5.10: Propriedades de notações assintóticas

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

$$cg(n) \leq f(n) \leq Cg(n) \quad (5.6)$$

para todo $n \geq n_0$. Assim, analisando as desigualdades em (5.6), concluímos que $(\frac{1}{C})f(n) \leq g(n) \leq (\frac{1}{c})f(n)$ 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, pelos itens 1 e 6 do Teorema 5.10, se uma função $f(n)$ é uma soma de funções logarítmicas, exponenciais e polinomiais 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

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

então conseguimos provar 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*.

Definição 5.11: Notações o e ω

Sejam n um inteiro positivo e $f(n)$ 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 $\lim_{n \rightarrow \infty} (f(n)/g(n)) = 0$. 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 .

Fato 5.12

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

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

$$10n + 3 \lg n < 13n < cn^2.$$

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 $10n + 3 \lg n = o(n^{1+\varepsilon})$ 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. 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, ao final dele, de que você entendeu *por que* podemos dizer essas frases.

Analisamos, na Seção 4.1, os tempos no melhor caso, pior caso e caso médio da BUSCALINEAR. No melhor caso, ela leva tempo $T_{BL}(n) = 12t$. Veja que a expressão $12t$ 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 $T_{BL}(n) = 7tn + 8t$. Veja que $7tn + 8t \leq 7tn + 8tn = 15tn$ para qualquer $n \geq 1$. Assim, tomando $C = 15t$ e $n_0 = 1$, podemos concluir que $7tn + 8t = O(n)$. Também é verdade que

Algoritmo 6.1: BUSCALINEAR(A, n, k)

```

1  $i = 1$ 
2 enquanto  $i \leq n$  e  $A[i].chave \neq k$  faça
3    $i = i + 1$ 
4 se  $i \leq n$  e  $A[i].chave == k$  então
5    $\downarrow$  devolve  $i$ 
6 devolve  $-1$ 

```

$7tn + 8t \leq 7tn + 8tn \leq 7tn^3 + 8tn^3 = 15tn^3$ sempre que $n \geq 1$, de onde podemos concluir que $7tn + 8t$ é $O(n^3)$. Podemos ainda concluir que $7tn + 8t$ é $O(n^5)$, $O(n^{10})$, $O(2^n)$ ou $O(n^n)$. Acontece que o limitante $O(n)$ é mais justo, pois $7tn + 8t$ é também $\Omega(n)$. Isso por sua vez vale porque $7tn + 8t \geq 7tn$ para qualquer $n \geq 1$, e podemos tomar $c = 7t$ e $n_0 = 1$ para chegar a essa conclusão. Também é verdade que $7tn + 8t$ é $\Omega(1)$, $\Omega(\lg n)$ e $\Omega(\sqrt{n})$. Sabendo que $7tn + 8t$ é $O(n)$ e $\Omega(n)$, podemos concluir que também é $\Theta(n)$.

Agora veja que $7tn + 8t$ não é $\Omega(n^2)$, por exemplo. Se isso fosse verdade, então pela definição deveriam existir constantes c e n_0 tais que $7tn + 8t \geq cn^2$ sempre que $n \geq n_0$. Mas veja que isso é equivalente a dizer que deveria valer que $\frac{7t}{n} + \frac{8t}{n^2} \geq c$, o que é impossível, pois c é uma constante positiva (estritamente maior do que 0) e $\frac{7t}{n} + \frac{8t}{n^2}$ 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 $7tn + 8t$ não é $O(\sqrt{n})$, porque se fosse, haveria constantes C e n_0 tais que $7tn + 8t \leq C\sqrt{n}$ sempre que $n \geq n_0$. Mas isso é equivalente a dizer que $7t\sqrt{n} + \frac{8t}{\sqrt{n}} \leq C$ sempre que $n \geq n_0$, o que é impossível, pois C é uma constante positiva e $7t\sqrt{n} + \frac{8t}{\sqrt{n}}$ 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 o seguinte sobre o algoritmo da busca linear:

- o tempo da BUSCALINEAR é $\Omega(n)$ no pior caso,
- o tempo da BUSCALINEAR é $\Omega(1)$ no pior caso,
- o tempo da BUSCALINEAR é $O(n)$ no pior caso,
- o tempo da BUSCALINEAR é $O(n^3)$ no pior caso,

- o tempo da BUSCALINEAR é $O(n^5)$ no pior caso,
- o tempo da BUSCALINEAR é $\Theta(n)$ no pior caso,
- o tempo da BUSCALINEAR é $O(1)$ no melhor caso,
- o tempo da BUSCALINEAR é $O(n)$ no melhor caso,
- o tempo da BUSCALINEAR é $O(n^2)$ no melhor caso,
- o tempo da BUSCALINEAR é $\Omega(1)$ no melhor caso,
- o tempo da BUSCALINEAR é $\Theta(1)$ no melhor caso.

Observe que usando a notação Θ só é possível dizer que ela leva 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, bastando explicitar a qual caso estamos nos referindo. Quando não explicitarmos o caso, estaremos nos referindo à execução sobre *qualquer instância*, e é aí que precisamos tomar muito cuidado. Por exemplo, não podemos dizer “o tempo da BUSCALINEAR é $\Theta(n)$ ”, pois isso significaria que “o tempo de execução da BUSCALINEAR sobre qualquer entrada de tamanho n é uma função $f(n)$ tal que $f(n) = \Theta(n)$ ”. Mas isso é claramente falso, já que há entradas que executam em tempo constante, por exemplo.

Agora, é importante perceber que esse erro não ocorre sem fundamento. 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 qualquer entrada, e apenas sobre as entradas que dão o pior caso. De forma equivalente, dizer que o melhor caso da busca leva tempo $O(\sqrt{n})$ é verdade, mas não é tão acurado e não nos diz muito sobre qualquer entrada. 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.

Como um exemplo da discussão acima, voltemos à BUSCALINEAR. Podemos dizer que “o tempo da BUSCALINEAR é $O(n)$ ” porque essa frase, sem menção de caso específico, diz algo sobre todas as instâncias e é verdade que *o tempo de qualquer instância é menor ou igual ao tempo de execução do pior caso*. Como o tempo do pior caso da busca é $\Theta(n)$ ¹, o resultado segue.

Podemos dizer que “o tempo da BUSCALINEAR é $\Omega(1)$ ” porque essa frase, sem menção de caso, diz algo sobre todas as instâncias e é verdade que *o tempo de qualquer instância é maior ou igual ao tempo de execução do melhor caso*. Como o tempo do melhor caso da busca é $\Theta(1)$ ², o resultado segue.

Antes de prosseguir com mais exemplos, perceba que não precisávamos ter encontrado explicitamente as expressões $7tn + 8t$ e $12t$. 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,

¹Mais especificamente, o tempo do pior caso é $T(n) = 7tn + 8t$ e $7tn + 8t = \Theta(n)$.

²Mais especificamente, o tempo do melhor caso é $T(n) = 12t$ e $12t = \Theta(n)$.

e pode ter valor *no máximo* $n + 1$. Como podemos ter bem menos valores de i , só conseguimos dizer que são $O(n)$ execuções. 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 $\Theta(\lg n)$. 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 $\lg n$, 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.

Problema 6.1: Soma em Vetor

Entrada: $\langle A, n, k \rangle$, onde $A[1..n]$ é um vetor de tamanho n e k é um inteiro.

Saída: Par de posições i e j , $i \neq j$, tais que $A[i] + A[j] = k$, caso existam, ou o par $-1, -1$, caso contrário.

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 uma possível resposta é o par 2, 5, pois $A[2] = 6$ e $A[5] = 4$ somam 10. Veja que $A[6] = 9$ e $A[8] = 1$ também somam 10 (o problema requer algum par apenas). Se $k = 2$, a resposta é o par $-1, -1$.

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 nesse caso poderemos reutilizar algoritmos que já temos.

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 outro³. Uma formalização dessa ideia encontra-se no Algoritmo 6.2. Considere que a BUSCALINEAR foi alterada para comparar o próprio elemento armazenado em uma posição ao invés de um campo chave.

Algoritmo 6.2: SOMAK(A, n, k)

```

1 para  $j = 1$  até  $n$ , incrementando faça
2    $i = \text{BUSCALINEAR}(A, n, k - A[j])$ 
3   se  $i \neq -1$  e  $i \neq j$  então
4      $\quad$  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. Não se esqueça de instâncias específicas. Por exemplo, 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. Depois disso, um bom exercício é verificar a corretude dele usando uma invariante de laço. Note que esta invariante pode considerar que a chamada à BUSCALINEAR funciona corretamente, pois já provamos isso.

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**. 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

³O conceito de redução será formalizado no Capítulo 28.

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 $\lg n$, n , $n \lg n$, $n^{3/2}$, para citar alguns.

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 outro algoritmo para o problema. Vamos então tentar resolver o problema diretamente, sem ajuda de algoritmos que já foram criados para outros problemas. 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 $n - 1$, 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 criar um algoritmo com um tempo de execução de pior caso ainda melhor, sendo $O(n \lg n)$. Com tudo que já foi visto até este capítulo, você já deve ser capaz de pensar nessa solução melhor.

Recursividade

Você quis dizer: *recursividade*.

Google

Imagine a seguinte situação. Você está tentando resolver um problema X e começou a escrever um algoritmo A para ele. Em um certo momento você percebe que conseguiria resolver a instância de entrada se soubesse também resolver uma outra instância do mesmo problema X . Mas então você pode então usar o próprio algoritmo A para resolver essa segunda instância! Esse algoritmo acabou de se tornar *recursivo*.

Antes de continuarmos, é importante lembrar 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.

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 que têm a seguinte característica: toda instância contém uma ou mais instâncias menores **do mesmo problema** (estrutura recursiva). Porém não basta apenas encontrar uma (ou mais de uma) instância menor. Deve-se encontrar uma (ou mais de uma) instância menor 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 com chave k em um vetor A que contém n elementos e suponha

que n é suficientemente grande. Para simplificar, vamos nos referir a esse problema apenas como “a chave k 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 um elemento com chave k poderia estar. Note ainda que qualquer subvetor de A é também um vetor e, portanto, uma entrada válida para o problema. Por exemplo, “a chave k 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 “a chave k está em $A[n-10..n]$?”, ou “a chave k está em $A[\frac{n}{4}..\frac{2n}{3}]$?”. Assim, se nós temos um algoritmo que resolve o problema de buscar um elemento com chave k 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 “a chave k está em $A[1..n]$?”, e resolver esses subproblemas não parece ajudar muito nessa tarefa. O subproblema “a chave k está em $A[1..n-1]$?”, por outro lado, é mais útil, porque se sabemos resolvê-lo, então podemos combinar sua resposta com a resposta para o teste “a chave k é igual a $A[n].\text{chave}$?” e assim resolver nosso problema original.

Agora, como procurar pela chave k 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 o elemento com chave k nesse vetor menor, comparamos k com a chave do 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: não pode haver elemento com chave k 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: $\text{BUSCALINEARRECURSIVA}(A, n, k)$

```

1 se  $n == 0$  então
2   └ devolve  $-1$ 
3 se  $A[n].\text{chave} == k$  então
4   └ devolve  $n$ 
5 devolve  $\text{BUSCALINEARRECURSIVA}(A, n-1, k)$ 
```

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 um elemento com chave k pertence a A para *qualquer* vetor A e valor k ? Se nosso vetor é vazio, note que a chamada `BUSCALINEARRECURSIVA(A , 0, k)` funciona corretamente, pois devolve -1 . Se nosso vetor tem um elemento, quando fazemos a chamada `BUSCALINEARRECURSIVA(A , 1, k)`, comparamos k com $A[1].\text{chave}$. Caso sejam iguais, a busca para e devolve 1; caso sejam diferentes, a chamada `BUSCALINEARRECURSIVA(A , 0, k)` é feita e acabamos de verificar que esta funciona corretamente. Como o elemento com chave k só pode estar ou não na posição 1 de um vetor de tamanho 1, concluímos então que `BUSCALINEARRECURSIVA(A , 1, k)` funciona corretamente. Se nosso vetor tem dois elementos, quando fazemos uma chamada `BUSCALINEARRECURSIVA(A , 2, k)`, comparamos k com $A[2].\text{chave}$. Caso sejam iguais, a busca para e devolve 2; caso sejam diferentes, a chamada `BUSCALINEARRECURSIVA(A , 1, k)` é feita e já verificamos que esta funciona. Como o elemento de chave k só pode estar na posição 2 ou então no subvetor de tamanho 1 que termina antes desta posição, concluímos que `BUSCALINEARRECURSIVA(A , 2, k)` funciona corretamente. 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 do

elemento com chave k no vetor se ele existir, ou devolve -1 ". Quando $n = 0$, o algoritmo devolve -1 e, de fato, um elemento com chave k não existe no vetor, que é vazio.

Considere então um $n > 0$ e suponha que para qualquer vetor de tamanho m , com $0 \leq m < n$, o algoritmo devolve a posição do elemento com chave k em tal vetor se ele existir, ou devolve -1 . Na chamada para um vetor de tamanho n , o algoritmo começa comparando k com $A[n]$. **chave**. Se forem iguais, o algoritmo devolve n , que é a posição do elemento com a chave k em A . Se não forem, o algoritmo faz uma chamada a `BUSCA LINEAR RECURSIVA`($A, n - 1, k$). Por hipótese, porque $n - 1 < n$, essa chamada corretamente devolve a posição do elemento de chave k 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, $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 2 \cdot 1$, e $0!$ é definido como sendo 1.

A seguir, considere um n suficientemente grande. Note que dentro da solução de $n!$ temos a solução de problemas menores. Por exemplo, a solução para $3!$ está contida ali, $3 \cdot 2 \cdot 1$, ou a solução para $(\frac{n}{2})!$. Porém, a solução para $(n - 1)!$ também está contida ali, e este subproblema, se resolvido, é bastante útil, uma vez que 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.

Por exemplo, ao chamar `FATORIAL(3)`, o algoritmo vai falhar no teste da linha 1 e vai executar a linha 3, fazendo " $3 \times \text{FATORIAL}(2)$ ". Antes de poder devolver esse valor, é necessário calcular `FATORIAL(2)`. Nesse ponto, o computador salva o estado atual na pilha de execução e faz a chamada `FATORIAL(2)`, que vai falhar no teste

Algoritmo 7.2: FATORIAL(n)

```

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

```

da linha 1 novamente e vai executar a linha 3 para devolver “ $2 \times \text{FATORIAL}(1)$ ”. Novamente, o estado atual é salvo na pilha de execução e a chamada FATORIAL(1) é realizada. O teste da linha 2 falha novamente e a linha 3 será executada, para devolver “ $1 \times \text{FATORIAL}(0)$ ”, 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 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 $n!$. Primeiro note que se $n = 0$, o algoritmo devolve 1. De fato, $0! = 1$, então nesse caso ele funciona corretamente.

Agora considere um $n > 0$. Vamos supor que FATORIAL(k) devolve $k!$ sempre 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 $n - 1 < n$, pois $n > 0$, então por hipótese sabemos que FATORIAL($n - 1$) devolve $(n - 1)!$. Como $n \cdot (n - 1)! = n!$, concluímos que a chamada atual funciona corretamente.

7.3 Potência de um número

Seja n um número inteiro não negativo. 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

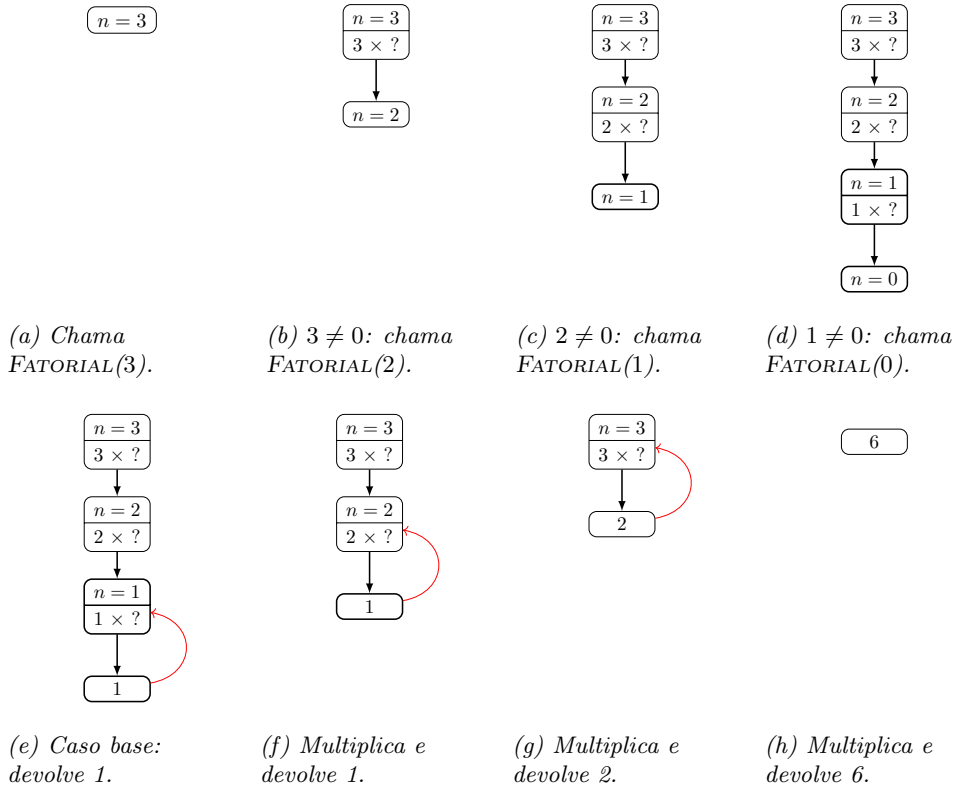


Figura 7.1: Exemplo de execução de $FATORIAL(3)$ (Algoritmo 7.2). Cada nó é uma chamada ao algoritmo. Setas vermelhas indicam o retorno de uma função.

$x^0 = 1$:

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

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 correte



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

por indução segue de forma simples e é deixada como exercício. 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  $x \times POTENCIAV1(x, n - 1)$ 

```

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. Veja essa ideia formalizada no Algoritmo 7.4. A verificação de sua corretude por indução também segue de forma simples. A Figura 7.4 mostra um exemplo simples de execução enquanto a Figura 7.5 mostra a árvore de recursão completa.

Algoritmo 7.4: $POTENCIAV2(x, n)$

```

1 se  $n == 0$  então
2   | devolve 1
3 se  $n$  é par então
4   | devolve  $POTENCIAV2(x, n/2) \times POTENCIAV2(x, n/2)$ 
5 senão
6   | devolve  $x \times POTENCIAV2(x, (n - 1)/2) \times POTENCIAV2(x, (n - 1)/2)$ 

```

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 a chamada $POTENCIAV2(x, n/2)$ sempre devolve o mesmo valor, de modo que não faz

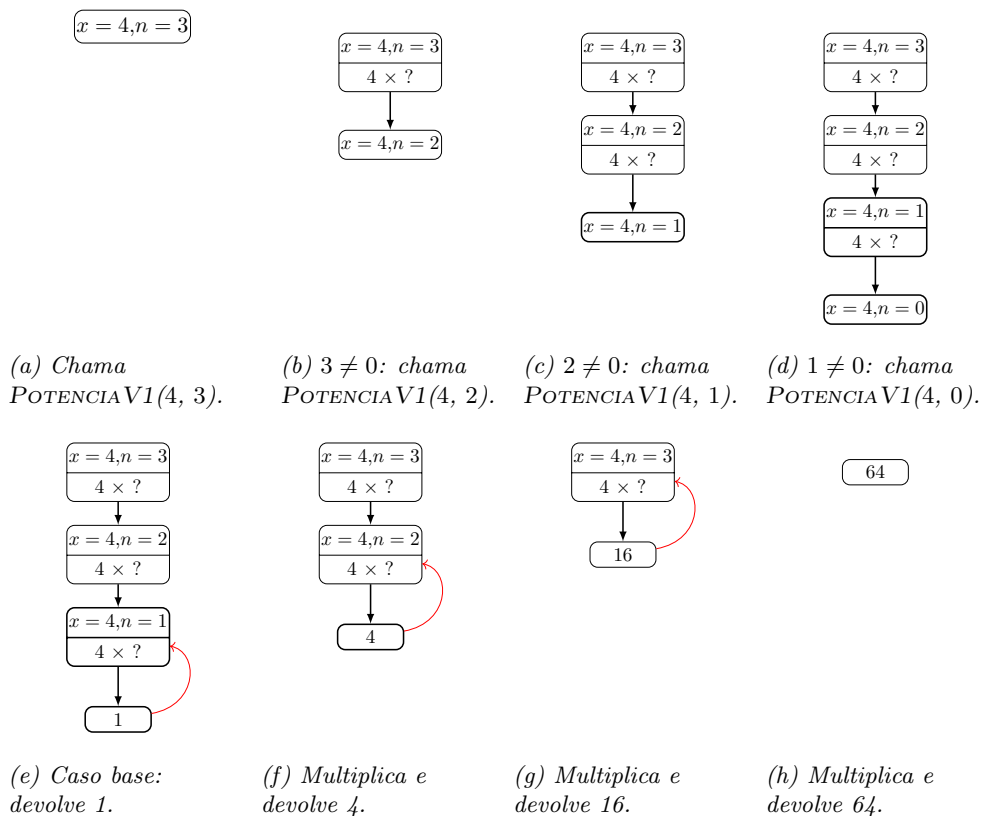


Figura 7.3: Exemplo de execução de $POTENCIAV1(4, 3)$ (Algoritmo 7.3). Cada nó é uma chamada ao algoritmo. Setas vermelhas indicam o retorno de uma função.

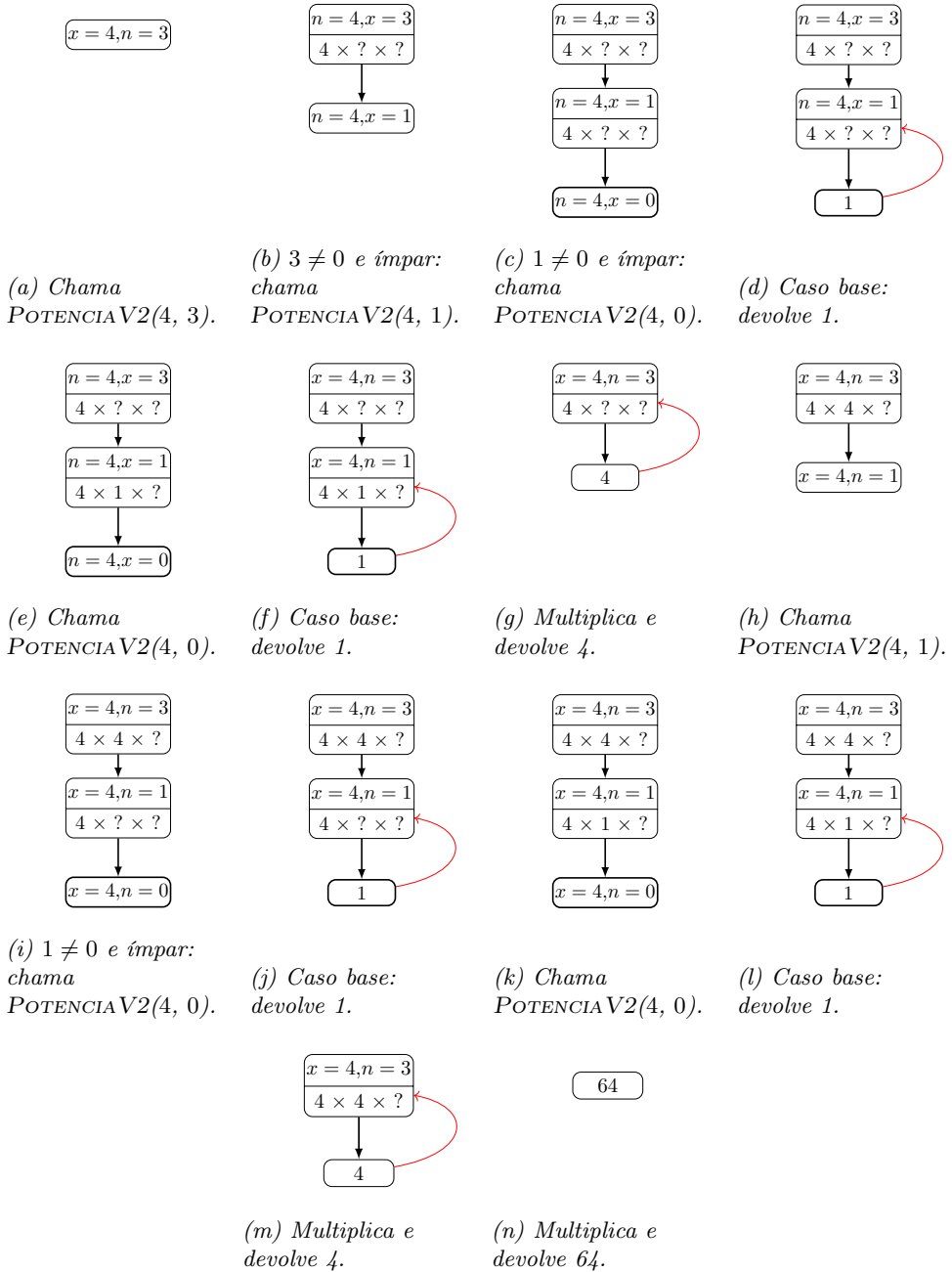


Figura 7.4: Exemplo de execução de $POTENCIAV2(4, 3)$ (Algoritmo 7.4). Cada nó é uma chamada ao algoritmo. Setas vermelhas indicam o retorno de uma função.

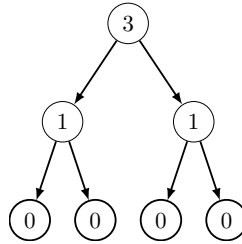


Figura 7.5: Árvore de recursão completa de $POTENCIAV2(4,3)$. Cada nó é rotulado com o valor de n correspondente.

sentido realizá-la duas vezes. Uma versão mais eficiente do Algoritmo 7.4 é descrita no Algoritmo 7.5.

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  $aux \times aux$ 
6 senão
7   └  $aux = POTENCIAV3(x, (n-1)/2)$ 
8   └ devolve  $x \times aux \times aux$ 

```

Com esses dois algoritmos para resolver o problema da potência, qual deles é melhor? Qual é o mais eficiente? Agora que temos recursão envolvida, precisamos tomar alguns cuidados para calcular o tempo de execução desses algoritmos. Além disso, observe que o tamanho da entrada é m , sendo que $m = \lg n$, onde n é o valor dado na entrada (lembre-se que o tamanho da entrada quando a entrada é um número é a quantidade de bits necessários para representá-lo). No caso de $POTENCIAV1$, podemos ver que o valor de n decresce de uma unidade a cada chamada, e que só não fazemos uma chamada recursiva quando $n = 0$. Assim, existem n chamadas recursivas. O tempo gasto dentro de uma única chamada é constante (alguns testes, multiplicações e atribuições). Assim, $POTENCIAV1(x, n)$ leva tempo $\Theta(n) = \Theta(2^m)$, ou seja, é *exponencial no tamanho da entrada*, que é m . 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 $\lg n$ chamadas recursivas. Também temos que o tempo gasto dentro de uma única chamada é constante, de forma que $POTENCIAV3(x, n)$ leva tempo $\Theta(\lg n) = \Theta(m)$, ou seja, é *linear no tamanho da entrada*, que é m . 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 por um elemento com chave k em um vetor A com n elementos cujas chaves estão em ordem crescente. Se $A[\lfloor n/2 \rfloor].\text{chave} = k$, então a busca está encerrada. Caso contrário, se $k < A[\lfloor n/2 \rfloor].\text{chave}$, então basta verificar se o vetor $A[1..\lfloor n/2 \rfloor - 1]$ contém um elemento com chave k . Mas isso é exatamente o mesmo problema (procurar um elemento com chave k em um vetor), só que menor! Então isso pode ser feito recursivamente. Se $k > A[\lfloor n/2 \rfloor].\text{chave}$, verificamos 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, k)`. 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.

Algoritmo 7.6: `BUSCABINARIARECURSIVA(A, esq, dir, k)`

```

1 se esq > dir então
2   └─ devolve -1
3 meio =  $\lfloor (\textit{esq} + \textit{dir}) / 2 \rfloor$ 
4 se  $A[\textit{meio}].\text{chave} == k$  então
5   └─ devolve meio
6 senão se  $k < A[\textit{meio}].\text{chave}$  então
7   └─ devolve BUSCABINARIARECURSIVA(A, esq, meio - 1, k)
8 senão
9   └─ devolve BUSCABINARIARECURSIVA(A, meio + 1, dir, k)
```

Para mostrar que `BUSCABINARIARECURSIVA` está correto, podemos fazer uma prova por indução no tamanho do vetor. Queremos mostrar que se $n = \textit{dir} - \textit{esq} + 1$, então `BUSCABINARIARECURSIVA(A, esq, dir, k)` devolve -1 se não há elemento com chave k em $A[\textit{esq}..\textit{dir}]$ e devolve i , com $A[i].\text{chave} = k$, se há tal elemento, para qualquer vetor e valor de n .

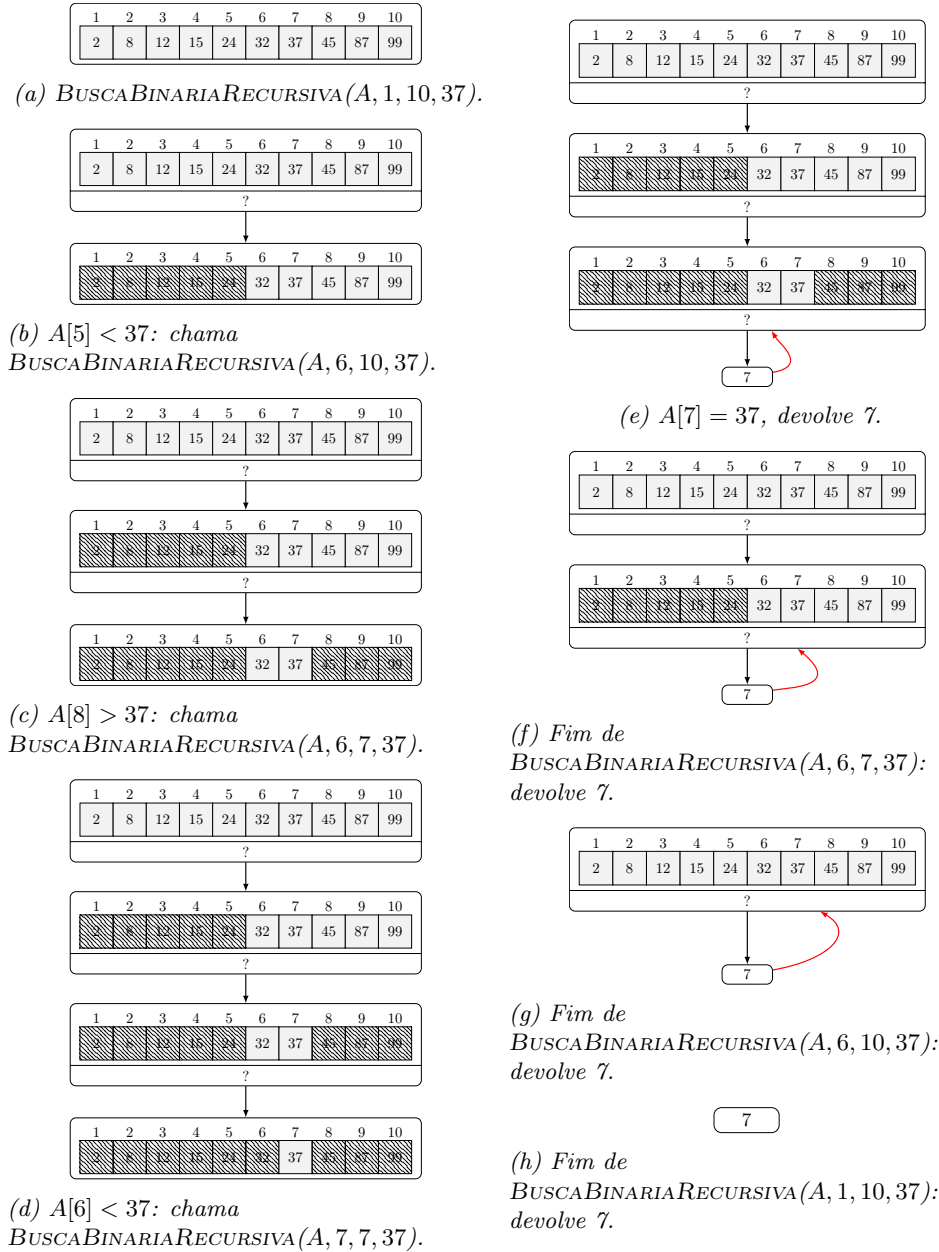


Figura 7.6: Exemplo de execução de $BUSCABINARIARECURSIVA(A, 1, 10, 37)$ (Algoritmo 7.6) para $A = (2, 8, 12, 15, 24, 32, 37, 45, 87, 99)$. Cada nó é uma chamada ao algoritmo. Setas vermelhas indicam o retorno de uma função.

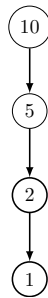


Figura 7.7: Árvore de recursão completa de `BUSCABINARIARECURSIVA(A, 1, 10, 37)` para $A = (2, 8, 12, 15, 24, 32, 37, 45, 87, 99)$. Cada nó é rotulado com o tamanho do subproblema ($dir - esq + 1$) correspondente.

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

Considere agora algum $n > 0$ e suponha que o algoritmo corretamente decide se um elemento com chave k pertence a qualquer vetor de tamanho m , para $0 \leq m < n$. 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 $A[meio].chave$ com k . Se $A[meio].chave = k$, então o algoritmo devolve $meio$ e, portanto, funciona corretamente nesse caso.

Se $A[meio].chave > k$, então o algoritmo devolve o mesmo que a chamada recursiva `BUSCABINARIARECURSIVA(A, esq, meio - 1, k)` 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, que garante que a chamada `BUSCABINARIARECURSIVA(A, esq, meio - 1, k)` corretamente detecta se um elemento com chave k pertence a $A[esq..meio - 1]$. Como o vetor está ordenado e $k < A[meio].chave$, concluímos que a chamada atual detecta se um elemento com chave k pertence a $A[esq..dir]$. Se $A[meio].chave < k$, a análise é similar.

Veja que é suficiente verificar se o elemento está na posição do meio, à direita dela, ou à esquerda dela, para resolver o problema original. Por isso, concluímos que a chamada atual funciona corretamente.

Como esse procedimento analisa, passo a passo, somente metade do tamanho do vetor

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

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

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

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

Entrada: $\langle n \rangle$, onde $n \geq 1$ é um inteiro.

Saída: F_n .

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

Algoritmo 7.7: FIBONACCIRECURSIVO(n)

1 se $n \leq 2$ 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 tenha sido calculado antes, e esse fenômeno cresce exponencialmente até chegar à base da recursão.

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.

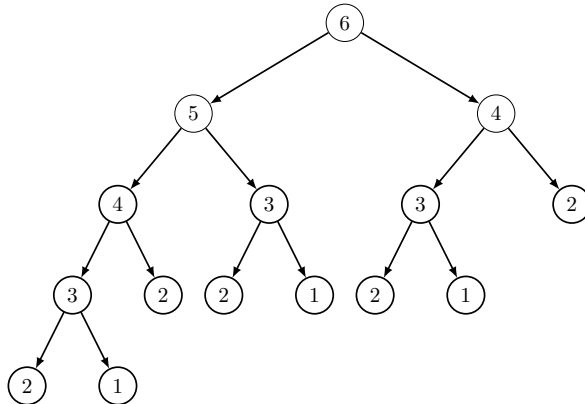


Figura 7.8: Árvore de recursão completa de `FIBONACCI_RECURSIVO(6)` (Algoritmo 7.7). Cada nó representa uma chamada ao algoritmo e é rotulado com o tamanho do subproblema correspondente. Note como várias chamadas são repetidas.

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  $F[n]$ 
  
```

Atenção! A análise acima sobre o tempo de execução do Algoritmo 7.7, versão recursiva, não está 100% correta de acordo com o número total de operações básicas. Note que acima dizemos que a soma e a comparação envolvem um número constante de operações e usamos o número 1 na fórmula $T(n) = T(n-1) + T(n-2) + 1$. De fato, temos um número constante de operações, que são a comparação, uma soma e um comando de retorno. Acontece que essa operação de soma pode não levar um número constante de operações básicas para ser realizada, ou seja, pode não levar tempo constante para ser realizada. É razoável imaginar que um número de 32 bits ou de 64 bits possa ser somado com outro rapidamente (os processadores atuais fazem isso), mas o n -ésimo número da sequência de Fibonacci precisa de aproximadamente $0.694n$ bits para ser armazenado e isso é bem maior do que 64 conforme n cresce. Essa análise não cuidadosa foi proposital para ser simples, pois mesmo com ela podemos ver a diferença entre os dois algoritmos para o problema do número de Fibonacci. Estritamente falando, o Algoritmo 7.7 faz cerca de F_n somas mas usa um número de passos básicos proporcional a nF_n .

Recorrências

Relações como $f(n) = f(n-1) + f(n-2)$, $T(n) = 2T(n/2) + n$ ou $T(n) \leq T(n/3) + T(n/4) + 3 \lg n$ 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 retorno¹.

¹Lembre-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 $\text{POTENCIAV3}(x, n)$, 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, $\text{FIBONACCIRECURSIVO}$, 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 $\text{FIBONACCIRECURSIVO}(n-1)$, $T(n-2)$ se refere ao tempo de execução de $\text{FIBONACCIRECURSIVO}(n-2)$ 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-los².

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.

No restante deste 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. Assim, pode ser útil revisar esses conceitos antes (veja a Parte I).

²Nesses 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.

Utilizaremos ainda algumas convenções. A primeira delas é a de que, em geral, o tempo gasto nos casos base dos algoritmos é constante ($\Theta(1)$), de forma que é comum descrevermos apenas a segunda parte. Por exemplo, descrevemos o tempo de execução $T(n)$ do Algoritmo 7.6, BUSCABINARIARECURSIVA, apenas como $T(n) \leq T(\lceil n/2 \rceil) + \Theta(1)$. A segunda é a de que tetos e pisos podem ser desconsiderados. Assim, $T(n) \leq T(\lceil n/2 \rceil) + \Theta(1)$ será resolvida considerando $T(n) \leq T(n/2) + \Theta(1)$. Por fim, quando pisos e tetos forem desconsiderados, a entrada será considerada uma potência conveniente, isto é, se $\lceil n/a \rceil$ e/ou $\lfloor n/a \rfloor$ forem escritos como n/a , então assumiremos que $n = a^k$ para algum k .

8.1 Método da substituição

Esse método consiste em provar por indução matemática que uma recorrência $T(n)$ é $O(f(n))$ e/ou $\Omega(f(n))$, de forma 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 $f(n)$.

Veja que para provar que alguma função $T(n)$ é, por exemplo, $O(f(n))$, precisamos provar que existem constantes c e n_0 tais que $T(n) \leq cf(n)$ para todo $n \geq n_0$. Então, para provar que $T(n)$ é $O(f(n))$ o que nós podemos fazer é provar por indução que a expressão $T(n) \leq cf(n)$ vale, por exemplo. Ou então podemos tentar provar que a expressão $T(n) \leq g(n)$ vale, sendo que $g(n)$ é $O(f(n))$. Por exemplo, se queremos provar que $T(n)$ é $O(n^2)$, podemos tentar provar que a expressão $T(n) \leq cn^2$ vale, ou então podemos tentar provar que a expressão $T(n) \leq cn^2 + dn$ vale, já que $cn^2 + dn$ é $O(n^2)$.

Agora, é muito importante observar que no passo indutivo é necessário provar **exatamente a expressão que foi proposta**. No exemplo acima, se nos propusemos a mostrar que $T(n) \leq cn^2$, não podemos chegar em uma expressão $T(n) \leq cn^2 + 10$, por exemplo, durante o passo indutivo. Novamente: isso é porque nos propusemos a provar que $T(n) \leq cn^2$ e a expressão $cn^2 + 10$ é diferente da expressão cn^2 . Da mesma forma, se tivéssemos proposto provar a expressão $T(n) \leq cn^2 + dn$, não poderíamos chegar na expressão $T(n) \leq cn^2 + dn + c$, pois ela é diferente. Esse é um ponto de muita confusão nesse método, porque acabamos misturando os conceitos de notação assintótica. Como $cn^2 + 10$ é de fato $O(n^2)$, é comum se confundir e achar que a prova acabou.

Resumindo: existem várias formas de provar que uma expressão é $O(f(n))$ (ou $\Omega(f(n))$),

porém uma vez que você escolheu uma dessas formas, a indução deve provar exatamente a forma escolhida. Veremos isso nos exemplos a seguir.

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 $1 \leq i \leq \lg n$. Nossa intuição inicial é que $T(n) = O(n^2)$. Uma forma de mostrar isso é provando que

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

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.

Começemos 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

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

onde a última desigualdade vale sempre que $c(n^2/2) + n \leq cn^2$, o que é verdade quando $c \geq 2/n^3$. 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 $c = 2, c = 3, c = 5, \dots$). 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 ?⁴ Por exemplo, será que se $T(n) = 2T(n/2) + n$, então temos $T(n) = O(n \lg n)$?

³Veja que se tivéssemos chegado em $cn^2 + n$ ao invés de $c(n^2/2) + n$, a prova estaria errada, pois $cn^2 + n$ não é menor ou igual a cn^2 nunca.

⁴Aqui queremos obter um limitante $f(n)$ tal que $f(n) = o(n^2)$.

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

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

Lembre que assumimos $T(1) = 1$. Quando $n = 1$, como $T(1) = 1$ e queremos que $T(1) \leq c \cdot 1 \lg 1 = 0$, 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 $T(2) \leq c \cdot 2 \lg 2$. 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 $T(m) \leq cm \lg m$. Precisamos mostrar que $T(n) \leq cn \lg n$ quando $n > 2$. Sabemos apenas que $T(n) = 2T(n/2) + n$, mas podemos usar a hipótese de que $T(n/2) \leq c(n/2) \lg(n/2)$, pois $n/2 < n$ quando $n > 2$. Temos então

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

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 $T(n) \leq 3n \lg n$ sempre que $n \geq n_0 = 2$, de onde concluímos que $T(n) = O(n \lg n)$.

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

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

Assim,

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

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 $T(n) = \Omega(n \lg n)$.

8.1.1 Diversas formas de obter o mesmo resultado

Como já foi discutido, podem existir diversas formas de encontrar um limitante assintótico utilizando indução. Lembre-se que, anteriormente, para mostrar que $T(n) = O(n \lg n)$, escolhemos mostrar que $T(n) \leq dn \lg n$. Mostraremos agora que $T(n) = O(n \lg n)$ provando que $T(n) \leq c(n \lg n + n)$.

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

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

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, $T(n) \leq n \lg n + n$. Como $n \lg n + n \leq n \lg n + n \lg n = 2n \lg n$, temos que $n \lg n + n = O(n \lg n)$, o que significa que $T(n) = O(n \lg n)$.

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

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

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

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

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 $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n = \Theta(n \lg n)$ 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 haja perda de generalidade ao se 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 $T(n) = \Theta(n \lg n)$ no caso em que n é potência de 2. Em particular, $T(2^k) \leq d2^k \lg(2^k)$ para alguma constante d e $T(2^{k-1}) \geq d'2^{k-1} \lg(2^{k-1})$ para alguma constante d' . Assim,

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

Similarmente,

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

Como existem constantes $d'/20$ e $4d$ tais que para todo $n \geq 3$ temos $(d'/20)n \lg n \leq T(n) \leq (4d)n \lg n$, então $T(n) = \Theta(n \lg n)$ 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

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

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

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

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

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

$16 \leq m < n$. Assim,

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

onde a última desigualdade vale se $c \geq 5/4$. Como $3/2 > 5/4$, basta tomar $c = 3/2$ para concluir que $T(n) = O(\sqrt{n}\lg n)$. A prova de que $T(n) = \Omega(\sqrt{n}\lg n)$ 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 $T(n) = O(\lg n)$. Para $n = 2$ temos $T(2) = 2 \leq c = c\lg 2$ sempre que $c \geq 2$. Suponha que $T(m) \leq c\lg m$ para todo $2 \leq m < n$. Logo,

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

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

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 $T(n) = O(\lg n)$, 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 $T(n) \leq c\lg n$ para um valor de c apropriado. Seja $n \geq 4$ e note que $T(4) = 1 \leq c\lg 4$ para $c \geq 1/2$. Suponha que $T(m) \leq c\lg m$ para todo

$4 \leq m < n$. Temos

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

onde a penúltima desigualdade vale para $n \geq 8$ e a última desigualdade vale sempre que $c \geq 1/(2 - \lg 3)$. Portanto, temos $T(n) = O(\lg n)$.

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

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

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

8.2 Método iterativo

Esse método consiste em expandir a recorrência até se chegar no caso base, que sabemos 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:

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

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

$$T(n) \leq T\left(\frac{n}{2^i}\right) + i = T\left(\frac{n}{2^{\lg n}}\right) + \lg n = T(1) + \lg n = O(\lg n).$$

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 $i = \lg n$, de onde temos que

$$T(n) = 2^{\lg n}T\left(\frac{n}{2^{\lg n}}\right) + n \lg n = nT(1) + n \lg n = n + n \lg n = \Theta(n \lg n).$$

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 $T_2(n) = \Theta(n \lg n)$, 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

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

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

$$T(n) = 2 \cdot 2^{\log_3 n} - 1 = 2 \cdot n^{\log_3 2} - 1 = 2 \cdot n^{\lg 2 / \lg 3} - 1 = \Theta(n^{1/\lg 3}).$$

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 $T(n) = \Omega(n^{1/\lg 3})$, 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

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

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

$$T(n) \geq 2^{\log_3 n} + \log_3 n = n^{1/\lg 3} + \log_3 n = \Omega(n^{1/\lg 3}).$$

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 internamente com o tempo feito naquela chamada, desconsiderando os tempos nas chamadas recursivas que ela faz, e é rotulado externamente com o tamanho do problema relacionado. 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. Interno em cada nó nós apresentamos o tempo feito na chamada representada pelo mesmo desconsiderando o tempo das chamadas recursivas. Externo a cada nó nós apresentamos o tamanho do subproblema que está sendo considerado ali. No lado direito da figura temos os níveis da árvore (que vão até $\lg n$ pois cada subproblema é reduzido pela metade) e a quantidade de

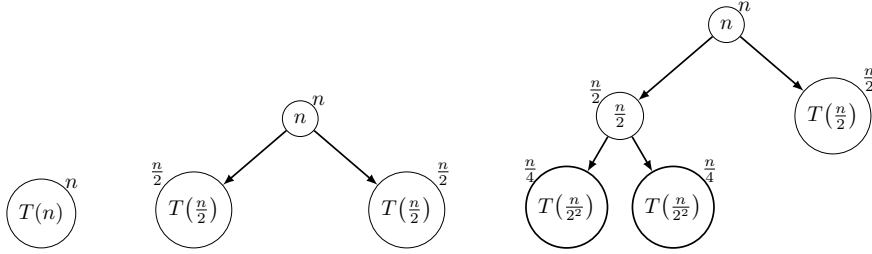


Figura 8.1: Começo da construção da árvore de recursão para $T(n) = 2T(n/2) + n$.

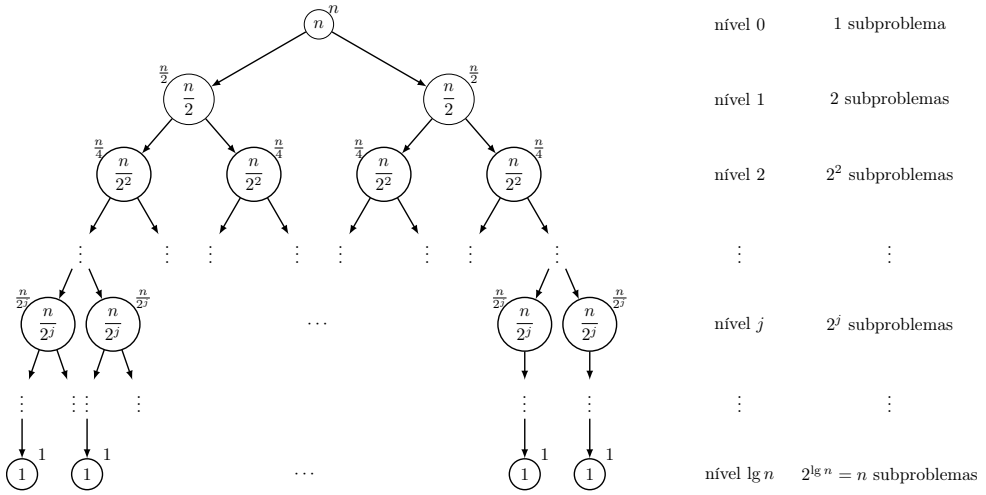


Figura 8.2: Árvore de recursão para $T(n) = 2T(n/2) + n$.

subproblemas por nível. Assim, temos que

$$T(n) = \sum_{j=0}^{\lg n} \left(2^j \cdot \frac{n}{2^j} \right) = \sum_{j=0}^{\lg n} n = n(\lg n + 1).$$

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

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

$$T(n) = \sum_{j=0}^{\log_3 n} (2^j \cdot 1) = \frac{2^{\log_3 n + 1} - 1}{2 - 1} = 2n^{\log_3 2} - 1,$$

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

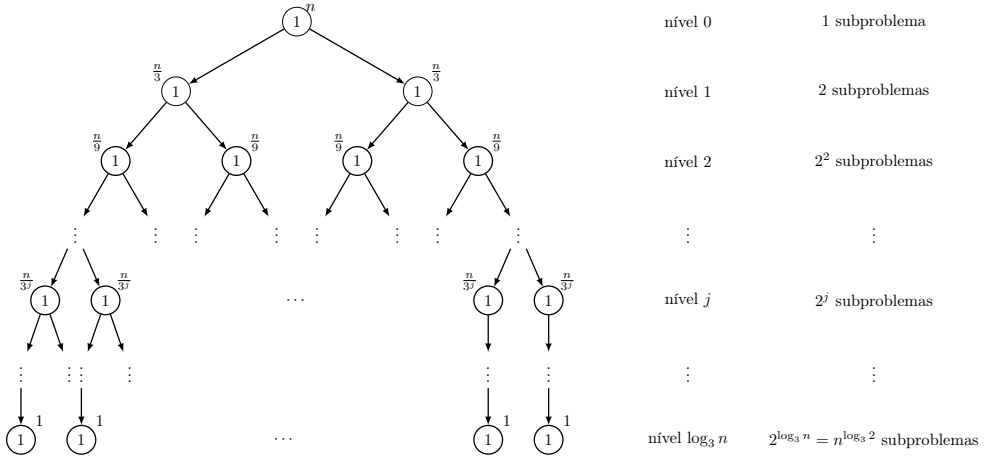


Figura 8.3: Árvore de recursão para $T(n) = 2T(n/3) + 1$.

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

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

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

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

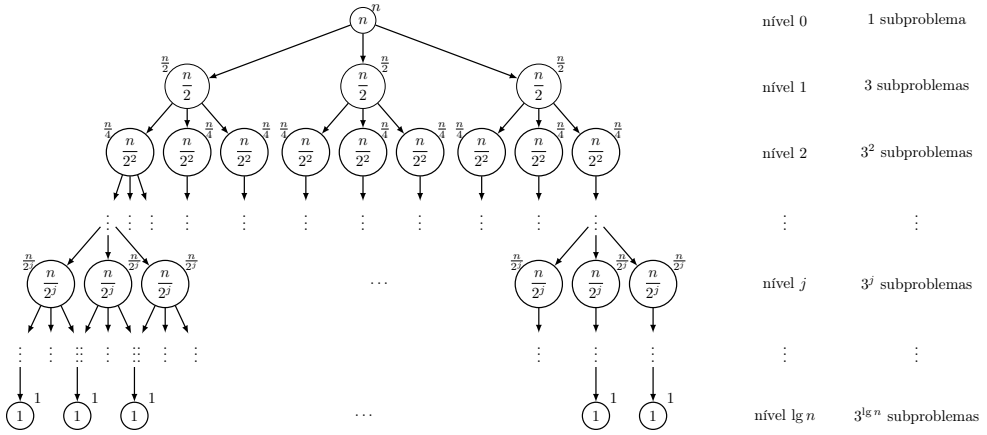


Figura 8.4: Árvore de recursão para $T(n) = 3T(n/2) + n$.

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 parte da árvore de recorrência de $T(n) = aT(n/b) + f(n)$.

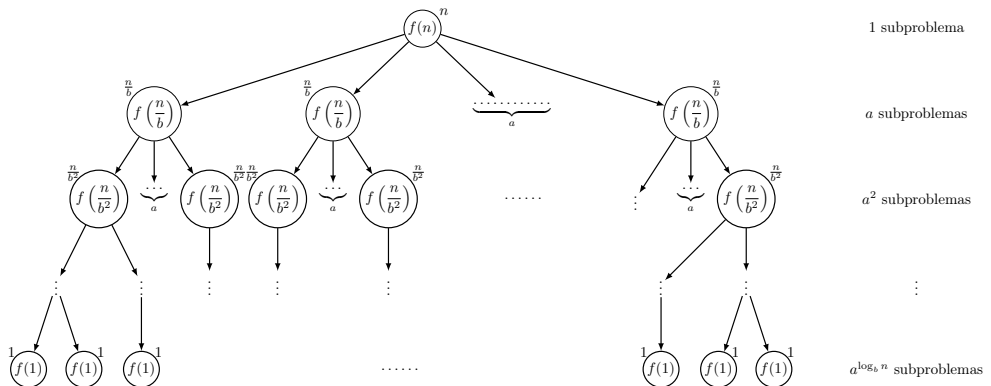


Figura 8.5: Árvore de recorrência para $T(n) = aT(n/b) + f(n)$.

Note que temos

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

Portanto, considerando somente o tempo para dividir o problema em subproblemas recursivamente, temos que é gasto tempo $\Theta(n^{\log_b a})$. 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., $f(n) = \Theta(n^{\log_b a} / \lg n)$ ou $\Theta(n^{\log_b a} / \lg(\lg n))$. 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, como pode ser visto no Exemplo 1 a seguir. 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.

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, \dots, 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 $T(n) = \Theta(n^{\log_b a} \lg n)$;
- (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.

Para provar que $af(n/b) \leq cf(n)$ para algum $c < 1$, basta provar que $cf(n) - af(n/b) \geq 0$ para algum $c < 1$. Assim,

$$\begin{aligned}
 cf(n) - af(n/b) &= c \sum_{i=0}^k a_i n^i - a \sum_{i=0}^k a_i \frac{n^i}{b^i} \\
 &= a_k \left(c - \frac{a}{b^k} \right) n^k + \sum_{i=0}^{k-1} a_i \left(c - \frac{a}{b^i} \right) n^i \\
 &\geq a_k \left(c - \frac{a}{b^k} \right) n^k - \sum_{i=0}^{k-1} a_i \left(\frac{a}{b^i} \right) n^i \\
 &\geq a_k \left(c - \frac{a}{b^k} \right) nn^{k-1} - \left(a \sum_{i=0}^{k-1} a_i \right) n^{k-1} \\
 &= (c_1 n) n^{k-1} - (c_2) n^{k-1},
 \end{aligned} \tag{8.2}$$

onde c_1 e c_2 são constantes e em (8.2) 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 $cf(n) - af(n/b) \geq 0$. \square

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, \dots, 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 $T(n) = \Theta(n^k \lg n)$;
- (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

$$T(n) \leq \sum_{j=0}^{\log_b n} c_1 n^k \left(\frac{a}{b^k}\right)^j = c_1 n^k \sum_{j=0}^{\log_b n} \left(\frac{a}{b^k}\right)^j, \quad (8.3)$$

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 (8.3) pode ser desenvolvida da seguinte

forma:

$$\begin{aligned}
 T(n) &\leq c_1 n^k \left(\frac{\left(\frac{a}{b^k}\right)^{\log_b n+1} - 1}{\frac{a}{b^k} - 1} \right) = \frac{c_1 n^k}{\frac{a}{b^k} - 1} \left(\left(\frac{a}{b^k}\right)^{\log_b n+1} - 1 \right) \\
 &\leq \frac{c_1 n^k}{\frac{a}{b^k} - 1} \left(\frac{a}{b^k}\right)^{\log_b n+1} = \frac{ac_1 n^k}{\left(\frac{a}{b^k} - 1\right) b^k} \left(\frac{a}{b^k}\right)^{\log_b n} \\
 &= \frac{ac_1 n^k}{\left(\frac{a}{b^k} - 1\right) b^k} n^{\log_b a/b^k} = c' n^k \frac{n^{\log_b a}}{n^{\log_b b^k}} \\
 &= c' n^{\log_b a},
 \end{aligned}$$

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 (8.3) pode ser desenvolvida da seguinte forma:

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

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

(3) se $a < b^k$, temos $a/b^k < 1$, e a Equação (8.3) 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

$$T(n) \geq \sum_{j=0}^{\log_b n} c_2 n^k \left(\frac{a}{b^k}\right)^j = c_2 n^k \sum_{j=0}^{\log_b n} \left(\frac{a}{b^k}\right)^j, \quad (8.4)$$

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 $T(n) = \Omega(n^k \lg n)$, 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 $f(n) = n = n^{\lg 2}$, o caso do Teorema Mestre em que esses parâmetros se encaixam é o caso (2). Assim, pelo Teorema Mestre, $T(n) = \Theta(n \lg n)$.

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, $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_{16} 4} \lg n) = \Theta(\sqrt{n} \lg n)$.

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

Neste caso temos $a = 4$, $b = 2$ e $f(n) = 10n^3$. Assim, $\log_b a = \lg 4 = 2$. 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. $T(n) = 2T(n/2) + n \lg n$.

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

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

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

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

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

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

Começamos notando que $a = 16$, $b = 4$ e $f(n) = n^2/\lg n$. Logo, $n^{\log_b a} = n^2$. Para todo n suficientemente grande e qualquer constante C vale que $n \geq C \lg n$. Assim, para qualquer $\varepsilon > 0$, temos que $n^2/\lg n \neq O(n^{2-\varepsilon})$, de onde concluímos que a recorrência $T(n)$ não se encaixa no caso (1). Como $n^2/\lg n \neq \Theta(n^2)$, também não podemos utilizar o caso (2). Por fim, como $n^2/\lg n \neq \Omega(n^{2+\varepsilon})$ 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 $T(n) = 16T(n/4) + n^2/\lg n$ 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 $T'(n) = \Theta(n^2 \lg n)$. Portanto, como $T(n) \leq T'(n)$, concluímos que $T(n) = O(n^2 \lg n)$, obtendo um limitante assintótico superior para $T(n)$. Por outro lado, temos que $T(n) = 16T(n/4) + n^2/\lg n \geq T''(n) = 16T''(n/4) + n$. 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 $n^2 \lg n$.

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. A escolha da estrutura correta pode ter grande impacto no tempo de execução de um algoritmo⁵. Estruturas diferentes suportam operações diferentes em tempos de execução 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.

⁵Veja um exemplo na Seção 7.5.

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 estrutura de dados linear que agrupa elementos, ou variáveis, de mesmo tipo, permitindo que todas sejam referenciadas por um mesmo 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 que indica a posição do elemento na estrutura.

Um vetor tem uma **capacidade** pré-definida, que indica quantos elementos ele agrupa e é indicada no momento da sua declaração. Distinguiremos este valor do **tamanho** do vetor, que é a quantidade de elementos efetivamente armazenados no mesmo. Denotamos um vetor A com capacidade para armazenar m elementos por $A[1..m]$. Se o tamanho do vetor é n , o denotaremos por $A[1..n]$ quando a capacidade não for importante. Isso ocorre em algoritmos que já recebem o vetor preenchido e não fazem inserção e remoção de elementos, por exemplo. Quando precisamos destacar os elementos contidos no vetor, a notação será $A = (a_1, a_2, \dots, a_n)$.

O acesso a um elemento específico se dá com o uso de colchetes. Assim, denotamos

por $A[i]$ o elemento que está armazenado na i -ésima posição de A (i.e., a_i), para todo i com $1 \leq i \leq n$ ¹. 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 $A[i], A[i+1], \dots, A[j]$.

Será muito comum utilizarmos vetores do tipo numérico, de forma que cada elemento é um único número. No entanto, em várias aplicações é útil que cada elemento seja um **registro**, contendo diversos atributos para armazenar os dados. Quando um elemento de um vetor for um registro, teremos um atributo **chave**, que em geral é numérico e nos permite identificar unicamente os elementos.

A seguir temos uma representação gráfica do vetor numérico $A = (12, 99, 37, 24, 2, 15)$, em que os elementos aparecem dentro de quadrados e as posições dos elementos aparecem acima dos quadrados:

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

Nas discussões a seguir consideraremos que o vetor A é numérico apenas e tem tamanho n . Como já foi discutido no Capítulo 2, o tempo para buscar um elemento em A 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 pode ser feita em tempo constante, $\Theta(1)$, pois basta copiar x na primeira posição disponível do vetor, a posição $n+1$, e atualizar o tamanho do vetor. Já a remoção de algum elemento do vetor envolve saber em qual posição i encontra-se tal elemento. Uma vez conhecida, podemos simplesmente copiar o elemento $A[n]$ para a posição i e decrementar o tamanho do vetor. Assim, se a posição i já for indicada, a remoção leva tempo $\Theta(1)$. Caso contrário, uma busca pelo elemento ainda precisa ser feita, e a remoção leva 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 $O(\lg n)$. A inserção, no entanto, não pode mais ser feita em tempo constante, pois precisamos garantir que o vetor continuará ordenado após a operação. Assim, primeiro precisamos descobrir em qual posição i gostaríamos de inserir o novo elemento. Isso pode ser feito com uma busca linear ou uma busca binária, por exemplo. Em seguida, precisamos mover cada elemento do subvetor $A[i..n]$ uma posição para a direita, para então copiarmos o novo elemento na posição i . Assim, mesmo usando busca binária

¹Em alguns capítulos futuros, será conveniente ter uma posição 0 em um vetor, de forma que a notação se torna $A[0..m]$.

para encontrar i , a inserção leva tempo $O(n)$. De forma similar, a remoção de um elemento que está em uma posição i precisa de tempo $O(n)$ pois é necessário deslocar cada elemento do subvetor $A[i + 1..n]$ uma posição para a esquerda.

9.2 Lista encadeada

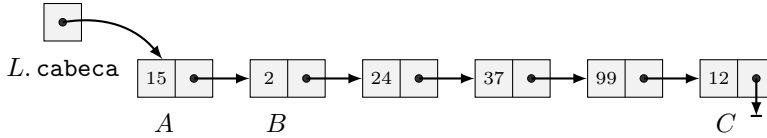
Uma **lista encadeada** é uma estrutura de dados linear onde cada elemento é armazenado em um **nó**, que armazena dados e também **endereços** para outros nós da lista. Por isso, cada nó de uma lista pode estar em uma posição diferente da memória, sendo diferente de um vetor, onde os elementos são armazenados 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 pode também conter outros atributos importantes. Iremos inserir, remover ou modificar elementos de uma lista baseados nos atributos chave, que devem ser numéricos. 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 **prox**, que dá acesso ao nó que está imediatamente após o nó atual na lista. Em uma **lista duplamente encadeada** existe, além do atributo **prox**, o atributo **ante**, que dá acesso ao nó que está imediatamente antes do nó atual na lista. Seja x um nó qualquer. Se $x.\text{ante} = \text{NULO}$, então x não tem predecessor, de modo que é o primeiro nó da lista, chamado também **cabeça** da lista. Se $x.\text{prox} = \text{NULO}$, então x não tem sucessor e é chamado de **cauda** da lista, sendo o último nó da mesma. Em uma **lista circular**, o atributo **prox** da cauda tem o endereço da cabeça da lista, enquanto que o atributo **ante** da cabeça da lista tem o endereço da cauda. Dada uma lista L , o atributo $L.\text{cabeça}$ guarda o endereço do primeiro nó de L , sendo que $L.\text{cabeça} = \text{NULO}$ quando a lista estiver vazia.

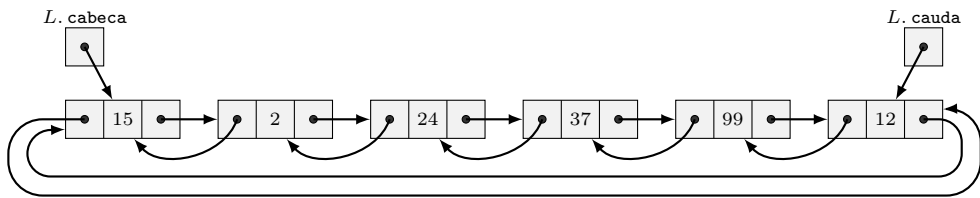
A seguir temos uma representação gráfica de uma lista encadeada simples que armazena elementos que têm como chave os números 12, 99, 37, 24, 2, 15, sendo que outros atributos não são representados. As chaves aparecem internas aos quadrados

e os endereços são representados pelas setas. Assim, $L.cabeca = A$, $A.prox = B$ e $C.prox = \text{NULO}$, por exemplo:



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

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



A seguir vamos descrever os procedimentos de busca, inserção e remoção em uma lista L duplamente encadeada, não ordenada e não-circular que armazena n elementos. Para algumas aplicações pode 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 $L.cauda = \text{NULO}$ quando L é uma lista vazia.

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 será percorrida, de modo que o tempo de execução de `BUSCANALISTA` é $O(n)$.

Algoritmo 9.1: `BUSCANALISTA`(L, k)

```

1  $x = L.cabeca$ 
2 enquanto  $x \neq \text{NULO}$  e  $x.chave \neq k$  faça
3    $x = x.prox$ 
4 devolve  $x$ 

```

A inserção de um elemento em uma lista é realizada, em geral, no começo da lista

(a menos que haja necessidade de manter os elementos em ordem). 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*. Caso *L* não esteja vazia, *x.prox* terá o endereço da cabeça atual de *L* e *L.cabeca.ante* terá o endereço de *x*. Caso *L* esteja vazia, então *x.prox* será NULO e *L.cauda* terá o endereço de *x*. Ao fim do processo, *x* deverá ser a nova cabeça da lista. Por isso, *x.ante* sempre será NULO.

Algoritmo 9.2: INSERENOINICIOLISTA(*L*, *x*)

```

1 x.ante = NULO
2 x.prox = L.cabeca
3 se L.cabeca ≠ NULO então
4   /* Se há elemento em L, a cabeça deve ter como anterior o nó x */
   L.cabeca.ante = x
5 senão
6   /* Se não há cabeça, também não há cauda, e x será o último nó */
   L.cauda = x
7 L.cabeca = x

```

Como somente uma quantidade constante de operações é executada, o procedimento INSERENOINICIOLISTA leva tempo $\Theta(1)$. 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.

Algoritmo 9.3: INSERENOFIMLISTA(*L*, *x*)

```

1 x.ante = L.cauda
2 x.prox = NULO
3 se L.cauda ≠ NULO então
4   /* Se há elemento em L, a cauda deve ter como próximo o nó x */
   L.cauda.prox = x
5 senão
6   /* Se não há cauda, também não há cabeça, e x será o primeiro nó */
   L.cabeca = x
7 L.cauda = x

```

Finalmente, o Algoritmo 9.4 mostra o procedimento REMOVEDALISTA, que remove um nó com chave k de uma lista L . A remoção é simples, sendo necessário efetuar uma busca para encontrar o nó x que possui chave k e atualizar os ponteiros $x.\text{ante}.\text{prox}$ e $x.\text{prox}.\text{ante}$, 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 $L.\text{cauda}$ 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 $O(n)$, como o algoritmo de busca.

Algoritmo 9.4: REMOVEDALISTA(L, k)

```

1  $x = L.\text{cabeca}$ 
2 enquanto  $x \neq \text{NULO}$  e  $x.\text{chave} \neq k$  faça
3    $x = x.\text{prox}$ 
4 se  $x = \text{NULO}$  então
5    $\mid$  devolve NULO
   /* Se  $x$  é a cauda de  $L$ , então o penúltimo nó passa a ser  $L.\text{cauda}$  */
6 se  $x.\text{prox} == \text{NULO}$  então
7    $\mid$   $L.\text{cauda} = x.\text{ante}$ 
8 senão
9    $\mid$   $x.\text{prox}.\text{ante} = x.\text{ante}$ 
   /* Se  $x$  é a cabeça de  $L$ , então o segundo nó passa a ser  $L.\text{cabeca}$  */
10 se  $x.\text{ante} == \text{NULO}$  então
11    $\mid$   $L.\text{cabeca} = x.\text{prox}$ 
12 senão
13    $\mid$   $x.\text{ante}.\text{prox} = x.\text{prox}$ 

```

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 qual elemento deve ser removido).

10.1 Pilha

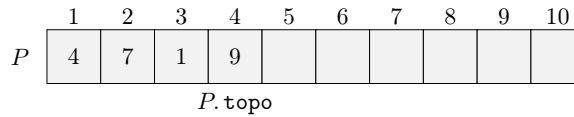
Pilha é uma coleção dinâmica de dados cuja operação de remoção deve remover o elemento que está na coleção há menos tempo. Essa política de remoção é conhecida como “LIFO”, acrônimo para “*last in, first out*” (último a entrar, primeiro a sair). Nesta seção vamos apresentar uma forma de implementar uma pilha em que tanto a operação de remoção quanto a de inserção levam tempo $\Theta(1)$.

Existem inúmeras aplicações para pilhas. Por exemplo, podemos verificar se uma palavra é um palíndromo ao inserir as letras na ordem de leitura 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, o que não limitaria a quantidade de elementos que podem ser armazenados na estrutura.

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

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



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 executados. 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 $P.\text{topo}$ e copia x em $P[P.\text{topo}]$.

Algoritmo 10.1: EMPILHA(P, x)

```

1 se  $P.\text{topo} \neq P.\text{capacidade}$  então
2    $P.\text{topo} = P.\text{topo} + 1$ 
3    $P[P.\text{topo}] = x$ 
```

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

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

Algoritmo 10.2: DESEMPILHA(P)

```

1 se  $P.\text{topo} \neq 0$  então
2    $x = P[P.\text{topo}]$ 
3    $P.\text{topo} = P.\text{topo} - 1$ 
4   devolve  $x$ 
5 devolve NULO

```

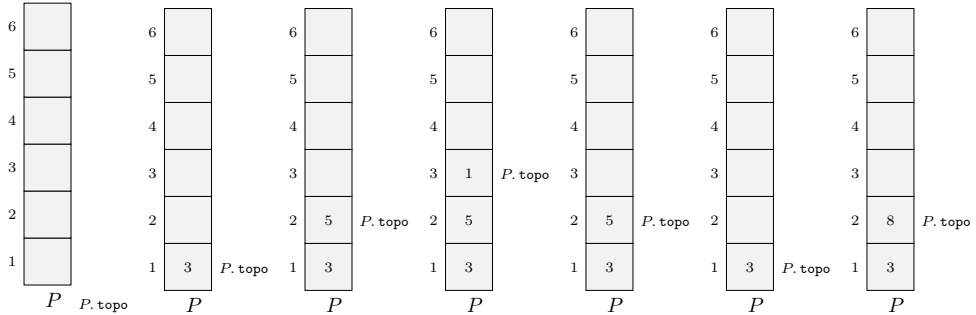


Figura 10.1: Operações em uma pilha P inicialmente vazia. Da esquerda para a direita, mostramos a pilha após cada uma das seguintes operações: EMPILHA(P , 3), EMPILHA(P , 5), EMPILHA(P , 1), DESEMPILHA(P), DESEMPILHA(P), DESEMPILHA(P), EMPILHA(P , 8).

10.2 Fila

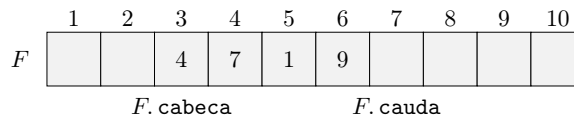
Fila é uma coleção dinâmica de dados cuja operação de remoção deve remover o elemento que está na coleção há mais tempo. Essa política de remoção é conhecida como “FIFO”, acrônimo para “*first in, first out*” (primeiro a entrar, primeiro a sair). Nesta seção vamos apresentar uma forma de implementar uma fila em que tanto a operação de remoção quanto a de inserção levam 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 $F.\text{cabeca}$, que deve sempre

armazenar o índice do elemento que está há mais tempo na fila. Teremos também um atributo $F.cauda$, 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 $F.cabeca$ e $F.cauda$ são feitas módulo $F.capacidade = m$. Com isso, manteremos a invariante de que se $F.cabeca < F.cauda$, então os elementos da fila encontram-se em $F[F.cabeca..F.cauda - 1]$, e se $F.cabeca > F.cauda$, então os elementos encontram-se em $F[F.cabeca..F.capacidade]$ e $F[1..F.cauda - 1]$.

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



Na inicialização da estrutura, faremos $F.cabeca = F.cauda = 1$. Teremos ainda um campo $F.tamanho$, 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 . Os 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, copia o elemento x em $F[F.cauda]$ e incrementa $F.cauda$ e $F.tamanho$.

Algoritmo 10.3: ENFILEIRA(F, x)

```

1 se  $F.tamanho \neq F.capacidade$  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

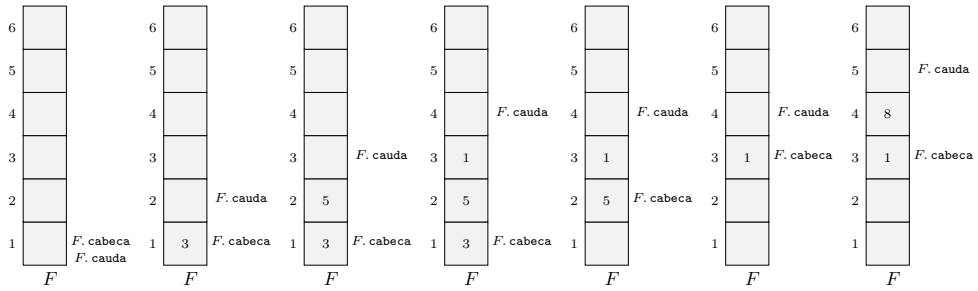


Figura 10.2: Operações em uma fila F inicialmente vazia. Da esquerda para a direita, mostramos a fila pós cada uma das seguintes operações: $ENFILEIRA(F, 3)$, $ENFILEIRA(F, 5)$, $ENFILEIRA(F, 1)$, $DESENFILEIRA(F)$, $DESENFILEIRA(F)$, $ENFILEIRA(F, 8)$.

elemento em $F[F.cabeca]$. Veja que o valor de $F.cabeca$ precisa ser incrementado e $F.tamanho$ precisa ser decrementado. Um outro procedimento interessante é o **CONSULTA**, que apenas devolve o valor em $F[F.cabeca]$.

Algoritmo 10.4: $DESENFILEIRA(F)$

```

1 se  $F.tamanho \neq 0$  então
2    $x = F[F.cabeca]$ 
3   se  $F.cabeca == F.capacidade$  então
4      $F.cabeca = 1$ 
5   senão
6      $F.cabeca = F.cabeca + 1$ 
7    $F.tamanho = F.tamanho - 1$ 
8   devolve  $(x)$ 
9 devolve NULO
```

Árvores

Árvores são estruturas de dados não lineares onde cada elemento é armazenado em um nó, que armazena dados e também endereços 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 para exemplos dessa discussão.

Uma árvore pode ser definida recursivamente: ela ou é vazia ou é um nó que possui endereços para outras árvores. De fato, muitos algoritmos que lidam com árvores são

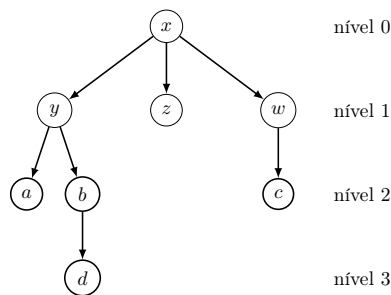


Figura 11.1: Árvore com 4 níveis e altura 3, onde cada nó é representado por um círculo e as relações pai e filho são representadas pelas setas. Além disso, (i) x é o nó raiz (nível 0), (ii) y , z e w são filhos de x , (iii) y é pai de a e b , (iv) a , d , z e c são folhas. Note que a altura do nó z 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 .

descritos de forma simples recursivamente.

Em uma árvore, somente temos acesso direto ao nó raiz e qualquer manipulação, portanto, deve utilizar os endereços armazenados nos 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 encadeada.

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 $O(\lg n)$, 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 endereços para no máximo duas árvores binárias.

O filho esquerdo (resp. direito) de um nó x é raiz da **subárvore esquerda** (resp. **subárvore direita**) de x . Formalmente, se x é um nó, então x contém os atributos $x.chave$, $x.esq$ e $x.dir$, entre outros, onde $x.chave$ contém a chave do elemento x e em $x.esq$ e $x.dir$ temos, respectivamente, as raízes das subárvores esquerda e direita (ou NULO, caso x não tenha aquele filho). Para toda folha x temos $x.esq = x.dir = \text{NULO}$. Por simplicidade, 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 $x.chave$ e todos os nós da subárvore direita possuem chaves maiores do que $x.chave$.

A Figura 11.2a tem um exemplo de uma árvore binária de busca. 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 $\text{BUSCAABB}(r, k)$ devolve um endereço para o nó que contém a chave k ou NULO, caso não exista elemento com chave k na árvore dada.

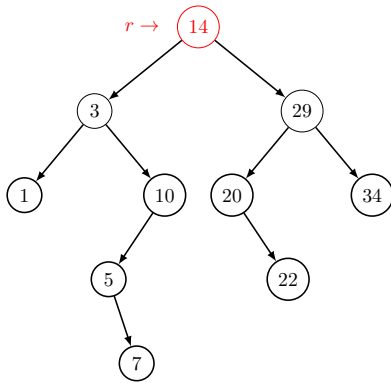
A ideia do algoritmo é semelhante à ideia utilizada na busca binária em vetores ordenados. Começamos comparando k com $r.chave$, de forma que temos 3 resultados possíveis: (i) caso $k = r.chave$, encontramos o elemento e o endereço para r é devolvido pelo algoritmo; (ii) caso $k < r.chave$, sabemos que 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 $r.esq$; e (iii) caso $k > r.chave$, então o elemento de chave k só pode estar na subárvore direita, de forma que executamos o procedimento sobre $r.dir$. Por isso, é bastante simples descrever esse procedimento de forma recursiva, como faz o Algoritmo 11.1. Veja a Figura 11.2 para um exemplo de execução.

Algoritmo 11.1: $BUSCAABB(r, k)$

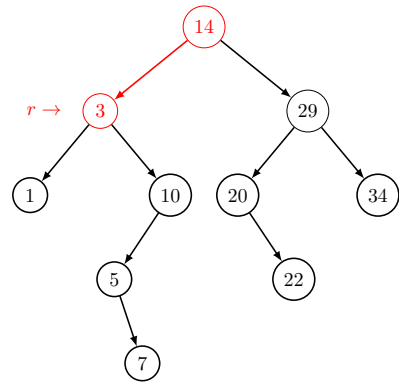
```

1 se  $r == \text{NULO}$  ou  $k == r.\text{chave}$  então
2   | devolve  $r$ 
3 senão se  $k < r.\text{chave}$  então
4   | devolve  $BUSCAABB(r.\text{esq}, k)$ 
5 senão
6   | devolve  $BUSCAABB(r.\text{dir}, k)$ 

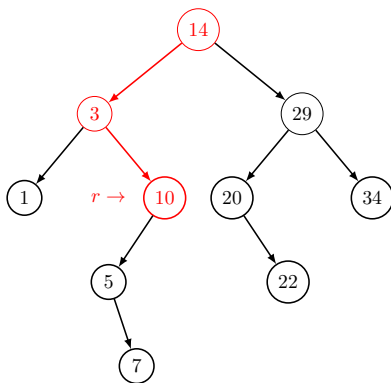
```



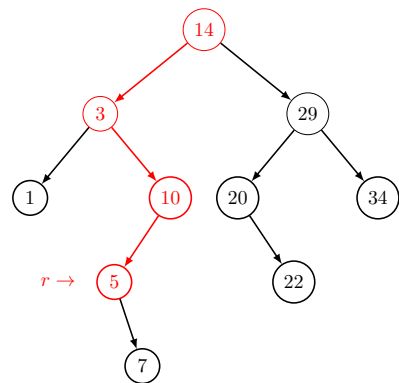
(a) $5 < 14$, chama $BUSCAABB(r.\text{esq}, 5)$.



(b) $5 > 3$, chama $BUSCAABB(r.\text{dir}, 5)$.



(c) $5 < 10$, chama $BUSCAABB(r.\text{esq}, 5)$.



(d) Encontrou 5.

Figura 11.2: Execução de $BUSCAABB(r, 5)$. O caminho percorrido está destacado em vermelho.

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 encadeada que armazene os mesmos elementos.

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 $x.\text{esq} = x.\text{dir} = \text{NULO}$.

Se a árvore está inicialmente vazia, então o nó x será a nova raiz. Caso contrário, o primeiro passo do algoritmo é buscar por $x.\text{chave}$ na árvore. Se $x.\text{chave}$ não estiver na árvore, então a busca terminou em um nó y que será o pai de x : se $x.\text{chave} < y.\text{chave}$, então inserimos x à esquerda de y e caso contrário o inserimos à direita. Note que qualquer busca posterior por $x.\text{chave}$ 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 $O(h)$, onde h é a altura da árvore. O Algoritmo 11.2 formaliza o procedimento INSEREABB, e a Figura 11.3 mostra um exemplo de execução.

Algoritmo 11.2: INSEREABB(r, x)

```

1 se  $r == \text{NULO}$  então
2   | devolve  $x$ 
3 se  $x.\text{chave} < r.\text{chave}$  então
4   |  $r.\text{esq} = \text{INSEREABB}(r.\text{esq}, x)$ 
5 senão se  $x.\text{chave} > r.\text{chave}$  então
6   |  $r.\text{dir} = \text{INSEREABB}(r.\text{dir}, x)$ 
7 devolve  $r$ 
```

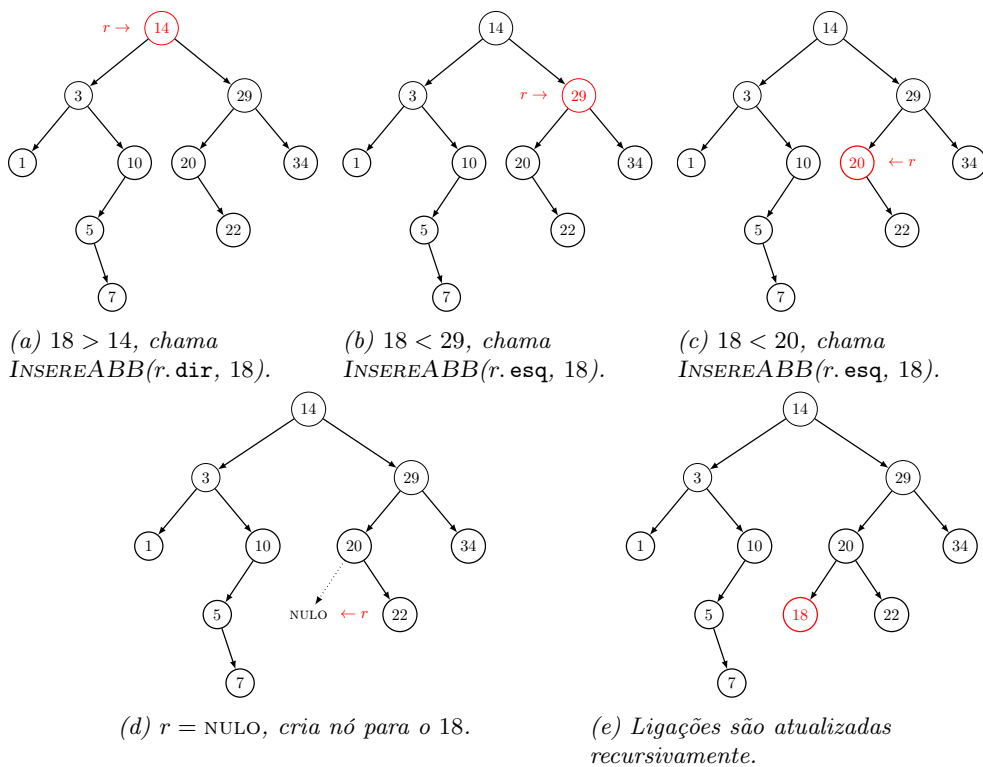


Figura 11.3: Execução de $INSEREABB(r, 18)$.

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 $\text{REMOVEABB}(r, k)$ remove o nó x tal que $x.\text{chave} = k$ da árvore (se ele existir) e devolve o endereço 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 a remoção de um nó x depende da quantidade de filhos de x . Quando ele não possui filhos, basta remover x , modificando o atributo $z.\text{esq}$ ou $z.\text{dir}$ 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 $z.\text{esq}$ ou $z.\text{dir}$ 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 , que é o nó cuja chave é o menor valor que é maior do que $x.\text{chave}$. Isso porque com x na árvore, temos $x.\text{esq}.\text{chave} \leq x.\text{chave} \leq x.\text{dir}.\text{chave}$. Ao substituir x pelo seu sucessor, continuaremos tendo $x.\text{esq}.\text{chave} \leq y.\text{chave} \leq x.\text{dir}.\text{chave}$, mantendo a propriedade de 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 $x.\text{dir}$ (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 $x.\text{esq}$;
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 $y.\text{chave}$ é o menor elemento da subárvore com raiz $x.\text{dir}$. Assim, basta

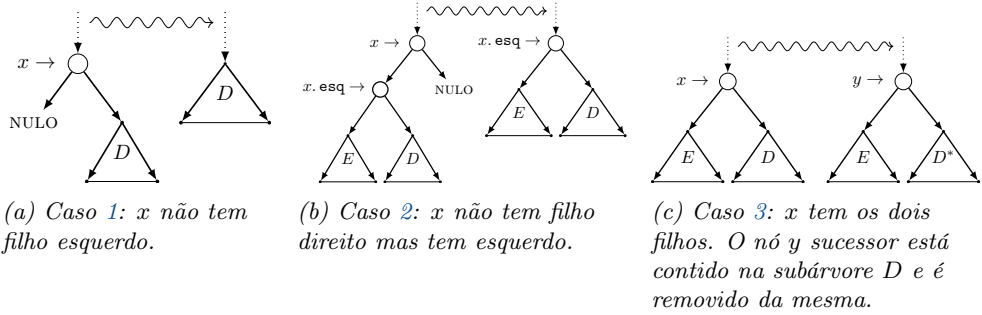


Figura 11.4: Esquema dos casos de remoção de um nó x em uma árvore binária de busca.

encontrarmos o menor elemento de uma subárvore binária de busca. Para encontrar o menor elemento de uma árvore com raiz r , executamos $\text{MINIMOABB}(r)$, que é apresentado no Algoritmo 11.3.

Algoritmo 11.3: $\text{MINIMOABB}(r)$

```

1 se  $r.\text{esq} \neq \text{NULO}$  então
2    $\lfloor$  devolve  $\text{MINIMOABB}(r.\text{esq})$ 
3 devolve  $r$ 

```

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 $\text{MINIMOABB}(r)$ é, no pior caso, proporcional à altura da árvore.

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.

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 $\text{REMOVEABB}(x.\text{dir}, y.\text{chave})$ 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 $\text{REMOVEABB}(r, k)$ 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 $\text{MINIMOABB}(x.\text{dir})$ para encontrar o sucessor y de x , o que leva tempo proporcional à altura de x , e executar $\text{REMOVEABB}(x.\text{dir}, y.\text{chave})$, 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 $\text{REMOVEABB}(r, k)$ é, ao todo, $O(h)$, onde h é a altura da árvore. A Figura 11.5 exemplifica a execução do algoritmo de remoção.

Algoritmo 11.4: $\text{REMOVEABB}(x, k)$

```

1 se  $x == \text{NULO}$  então
2   | devolve NULO
3 se  $k < x.\text{chave}$  então
4   |  $x.\text{esq} = \text{REMOVEABB}(x.\text{esq}, k)$ 
5 senão se  $k > x.\text{chave}$  então
6   |  $x.\text{dir} = \text{REMOVEABB}(x.\text{dir}, k)$ 
7 senão
8   | /* Aqui temos  $x.\text{chave} = k$  */
9   | se  $x.\text{esq} == \text{NULO}$  então
10    |  $x = x.\text{dir}$ 
11    | senão se  $x.\text{dir} == \text{NULO}$  então
12      |  $x = x.\text{esq}$ 
13    | /* O sucessor de  $x$  é o nó de menor chave da subárvore direita, que
14      | precisa ser removido de lá */
15    | senão
16      |  $y = \text{MINIMOABB}(x.\text{dir})$ 
17      |  $x.\text{chave} = y.\text{chave}$ 
18      | /* Copie outros dados, se necessário */
19      |  $x.\text{dir} = \text{REMOVEABB}(x.\text{dir}, y.\text{chave})$ 
20 devolve  $x$ 

```

11.1.4 Outras operações sobre árvores binárias de busca

Diversas outras operações podem ser realizadas em árvores binárias de busca de forma eficiente. No que segue, seja h a altura de uma árvore:

- 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

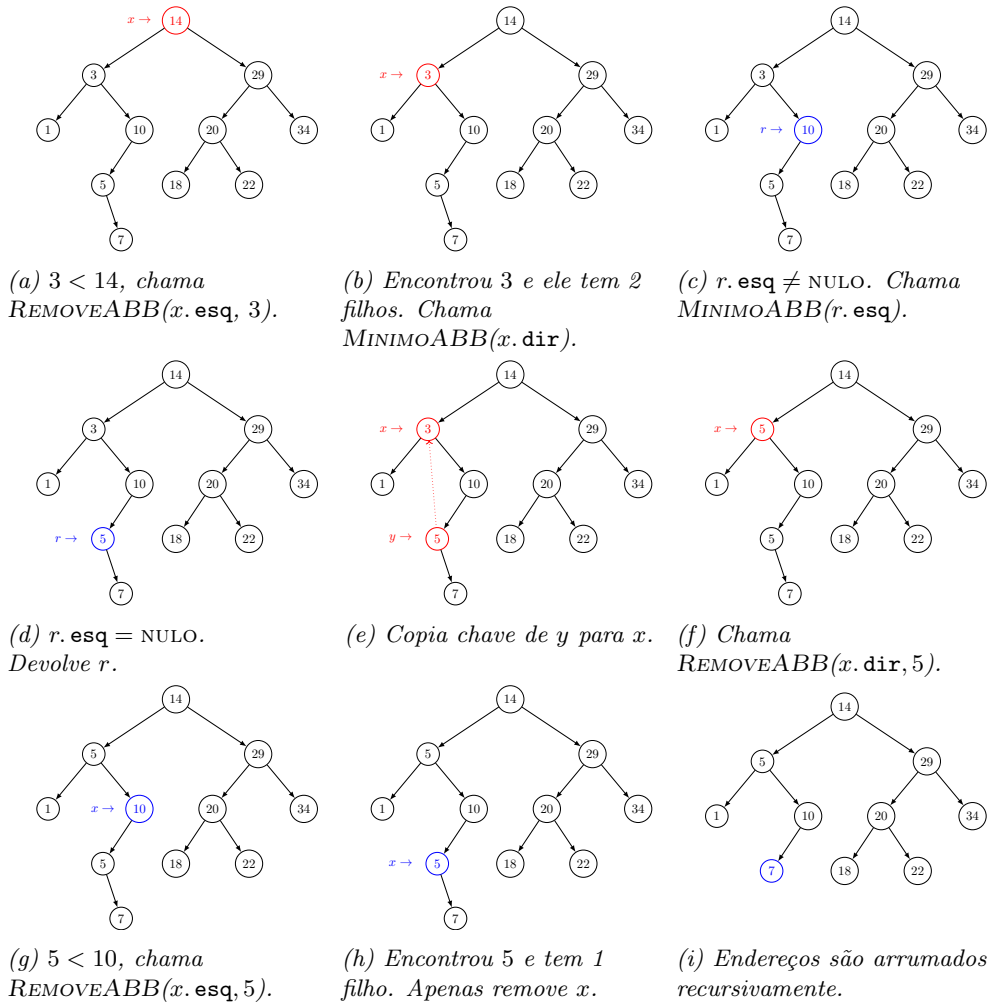


Figura 11.5: Execução de $REMOVEABB(r, 3)$.

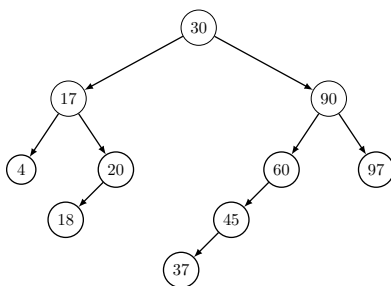


Figura 11.6: Exemplo de árvore binária de busca onde o sucessor de 30 é o 37 (menor nó da subárvore enraizada em 90) e o sucessor de 20 é o 30 (menor ancestral do 20 cujo filho esquerdo, o 17, também é ancestral do 20).

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 com chave k (o elemento com a menor que chave que é maior do que k): seja x o nó tal que $x.\text{chave} = k$. Pela estrutura da árvore, se x tem um filho direito, então o sucessor de x é o elemento com menor chave armazenado nessa subárvore direita. Caso x não tenha filho direito, então o primeiro nó que contém uma chave 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 com chave k : se x é o nó cuja chave é k , o predecessor de x é o elemento com maior chave da subárvore esquerda de x ou então é o ancestral de maior chave cujo filho direito também é ancestral de x . Tempo necessário: $O(h)$.

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 em que 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, \dots, x_n , onde $x_1 < \dots < x_n$. A ordem em que os elementos serão inseridos é x_1, x_2, \dots, x_n . Esse procedimento irá gerar uma árvore com altura n , composta por um único caminho. Por outro lado, uma mesma árvore pode ser gerada por diferentes ordens de inserções. Por exemplo, inserir elementos

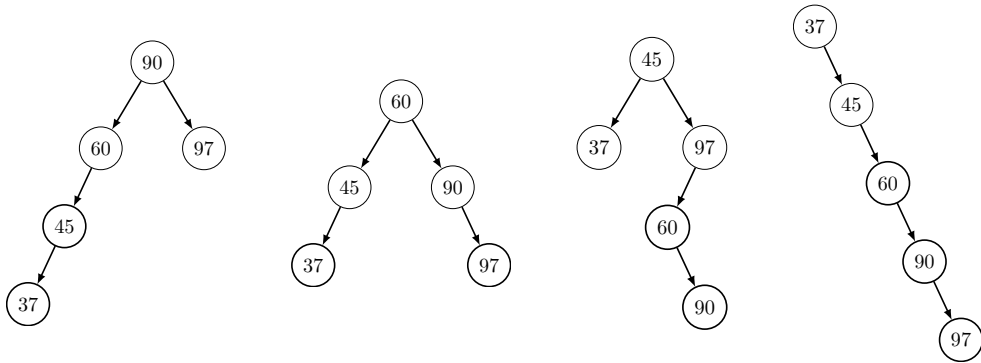


Figura 11.7: Árvores distintas formadas pela inserção dos elementos 37, 45, 60, 90 e 97 em diferentes ordens. Da esquerda para direita as ordens são (90, 60, 97, 45, 37), (60, 45, 37, 90, 97), (45, 37, 97, 60, 90), (37, 45, 60, 90, 97)

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 encadeada. 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 $n \leq 2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$. Assim, temos $2^{h+1} \geq n + 1$, de onde concluímos que (aplicando \lg dos dois lados) $h \geq \lg(n + 1) - 1$. Portanto, obtemos o seguinte resultado

Teorema 11.3

A altura de uma árvore binária com n nós é $\Omega(\lg n)$.

Uma árvore binária é dita **completa** se todos os seus níveis estão completamente preenchidos. Note que árvores binárias completas com altura h possuem exatamente $2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$ 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 $h = \lg(n + 1) - 1 = O(\lg n)$. Uma árvore binária com altura h é dita **quase completa** se os níveis $0, 1, \dots, h - 1$ 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

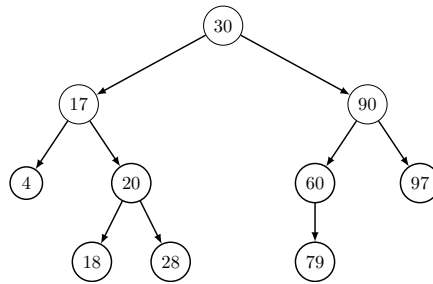


Figura 11.8: Uma árvore binária quase completa.

completa. Veja a Figura 11.8.

Como todas as operações discutidas nesta seção são executadas em tempo proporcional à altura da árvore e vimos que qualquer árvore binária tem altura $\Omega(\lg n)$, idealmente queremos árvores binárias de altura $O(\lg n)$. Na próxima seção apresentamos algumas árvores em que é possível manter essa propriedade, mesmo após qualquer sequência de inserções e remoções.

11.2 Árvores balanceadas

Árvores balanceadas são árvores de busca que garantem manter altura $O(\lg n)$ quando n elementos estão armazenados. Ademais, essa propriedade se mantém após qualquer sequência de inserções e remoções. Note que ao manter a altura $O(\lg n)$, diretamente garantimos que as operações de busca, inserção e remoção, e as operações vistas na Seção 11.1.4 levam tempo $O(\lg n)$.

Esta seção ainda está em construção.

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 ter o maior *valor* indicativo de prioridade. 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, Prim e Kruskal. Filas de prioridades podem ser implementadas de diversas formas, como por exemplo em um vetor ou lista encadeada 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 essas operações de modo eficiente. Com ela, é possível realizar inserção, remoção e alteração de um elemento em tempo $O(\lg n)$, consulta pelo elemento de maior prioridade em tempo $\Theta(1)$ e construção a partir de um conjunto já existente em tempo $O(n)$, onde n é a quantidade de elementos armazenados na estrutura.

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 $\ell + 1$. Por isso, os nós em um nível ℓ estão entre as posições 2^ℓ e $\min\{n, 2^\ell - 1\}$. Disso podemos concluir que um elemento que esteja em uma posição i de A está no nível $\lfloor \lg i \rfloor$. Como a altura da árvore é $\lfloor \lg n \rfloor$, também concluímos que um elemento na posição i tem altura $\lfloor \lg n - \lg i \rfloor = \lfloor \lg(n/i) \rfloor$.

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.

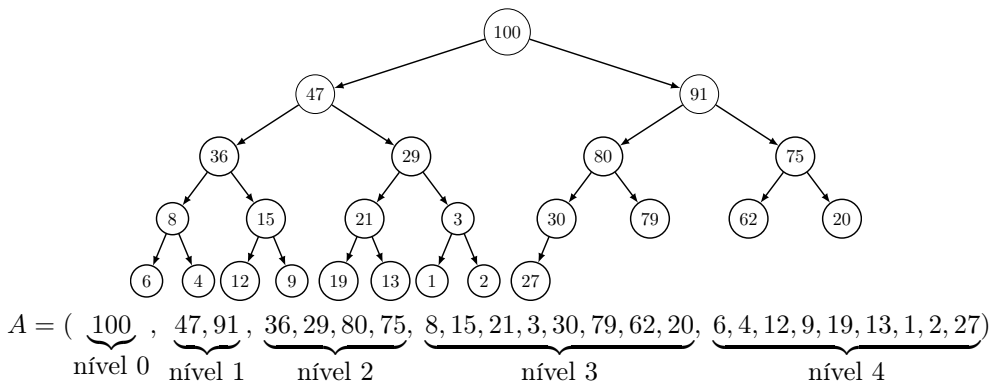


Figura 12.1: Exemplo de um vetor A observado como árvore binária. A árvore enraizada na posição 4 possui os elementos 36, 8, 15, 6, 4, 12 e 9 (que estão nas posições 4, 8, 9, 16, 17, 18 e 19 do vetor, respectivamente).

Finalmente, um **heap (binário)** é uma estrutura de dados que implementa o tipo

abstrato de dados fila de prioridades. Ele é implementado em um vetor e é bastante útil visualizá-lo como uma árvore binária como descrito acima.

No que segue vamos assumir que cada elemento x possui um atributo $x.\text{prioridade}$, que guarda o valor referente à prioridade de x , e um atributo $x.\text{indice}$, 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.

Assim, um vetor H com n elementos é um heap se, para todo i , com $2 \leq i \leq n$, temos $H[\lfloor i/2 \rfloor].\text{prioridade} \geq H[i].\text{prioridade}$, i.e., a prioridade do pai é sempre maior ou igual à prioridade de seus filhos. Considerando 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 $n = H.\text{tamanho}$ elementos em que assumimos que a quantidade máxima possível de elementos que podem ser armazenados em H está salva no atributo $H.\text{capacidade}$. 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, $\text{CONSULTAHEAP}(H)$, 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 $\text{CORRIGEHEAPDESCENDO}$ e $\text{CORRIGEHEAPSUBINDO}$, 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 $\text{CORRIGEHEAPDESCENDO}$ 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.

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  $2i \leq H.\text{tamanho}$  e  $H[2i].\text{prioridade} > H[\text{maior}].\text{prioridade}$  então
3   | maior =  $2i$ 
4 se  $2i + 1 \leq H.\text{tamanho}$  e  $H[2i + 1].\text{prioridade} > H[\text{maior}].\text{prioridade}$ 
   então
5   | maior =  $2i + 1$ 
/* Trocar maior filho com o pai, se o pai já não for o maior */
6 se maior  $\neq i$  então
7   | troca  $H[i].\text{indice}$  com  $H[\text{maior}].\text{indice}$ 
8   | troca  $H[i]$  com  $H[\text{maior}]$ 
   /* A troca pode ter estragado a subárvore enraizada em maior */
9   | CORRIGEHEAPDESCENDO( $H, \text{maior}$ )

```

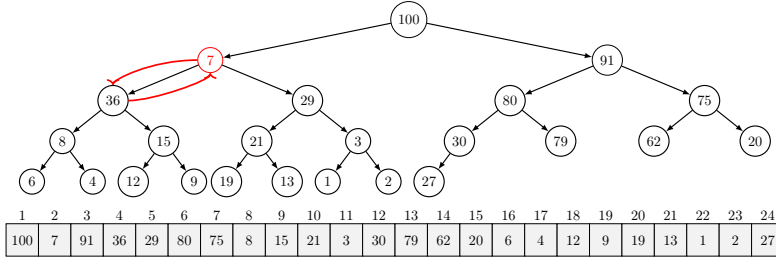
O Teorema 12.2 a seguir mostra que o CORRIGEHEAPDESCENDO de fato consegue transformar a árvore enraizada em $H[i]$ em um heap.

Teorema 12.2: Corretude de CORRIGEHEAPDESCENDO

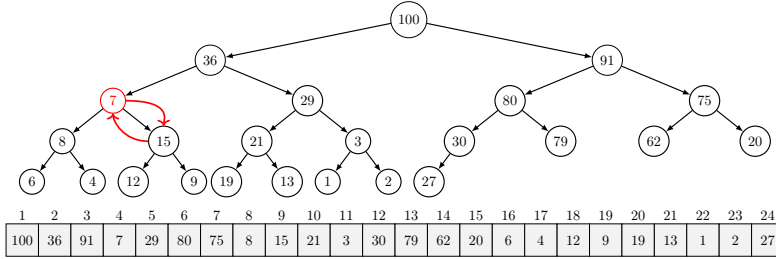
O algoritmo CORRIGEHEAPDESCENDO recebe um vetor H e um índice i tal que as subárvores enraizadas em $H[2i]$ e $H[2i+1]$ são heaps, e modifica H de modo que a árvore enraizada em $H[i]$ é um heap ao término.

Demonstração. Seja h_x a altura de um nó que está na posição x no heap (isto é, $h_x = \lfloor \lg(n/x) \rfloor$). Vamos provar o resultado por indução na altura h_i do nó i .

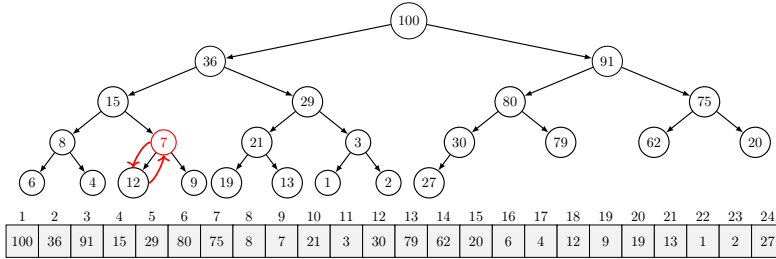
Se $h_i = 0$, o nó deve ser uma folha, que por definição é um heap (de tamanho 1). O algoritmo não faz nada, já que folhas não possuem filhos e, portanto, está correto.



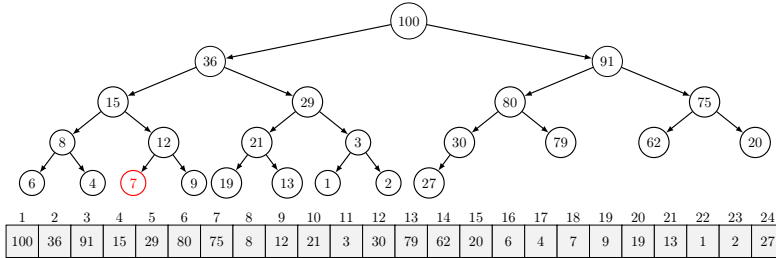
(a) $H[4] > H[2]$: troca e chama CORRIGEHEAPDESCENDO($H, 4$).



(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.

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, $O(\lg(n/i))$. 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 $O(\lg n)$ (veja Seção 11.1.4), o tempo de execução de `CORRIGEHEAPDESCENDO` também é, portanto, $O(\lg n)$.

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 $T(n) \leq 1 + \log_{3/2} n = O(\lg n)$.

Podemos também 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 $T'(n) = \Theta(\lg n)$. Portanto, $T(n) = O(\lg n)$.

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. Esse procedimento está formalizado no Algoritmo 12.2 e Figura 12.3 mostra um exemplo de execução.

O Teorema 12.3 a seguir mostra que CORRIGEHEAPSUBINDO(H, i) de fato consegue transformar o subvetor $H[1..i]$ em um heap.

Algoritmo 12.2: CORRIGEHEAPSUBINDO(H, i)

```

/* Se  $i$  tiver um pai e a prioridade do pai for menor, trocar          */
1  $pai = \lfloor i/2 \rfloor$ 
2 se  $i \geq 2$  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( $H, pai$ )

```

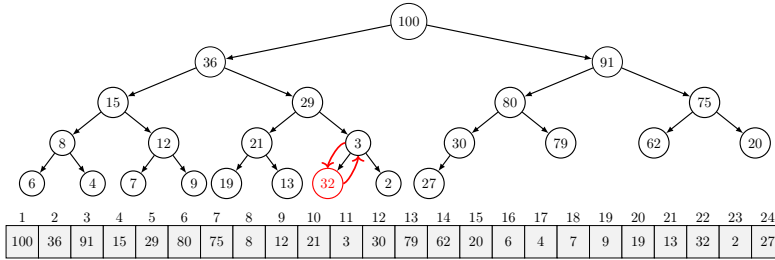
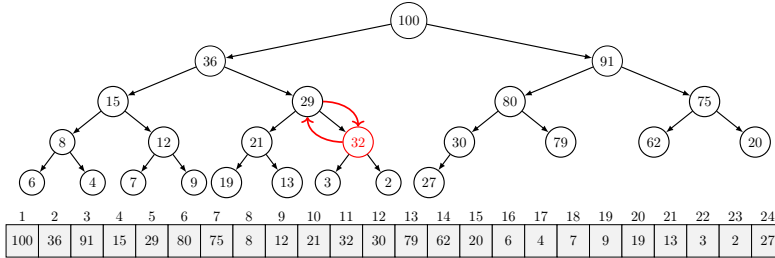
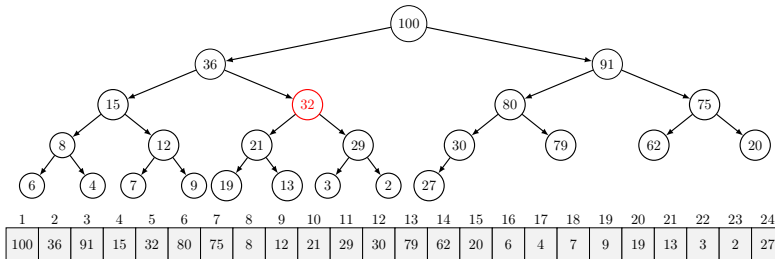
(a) $H[22] > H[11]$: troca e chama CORRIGEHEAPSUBINDO($H, 11$).(b) $H[11] > H[5]$: troca e chama CORRIGEHEAPSUBINDO($H, 5$).(c) $H[2] < H[5]$: faz nada.

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.

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 após o término.

Demonstração. Seja ℓ_x o nível de um nó que está na posição x do heap (isto é, $\ell_x = \lfloor \lg x \rfloor$). 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. \square

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 menor da 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, $O(\lg i)$. 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, $O(\lg n)$.

12.1.1 Construção de um heap binário

Suponha que temos um vetor H com capacidade total $H.\text{capacidade}$ em que suas primeiras $n = H.\text{tamanho}$ 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 na posição $\lfloor n/2 \rfloor$ é o primeiro elemento que tem filhos, e ele 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 na posição $\lfloor n/2 \rfloor - 1$ e todos os outros elementos que são pais das folhas. Com isso teremos vários heaps de altura 1, de forma que podemos aplicar o `CORRIGEHEAPDESCENDO` aos elementos que são pais desses também. A ideia do `CONSTROIHEAP` é justamente essa: aplicar `CORRIGEHEAPDESCENDO` a partir dos elementos de altura 1 até o elemento raiz, com maior altura, e isso está formalizado no Algoritmo 12.3.

Algoritmo 12.3: `CONSTROIHEAP(H, n)`

```

1  $H.\text{tamanho} = n$ 
  /* Cada elemento começa no índice em que está atualmente */
2 para  $i = 1$  até  $H.\text{tamanho}$ , incrementando faça
3    $H[i].\text{indice} = i$ 
  /* Corrigimos cada subárvore, da menor altura para a maior, a partir do
     primeiro índice que não é folha */
4 para  $i = \lfloor H.\text{tamanho} / 2 \rfloor$  até 1, decrementando faça
5    $\text{CORRIGEHEAPDESCENDO}(H, i)$ 

```

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 $H.\text{tamanho}$ e inicializando os índices. Para provar o resultado, vamos utilizar a seguinte invariante sobre o segundo laço **para** (da linha 4).

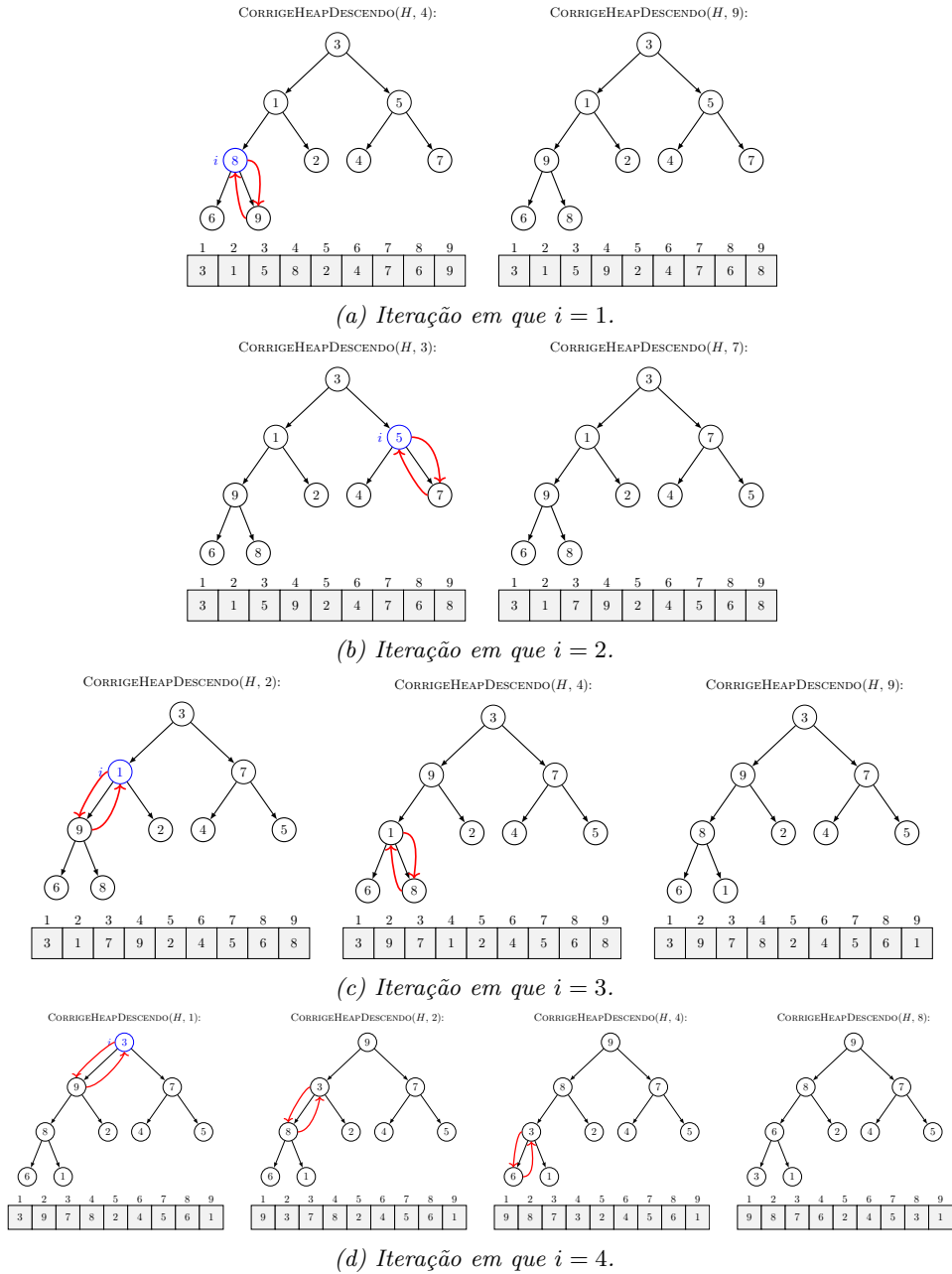


Figura 12.4: Execução de $\text{CONSTROIHEAP}(H, 9)$ (Algoritmo 12.3) sobre o vetor $H = (3, 1, 5, 8, 2, 4, 7, 6, 9)$. As setas vermelhas indicam trocas de valores.

Invariante: Segundo laço para – CONSTROIHEAP

$P(t)$ = “Antes da t -ésima iteração começar, vale que $i = \lfloor n/2 \rfloor - t + 1$ e a árvore enraizada em $H[j]$ é um heap para todo j tal que $i + 1 \leq j \leq n = H.\text{tamanho}$.”

Demonstração. Vamos provar que P é invariante por indução no número t de vezes que o teste da condição de parada é executado.

Caso base. Quando $t = 1$, o teste executou uma única vez. Antes da primeira iteração começar, fizemos $i = \lfloor n/2 \rfloor$. Como 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 $j > \lfloor n/2 \rfloor$, então o elemento $H[j]$ é folha e não tem filhos. Assim, de fato a árvore com raiz em $H[j]$ é um heap e $P(1)$ vale.

Passo. Seja $t \geq 1$ qualquer e suponha que $P(t)$ vale. Vamos mostrar que $P(t + 1)$ vale.

Como $P(t)$ vale, sabemos que no início da t -ésima iteração, $i = t$ e para todo j tal que $t + 1 \leq j \leq n$, a árvore com raiz $H[j]$ é um heap. Assim, se $H[t]$ tem filhos, então esses filhos são raízes de heaps, pois estão em posições $j > t$. Como a chamada a `CORRIGEHEAPDESCENDO(H, t)` na linha 5 funciona corretamente, ela transforma a árvore com raiz $H[t]$ em um heap. Com isso, para todo j tal que $t \leq j \leq n$, vale que a árvore com raiz $H[j]$ é um heap ao final dessa iteração e, conseqüentemente, antes do início da próxima. Então garantimos que $P(t + 1)$ vale, uma vez que i é atualizada para o valor $t - 1$.

Concluimos então que P é uma invariante de laço. \diamond

Como P é uma invariante, temos que, em particular, $P(\lfloor n/2 \rfloor + 1)$ vale, onde $\lfloor n/2 \rfloor + 1$ é a quantidade total de execuções do teste do laço. Isso nos diz que ao final do laço teremos que a árvore enraizada em $H[j]$ é um heap para todo $\lfloor n/2 \rfloor - \lfloor n/2 \rfloor + 1 = 1 \leq j \leq n$. Então, em particular, $H[1]$ é um heap. \square

No que segue seja $T(n)$ o tempo de execução de `CONSTROIHEAP` em um vetor H com n elementos. Uma análise rápida nos permite concluir que $T(n) = O(n \lg n)$, pois o laço **para** é executado $n/2$ vezes e, em cada uma dessas execuções, a rotina `CORRIGEHEAPDESCENDO`, que leva tempo $O(\lg n)$, é executada. Logo, concluímos que $T(n) = O(n \lg n)$.

Uma análise mais cuidadosa, no entanto, fornece um limitante melhor que $O(n \lg n)$.

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 $1 \leq h \leq \lfloor \lg n \rfloor$ e suponha que existem no máximo $\lceil n/2^h \rceil$ elementos com altura $h - 1$. Note que na altura h existe a 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 para alguma constante $C > 0$. Portanto, o tempo de execução $T(n)$ de CONS-TROIHEAP é tal que

$$T(n) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil Ch = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n/2^h}{2} \right\rceil Ch \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} Ch = Cn \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \leq 4Cn,$$

de onde vemos que $T(n) = O(n)$. Na última inequação acima, usamos o seguinte fato.

Seja $S = \sum_{i=0}^m \frac{i}{2^i} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{m}{2^m}$. Então $\frac{S}{2} = \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \cdots + \frac{m}{2^{m+1}}$, de

onde temos que $\frac{S}{2} = S - \frac{S}{2} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^m} - \frac{m}{2^{m+1}}$. Multiplicando essa última equação por 2, temos que

$$S = 2 \left(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^m} \right) - 2 \frac{m}{2^{m+1}} \leq 2 \left(\frac{1 - (\frac{1}{2})^{m+1}}{1 - \frac{1}{2}} \right) = 4 \left(1 - \left(\frac{1}{2} \right)^{m+1} \right) \leq 4.$$

12.1.2 Inserção em um heap binário

Para inserir um novo elemento x em um heap H , primeiro verificamos se há espaço em H para isso, lembrando que H comporta no máximo $H.\text{capacidade}$ elementos. Se sim, então inserimos x na primeira posição disponível, $H[H.\text{tamanho} + 1]$, o que potencialmente destruirá a propriedade de heap. No entanto, como $H[1..H.\text{tamanho}]$ já era heap, podemos simplesmente fazer uma chamada a CORRIGEHEAPSUBINDO a partir da posição $H.\text{tamanho} + 1$ para restaurar a propriedade em $H[1..H.\text{tamanho} + 1]$.

O Algoritmo 12.4 formaliza essa ideia, do procedimento INSERENAHEAP. Ele recebe um elemento x novo (que, portanto, tem atributos $x.\text{prioridade}$ e $x.\text{indice}$).

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( $H, H.tamanho$ )

```

Como CORRIGEHEAPSUBINDO($H, H.tamanho$) é executado em tempo $O(\lg n)$, com $n = H.tamanho$, o tempo de execução de INSERENAHEAP também é $O(\lg n)$.

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 remover $H[1]$ sem modificar muito a estrutura trocando $H[1]$ com $H[H.tamanho]$, 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  $x = \text{NULO}$ 
2 se  $H.tamanho \geq 1$  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 $O(\lg n)$, onde $n = H.tamanho$. Logo, o tempo de execução de REMOVEDAHEAP(H) é $O(\lg n)$ também.

12.1.4 Alteração em um heap binário

Ao alterarmos a prioridade de um elemento armazenado em um 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 $O(\lg n)$, onde $n = H.tamanho$.

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, \dots, A_k é uma *partição* de uma coleção B se $A_i \cap A_j = \emptyset$ para todo $i \neq j$ e $\cup_{i=1}^k A_i = B$. Todo grupo possui um **representante**, que é algum membro do grupo o identifica. Um *disjoint set* fornece operações de criação de um novo grupo (*make*), união de dois grupos existentes (*union*) e busca pelo grupo que contém um determinado elemento (*find*).

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. Cada nó da árvore aponta para um único elemento, que é seu pai na árvore. Assim, a criação de um novo grupo pode ser feita gerando-se uma árvore com apenas um nó, que é seu próprio representante. A operação de união pode ser feita fazendo com que a raiz de uma árvore aponte para a raiz da outra. A busca pelo grupo que contém um elemento pode ser feita percorrendo o caminho a partir daquele elemento até que se chegue à raiz. Perceba que as duas primeiras operações são eficientes, podendo ser realizadas em tempo constante, mas a operação de *find* pode levar tempo $O(n)$ se a sequência de operações de união que construiu uma árvore criar uma estrutura linear com n nós.

Nessa representação por árvores, duas ideias importantes auxiliam a melhorar o tempo total das operações: *união por rank* e *compressão de caminhos*. Com elas, o tempo de pior caso de execução é $O(m\alpha(n))$, onde n é o número total de elementos nos grupos,

m é o número total de operações *make*, *union* e *find* feitas e $\alpha(x)$ é uma função que cresce bem devagar conforme x cresce ($\alpha(x) \leq 4$ para valores práticos de x). A implementação que veremos na seção a seguir não garante um tempo de execução tão bom, sendo $O(m + n \lg n)$ seu pior caso, porém esse tempo é bom o suficiente para os propósitos deste livro.

13.1 Union-Find

Chamaremos de ***union-find*** a estrutura que apresentaremos nessa seção, que implementa as três operações de *disjoint set* mencionadas. Formalmente, teremos disponíveis as operações:

- **MAKESET**(x): cria um grupo novo contendo somente o elemento x ;
- **FINDSET**(x): devolve o representante do grupo que contém x ;
- **UNION**(x, y): une os grupos que contêm os elementos x e y .

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 . Assim, se x não é representante do seu próprio grupo, o conteúdo de x .**tamanho** é irrelevante. 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 . Novamente, o conteúdo em $L[x]$ é irrelevante se x não é um representante. 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, pois ela apenas 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 detalhes no Algoritmo 13.2.

Algoritmo 13.2: UNION(x, y)

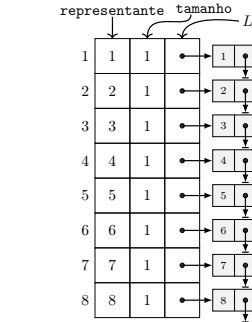
```

1   $X = \text{FINDSET}(x)$ 
2   $Y = \text{FINDSET}(y)$ 
3  se  $X.\text{tamanho} < Y.\text{tamanho}$  então
    /* Os elementos do grupo menor precisam ter seu representante
       modificado */
4    para todo  $v$  em  $L[X]$  faça
5       $v.\text{representante} = Y$ 
    /* Os tamanhos dos grupos envolvidos precisam ser atualizados */
6     $Y.\text{tamanho} = X.\text{tamanho} + Y.\text{tamanho}$ 
7     $X.\text{tamanho} = 0$ 
    /* Concatenamos a lista do trupo menor no fim da lista do grupo
       maior */
8     $L[Y].\text{cauda}.\text{prox} = L[X].\text{cabeca}$ 
9     $L[Y].\text{cauda} = L[X].\text{cauda}$ 
10    $L[X].\text{cabeca} = \text{NULO}$ 
11 senão
12   para todo  $v$  em  $L[Y]$  faça
13      $v.\text{representante} = X$ 
14    $X.\text{tamanho} = X.\text{tamanho} + Y.\text{tamanho}$ 
15    $Y.\text{tamanho} = 0$ 
16    $L[X].\text{cauda}.\text{prox} = L[Y].\text{cabeca}$ 
17    $L[X].\text{cauda} = L[Y].\text{cauda}$ 
18    $L[Y].\text{cabeca} = \text{NULO}$ 

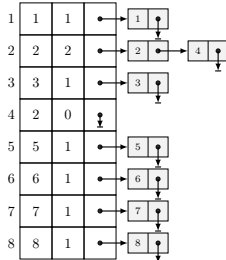
```

Perceba que graças à manutenção das listas encadeadas 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, 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 $t = \min\{x.\text{representante}.\text{tamanho}, y.\text{representante}.\text{tamanho}\}$.

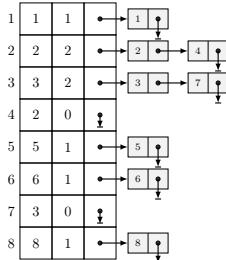
A Figura 13.1 apresenta um exemplo de operações de união feitas sobre um conjunto de dados considerando a implementação dada acima.



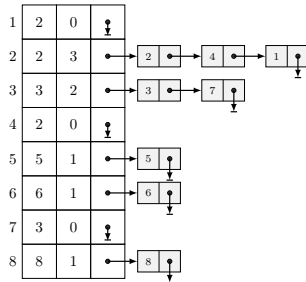
(a) MAKESET sobre todos os elementos.



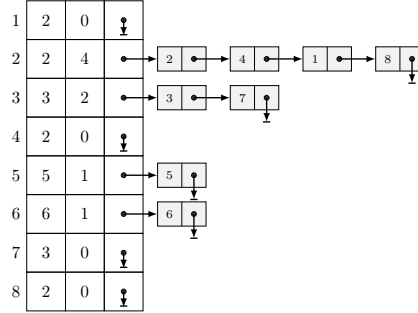
(b) UNION(2, 4).



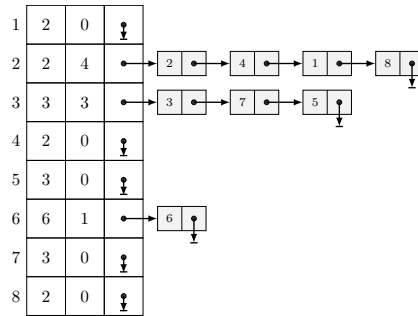
(c) UNION(7, 3).



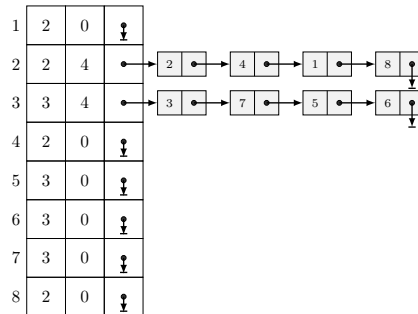
(d) UNION(1, 2).



(e) UNION(4, 8).



(f) UNION(3, 5).



(g) UNION(5, 6).

Figura 13.1: Execução de algumas chamadas a UNION (Algoritmo 13.2) sobre a coleção $\{1, 2, 3, 4, 5, 6, 7, 8\}$.

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 $O(\lg n)$, mas inserções e remoções passam a ser custosas. Uma outra opção é usar uma árvore binária de busca balanceada, que garante tempo $O(\lg n)$ 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 sendo o tamanho m da tabela bem menor do que M , não importa como é 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. Ele consiste em, dados alguns elementos, rearranjá-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. Apesar de existirem inúmeros algoritmos de ordenação, veremos apenas os mais clássicos nas seções a seguir, considerando a seguinte definição do problema.

Problema 14.1: Ordenação

Entrada: $\langle A, n \rangle$, onde $A = (a_1, a_2, \dots, a_n)$ é um vetor com n números.

Saída: Permutação $(a'_1, a'_2, \dots, a'_n)$ dos números de A de modo que $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

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 nossas discussões.

Note que estamos considerando apenas um vetor numérico, mas poderíamos ter um vetor que 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), como por exemplo o campo **chave** que consideramos nos algoritmos de busca.

Duas características são importantes de serem observadas em algoritmos de ordenação. 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 de alguns algoritmos importantes nas seções a seguir.

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* é certamente familiar a muitas pessoas, por ser a forma mais comum com que ordenamos cartas de baralho. 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 recebe 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 ordenado 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 então 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. Como descrevemos no funcionamento do algoritmo, a ideia dele é que a qualquer momento o vetor $A[1..i-1]$ já está em ordem não-decrescente e precisamos apenas colocar o elemento $A[i]$ no local correto. Intuitivamente, o que o laço **enquanto** precisa fazer, portanto, é “arrastar” uma posição para a direita os elementos que estão em $A[1..i-1]$ e que são maiores do que $A[i]$, mas sem alterar a ordem relativa entre eles e sem perder nenhum elemento que já estava ali. Apenas assim garantiremos que $A[1..i]$ contém os mesmos elementos que já estavam ali originalmente porém ordenados. E aumentando o valor de i a cada momento, eventualmente teremos que essa característica vale para $A[1..n]$, que é o que realmente queremos. Veremos a formalização dessas ideias a seguir, que pode até parecer exagerada quando comparada a essa intuição, porém é necessária para provar a corretude do INSERTIONSORT.

A primeira invariante que provaremos delas é referente ao laço **enquanto**, da linha 4.

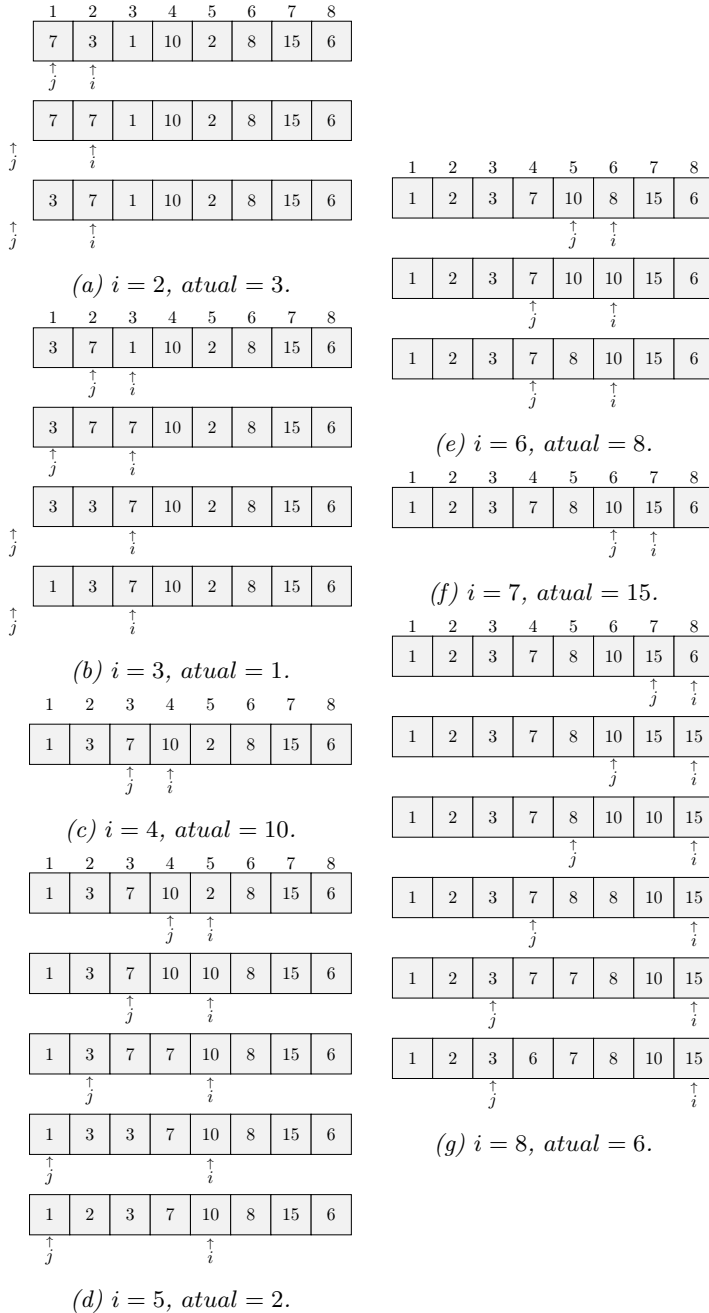


Figura 15.1: Execução de $\text{INSERTIONSORT}(A, 8)$ (Algoritmo 15.1), com $A = (7, 3, 1, 10, 2, 8, 15, 6)$.

Para facilitar a compreensão, consideraremos um vetor B que é uma cópia do vetor A no instante logo antes do laço **enquanto** começar¹.

Invariante: Laço enquanto – INSERTIONSORT

$R(y)$ = “Antes da y -ésima iteração começar, vale que $j = i - y$, os elementos contidos em $A[j + 2..i]$ são maiores do que *atual*, o subvetor $A[j + 2..i]$ é igual ao subvetor $B[j + 1..i - 1]$, e o subvetor $A[1..j]$ é igual ao subvetor $B[1..j]$.”

Vamos primeiro verificar que a frase acima realmente é uma invariante do laço **enquanto**, por indução no número y de vezes que o teste da condição de parada é executado. Quando $y = 1$, o teste executou uma única vez. Antes da primeira iteração, fazemos $j = i - 1$, de forma que $A[j + 2..i] = A[i + 1..i]$ é um vetor vazio. Além disso, B ainda é uma cópia exata de A , de forma que $R(1)$ vale trivialmente.

Seja $y \geq 1$ qualquer e suponha que $R(y)$ vale, isto é, que no início da y -ésima iteração temos $j = i - y$ e que os elementos de $A[i - y + 2..i]$ são maiores do que *atual*, o subvetor $A[i - y + 2..i]$ é igual ao subvetor $B[i - y + 1..i - 1]$, e o subvetor $A[1..i - y]$ é igual ao subvetor $B[1..i - y]$.

Vamos mostrar que $R(y+1)$ também vale. Como a y -ésima iteração começou, sabemos que as condições do laço são verdadeiras e, portanto, particularmente, $A[i - y] > \textit{atual}$. Então fazemos $A[i - y + 1] = A[i - y]$, o que faz com que $A[i - y + 1..i]$ contenha apenas elementos maiores do que *atual*. Além disso, essa iteração não alterou os elementos $A[i - y + 2..i]$ e $A[i - y + 1]$ recebeu uma cópia do elemento $A[i - y]$, que fazia parte do subvetor $B[1..i - y]$. Então o subvetor $A[i - y + 1..i]$ é igual ao subvetor $B[i - y..i - 1]$ e o subvetor $A[1..i - y + 1]$ é igual ao subvetor $B[1..i - y + 1]$. Como j é decrementado e passa a valer $i - y - 1$ ao final da iteração, acabamos de mostrar que $R(y + 1)$ é verdadeira. Assim, a frase de fato é uma invariante.

Suponha agora que o laço terminou com $j = r$, para algum $0 \leq r \leq i - 1$. Então a condição de parada foi executada $i - r$ vezes, de forma que $R(i - r)$ nos diz algo quando o laço termina. Sabemos então que os elementos contidos em $A[r + 2..i]$ são todos maiores do que *atual*, que o subvetor $A[r + 2..i]$ é igual ao subvetor $B[r + 1..i - 1]$, e que $A[1..r]$ é igual a $B[1..r]$. Vamos precisar destas informações para provar que a seguinte frase é uma invariante do laço **para** da linha 1. Também para facilitar a compreensão, consideraremos um vetor C que é uma cópia do vetor A logo antes do laço **para** começar (isto é, o próprio vetor que foi passado para o algoritmo).

¹Não confunda início do laço com início da iteração.

Invariante: Laço para – INSERTIONSORT

$P(x)$ = “Antes da x -ésima iteração começar, vale que $i = x + 1$ e o subvetor $A[1..i - 1]$ contém os mesmos elementos que o vetor $C[1..i - 1]$ porém em ordem não-decrescente.”

Vamos também provar que ela é de fato uma invariante de laço por indução no número x de vezes que o teste da condição de parada é executado. Quando $x = 1$, o teste executou uma única vez. Antes da primeira iteração, fazemos apenas $i = 2$, de forma que $A[1..i - 1] = A[1..1]$ de fato contém o mesmo elemento que $C[1]$ e em ordem não-decrescente. Portanto, $P(1)$ vale trivialmente.

Agora seja $x \geq 1$ qualquer e suponha que $P(x)$ vale. Vamos mostrar que $P(x + 1)$ também vale.

Como $P(x)$ vale, sabemos que $i = x + 1$ e que $A[1..x]$ contém os mesmos elementos de $C[1..x]$ porém em ordem não-decrescente. Observe ainda que B é igual a A neste momento. Nessa iteração, fazemos $atual = A[x + 1]$ e $j = x$. Como visto acima, sabemos que quando o laço **enquanto** termina, se r é o valor da variável j , então $A[1..r]$ é igual a $B[1..r]$ e $A[r + 2..x + 1]$ é igual a $B[r + 1..x]$.

Existem dois motivos para o laço **enquanto** ter parado. Se $r > 0$ e $A[r] \leq atual$, então como $A[1..x]$ estava em ordem não-decrescente, sabemos que todos os elementos em $A[1..r]$ são menores ou iguais a $atual$. Como os elementos em $A[r + 2..x + 1]$ são maiores do que $atual$, a operação $A[r + 1] = atual$ faz com que $A[1..x + 1]$ fique em ordem não-decrescente. Outro motivo para o laço **enquanto** terminar é que $r = 0$. Neste caso, a invariante do laço **enquanto** nos diz que $A[2..x + 1]$ é igual ao subvetor $B[1..x]$ e contém apenas elementos maiores do que $atual$. Então a operação $A[1] = atual$ faz com que $A[1..x + 1]$ fique em ordem não-decrescente. As outras garantias da invariante R terminam de mostrar que $A[1..x + 1]$ contém os elementos de $C[1..x + 1]$. Como o valor de i é atualizado para $x + 1$, acabamos de mostrar que $P(x + 1)$ vale. Logo, P é de fato invariante do laço **para**.

Mas lembre-se que nosso objetivo na verdade é mostrar que ao final da execução do algoritmo, o vetor A está ordenado. Como o laço **para** executa $n - 1$ vezes e o teste da condição de parada executa n vezes, então $P(n)$ nos diz algo ao fim da execução do laço. Ela nos diz que o subvetor $A[1..n + 1 - 1] = A[1..n]$ contém os elementos de $C[1..n]$ porém em ordem não-decrescente. Assim, 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 em ordem não-decrescente. No pior caso, o teste do laço **enquanto** é executado i vezes para cada valor de i do laço **para**, totalizando $2 + \dots + n = n(n + 1)/2 - 1 = \Theta(n^2)$ 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 entendermos 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 $H[m] = 1$.

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  $j \geq H[t]$  e  $A[j - H[t] + 1] > atual$  faça
6        $A[j + 1] = A[j - H[t] + 1]$ 
7        $j = j - H[t]$ 
8      $A[j + 1] = atual$ 
```

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, ... e ela dá bons resultados na prática e faz $O(n^{3/2})$ comparações. Uma sequência do tipo 1, 2, 4, 8, 16, ... dá resultados muito ruins, já que elementos em posições ímpares não são comparados com elementos em posições pares até a última iteração.

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.

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 $A[\text{inicio}..\text{fim}]$. Assim, para ordenar um vetor A inteiro de n posições, basta executar MERGESORT($A, 1, n$).

Algoritmo 16.1: MERGESORT($A, \text{inicio}, \text{fim}$)

```
1 se  $\text{inicio} < \text{fim}$  então
2    $\text{meio} = \lfloor (\text{inicio} + \text{fim}) / 2 \rfloor$ 
3   MERGESORT( $A, \text{inicio}, \text{meio}$ )
4   MERGESORT( $A, \text{meio} + 1, \text{fim}$ )
5   COMBINA( $A, \text{inicio}, \text{meio}, \text{fim}$ )
```

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.

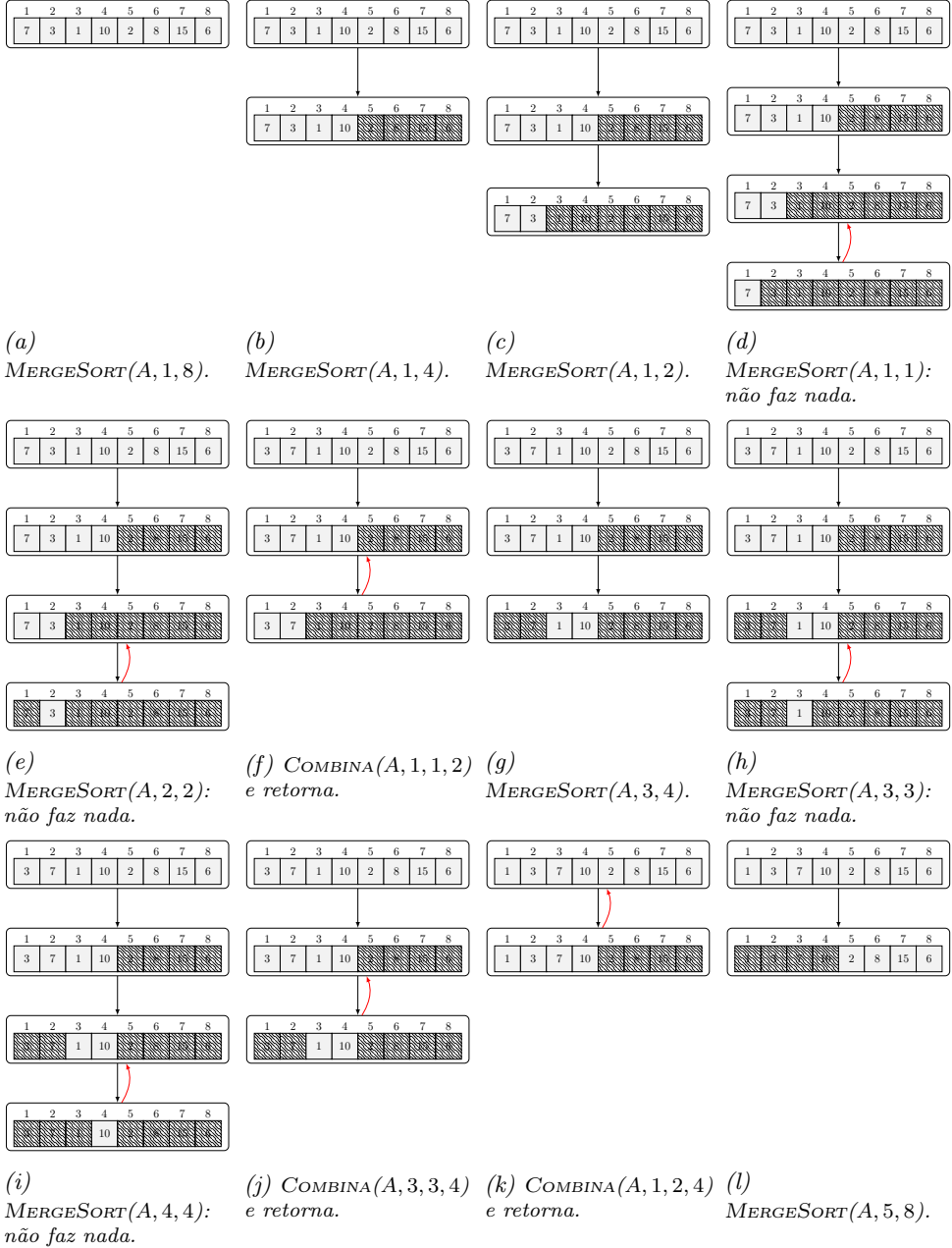


Figura 16.1: Parte 1 da execução de $\text{MERGESORT}(A, 1, 8)$ (Algoritmo 16.1) para $A = (7, 3, 1, 10, 2, 8, 15, 6)$.

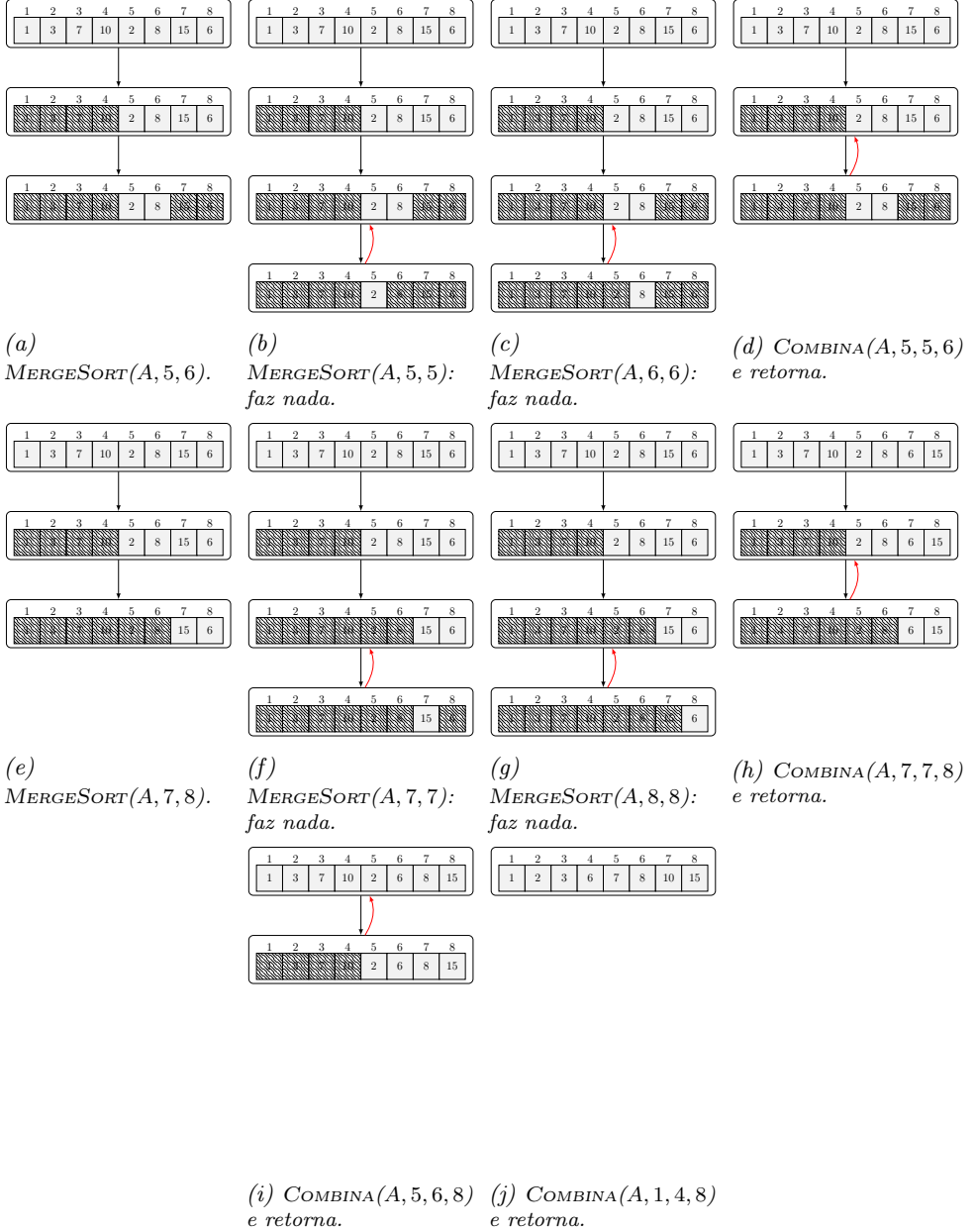


Figura 16.2: Parte 2 da execução de $MERGESORT(A, 1, 8)$ (Algoritmo 16.1) para $A = (7, 3, 1, 10, 2, 8, 15, 6)$.

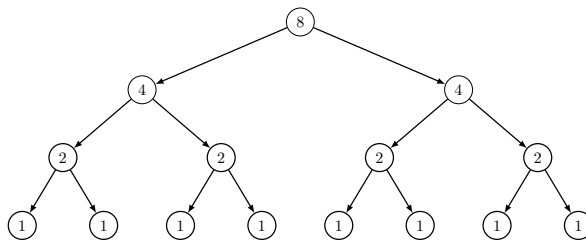


Figura 16.3: Árvore de recursão completa de MERGESORT($A, 1, 8$). Cada nó é rotulado com o tamanho do problema ($fim - inicio + 1$) correspondente.

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 $A[inicio..fim]$ 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 de os 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 copiados de B , um índice j para acessar elementos a serem copiados de 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.

Algoritmo 16.2: COMBINA(A , $inicio$, $meio$, fim)

```

1  $n_1 = meio - inicio + 1$  /* Qtd. de elementos em  $A[inicio..meio]$  */
2  $n_2 = fim - meio$  /* Qtd. de elementos em  $A[meio + 1..fim]$  */
3 Crie vetores auxiliares  $B[1..n_1]$  e  $C[1..n_2]$ 
4 para  $i = 1$  até  $n_1$ , incrementando faça
   | /* Copiando o conteúdo de  $A[inicio..meio]$  para  $B$  */
5   |  $B[i] = A[inicio + i - 1]$ 
6 para  $j = 1$  até  $n_2$ , 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  $i \leq n_1$  e  $j \leq n_2$  faça
   | /* Copia o menor dentre  $B[i]$  e  $C[j]$  para  $A[k]$  */
12   | se  $B[i] \leq C[j]$  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  $i \leq n_1$  faça
   | /* Termina de copiar elementos de  $B$ , se houver */
20   |  $A[k] = B[i]$ 
21   |  $i = i + 1$ 
22   |  $k = k + 1$ 
23 enquanto  $j \leq n_2$  faça
   | /* Termina de copiar elementos de  $C$ , se houver */
24   |  $A[k] = C[j]$ 
25   |  $j = j + 1$ 
26   |  $k = k + 1$ 

```

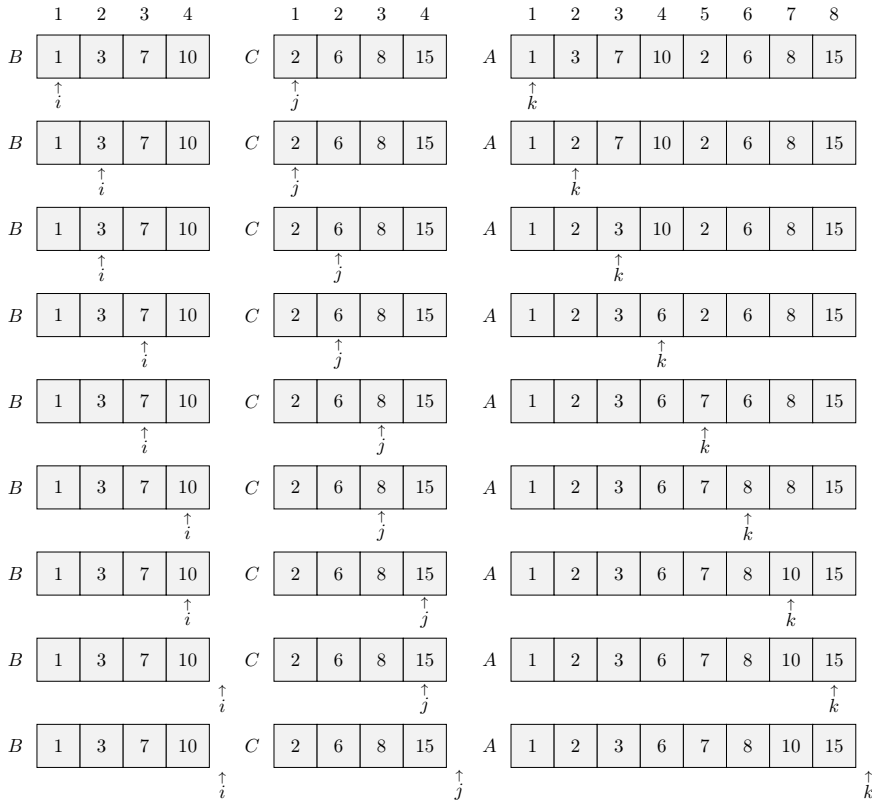


Figura 16.4: Execução de $\text{COMBINA}(A, 1, 4, 8)$ (Algoritmo 16.2) sobre o vetor $A = (1, 3, 7, 10, 2, 6, 8, 15)$.

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 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 usando a seguinte invariante de laço.

Invariante: Primeiro laço enquanto – COMBINA

$P(t)$ = “Antes da t -ésima iteração começar, vale que $k = inicio + t - 1$, o vetor $A[inicio..k - 1]$ está ordenado, e os elementos de $B[i..n_1]$ e $C[j..n_2]$ são maiores ou iguais aos elementos de $A[inicio..k - 1]$.”

Demonstração. Vamos provar que P é uma invariante por indução no número t de vezes que o teste da condição de parada é executado. Quando $t = 1$, o teste executou uma vez. Antes da primeira iteração, fazemos apenas $i = 1$, $j = 1$ e $k = inicio = inicio + t - 1$. Além disso, certamente $A[inicio..inicio - 1]$, que é vazio, está ordenado e seus elementos são menores ou iguais aos elementos de $B[1..n_1]$ e $C[1..n_2]$.

Agora seja $t \geq 1$ qualquer e suponha que $P(t)$ vale. Vamos mostrar que $P(t + 1)$ também vale. Como $P(t)$ vale, sabemos que $k = inicio + t - 1$, que $A[inicio..inicio + t - 2]$ está ordenado, e os elementos de $B[i..n_1]$ e $C[j..n_2]$ são maiores ou iguais aos elementos de $A[inicio..inicio + t - 2]$. Nessa iteração, apenas uma dentre duas coisas pode 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 $A[inicio + t - 1]$. Como $P(t)$ vale, $B[i]$ é maior do que os elementos que estão em $A[inicio..inicio + t - 2]$, de forma que $A[inicio..inicio + t - 1]$ fica em ordem ao fim da iteração. Como $B[i] \leq C[j]$ e B e C estão em ordem, então os elementos em $B[i + 1..n_1]$ e $C[j..n_2]$ são maiores do que os elementos em $A[inicio..inicio + t - 1]$. Como incrementamos apenas os valores das variáveis i e k , acabamos de mostrar que $P(t + 1)$ vale.

Para o caso em que $B[i] > C[j]$, uma análise similar nos mostra que $P(t+1)$ também vale. Nesse caso, note que incrementamos apenas j e k . \diamond

Vamos então utilizar essa invariante para mostrar que o algoritmo COMBINA está correto. Quando o laço **enquanto** da linha 11 termina, suponha que k_f , i_f e j_f são os valores de k , i e j , respectivamente. Então o teste do laço executou k_f vezes e $P(k_f)$ nos dá informações ao final da execução do laço. Ela 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 $A[inicio..k_f - 1]$. Além disso, note que o laço só pode ter acabado porque $i_f = n_1 + 1$ ou $j_f = n_2 + 1$, 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, 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. \square

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, o que pode ser observado pela quantidade de valores diferentes que k assume. Assim, 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 $n = fim - inicio + 1 = 0$ implica em $inicio = fim + 1$, o que por sua vez implica em $inicio > fim$. Quando $n = 1$, $n = fim - inicio + 1 = 1$ 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 o algoritmo faz é calcular a posição $meio = \lfloor (inicio + fim)/2 \rfloor$. Em seguida, faz uma chamada $MERGESORT(A, inicio, meio)$, isto é, uma chamada passando um vetor com $meio - inicio + 1$ elementos. Veja que

$$\begin{aligned} meio - inicio + 1 &= \left\lfloor \frac{inicio + fim}{2} \right\rfloor - inicio + 1 \\ &\leq \frac{inicio + fim}{2} - inicio + 1 \\ &= \frac{fim - inicio + 2}{2} = \frac{n + 1}{2}. \end{aligned}$$

Além disso, $(n + 1)/2 < n$ sempre que $n > 1$. Assim, a chamada $MERGESORT(A, inicio, meio)$ é de fato sobre um vetor de tamanho menor do que o tamanho inicial e, por hipótese, corretamente ordena $A[inicio..meio]$.

Em seguida, outra chamada recursiva é feita, $MERGESORT(A, meio + 1, fim)$, 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 $n > 1$. Portanto 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 acabamos de ver. Logo, $A[inicio..fim]$ termina totalmente ordenado. \square

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 aproximadamente 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, assintoticamente, no resultado obtido. Também podemos desconsiderar pisos e tetos¹, 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, podemos usar o método Mestre, como visto no Capítulo 8, para provar que o tempo de execução de MERGESORT é $T(n) = \Theta(n \lg n)$.

¹Mais detalhes sobre isso podem ser vistos na Seção 8.1.3.

Ordenação por seleção

Neste capítulo vamos introduzir dois algoritmos para o problema 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 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 a 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 do subvetor $A[1..i]$ é encontrado e colocado na posição i , aumentando o subvetor ordenado em uma unidade¹.

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 à direita da posição i ,

¹É possível adaptar 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. Assim, a cada iteração, o menor elemento do subvetor $A[i..n]$ deve ser encontrado e colocado na posição i .

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.

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 para qualquer vetor A de tamanho 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. Para obter esse resultado, vamos precisar da seguinte invariante de laço, que é referente ao laço **para** mais interno, da linha 3.

Invariante: Segundo laço para – SELECTIONSORT

$R(y)$ = “Antes da y -ésima iteração, temos que $j = y$ e que $A[iMax]$ é maior ou igual a qualquer elemento em $A[1..j - 1]$ e em $A[i]$.”

Demonstração. Vamos provar que R é uma invariante por indução no número y de vezes que o teste da condição de parada é executado. Quando $y = 1$ e o teste executou uma única vez, temos $j = 1$ e $iMax = i$. De fato, podemos dizer que $A[i]$ é maior ou igual a qualquer elemento em $A[1..0]$, que é vazio, e o próprio $A[i]$. Assim, $R(1)$ é verdade.

Agora seja $y \geq 1$ e suponha que $R(y)$ vale. Precisamos provar que $R(y+1)$ também vale. Como $R(y)$ vale, então sabemos que antes da y -ésima iteração começar temos $j = y$ e que $A[iMax]$ é maior ou igual a qualquer elemento em $A[1..y - 1]$ e em $A[i]$. Durante a iteração, uma dentre duas coisas pode ocorrer.

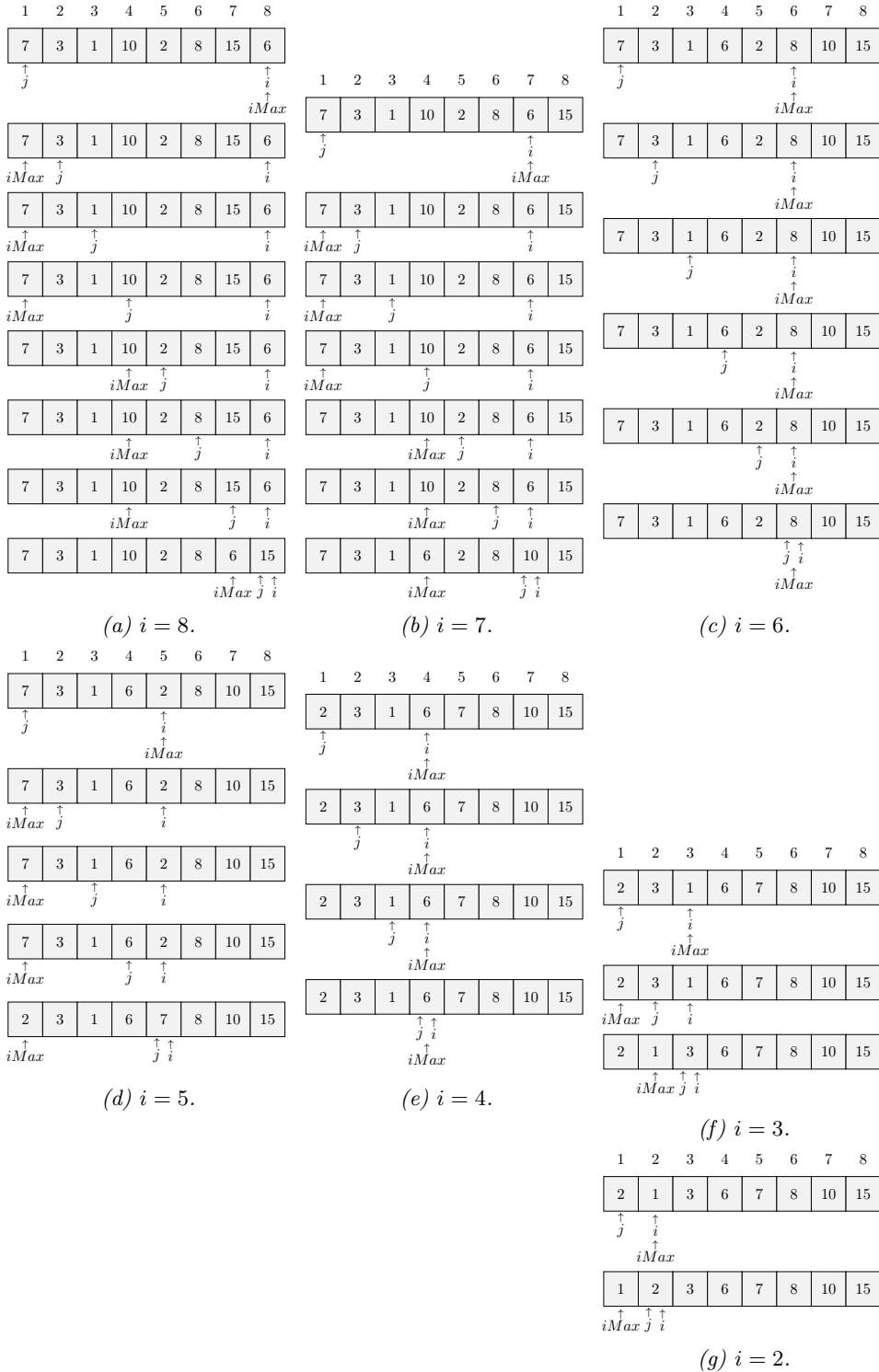


Figura 17.1: Execução de `SELECTIONSORT(A, 8)` (Algoritmo 17.1), com $A = (7, 3, 1, 10, 2, 8, 15, 6)$.

Se $A[y] > A[iMax]$, então atualizaremos a variável $iMax$ para ter valor igual a y . Note que como $A[iMax]$ era maior ou igual a todos os elementos em $A[1..y-1]$ no início da iteração, então ao fim teremos que $A[iMax] = A[y]$ é maior ou igual aos elementos em $A[1..y]$ e em $A[i]$. Por outro lado, se $A[y] \leq A[iMax]$, então o valor da variável $iMax$ não muda e também temos garantia de que $A[iMax]$ é maior ou igual aos elementos em $A[1..y]$ e em $A[i]$. Como o valor de j incrementa em uma unidade, acabamos de mostrar que $R(y+1)$ é válida e, portanto, a frase é de fato uma invariante. \diamond

Como o teste do laço **para** interno executa i vezes, ao final do laço temos que $R(i)$ vale. Ela nos diz então que $A[iMax]$ é maior ou igual a qualquer elemento em $A[1..i-1]$ e em $A[i]$. Usaremos este fato em 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 x -ésima iteração, vale que $i = n - x + 1$ e que o subvetor $A[i+1..n]$ está ordenado de modo não-decrescente e contém os $n - i$ maiores elementos de A .”

Demonstração. Vamos provar que P é uma invariante por indução no número x de vezes que o teste da condição de parada é executado. Quando $x = 1$ e o teste executou uma única vez, apenas fizemos $i = n$ e de fato o subvetor $A[n+1..n]$, vazio, está ordenado com os 0 maiores elementos de A . Portanto, $P(1)$ vale trivialmente.

Agora seja $x \geq 1$ e suponha que $P(x)$ vale. Precisamos provar que $P(x+1)$ vale também. Como $P(x)$ vale, então sabemos que na x -ésima iteração temos $i = n - x + 1$ e que o subvetor $A[n - x + 2..n]$ está ordenado de modo não-decrescente e contém os $n - (n - x + 1) = x - 1$ maiores elementos de A .

Durante a iteração, fazemos $iMax = i = n - x + 1$ e executamos o laço interno. Pela sua invariante de laço, sabemos que no seu término o valor em $A[iMax]$ é maior ou igual a qualquer elemento em $A[1..i] = A[1..n - x + 1]$. Na linha 6, trocamos tal valor com o elemento $A[n - x + 1]$. Agora então sabemos que $A[n - x + 1..n]$ está ordenado e contém os x maiores elementos de A , mostrando assim que $P(x+1)$ vale, uma vez que o valor de i é decrementado de uma unidade. \diamond

Como o teste do laço **para** externo executa n vezes, ao final de sua execução temos que $P(n)$ vale. Ela nos diz então que o vetor $A[(n - n + 1) + 1..n] = A[2..n]$ está

ordenado e contém os $n - (n - n + 1) = n - 1$ maiores elementos de A . Só pode ser o caso então de $A[1]$ conter o menor elemento de A , o que nos permite concluir o vetor $A[1..n]$ está ordenado ao fim da execução do algoritmo. \square

Agora que sabemos que o algoritmo está correto, vamos analisar seu tempo de execução. Note que cada linha de `SELECTIONSORT(A, n)`, individualmente pode ser executada em tempo constante. As linhas 1, 2 e 6 executam $\Theta(n)$ vezes 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 de o *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 $\Theta(n \lg n)$, 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`, que foram definidos e detalhados na Seção 12.1. O primeiro recebe um vetor H e uma posição i tal que as subárvores enraizadas em $H[2i]$ e $H[2i + 1]$ já são *heaps* e tem como objetivo transformar a árvore enraizada em $H[i]$ em um *heap*. O segundo recebe um vetor H qualquer de tamanho $H.tamanho$ e o transforma em um *heap*. Consideraremos aqui que os valores armazenados no vetor H 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.

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 $A[n]$, o que acaba colocando o maior elemento já em sua posição final (onde ele deve ficar após

Algoritmo 17.2: CORRIGEHEAPDESCENDO(H, i)

```

1 maior = i
2 se  $2i \leq H.tamanho$  e  $H[2i] > H[maior]$  então
3   | maior =  $2i$ 
4 se  $2i + 1 \leq H.tamanho$  e  $H[2i + 1] > H[maior]$  então
5   | maior =  $2i + 1$ 
6 se maior  $\neq i$  então
7   | troca  $H[i]$  com  $H[maior]$ 
8   | CORRIGEHEAPDESCENDO( $H, maior$ )

```

Algoritmo 17.3: CONSTROIHEAP(H)

```

1 para  $i = \lfloor H.tamanho / 2 \rfloor$  até 1, decrementando faça
2   | CORRIGEHEAPDESCENDO( $H, i$ )

```

a ordenação terminar). 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 similar, $A[1]$ contém o maior elemento de $A[1..n - 1]$ e, portanto, podemos repetir o mesmo procedimento acima, trocando $A[1]$ com $A[n - 1]$. Descrevemos formalmente o procedimento HEAPSORT no Algoritmo 17.4. Lembre-se que $A.tamanho$ é a quantidade de elementos armazenados em A , isto é, inicialmente $A.tamanho = n$. 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.

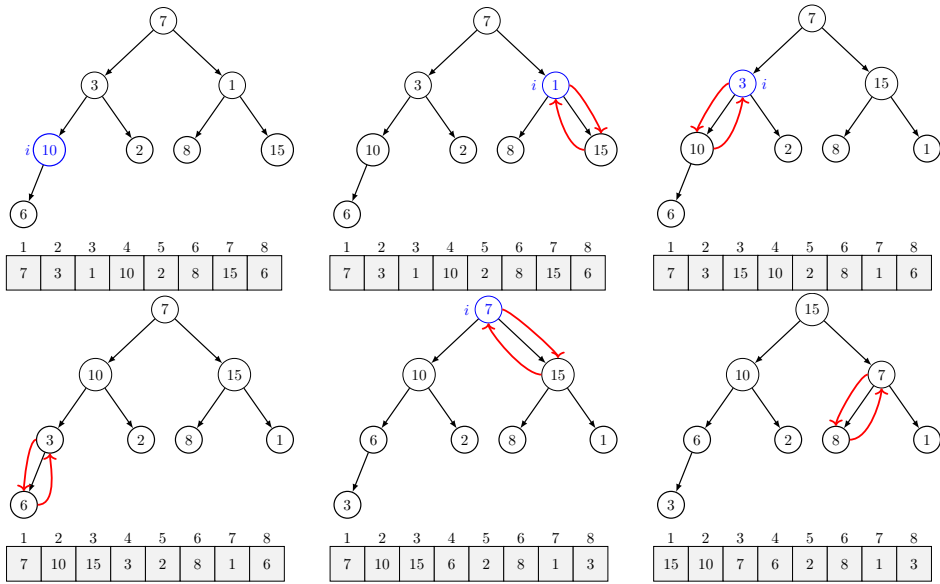


Figura 17.2: Parte 1 da execução de $\text{HEAPSORT}(A, 8)$ (Algoritmo 17.4), com $A = (7, 3, 1, 10, 2, 8, 15, 6)$: chamada a $\text{CONSTROIHEAP}(A)$.

Demonstração. Para provar esse resultado vamos utilizar a seguinte invariante para o laço **para** do algoritmo.

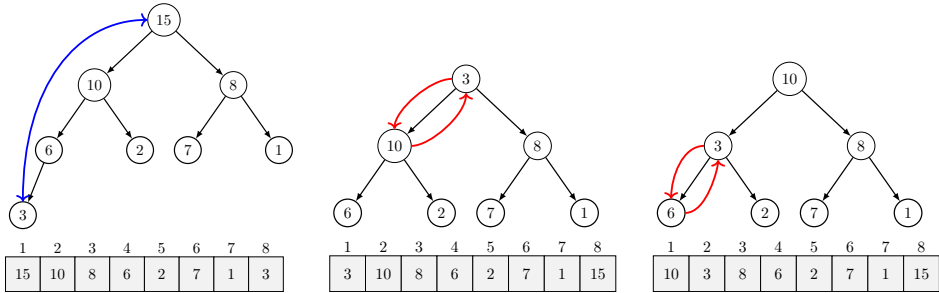
Invariante: Laço para – HEAPSORT

$P(t)$ = “Antes da t -ésima iteração começar, temos que $i = n - t + 1$ e que:

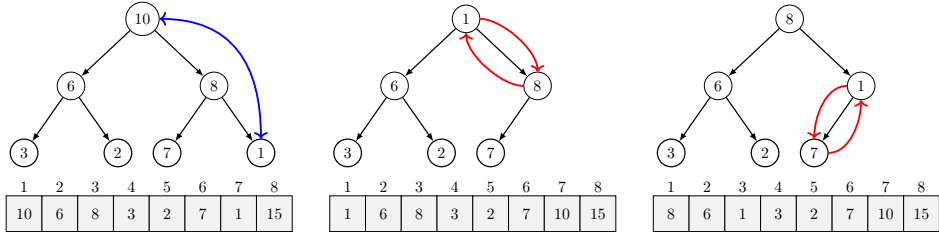
- o vetor $A[i + 1..n]$ está ordenado de modo não-decrescente e contém os $n - i$ maiores elementos de A ;
- $A.\text{tamanho} = i$ e o vetor $A[1..A.\text{tamanho}]$ é um heap.”

Demonstração. Vamos provar que P é uma invariante por indução no número t de vezes que o teste da condição de parada é executado. Quando $t = 1$ e o teste executou uma única vez, apenas fizemos $i = n$ e de fato o subvetor $A[n + 1..n]$, vazio, está ordenado com os 0 maiores elementos de A . Além disso, CONSTROIHEAP está correto e, portanto, A é um heap e $A.\text{tamanho} = n$. Portanto, $P(1)$ vale.

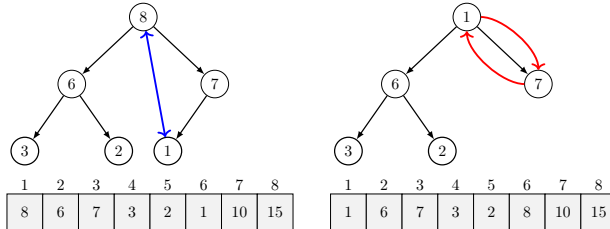
Agora seja $t \geq 1$ e suponha que $P(t)$ vale. Precisamos provar que $P(t + 1)$ vale também. Como $P(t)$ vale, então sabemos que na t -ésima iteração temos $i = n - t + 1$, que o subvetor $A[n - t + 2..n]$ está ordenado de modo não-decrescente com os $n - (n - t + 1) = t - 1$ maiores elementos de A , e que $A.\text{tamanho} = n - t + 1$



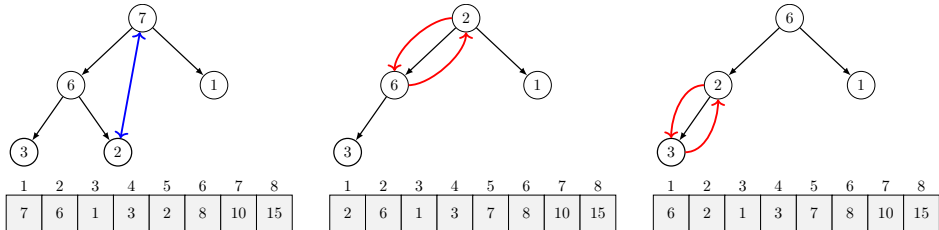
(a) Iteração $i = 8$. Troca $A[1]$ com $A[8]$, diminui heap e corrige descendo.



(b) Iteração $i = 7$. Troca $A[1]$ com $A[7]$, diminui heap e corrige descendo.

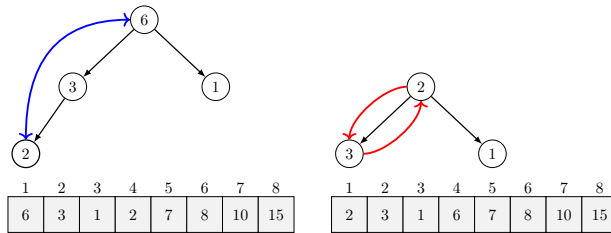


(c) Iteração $i = 6$. Troca $A[1]$ com $A[6]$, diminui heap e corrige descendo.

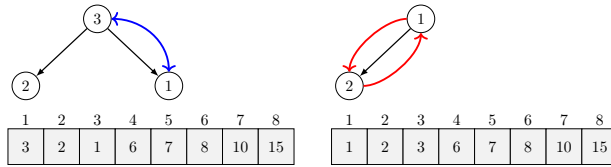


(d) Iteração $i = 5$. Troca $A[1]$ com $A[5]$, diminui heap e corrige descendo.

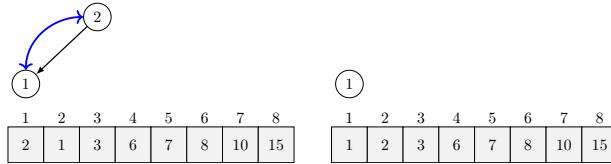
Figura 17.3: Parte 2 da execução de $\text{HEAPSORT}(A, 8)$ (Algoritmo 17.4), com $A = (7, 3, 1, 10, 2, 8, 15, 6)$: execução do laço **para**.



(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 $\text{HEAPSORT}(A, 8)$ (Algoritmo 17.4), com $A = (7, 3, 1, 10, 2, 8, 15, 6)$: laço para.

e $A[1..A.\text{tamanho}]$ é um heap.

Durante a iteração, a primeira ação é a troca de $A[1]$ com $A[n - t + 1]$, colocando portanto o maior elemento de $A[1..n - t + 1]$ em $A[n - t + 1]$. Em seguida, diminui-se o valor de $A.\text{tamanho}$ em uma unidade, fazendo com que $A.\text{tamanho} = n - t$. Por fim, chama-se $\text{CORRIGEHEAPDESCENDO}(A, 1)$, transformando $A[1..n - 1]$ em heap, pois o único elemento de $A[1..A.\text{tamanho}]$ que pode não satisfazer a propriedade de heap é $A[1]$ e sabemos que $\text{CORRIGEHEAPDESCENDO}(A, 1)$ funciona corretamente. Como o maior elemento de $A[1..n - t + 1]$ agora está em $A[n - t + 1]$ e dado que sabemos que $A[n - t + 2..n]$ estava ordenado de modo não-decrescente com os $t - 1$ maiores elementos de A (porque $P(t)$ é verdadeira), concluímos que agora o vetor $A[n - t + 1..n]$ está ordenado de modo não-decrescente e contém os t maiores elementos de A ao fim da iteração. Como i é decrementado de uma unidade, passando a valer $n - t$, acabamos de mostrar, portanto, que $P(t + 1)$ vale e, portanto, a frase é uma invariante. \diamond

Como o teste do laço **para** executa n vezes, ao final de sua execução temos que $P(n)$ vale. Ela nos diz então que $A[2..n]$ está ordenado de modo não-decrescente e contém os $n - 1$ 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 ao fim da execução do algoritmo. \square

Sobre o tempo de execução, note que CONSTROIHEAP executa em tempo $O(n)$. Note ainda que a cada execução do laço **para**, o heap tem tamanho i , para $2 \leq i \leq n$, e $\text{CORRIGEHEAPDESCENDO}$ é executado a partir da primeira posição do vetor, de forma que ele leva tempo $O(\lg i)$. Como são realizadas $n - 1$ execuções do laço, com $2 \leq i \leq n$, o tempo total é dado por

$$\sum_{i=2}^n O(\lg i) = O(n \lg n).$$

A expressão acima é válida pelo seguinte, onde c é uma constante. Note que $\sum_{i=2}^n c \lg i = c \lg 2 + c \lg 3 + \dots + c \lg n = c(\lg(2 \cdot 3 \cdot \dots \cdot n)) = c \lg n!$. Além disso, $n! \leq n^n$, de forma que $c \lg n! \leq c \lg n^n$, o que significa que $c \lg n! \leq cn \lg n$ e, por isso, $c \lg n! = O(n \lg n)$.

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 $O(n \lg n)$ gasto pelo *Heapsort* ou pelo *Mergesort*. No entanto, o *Quicksort* costuma ser a melhor escolha na prática. Isso porque, além de ter uma implementação bem simples, seu tempo de execução *esperado* é $\Theta(n \lg n)$ e as constantes escondidas em $\Theta(n \lg n)$ são bem pequenas. Esse algoritmo também faz uso do paradigma de divisão e conquista, assim como o *Mergesort*¹. Ao contrário do *Mergesort*, nenhum vetor auxiliar é necessário para a execução do *Quicksort*, e por isso ele é um algoritmo *in-place*. As implementações mais eficientes, e a que veremos nesta seção, não costumam ser estáveis.

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

¹Veja mais sobre esse paradigma no Capítulo 20.

o pivô, portanto, está em sua posição correta final com relação ao vetor ordenado e, por isso, não precisa mais ser considerado. 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 começa escolhendo um elemento pivô (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 do que o 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 $A[\text{inicio}..\text{fim}]$. 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($A, \text{inicio}, \text{fim}$)

```

1 se  $\text{inicio} < \text{fim}$  então
2    $p = \text{ESCOLHEPIVO}(A, \text{inicio}, \text{fim})$ 
3   troque  $A[p]$  com  $A[\text{fim}]$ 
4    $x = \text{PARTICIONA}(A, \text{inicio}, \text{fim})$ 
5   QUICKSORT( $A, \text{inicio}, x - 1$ )
6   QUICKSORT( $A, x + 1, \text{fim}$ )
```

O procedimento PARTICIONA recebe o vetor A e as posições *inicio* e *fim*, e considera que o pivô é $A[\text{fim}]$. Seu objetivo é particionar $A[\text{inicio}..\text{fim}]$ com relação ao pivô. Ele devolve a posição final do pivô após a partição.

O PARTICIONA funciona de maneira bastante eficiente pois faz uma única varredura no vetor, da direita para a esquerda. Assim, a qualquer momento, o que já foi visto está antes da posição atual e o que ainda será visto está depois. A ideia é que no subvetor que contém elementos já vistos, vamos manter ainda 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 copiado, baseando-se no fato do elemento ser maior, igual ou menor do que o pivô.

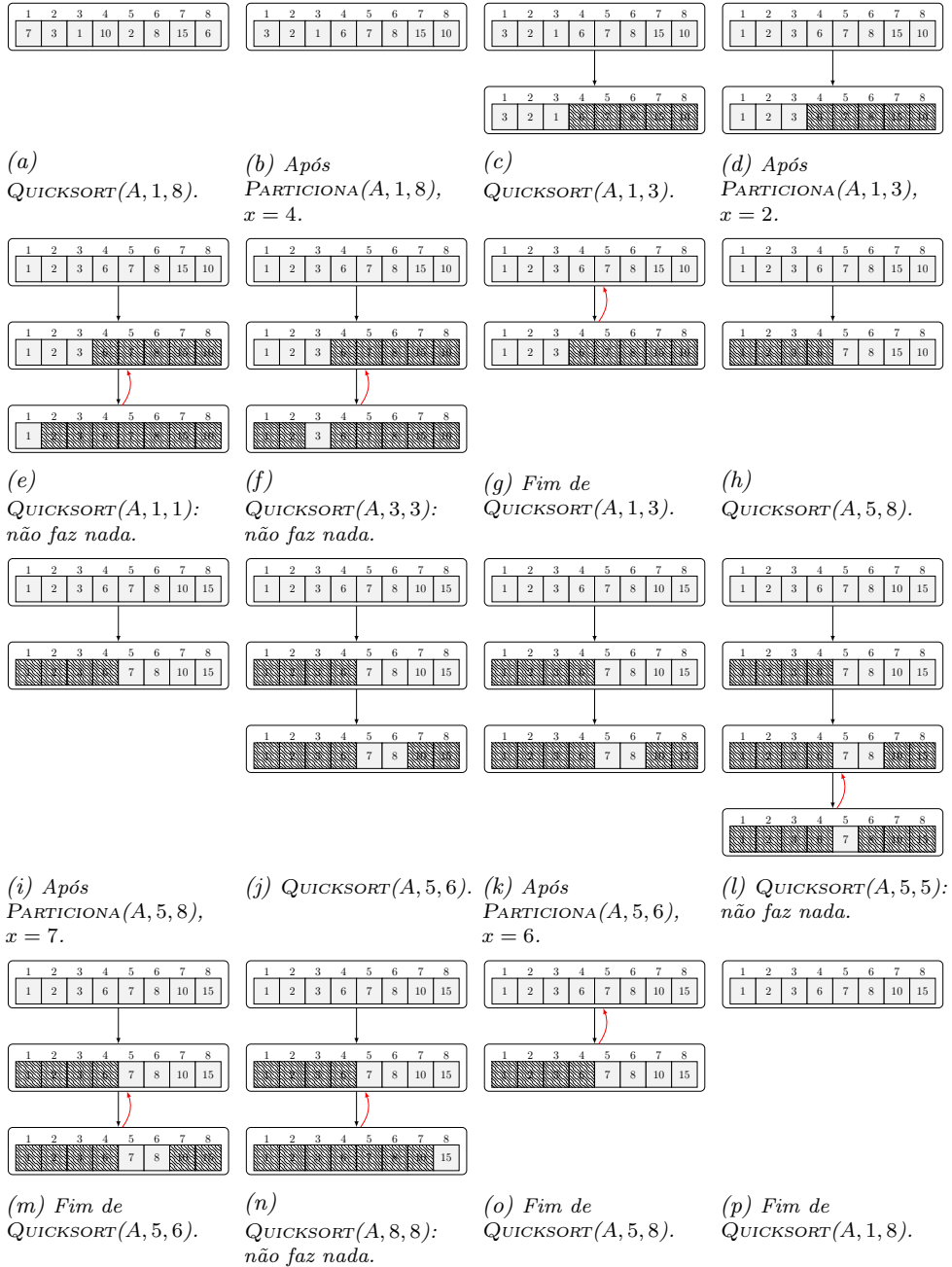


Figura 18.1: Execução de $\text{QUICKSORT}(A, 1, 8)$ (Algoritmo 18.1) para $A = (7, 3, 1, 10, 2, 8, 15, 6)$. Suponha que o pivô sempre é o último elemento do vetor.

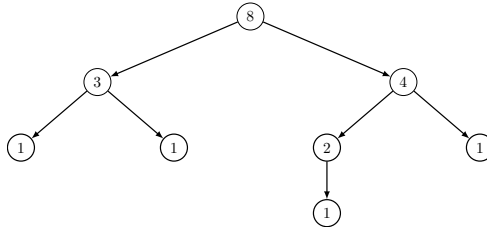
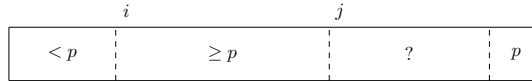


Figura 18.2: Árvore de recursão completa de $\text{QUICKSORT}(A, 1, 8)$. Cada nó é rotulado com o tamanho do problema ($\text{fim} - \text{inicio} + 1$) correspondente.

Formalmente, iremos manter um índice j que irá indicar uma separação do vetor em duas partes: $A[\text{inicio}..j-1]$ 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: $A[\text{inicio}..i-1]$ contém elementos menores ou iguais ao pivô e $A[i..j-1]$ contém elementos maiores do que o pivô. O desenho a seguir esquematiza essa ideia de forma gráfica, em que p é o valor do elemento pivô e $?$ indica uma porção de elementos ainda não examinados:



O **PARTICIONA** então consegue realizar uma única varredura no vetor porque decide imediatamente o que fazer com o elemento $A[j]$ atual. Se $A[j]$ é menor do que o pivô, então ele deve ser colocado próximo aos elementos de $A[\text{inicio}..i-1]$: troca-se $A[i]$ com $A[j]$. Caso contrário, $A[j]$ já está próximo aos elementos maiores, que estão em $A[i..j-1]$: não há troca. Finalmente, o **PARTICIONA** é formalizado no Algoritmo 18.2 e um exemplo de sua execução é mostrado na Figura 18.3.

Algoritmo 18.2: $\text{PARTICIONA}(A, \text{inicio}, \text{fim})$

```

1  $pivo = A[\text{fim}]$ 
2  $i = \text{inicio}$ 
3 para  $j = \text{inicio}$  até  $\text{fim} - 1$ , incrementando faça
4   se  $A[j] \leq pivo$  então
5     troca  $A[i]$  e  $A[j]$ 
6      $i = i + 1$ 
7 troca  $A[i]$  e  $A[\text{fim}]$ 
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.

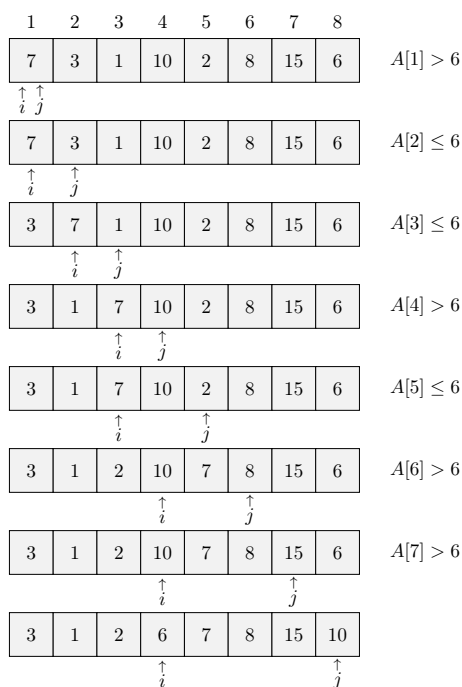


Figura 18.3: Execução de $\text{PARTICIONA}(A, 1, 8)$ (Algoritmo 18.2), onde $A = (7, 3, 1, 10, 2, 8, 15, 6)$ e $\text{pivo} = 6$.

Teorema 18.1

O algoritmo $\text{PARTICIONA}(A, \text{inicio}, \text{fim})$ devolve um índice x tal que o *pivô* está na posição x , todo elemento em $A[1..x-1]$ é menor do que o *pivô*, e todo elemento em $A[x+1..n]$ é maior ou igual ao *pivô*.

Demonstração. Para provar esse resultado, vamos utilizar a seguinte invariante do laço **para** do algoritmo.

Invariante: Laço para – PARTICIONA

$P(t)$ = “Antes da t -ésima iteração, temos que $j = \text{inicio} + t - 1$ e vale que

- (i) os elementos de $A[\text{inicio}..i-1]$ são menores do que *pivô*;
- (ii) os elementos de $A[i..j-1]$ são maiores ou iguais a *pivô*.”

Demonstração. Vamos provar que P é uma invariante por indução no número t de vezes que o teste da execução de parada é executado. Quando $t = 1$ e o teste executou uma única vez, apenas fizemos $i = \text{inicio}$ e $j = \text{inicio}$. De fato, os elementos de $A[\text{inicio}..i-1] = A[\text{inicio}..\text{inicio}-1]$ (vetor vazio) são menores do que *pivô* e os elementos de $A[i..j-1] = A[\text{inicio}..\text{inicio}-1]$ são maiores ou iguais a *pivô*. Então $P(1)$ vale trivialmente.

Agora seja $t \geq 1$ e suponha que $P(t)$ vale. Precisamos provar que $P(t+1)$ também vale. Como $P(t)$ vale, sabemos que $j = \text{inicio} + t - 1$, que os elementos de $A[\text{inicio}..i-1]$ são menores do que *pivô* e que os elementos de $A[i..\text{inicio}+t-2]$ são maiores ou iguais a *pivô*. Durante essa t -ésima iteração, uma dentre duas coisas pode acontecer.

Se $A[j] = A[\text{inicio} + t - 1] > \text{pivô}$, nada é feito. Juntando essa nova informação com a informação dada por $P(t)$, acabamos de garantir que $A[i..\text{inicio}+t]$ contém elementos maiores ou iguais a *pivô* e continua valendo que $A[\text{inicio}..i-1]$ contém elementos menores do que *pivô*. Como j é incrementado de uma unidade, mostramos que, neste caso, $P(t+1)$ vale.

Por outro lado, se $A[\text{inicio} + t - 1] \leq \text{pivô}$, então trocamos $A[i]$, que sabemos ser maior ou igual a *pivô* com $A[\text{inicio} + t - 1]$. Juntamente com os fatos de $P(t)$, sabemos agora que todo elemento em $A[\text{inicio}..i]$ é menor ou igual a *pivô* e que todo elemento em $A[i+1..\text{inicio} + t - 1]$ é maior ou igual a *pivô*. Como i é incrementado de 1 unidade e j também, acabamos de provar que $P(t+1)$ também

vale nesse caso.

Assim, P é de fato uma invariante do laço **para**. ◇

Como o teste do laço **para** executa $fim - inicio + 1$ vezes, ao final de sua execução temos que $P(fim - inicio + 1)$ vale. Ela nos diz então que $j = inicio + (fim - inicio + 1) - 1 = fim$, que os elementos de $A[inicio..i - 1]$ são menores do que *pivo* e que os elementos de $A[i..fim - 1]$ são maiores ou iguais a *pivo*. Trocar $A[i]$ com $A[fim]$ na linha 7, portanto, irá de fato particionar o vetor $A[inicio..fim]$ com relação a *pivo* e devolver o índice correto. □

Com relação ao tempo, como o laço **para** é executado $fim - inicio$ vezes, e cada iteração leva tempo constante, 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($A, inicio, fim$) simplesmente 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 zero ou um 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 $QUICKSORT(A, x+1, fim)$. Também note que $fim - (x+1) + 1 = fim - x \leq fim - inicio < n$, 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. \square

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. Nessa seção vamos então discutir como implementar $ESCOLHEPIVO$. Nas discussões a seguir, considere $n = fim - inicio + 1$ a quantidade de elementos do vetor de entrada.

Suponha que $ESCOLHEPIVO$ sempre 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 $\Theta(n)$ quando n 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

$$\begin{aligned}
 T(n) &= T(n-1) + n \\
 &= T(n-2) + (n-1) + n \\
 &\vdots \\
 &= T(1) + \sum_{j=2}^n j \\
 &= 1 + \frac{(n-1)(n+2)}{2} \\
 &= \Theta(n^2).
 \end{aligned}$$

Intuitivamente, conseguimos perceber que esse é o pior caso possível. Formalmente, o

tempo de execução de pior caso é dado por $T(n) = \max_{0 \leq x \leq n-1} (T(x) + T(n-x-1)) + n$. Vamos utilizar indução para mostrar que $T(n) \leq n^2$. Supondo que $T(m) \leq m^2$ para todo $m < n$, obtemos

$$\begin{aligned} T(n) &\leq \max_{0 \leq x \leq n-1} (cx^2 + c(n-x-1)^2) + n \\ &\leq (n-1)^2 + n \\ &= n^2 - (2n-1) + n \\ &\leq n^2, \end{aligned}$$

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 $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$.

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 $T(n) = O(n \lg n)$.

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 $O(n \lg n)$, como veremos a seguir.

Vamos utilizar o método da substituição para mostrar que $T(n) = T(n/k) + T((k-1)n/k) + n$ tem solução $O(n \lg n)$. Assuma que se $n \leq k-1$, então $T(n) \leq c$ para alguma constante $c \geq 1$.

Vamos provar que $T(n) = T(n/k) + T((k-1)n/k) + n$ é no máximo $dn \lg n + n$ para todo $n \geq k$ e alguma constante $d > 0$. Para o caso base, note que $T(k) =$

$T(1) + T(k-1) + k \leq 2c + k$ e que $2c + k \leq dk \lg k + k$ sempre que $d \geq 2c/(k \lg k)$. Suponha então que $T(m) \leq dm \lg m + m$ para todo $k < m < n$ e vamos analisar $T(n)$:

$$\begin{aligned}
 T(n) &= T\left(\frac{n}{k}\right) + T\left(\frac{(k-1)n}{k}\right) + n \\
 &\leq d\left(\frac{n}{k} \lg\left(\frac{n}{k}\right)\right) + \frac{n}{k} + d\left(\frac{(k-1)n}{k} \lg\left(\frac{(k-1)n}{k}\right)\right) + \frac{(k-1)n}{k} + n \\
 &= d\left(\frac{n}{k} (\lg n - \lg k)\right) + d\left(\frac{(k-1)n}{k} (\lg(k-1) + \lg\left(\frac{n}{k}\right))\right) + 2n \\
 &= dn \lg n + n - dn \lg k + \left(\frac{d(k-1)n}{k} \lg(k-1) + n\right) \\
 &\leq dn \lg n + n,
 \end{aligned}$$

onde a última desigualdade vale se $d \geq k/\lg k$, pois para tal valor de d temos

$$dn \lg k \geq \left(\frac{d(k-1)n}{k} \lg(k-1) + n\right).$$

Portanto, acabamos de mostrar que $T(n) = O(n \lg n)$ 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 $\lg_{k/(k-1)} n = \Theta(\lg n)$ 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 $O(n \lg n)$. 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. Nós poderíamos então fazer uma análise de caso médio, considerando que cada uma das $n!$ possíveis ordenações dos elementos de A tem a mesma chance de ser a distribuição do vetor de entrada A . Por mais que essa análise forneça um limite de $O(n \lg n)$ no caso médio, isso ainda permite que usuários maliciosos forcem o pior caso. Para evitar isso, vamos implementar ESCOLHEPIVO para sempre fazer uma escolha aleatória. Assim, suponha que o pivô é escolhido *uniformemente ao acaso* dentre as chaves contidas em A , i.e., com probabilidade $1/n$. A análise a seguir fornece um tempo *esperado* de execução².

Perceba que o tempo de execução de QUICKSORT é dominado pela quantidade de

²A análise de caso médio mencionada é muito similar à análise do tempo esperado e por isso elas fornecem o mesmo tempo de execução. A vantagem é que temos agora um algoritmo que tem probabilidade muito pequena de atingir o pior caso.

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, \dots, o_n os elementos de A em sua ordenação final (após estarem ordenados de modo não-decrescente), i.e., $o_1 \leq o_2 \leq \dots \leq o_n$ 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

$$\begin{aligned} \mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{P}(o_i \text{ ser comparado com } o_j). \end{aligned} \quad (18.1)$$

Vamos então calcular $\mathbb{P}(o_i \text{ ser comparado com } o_j)$. Começemos notando que para o_i ser comparado com o_j , um dos dois precisa ser o primeiro elemento de $O_{ij} = \{o_i, o_{i+1}, \dots, o_j\}$ 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,

$$\begin{aligned} &\mathbb{P}(o_i \text{ ser comparado com } o_j) \\ &= 1 - \mathbb{P}(o_1 \text{ não ser comparado com } o_j) \\ &= 1 - \mathbb{P}(\text{qualquer elemento em } O_{ij} \setminus \{o_i, o_j\} \text{ ser escolhido primeiro como pivô em } O_{ij}) \\ &= 1 - \frac{j-i+1-2}{j-i+1} = \frac{2}{j-i+1}. \end{aligned}$$

Assim, voltando à (18.1), temos

$$\begin{aligned}
 \mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
 &< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i+1} \frac{2}{k} < \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} \\
 &= \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n).
 \end{aligned}$$

Portanto, concluímos que o *tempo esperado de execução* de QUICKSORT é $O(n \lg n)$.

Assim, sem supor nada sobre a entrada do algoritmo, garantimos um tempo de execução esperado de $O(n \lg n)$.

Ordenação sem comparação

Vimos, nos capítulos anteriores, alguns algoritmos com tempo de execução (de pior caso ou caso médio) $\Theta(n \lg n)$. *Mergesort* e *Heapsort* têm esse limitante no pior caso e *Quicksort* possui tempo de execução esperado da ordem de $n \lg n$. Acontece que todos os algoritmos anteriores são baseados em *comparações* entre os elementos de entrada.

Suponha um algoritmo correto para o problema 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. Assim, a string tem valor 1 na i -ésima posição se a i -ésima comparação feita foi verdadeira, e tem 0 caso contrário. Como existem 2^k strings binárias de tamanho k , então temos 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 $k \geq (n/2) \lg(n/2)$, isto é, $k = \Omega(n \lg n)$.

Pela discussão acima, percebemos que qualquer algoritmo baseado em comparações requer $\Omega(n \lg n)$ 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 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 $n \lg n$. Formalizaremos o tempo a seguir. O COUNTINGSORT é formalizado no

Algoritmo 19.1 e a Figura 19.1 apresenta um exemplo de execução.

Algoritmo 19.1: COUNTINGSORT(A, n, k)

```

/* C é um vetor auxiliar de contadores e B manterá o vetor ordenado */
1 Sejam  $B[1..n]$  e  $C[0..k]$  vetores
2 para  $i = 0$  até  $k$ , incrementando faça
3    $C[i] = 0$ 
   /*  $C[i]$  inicialmente guarda a quantidade de ocorrências de  $i$  em  $A$  */
4 para  $j = 1$  até  $n$ , incrementando faça
5    $C[A[j]] = C[A[j]] + 1$ 
   /*  $C[i]$  deve guardar a qtd. de ocorrências de elementos  $\leq i$  em  $A$  */
6 para  $i = 1$  até  $k$ , incrementando faça
7    $C[i] = C[i] + C[i - 1]$ 
   /* Colocando o resultado da ordenação de  $A$  em  $B$ :  $A[j]$  deve ir para a
      posição  $C[A[j]]$  */
8 para  $j = n$  até  $1$ , decrementando faça
9    $B[C[A[j]]] = A[j]$ 
10   $C[A[j]] = C[A[j]] - 1$ 
11 devolve  $B$ 

```

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.

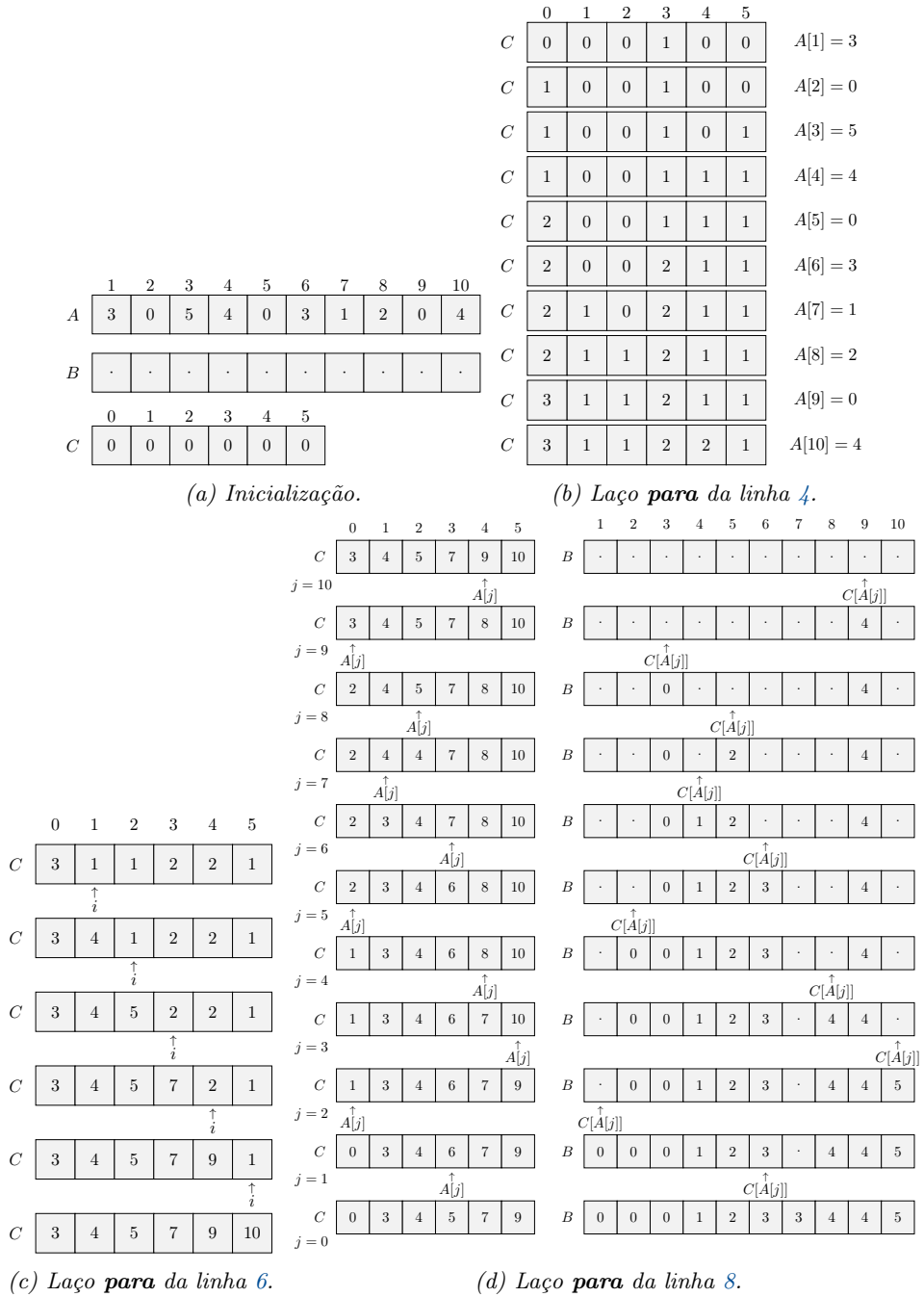


Figura 19.1: Execução de COUNTINGSORT(A , 10, 5) (Algoritmo 19.1), com $A = (3, 0, 5, 4, 0, 3, 1, 2, 0, 4)$.

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.

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

Entrada: $\langle x, y, n \rangle$, onde x e y são inteiros contendo n dígitos cada.

Saída: O produto xy .

Esse problema pode, a princípio, parecer muito simples: toda linguagem de programação oferece um operador aritmético de multiplicação, então basta utilizá-lo. Acontece que toda linguagem de programação também tem um limite no tamanho do inteiro que pode ser armazenado em uma variável do tipo inteiro. Assim, nesse problema estamos realmente imaginando valores muito grandes de n ; tão grandes que não podem ser armazenados em uma única variável.

Todos nós conhecemos o algoritmo clássico de multiplicação, exemplificado a seguir. Sejam $x = 5678$ e $y = 1234$ (ou seja, $n = 4$):

$$\begin{array}{r}
 5678 \\
 \times 1234 \\
 \hline
 22712 \\
 170340 \\
 + 1135600 \\
 5678000 \\
 \hline
 7006652
 \end{array}$$

Para mostrar que esse algoritmo está de fato correto, precisamos mostrar que para quaisquer dois inteiros x e y , ele devolve xy . Sejam $x = x_1x_2 \dots x_n$ e $y = y_1y_2 \dots y_n$, onde cada x_i e y_i é um dígito de 0 a 9, para $1 \leq i \leq n$. Note que o algoritmo faz

$$\begin{aligned}
 & \sum_{j=1}^n 10^{n-j} y_j \sum_{i=1}^n 10^{n-i} x_i \\
 &= y_n \sum_{i=1}^n 10^{n-i} x_i + 10y_{n-1} \sum_{i=1}^n 10^{n-i} x_i + \dots + 10^{n-1} y_1 \sum_{i=1}^n 10^{n-i} x_i \\
 &= (x \times y_n) + (x \times y_{n-1} \times 10) + \dots + (x \times y_2 \times 10^{n-2}) + (x \times y_1 \times 10^{n-1}) \\
 &= x((y_n) + (y_{n-1} \times 10) + \dots + (y_2 \times 10^{n-2}) + (y_1 \times 10^{n-1})) \\
 &= xy.
 \end{aligned}$$

Com relação ao tempo, observe que somar ou multiplicar dois dígitos simples é uma operação básica. Então, para obter o primeiro produto parcial $(x \times y_n)$, precisamos de n multiplicações de dois dígitos 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 são 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, que é a quantidade de dígitos recebida.

Felizmente, e talvez surpreendentemente, existem algoritmos melhores para resolver o problema Multiplicação de Inteiros. 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 $\lfloor \frac{n}{2} \rfloor$ dígitos cada. No exemplo anterior, se $x = 5678$ e $y = 1234$, então temos $a = 56$, $b = 78$, $c = 12$ e $d = 34$. Podemos então escrever

$$xy = (10^{\lceil n/2 \rceil}a + b)(10^{\lceil n/2 \rceil}c + d) = 10^{2\lceil n/2 \rceil}ac + 10^{\lceil n/2 \rceil}(ad + bc) + bd. \quad (20.1)$$

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! Como $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$ apenas quando $n > 2$, nosso caso base será a multiplicação de números com 1 ou 2 dígitos.

Um procedimento de divisão e conquista simples para o problema Multiplicação de Inteiros é descrito no Algoritmo 20.1. Ele usa a função $\text{IGUALATAM}(x, y)$, que deixa os números x e y com o mesmo número de dígitos, sendo este igual ao número de dígitos do maior deles, colocando zeros à esquerda se necessário e devolve o número de dígitos dos números após a alteração.

É possível provar por indução em n que $\text{MULTIPLICAINTEIROS}$ 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 $\text{IGUALATAM}(x, y)$ também é $O(n)$ ¹. Pelo Método Mestre (Seção 8.4), temos $T(n) = O(n^2)$, isto é, não houve melhora, assintoticamente falando, com relação ao algoritmo simples.

¹ Assim como não conseguimos multiplicar diretamente dois números com muitos dígitos, também não conseguimos somá-los. Por isso, o tempo da linha 9 é proporcional à quantidade de dígitos sendo somados.

Algoritmo 20.1: MULTIPLICAÍNTeiros(x, y)

```

1  $n = \text{IGUALATAM}(x, y)$ 
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  $p_1 = \text{MULTIPLICAÍNTeiros}(a, c)$ 
6  $p_2 = \text{MULTIPLICAÍNTeiros}(a, d)$ 
7  $p_3 = \text{MULTIPLICAÍNTeiros}(b, c)$ 
8  $p_4 = \text{MULTIPLICAÍNTeiros}(b, d)$ 
9 devolve  $10^{2\lceil n/2 \rceil}p_1 + 10^{\lceil n/2 \rceil}(p_2 + p_3) + p_4$ 

```

Veremos agora o algoritmo de Karatsuba, que também usa o paradigma de divisão e conquista mas se aproveita do fato de que $(a + b)(c + d) = ac + ad + bc + bd$ e faz apenas 3 chamadas recursivas. Assim, calculando apenas os produtos ac , bd e $(a + b)(c + d)$ conseguiremos calcular (20.1), pois $(a + b)(c + d) - ac - bd = ad + bc$. 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.

Algoritmo 20.2: KARATSUBA(x, y, n)

```

1  $n = \text{IGUALATAM}(x, y)$ 
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  $p_1 = \text{KARATSUBA}(a, c)$ 
6  $p_2 = \text{KARATSUBA}(b, d)$ 
7  $p_3 = \text{KARATSUBA}(a + b, c + d)$ 
8 devolve  $10^{2\lceil n/2 \rceil}p_1 + 10^{\lceil n/2 \rceil}(p_3 - p_1 - p_2) + p_2$ 

```

Novamente, é possível provar por indução em n que KARATSUBA corretamente calcula xy , 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.

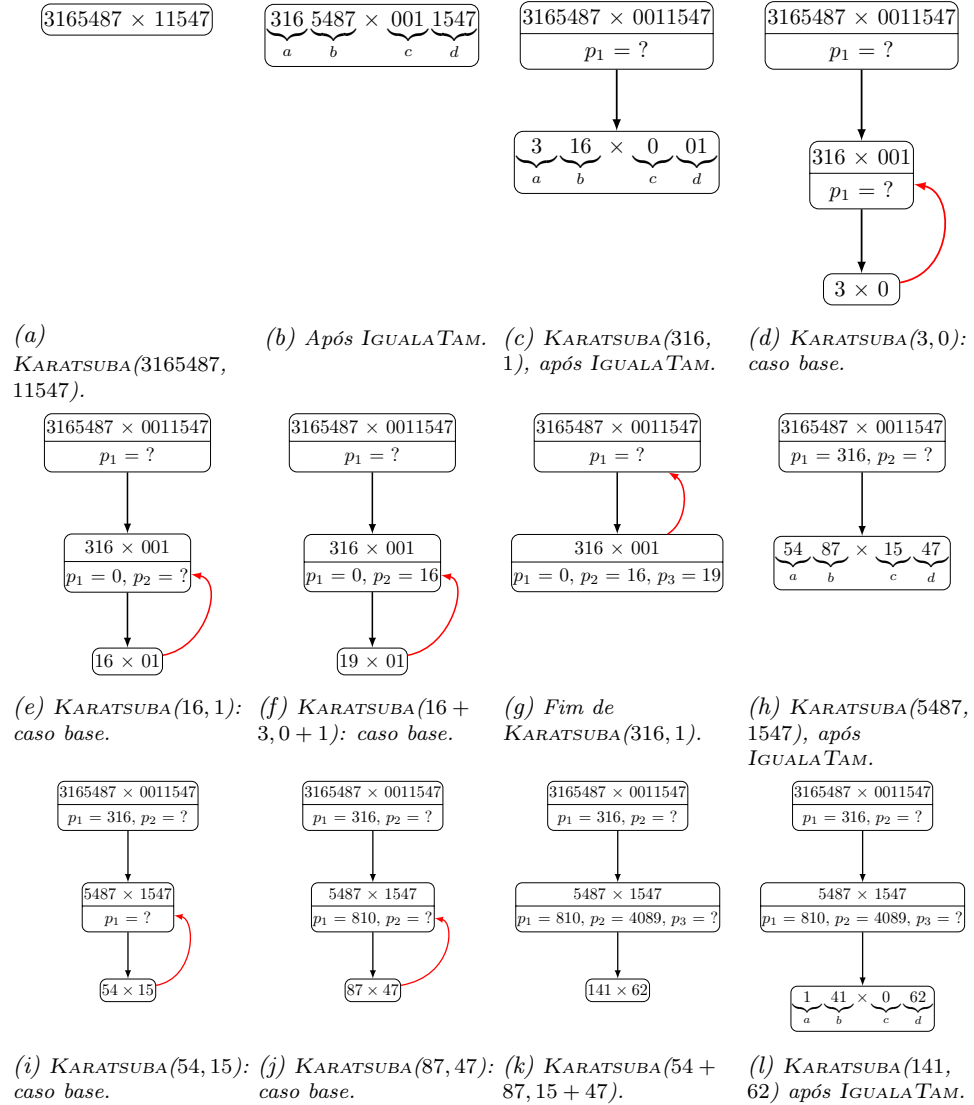


Figura 20.1: Parte 1 da execução de $KARATSUBA(3165487, 11547)$ (Algoritmo 20.2).

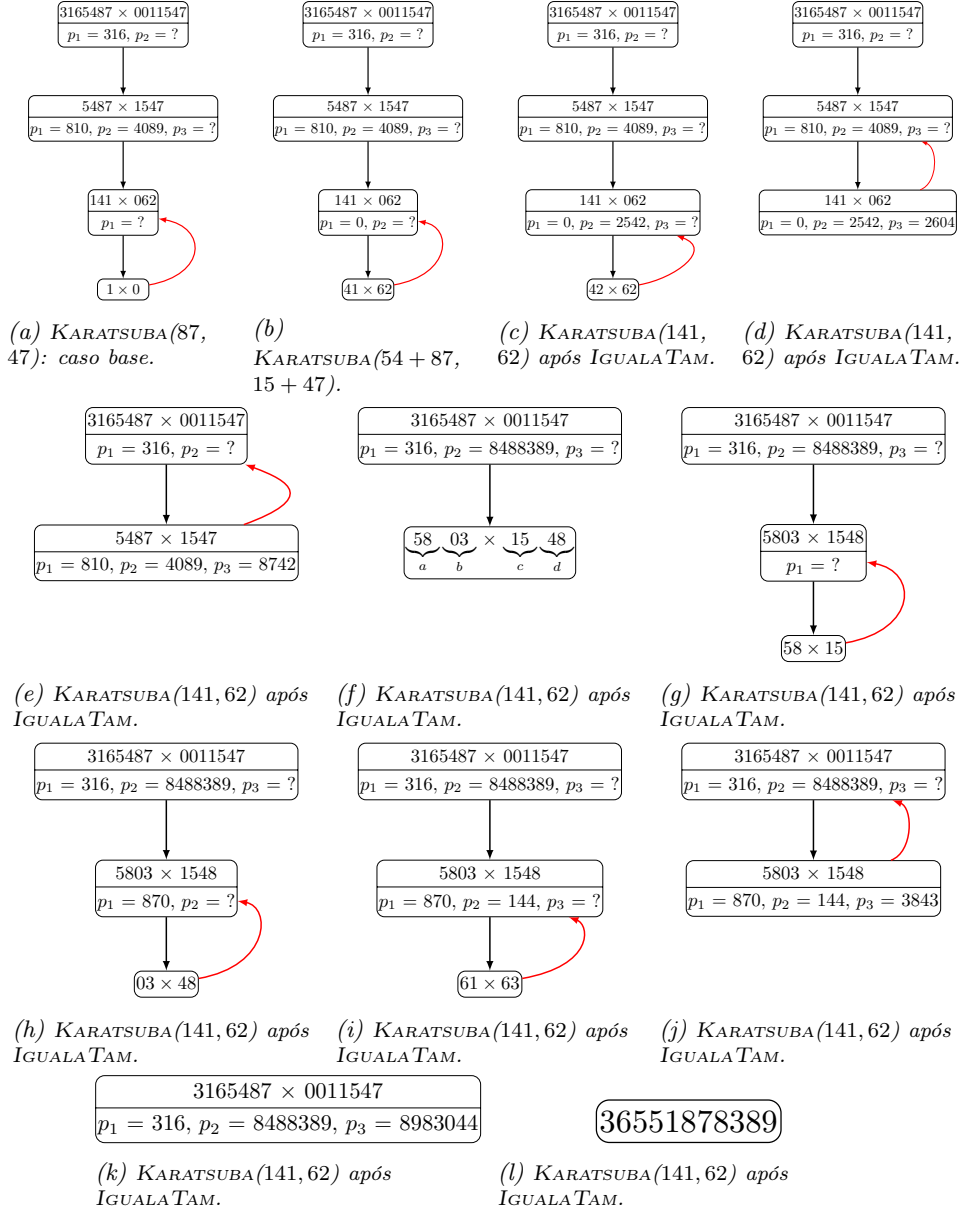


Figura 20.2: Parte 2 da execução de $\text{KARATSUBA}(3165487, 11547)$ (Algoritmo 20.2).

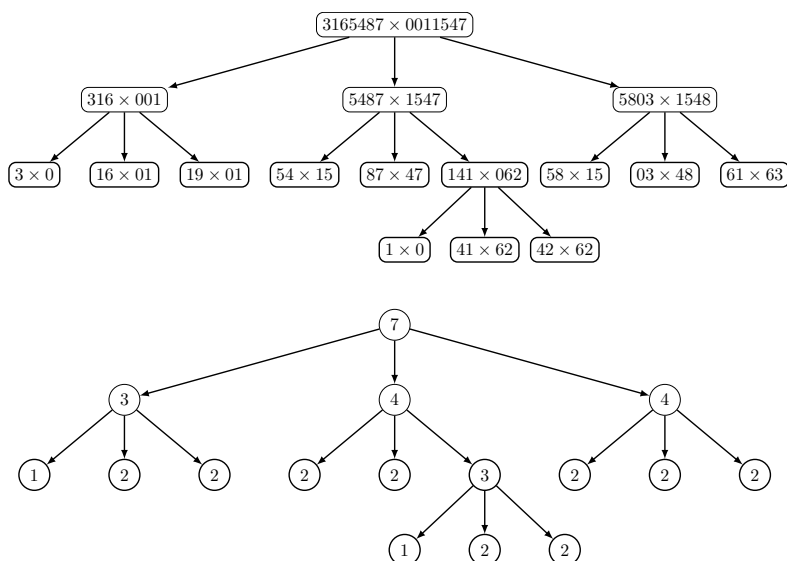


Figura 20.3: Árvore de recursão completa de KARATSUBA(3165487, 11547). Na parte superior, cada nó é rotulado com o problema enquanto que na parte inferior cada nó é rotulado com o tamanho do problema.

Algoritmos gulosos

Um algoritmo é dito guloso quando constrói uma solução através de uma sequência de decisões que visam o melhor cenário no momento da decisão, 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

No problema que veremos, temos um conjunto de tarefas que precisam ser executadas. 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

Entrada: $\langle T, n, s, f \rangle$, onde $T = \{t_1, \dots, t_n\}$ é um conjunto com n tarefas onde cada $t_i \in T$ tem tempo inicial s_i e tempo final f_i .

Saída: Maior subconjunto $S \subseteq T$ de tarefas mutuamente compatíveis.

Veja a Figura 21.1 para um exemplo do problema.

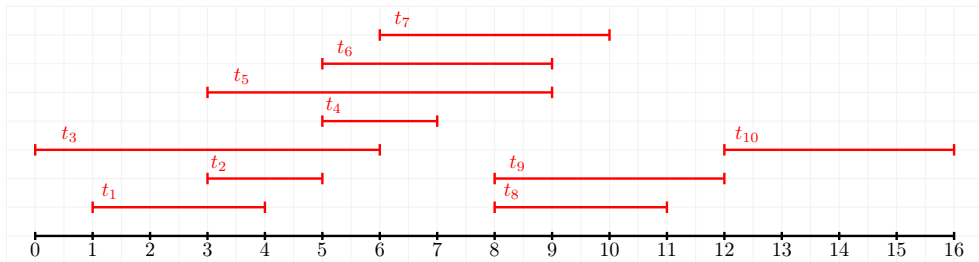


Figura 21.1: Conjunto $T = \{t_1, t_2, \dots, t_{10}\}$ de tarefas e seus respectivos intervalos, representados pelas barras vermelhas. Neste tipo de representação, a incompatibilidade é facilmente observada quando traçamos uma reta vertical: todas as tarefas que intersectam a reta são incompatíveis. Note que $S_1 = \{t_3, t_9, t_{10}\}$ é uma solução viável para essa instância. As soluções viáveis $S_2 = \{t_1, t_4, t_8, t_{10}\}$ e $S_3 = \{t_2, t_4, t_8, t_{10}\}$, no entanto, são ótimas.

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, isto é, com menor valor de f_i . Assim, dado $T = \{t_1, \dots, t_n\}$, começamos escolhendo a tarefa t_x tal que $f_x \leq f_z$ para toda $t_z \in T$. Em seguida, devemos escolher uma

tarefa t_y que seja compatível com t_x e que tenha o menor valor f_y (dentre as compatíveis com t_x). Note que isso é equivalente a escolher a tarefa que termina primeiro no conjunto $T' = \{t_z \in T: s_z \geq f_x\}$, isto é, o conjunto que contém todas as tarefas que são compatíveis com t_x (como f_x é o menor possível em T , não há tarefas t_z para as quais $s_x \geq f_z$). Mas note que T' também é um conjunto de tarefas. Temos então um subproblema em mãos e, portanto, podemos usar recursão para resolvê-lo: uma solução para T' combinada com t_x é uma solução para T . O Algoritmo 21.1 formaliza essa ideia. É possível mostrar, por indução na quantidade de tarefas, que ele sempre devolve uma solução viável.

Algoritmo 21.1: ESCALONACOMPATIVELREC(T)

```

1 se  $|T| == 1$  então
2   devolve  $T$ 
3 Seja  $t_x$  a tarefa que termina primeiro em  $T$ 
4  $T' = \{t_z \in T: s_z \geq f_x\}$ 
5  $S' = \text{ESCALONACOMPATIVELREC}(T')$ 
6 devolve  $S' \cup \{t_x\}$ 

```

Observe que o fato de ESCALONACOMPATIVELREC 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. Isso está formalizado no teorema a seguir.

Teorema 21.2

Dado um conjunto T de tarefas, onde cada $t_i \in T$ tem tempo inicial s_i e tempo final f_i , o algoritmo ESCALONACOMPATIVELREC(T) devolve uma solução ótima para o problema Escalonamento de Tarefas Compatíveis.

Demonstração. Vamos provar o resultado por indução na quantidade n de tarefas em T . Se $n = 1$, o algoritmo simplesmente devolve T . Claramente, essa é a solução ótima neste caso.

Seja então $n > 1$ e suponha que o algoritmo devolve uma solução ótima para qualquer conjunto que tenha menos do que n tarefas. Com $n > 1$, o algoritmo primeiro encontra t_x , que termina primeiro em T , e monta T' , que contém apenas tarefas que são compatíveis com t_x . Como $|T'| < |T|$, já que ao menos $t_x \notin T'$, então vale, por hipótese de indução, que S' é uma solução ótima para T' . Resta provar que $S = S' \cup \{t_x\}$ é ótima para T . Vamos fazer isso por contradição, assumindo que S não é ótima para T .

Primeiro note que qualquer solução ótima para T deve conter uma tarefa $t_y \in T \setminus T'$: se não fosse esse o caso, t_y poderia ser adicionada a essa solução ótima sem que ela deixasse de ser uma solução viável e seu tamanho seria aumentado, o que é uma contradição. Como $T \setminus T'$ contém t_x e todas as tarefas incompatíveis com t_x , podemos concluir que existe uma solução ótima para T que contém t_x .

Assim, seja S^* uma solução ótima para T que contém t_x . Como S não é ótima, vale que $|S^*| > |S|$. Perceba que $R' = S^* \setminus \{t_x\}$ é tal que $R' \subseteq T'$ e, ademais, R' é uma solução ótima para T' : se não fosse esse o caso, haveria uma solução $R^* \subseteq T'$ ótima para T' , o que significaria $|R^*| > |R'|$, e certamente $R^* \cup \{t_x\}$ seria uma solução viável para T ; porém, $|R^* \cup \{t_x\}| = |R^*| + 1 > |R'| + 1 = |S^* \setminus \{t_x\}| + 1 = |S^*|$, ou seja, $R^* \cup \{t_x\}$ teria custo melhor do que S^* , o que é uma contradição. Mas então temos $|R'| = |S^* \setminus \{t_x\}| = |S^*| - 1 > |S| - 1 = |S'|$, o que é uma contradição com o fato de S' ser ótima para T' . Concluimos então que S é ótima para T . \square

Com relação ao tempo de execução, primeiro considere que os valores de início e fim das tarefas estão armazenados em vetores s e f , de forma que têm-se acesso constante a eles para uma dada tarefa i . Também é simples implementar S como um vetor de n posições tal que $S[i] = 1$ se t_i foi escolhida para a solução e $S[i] = 0$ caso contrário. Assim, podemos ver que o Algoritmo 21.1 tem tempo de execução polinomial no tamanho n da entrada: a execução de uma única chamada leva tempo $O(n)$, pois encontrar a tarefa t_x que termina primeiro leva tempo $O(n)$ e montar a instância T' também leva tempo $O(n)$, e existem $O(n)$ chamadas recursivas, pois no pior caso o tamanho de T' diminui de uma única unidade a cada chamada. Com isso, o tempo total é $O(n^2)$.

Agora veja que por ter uma única chamada recursiva ao fim, o algoritmo ESCALONACOMPATIVELREC pode ser facilmente transformado em um algoritmo iterativo. Além disso, se o conjunto de tarefas estiver ordenado de acordo com os tempos de término, encontrar as tarefas com menor valor de término também pode ser realizado de forma mais eficiente. Esse novo formato é descrito no Algoritmo 21.2.

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, é fácil mostrar que o algoritmo mantém a *invariante* de que S é um conjunto de tarefas mutuamente compatíveis no início de qualquer iteração do laço. Assim, o conjunto S devolvido é de fato uma solução viável para o problema.

Algoritmo 21.2: ESCALONACOMPATIVEL(T, n)

```

1 Ordene as tarefas em ordem não-decrescente de tempo final
2 Renomeie-as para que  $f_1 \leq f_2 \leq \dots \leq f_n$ 
3  $S = \{t_1\}$ 
4  $k = 1$  /*  $k$  mantém o índice da última tarefa que foi adicionada à  $S$  */
5 para  $i = 2$  até  $n$ , incrementando faça
6   se  $s_i \geq f_k$  então
7      $S = S \cup \{t_i\}$ 
8      $k = i$ 
9 devolve  $S$ 

```

Apesar de já termos mostrado que ESCALONACOMPATIVELREC é um algoritmo ótimo e de termos afirmado que ESCALONACOMPATIVEL é apenas outra forma de escrever a mesma ideia, é possível mostrar diretamente que ESCALONACOMPATIVEL é ótimo. Para isso, sugerimos a demonstração da seguinte invariante de laço. Para ela, considere que $T_j = \{t_1, t_2, \dots, t_j\}$, para todo $j = 1, 2, \dots, n$, e que $f(X) = \max\{f_j : t_j \in X\}$ para qualquer conjunto $X \subseteq T$ que contém tarefas mutuamente compatíveis.

Invariante: Laço para – ESCALONACOMPATIVEL

$P(x)$ = “Antes da x -ésima iteração começar, temos que $i = x + 1$ e:

1. o conjunto S é uma solução ótima para o subproblema T_{i-1} que minimiza $f(S)$, e
2. $f_k \geq f_j$ para todo $t_j \in S$.”

Com relação ao tempo de execução, considere também que os valores de início e fim das tarefas estão armazenados em vetores s e f , e que S é um vetor de n posições tal que $S[i] = 1$ se t_i foi escolhida para a solução e $S[i] = 0$ caso contrário. Com isso, as linhas 1 e 2 levam tempo $\Theta(n \lg n)$ 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 analisamos todas as tarefas fazendo operações de tempo constante e a inserção no vetor S também leva tempo constante. Assim, o tempo desse algoritmo é dominado pela ordenação das tarefas, tendo tempo total portanto $\Theta(n \lg n)$.

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

Entrada: $\langle I, n, w, v, W \rangle$, onde $I = \{1, 2, \dots, n\}$ é um conjunto de n itens sendo que cada $i \in I$ tem um peso w_i e um valor v_i associados, e W é a capacidade total da mochila.

Saída: Sequência $S = (f_1, \dots, f_n)$ em que $f_i \in [0, 1]$, para todo $i \in I$, 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.

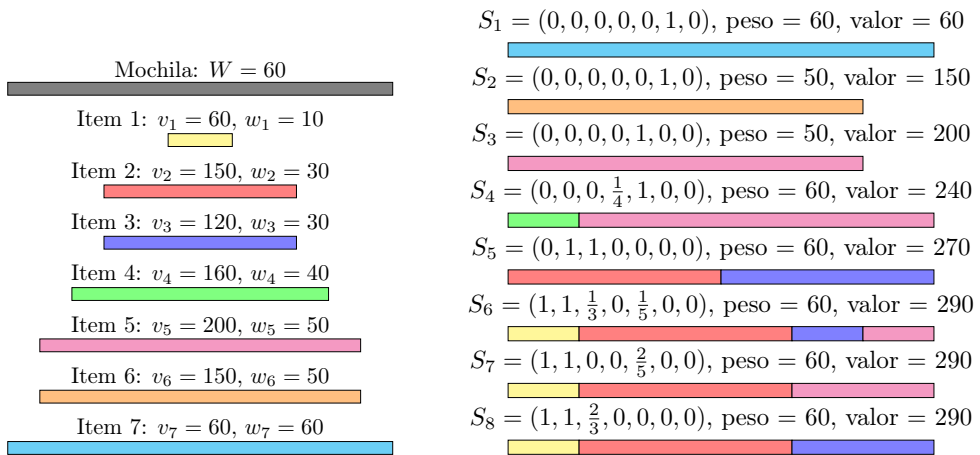


Figura 21.2: Instância do problema da mochila à esquerda com 7 itens e os respectivos pesos e valores. À direita, 8 soluções viáveis, onde as soluções S_6 , S_7 e S_8 são ótimas para a instância dada.

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 por inteiro, deixando uma capacidade restante de peso 10. O próximo item de maior valor é o 4, que não cabe por inteiro. Pegamos então a maior fração possível sua que caiba, que é $1/4$. Com isso, geramos a solução $S_4 = (0, 0, 0, 1/4, 1, 0, 0)$ 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 para o qual ele devolve uma solução que não é ótima. E veja que para fazer isso, basta mostrar alguma outra solução que seja melhor do que a devolvida pelo algoritmo, isto é, nem é necessário mostrar a solução ótima daquela instância. Enfatizamos 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 maior valor ignora totalmente outro aspecto do problema, que é a restrição do peso da mochila. Na prática, 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 21.3.

O Algoritmo 21.3 funciona inicialmente ordenando os itens e renomeando-os para ter $v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n$. Assim, o item 1 tem a maior razão valor/peso. Mantemos na variável *capacidade* a capacidade restante da mochila. No laço **enquanto** da linha 4 o algoritmo seleciona itens inteiros ($f_i = 1$) na ordem da razão valor/peso enquanto eles couberem por inteiro na mochila ($w_i \leq \text{capacidade}$). Do próximo item, se ele existir, pegamos a maior fração possível que caiba no restante do espaço (linha 9). Nenhum outro item é considerado, tendo $f_i = 0$ (laço da linha 10).

Vamos agora mostrar que temos a seguinte invariante para esse algoritmo.

Invariante: Laço enquanto – MOCHILAFRACIONARIA

$P(t) = \text{“Antes da } t\text{-ésima iteração começar, } i = t \text{ e } \textit{capacidade} = W - \sum_{j=1}^{i-1} f_j w_j \geq 0.”$

Algoritmo 21.3: MOCHILAFRACIONARIA(I, n, W)

```

1 Ordene os itens pela razão valor/peso e os renomeie de forma que
   $v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n$ 
2  $capacidade = W$ 
3  $i = 1$ 
4 enquanto  $i \leq n$  e  $capacidade \geq w_i$  faça
5    $f_i = 1$ 
6    $capacidade = capacidade - w_i$ 
7    $i = i + 1$ 
8 se  $i \leq n$  então
9    $f_i = capacidade/w_i$ 
10 para  $j = i + 1$  até  $n$ , incrementando faça
11    $f_j = 0$ 
12 devolve  $(f_1, \dots, f_n)$ 

```

Demonstração. Vamos provar que P é invariante por indução no número t de vezes que o teste da condição de parada é executado.

Caso base. Quando $t = 1$, o teste executou uma vez. Antes da primeira iteração começar, claramente temos $i = 1$ e $capacidade = W$. Além disso, $\sum_{j=1}^{1-1} f_j w_j = 0$, já que não há o que ser somado. Então de fato $P(1)$ é verdade e o caso base fale.

Passo. Considere então que o teste executou t vezes, com $t \geq 1$. Isso significa que a t -ésima iteração vai começar. Suponha que $P(t)$ vale e vamos mostrar que $P(t+1)$ vale. Para isso, argumentamos sobre o que está sendo feito **durante** a t -ésima iteração.

Denote por $capacidade_t$ o valor de $capacidade$ no início da t -ésima iteração. Como temos que $P(t)$ vale, sabemos que $i = t$ e que $capacidade_t = W - \sum_{j=1}^{t-1} f_j w_j \geq 0$. Como a iteração começou, sabemos ainda que $capacidade_t \geq w_t$. Nessa iteração, fazemos $f_t = 1$ e $capacidade = capacidade_t - w_t$. Pela observação anterior, já podemos afirmar que $capacidade \geq 0$. Além disso, note que

$$capacidade = capacidade_t - w_t = \left(W - \sum_{j=1}^{t-1} f_j w_j \right) - f_t w_t = W - \sum_{j=1}^t f_j w_j,$$

onde usamos $P(t)$ e o fato de $f_t = 1$. Como, ao fim, $i = t + 1$, temos que $P(t+1)$ vale. \diamond

E vamos usar essa invariante para mostrar que o algoritmo devolve uma solução

viável (nesse caso, uma solução que caiba dentro da mochila). Suponha que o laço **enquanto** acabou e que i_t é o valor da variável i ao final. A invariante nos diz que $capacidade = W - \sum_{j=1}^{i_t-1} f_j w_j \geq 0$. Note que o laço termina por um dentre dois motivos. Se ele terminou porque $i_t \leq n$ e $capacidade > w_{i_t}$, o algoritmo ainda faz $f_{i_t} = capacidade/w_{i_t}$ que claramente é um valor menor ou igual a 1, e atribui 0 a todos os valores de f_j com $j > i_t$. Assim, temos que o peso total da solução é

$$\begin{aligned} \sum_{j=1}^n f_j w_j &= \sum_{j=1}^{i_t-1} f_j w_j + f_{i_t} w_{i_t} + \sum_{j=i_t+1}^n f_j w_j \\ &= W - capacidade + \frac{capacidade}{w_{i_t}} w_{i_t} + 0 = W, \end{aligned}$$

o que mostra que é uma solução viável. Por outro lado, se o laço terminou porque $i_t = n + 1$, então o algoritmo não faz mais nenhuma operação. Como $capacidade = W - \sum_{j=1}^{i_t-1} f_j w_j \geq 0$ e $i_t = n + 1$, então temos que $\sum_{j=1}^n f_j w_j \leq W$, mostrando que nesse caso também temos uma solução viável.

Com relação ao tempo de execução, considere que vamos armazenar os valores v_i , w_i e f_i em vetores, de forma que o acesso a cada um é constante, dado i . Com isso, a linha 1 leva tempo $\Theta(n \lg n)$ (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 é dominado pela ordenação, tendo tempo total $\Theta(n \lg n)$.

Finalmente, no lema a seguir vamos mostrar que o resultado desse algoritmo é na verdade uma solução ótima para o problema.

Lema 21.4

Dado um conjunto $I = \{1, 2, \dots, n\}$ de n itens, em que 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 Mochila Fracionária.

Demonstração. Seja $f = (f_1, \dots, f_n)$ a solução devolvida por MOCHILAFRACIONARIA(I, n, W). Seja $f^* = (f_1^*, \dots, f_n^*)$ uma solução ótima para a mesma instância. Se $f = f^*$, então não há o que provar. Então podemos supor que f difere de f^* em alguns valores e seja i o menor índice em que elas diferem. Como $f_j = f_j^*$ para todo $j < i$, pela escolha do i , então só pode ser que $f_i > f_i^*$, já que o algoritmo sempre toma a maior fração possível de todo item. Note ainda 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, vale que $\sum_{j=i}^n f_j w_j = \sum_{j=i}^n f_j^* w_j$.

Vamos agora criar uma outra solução $f' = (f'_1, \dots, f'_n)$ a partir de f^* . Nossa intenção é que f' não seja tão diferente de f^* e por isso fazemos inicialmente $f'_j = f_j^*$ ($= f_j$) 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.

Como $f'_i = f_i > f_i^*$, não podemos simplesmente copiar para o restante de f' os mesmos valores de f^* , pois nesse caso teríamos uma solução de peso $W + (f'_i - f_i^*)w_i$. Para garantir que f' será uma solução viável e o mais próxima possível de f^* , precisamos garantir então que o peso $\sum_{j=i}^n f'_j w_j$ seja igual ao peso da solução ótima nessas mesmas posições, isto é, igual a $\sum_{j=i}^n f_j^* w_j$ (já que o peso até a posição i é o mesmo). Assim, preencha f'_{i+1}, \dots, f'_n a partir dos valores em f_{i+1}^*, \dots, f_n^* , modificando-os para atingir esse objetivo. Veja que isso é possível, pois sabemos que existe ao menos uma solução que satisfaz nossos requisitos, já que $\sum_{j=i}^n f_j w_j = \sum_{j=i}^n f_j^* w_j$.

Agora que temos $\sum_{j=i}^n f'_j w_j = \sum_{j=i}^n f_j^* w_j$, isolamos os valores referentes a i para obter a seguinte propriedade:

$$w_i(f'_i - f_i^*) = \sum_{j=i+1}^n w_j(f_j^* - f'_j). \quad (21.1)$$

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 a propriedades do nosso algoritmo a fim de compará-lo com o valor de f^* :

$$\begin{aligned}
 \sum_{j=1}^n f'_j v_j &= \left(\sum_{j=1}^{i-1} f_j^* v_j \right) + f'_i v_i + \sum_{j=i+1}^n f'_j v_j \\
 &= \left(\sum_{j=1}^n f_j^* v_j - f_i^* v_i - \sum_{j=i+1}^n f_j^* v_j \right) + f'_i v_i + \sum_{j=i+1}^n f'_j v_j \\
 &= \sum_{j=1}^n f_j^* v_j + v_i (f'_i - f_i^*) - \sum_{j=i+1}^n v_j (f_j^* - f'_j) \\
 &= \sum_{j=1}^n f_j^* v_j + v_i (f'_i - f_i^*) \frac{w_i}{w_i} - \sum_{j=i+1}^n v_j (f_j^* - f'_j) \frac{w_j}{w_j} \\
 &\geq \sum_{j=1}^n f_j^* v_j + \frac{v_i}{w_i} (f'_i - f_i^*) w_i - \sum_{j=i+1}^n \frac{v_i}{w_i} (f_j^* - f'_j) w_j \quad (21.2)
 \end{aligned}$$

$$\begin{aligned}
 &= \sum_{j=1}^n f_j^* v_j + \frac{v_i}{w_i} \left((f'_i - f_i^*) w_i - \sum_{j=i+1}^n (f_j^* - f'_j) w_j \right) \\
 &= \sum_{j=1}^n f_j^* v_j, \quad (21.3)
 \end{aligned}$$

onde (21.2) vale pois $v_i/w_i \geq v_j/w_j$ para todo $j > i$, e (21.3) vale devido a (21.1), da construção de f' . Assim, 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 uma solução igual à f , e, portanto, concluímos que f também é ótima. \square

21.3 Compressão de dados

Ao comprimir um conjunto de dados, queremos diminuir o espaço que está sendo ocupado por ele. Na definição a seguir, simplificamos a ideia do problema considerando que estamos lidando com arquivos que possuem caracteres.

Problema 21.5: Compressão de Dados

Entrada: $\langle X, \Sigma, f \rangle$, onde X é um arquivo com caracteres pertencentes a um alfabeto Σ e cada $i \in \Sigma$ possui uma frequência f_i de aparição.

Saída: Sequência de bits (código) para representar cada caractere de modo que o arquivo binário tenha tamanho mínimo.

Por exemplo, suponha que o alfabeto é $\Sigma = \{a, b, c, d\}$. Poderíamos usar um código de **largura fixa**, fazendo $a = 00$, $b = 01$, $c = 10$ e $d = 11$. Assim, “acaba” pode ser representada em binário por “0010000100”. Mas note que o símbolo a aparece bastante, 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 “acaba” ficaria representada por “0100010”. No entanto, “0100010” poderia ser interpretada também como “baaac”, ou seja, esse código escolhido é ambíguo. De fato, o problema se deve ao bit 0 poder tanto representar o símbolo a quanto um prefixo do código do símbolo b . Podemos nos livrar desse problema com um código de largura variável que seja **livre de prefixo**. Assim, podemos fazer, $a = 0$, $b = 10$, $c = 110$ e $d = 111$.

Podemos representar os códigos dos símbolos de um alfabeto Σ por uma árvore binária onde há 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 Σ . Assim, o código formado no caminho entre a raiz e o nó rotulado por um símbolo em Σ é o código binário desse símbolo. Note que uma árvore como essa é livre de prefixo se e somente se os nós rotulados são folhas. Veja a Figura 21.3 para exemplos.

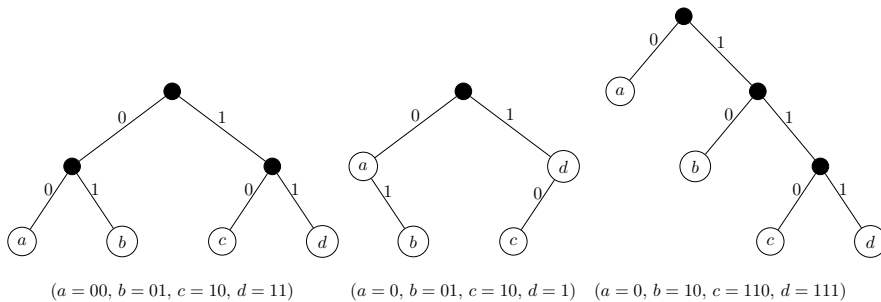


Figura 21.3: Árvores representando três códigos diferentes para o alfabeto $\Sigma = \{a, b, c, d\}$.

Seja T uma árvore que representa os códigos dos símbolos de um alfabeto Σ . Note que o **comprimento** do código de $i \in \Sigma$ é exatamente o nível do nó rotulado com i na árvore T e isso independe da quantidade exata de 0s e 1s no código. Denotaremos tal valor por $d_T(i)$. Com essa nova representação e notações, podemos redefinir o problema Compressão de Dados da seguinte forma.

Problema 21.6: Compressão de Dados

Entrada: $\langle \Sigma, f \rangle$, onde Σ é um alfabeto e cada símbolo $i \in \Sigma$ possui uma frequência f_i .

Saída: 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.

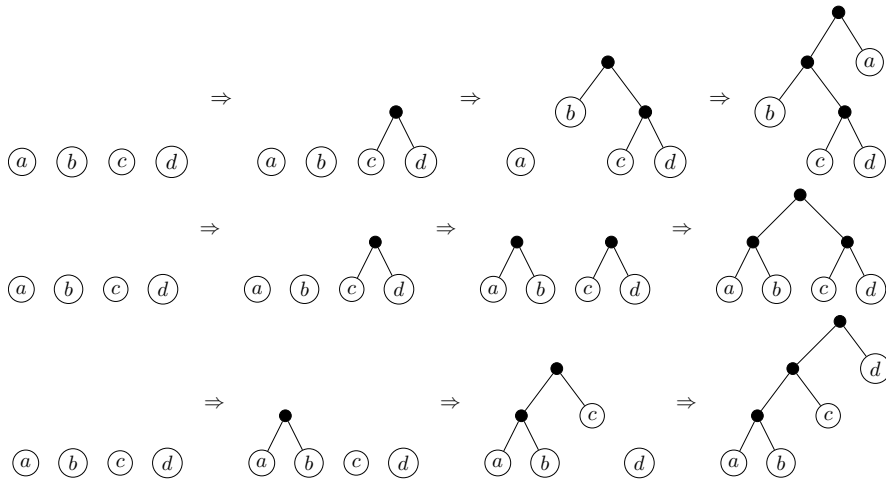


Figura 21.4: Construção de árvores representativas de códigos binários tendo início com $n = |\Sigma|$ árvores triviais.

No que seque, seja $n = |\Sigma|$. Uma forma de construir uma árvore pode ser partir de n árvores que contêm um único nó cada, um para cada $i \in \Sigma$, e repetidamente 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 21.4, conhecido como algoritmo de Huffman. Um exemplo de execução é dado na Figura 21.5.

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

$$\Sigma = \{a, b, c, d\}, f_a = 60, f_b = 25, f_c = 10, f_d = 5$$

(a) Chamada a $HUFFMAN(\{a, b, c, d\}, f)$: $|\Sigma| > 2$, junta os símbolos c e d .

$$\Sigma = \{a, b, c, d\}, f_a = 60, f_b = 25, f_c = 10, f_d = 5$$

$$T' = ?$$

$$\Sigma = \{a, b, cd\}, f_a = 60, f_b = 25, f_{cd} = 15$$

(b) Chamada a $HUFFMAN(\{a, b, cd\}, f)$: $|\Sigma| > 2$, junta os símbolos b e cd .

$$\Sigma = \{a, b, c, d\}, f_a = 60, f_b = 25, f_c = 10, f_d = 5$$

$$T' = ?$$

$$\Sigma = \{a, b, cd\}, f_a = 60, f_b = 25, f_{cd} = 15$$

$$T' = ?$$

$$\Sigma = \{a, bcd\}, f_a = 60, f_{bcd} = 40$$

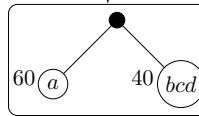
(c) Chamada a $HUFFMAN(\{a, bcd\}, f)$: $|\Sigma| = 2$.

$$\Sigma = \{a, b, c, d\}, f_a = 60, f_b = 25, f_c = 10, f_d = 5$$

$$T' = ?$$

$$\Sigma = \{a, b, cd\}, f_a = 60, f_b = 25, f_{cd} = 15$$

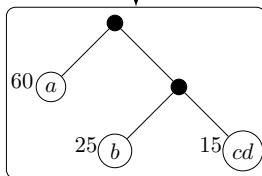
$$T' = ?$$



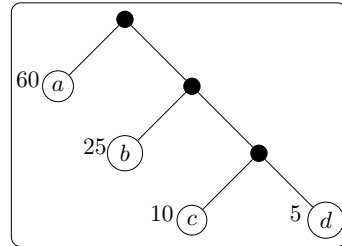
(d) Devolve árvore com dois nós.

$$\Sigma = \{a, b, c, d\}, f_a = 60, f_b = 25, f_c = 10, f_d = 5$$

$$T' = ?$$



(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 $c(T) = f_a + 2f_b + 3f_c + 3f_d$.

Algoritmo 21.4: HUFFMAN(Σ, f)

-
- 1 Sejam i e j os símbolos de menor frequência em Σ
 - 2 **se** $|\Sigma| = 2$ **então**
 - 3 **devolve** *Árvore com um nó pai não rotulado e i e j como nós filhos*
 - 4 Seja $\Sigma' = (\Sigma \setminus \{i, j\}) \cup \{ij\}$
 - 5 Defina $f_{ij} = f_i + f_j$
 - 6 $T' = \text{HUFFMAN}(\Sigma', f)$
 - 7 Construa T a partir de T' separando a folha rotulada por ij em folhas i e j
irmãs
 - 8 **devolve** T
-

frequência. Assim, podemos usar a estrutura *heap*, que fornece remoção do elemento de maior prioridade (no caso, o de menor frequência) em tempo $O(\lg n)$ sobre um conjunto de n elementos. Ela também fornece inserção em tempo $O(\lg n)$, 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 $\Theta(n \lg n)$ 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.

Lema 21.7

Dado um alfabeto Σ onde cada $i \in \Sigma$ possui uma frequência f_i , HUFFMAN(Σ, f) devolve uma solução ótima para o problema Compressão de Dados.

Demonstração. Perceba que na árvore binária T devolvida pelo algoritmo apenas as folhas são rotuladas com símbolos de Σ . Vamos mostrar por indução em $n = |\Sigma|$ que $c(T)$ é mínimo.

Quando $n = 2$, a árvore construída pelo algoritmo é claramente ótima.

Seja então $n > 2$ e Σ um alfabeto com n símbolos. Suponha que o algoritmo constrói uma árvore ótima para qualquer alfabeto de tamanho menor do que n , dadas as frequências dos símbolos.

Sejam $i, j \in \Sigma$ os dois símbolos de menor frequência em Σ . Construa Σ' a partir de Σ 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 Σ' ” e “árvores cujas folhas são rotuladas com símbolos de Σ 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 Σ' e seja \hat{T} uma árvore de $\mathcal{T}_{i,j}$. Por definição,

$$c(\hat{T}) = \sum_{k \in \Sigma \setminus \{i,j\}} f_k d_{\hat{T}}(k) + f_i d_{\hat{T}}(i) + f_j d_{\hat{T}}(j), \text{ e}$$

$$c(\hat{T}') = \sum_{k \in \Sigma' \setminus \{ij\}} f_k d_{\hat{T}'}(k) + f_{ij} d_{\hat{T}'}(ij).$$

Como $\Sigma \setminus \{i,j\} = \Sigma' \setminus \{ij\}$, 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 Σ' (isto é, minimiza $c(T')$ dentre todas as árvores para Σ'). Então, diretamente pela observação acima, a árvore correspondente T construída para Σ é ótima *dentre as árvores contidas em $\mathcal{T}_{i,j}$* . Com isso, basta mostrar que existe uma árvore ótima para Σ (dentre todas as árvores ótimas para Σ) que está contida em $\mathcal{T}_{i,j}$ para provar que T é de fato ótima para Σ .

Seja T^* qualquer árvore ótima para Σ 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 $\Psi = \Sigma \setminus \{x, y, i, j\}$. Temos, por definição,

$$c(T^*) = \sum_{k \in \Psi} f_k d_{T^*}(k) + f_x d_{T^*}(x) + f_y d_{T^*}(y) + f_i d_{T^*}(i) + f_j d_{T^*}(j), \text{ e}$$

$$c(\bar{T}) = \sum_{k \in \Psi} f_k d_{T^*}(k) + f_x d_{T^*}(i) + f_y d_{T^*}(j) + f_i d_{T^*}(x) + f_j d_{T^*}(y).$$

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

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 escrito de forma recursiva natural, com a diferença que, sempre que um subproblema for resolvido, o resultado é salvo na tabela. Assim, quando a chamada começa, o algoritmo primeiro consulta a tabela para veri-

ficar se já existe solução para o subproblema recebido. Por isso, algoritmos *top-down* são compostos por dois procedimentos, sendo 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 escrito 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. Os algoritmos Bellman-Ford (Seção 27.1.2) e Floyd-Warshall (Seção 27.2.1), clássicos em grafos, também usam programação dinâmica.

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, denotado por F_n , é dado por

$$F_n = \begin{cases} 1 & \text{se } n = 1 \text{ ou } n = 2 \\ F_{n-1} + F_{n-2} & \text{se } n > 2. \end{cases} \quad (22.1)$$

Já introduzimos, na Seção 7.5, o problema Número de Fibonacci e apresentamos algoritmos para o mesmo. Repetiremos alguns trechos daquela discussão aqui, por conveniência.

Problema 22.1: Número de Fibonacci**Entrada:** $\langle n \rangle$, onde $n \geq 1$ é um inteiro.**Saída:** F_n .

Pela definição de F_n , o Algoritmo 22.1, recursivo, segue de forma natural.

Algoritmo 22.1: FIBONACCIRECURSIVO(n)

```

1 se  $n \leq 2$  então
2   devolve 1
3 devolve FIBONACCIRECURSIVO( $n - 1$ ) + FIBONACCIRECURSIVO( $n - 2$ )

```

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 chamada a FIBONACCIRECURSIVO(6).

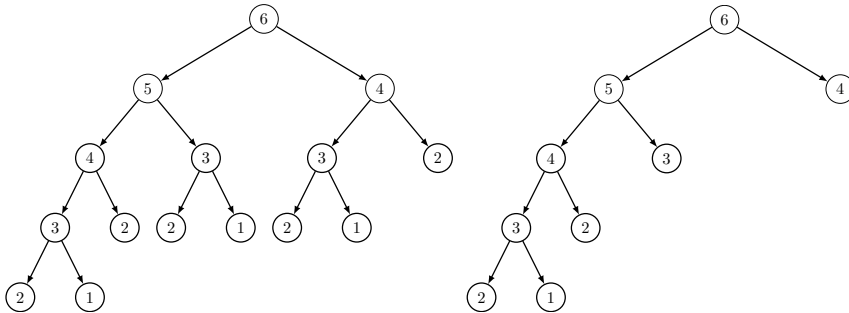


Figura 22.1: Árvore de recursão completa de FIBONACCI(6) à esquerda (Algoritmo 22.1) e de FIBONACCI-TOPDOWN(6) à direita (Algoritmo 22.2).

Cada nó representa uma chamada ao algoritmo e é rotulado com o valor de n correspondente.

O tempo de execução $T(n)$ de FIBONACCI(6) 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 e o número F_n é aproximadamente $2^{0.684n}$, precisando da ordem de n bits para ser armazenado. Considere ainda $T(1) = 1$ e $T(2) = 3$.

Podemos então usar o método da substituição para mostrar que $T(n)$ é $\Omega(((1 + \sqrt{5})/2)^n)$. Para ficar claro de onde tiramos o valor $((1 + \sqrt{5})/2)^n$, 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. Vamos provar o resultado para todo $n \geq 2$. No caso base,

temos que $T(2) \geq x^2$ para todo $x \leq \sqrt{3} \approx 1,732$.

Seja então $n > 2$ e suponha que $T(m) \geq x^m$ para todo $2 \leq m < n$. Assim, aplicando isso à definição de $T(n)$, 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 FIBONACCI-RECURSIVO é de fato muito ineficiente, tendo tempo de execução $T(n) = \Omega((1,618)^n)$.

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 FIBONACCI-RECURSIVO 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 FIBONACCI-RECURSIVO-TOPDOWN( $n$ )
```

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

Algoritmo 22.3: FIBONACCI-RECURSIVO-TOPDOWN(n)

```

1 se  $F[n] == -1$  então
2    $F[n] = \text{FIBONACCI-RECURSIVO-TOPDOWN}(n-1) +$ 
    $\text{FIBONACCI-RECURSIVO-TOPDOWN}(n-2)$ 
3 devolve  $F[n]$ 

```

procedimento FIBONACCI-RECURSIVO-TOPDOWN é chamado para calcular $F[n]$. Note que FIBONACCI-RECURSIVO-TOPDOWN tem a mesma estrutura do algoritmo recursivo natural FIBONACCI-RECURSIVO, com a diferença que em FIBONACCI-RECURSIVO-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 FIBONACCI-RECURSIVO-TOPDOWN, que todas as operações realizadas levam tempo constante, e que existem n subproblemas (calcular F_1, F_2, \dots, 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 FIBONACCI-RECURSIVO-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  $n \leq 2$  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  $F[n]$ 

```

22.2 Corte de barras de ferro

Considere uma empresa que corta e vende pedaços de barras de ferro. As barras são vendidas em pedaços de tamanho inteiro, onde um pedaço 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 e vender os pedaços de modo a maximizar o lucro obtido.

Problema 22.2: Corte de Barras de Ferro

Entrada: $\langle n, p \rangle$, onde n é um inteiro positivo indicando o tamanho da barra, sendo que cada tamanho i de barra, com $1 \leq i \leq p$, tem um preço p_i de venda.

Saída: Maior lucro obtido com a venda de pedaços inteiros de uma barra de tamanho n .

Como exemplo, considere uma barra de tamanho 6 cujos preços dos pedaços são $p_1 = 1$, $p_2 = 3$, $p_3 = 11$, $p_4 = 16$, $p_5 = 19$ e $p_6 = 10$. 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**. Em geral eles utilizam muito esforço computacional 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 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 $p_4 + p_2$. Como 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. Além disso, 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). Na verdade, nenhuma abordagem gulosa funciona para resolver o problema Corte de Barras de Ferro de forma ótima. Vamos analisá-lo 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 nosso caso base ocorre 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? Visto que temos apenas n valores diferentes para 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 $n - i$. Essa ideia recursiva é mostrada 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 uma folha. De fato, perceba que se L_k é o maior lucro obtido ao cortar

Algoritmo 22.5: CORTEBARRAS(n)

```

1 se  $n == 0$  então
2   | devolve 0
3 lucro = -1
4 para  $i = 1$  até  $n$ , incrementando faça
5   | valor =  $p_i + \text{CORTEBARRAS}(n - i)$ 
6   | se valor > lucro então
7   |   | lucro = valor
8 devolve lucro

```

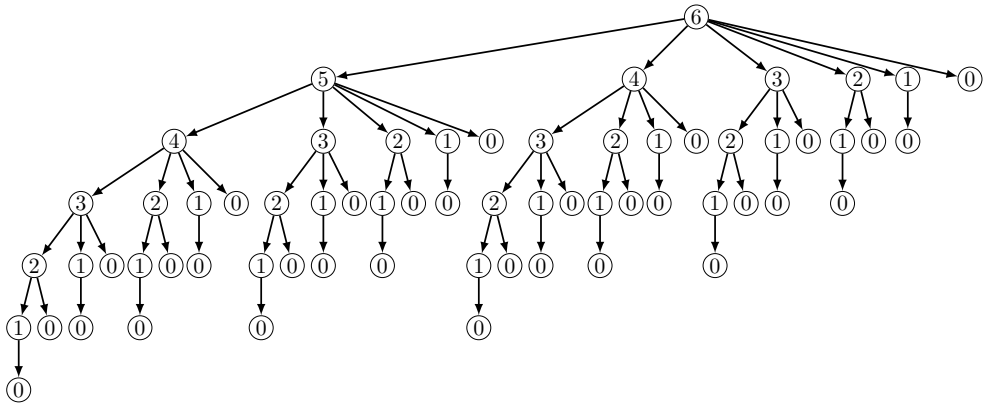


Figura 22.2: Árvore de recursão completa de CORTEBARRAS(6, p) (Algoritmo 22.5). Cada nó representa uma chamada ao algoritmo e é rotulado com o tamanho do problema correspondente.

uma barra de tamanho k (valor de solução ótima), então vale que

$$L_k = \max_{1 \leq i \leq k} \{p_i + L_{k-i}\}, \quad (22.2)$$

porque uma solução ótima para uma barra de tamanho k contém soluções ótimas para barras menores, conforme provamos a seguir. Considere uma solução ótima 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 ainda que $S^* \setminus \{j\}$ é na verdade ótima para $k - j$, pois se houvesse uma solução melhor S' para $k - j$, então 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 CORTE-

$BARRAS(n)$. Claramente, $T(0) = 1$ e $T(n) = n + \sum_{i=1}^n T(n-i)$. Vamos utilizar o método da substituição para provar que $T(n) \geq 2^n$. No caso base, claramente temos $T(0) = 1 = 2^0$. Dado então $n > 0$, suponha que $T(m) \geq 2^m$ para todo $0 \leq m \leq n-1$. Por definição de $T(n)$ e utilizando a hipótese de indução, temos que

$$T(n) = n + T(0) + T(1) + \cdots + T(n-1) \geq n + (2^0 + 2^1 + \cdots + 2^{n-1}) = 2^n - 1 + n \geq 2^n,$$

onde a última inequação vale pois $n \geq 1$. Assim, vemos que o tempo de execução continua tão ruim quanto o do algoritmo de força bruta.

Porém, existem apenas $n + 1$ subproblemas diferentes: cada um é descrito por um valor i e vale que $0 \leq i \leq n$. Podemos então utilizar um vetor para armazenar os valores de cada subproblema e acessar o valor diretamente quando necessário. O Algoritmo 22.6 é uma variação de CORTEBARRAS em que, cada vez que um subproblema é resolvido, o valor é salvo em um vetor B . É uma programação dinâmica escrita 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)

```

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( $n$ )
```

Algoritmo 22.7: CORTEBARRASRECURSIVO-TOPDOWN(k)

```

1 se  $B[k] == -1$  então
2    $lucro = -1$ 
3   para  $i = 1$  até  $k$ , incrementando faça
4      $valor = p_i + \text{CORTEBARRASRECURSIVO-TOPDOWN}(k-i)$ 
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) 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) é executado e seu primeiro passo é verificar se o subproblema em questão já foi resolvido (linha 1). Caso

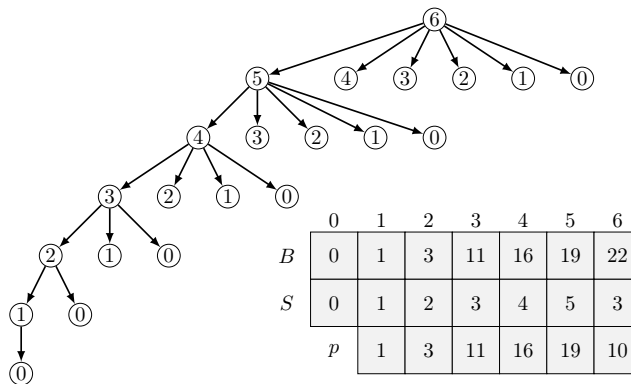


Figura 22.3: Árvore de execução completa de *CORTEBARRASRECURSIVO-TOPDOWN*(6) (Algoritmo 22.6). Cada nó representa uma chamada ao algoritmo e é rotulado com o tamanho do problema correspondente.

não tenha sido, 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) que tem, assintoticamente, o mesmo tempo de execução de *CORTEBARRASRECURSIVO-TOPDOWN*(n). 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 $0, 1, \dots, n-1$, o tempo de execução $T(n)$ de *CORTEBARRASRECURSIVO-TOPDOWN*(n) é assintoticamente dado por $T(n) = 1 + 2 + \dots + n = \Theta(n^2)$. De fato, isso também pode ser observado em sua árvore de recursão, na Figura 22.3.

Ambos algoritmos apenas devolvem o lucro obtido pelos cortes da barra. Caso precisemos de fato descobrir o tamanho dos pedaços nos quais 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, sobrando uma barra de tamanho $n - S[n]$. Para cortá-la 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), 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 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*.

Algoritmo 22.9: CORTEBARRAS-BOTTOMUP(n)

```

1 Cria vetores  $B[0..n]$  e  $S[0..n]$ 
2  $B[0] = 0$ 
3 para  $k = 1$  até  $n$ , incrementando faça
4   |  $lucro = -1$ 
5   | para  $i = 1$  até  $k$ , incrementando faça
6   |   |  $valor = p_i + B[k - i]$ 
7   |   | se  $valor > lucro$  então
8   |   |   |  $lucro = valor$ 
9   |   |   |  $S[k] = i$ 
10  |  $B[k] = lucro$ 
11 devolve  $B[n]$ 

```

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 problema muito clássico em computação. Nessa seção veremos a versão da mochila inteira ou binária. A Seção 21.2 apresenta a versão da mochila fracionária.

	0	1	2	3	4	5	6	
B	0							
B	0	1						$B[1] = \max\{p_1 + B[0]\}$ $= \max\{1 + 0\}$
B	0	1	3					$B[2] = \max\{p_1 + B[1], p_2 + B[0]\}$ $= \max\{1 + 1, 3 + 0\}$
B	0	1	3	11				$B[3] = \max\{p_1 + B[2], p_2 + B[1], p_3 + B[0]\}$ $= \max\{1 + 3, 3 + 1, 11 + 0\}$
B	0	1	3	11	16			$B[4] = \max\{p_1 + B[3], p_2 + B[2], p_3 + B[1], p_4 + B[0]\}$ $= \max\{1 + 11, 3 + 3, 11 + 1, 16 + 0\}$
B	0	1	3	11	16	19		$B[5] = \max\{p_1 + B[4], p_2 + B[3], p_3 + B[2], p_4 + B[1], p_5 + B[0]\}$ $= \max\{1 + 16, 3 + 11, 11 + 3, 16 + 1, 19 + 0\}$
B	0	1	3	11	16	19	22	$B[6] = \max\{p_1 + B[5], p_2 + B[4], p_3 + B[3], p_4 + B[2], p_5 + B[1], p_6 + B[0]\}$ $= \max\{1 + 19, 3 + 16, 11 + 11, 16 + 3, 19 + 1, 10 + 0\}$

Figura 22.4: Execução de CORTEBARRAS-TOPDOWN(6) e CORTEBARRAS-BOTTOMUP(6), com $p_1 = 1$, $p_2 = 3$, $p_3 = 11$, $p_4 = 16$, $p_5 = 19$ e $p_6 = 10$.

Problema 22.3: Mochila Inteira

Entrada: $\langle I, n, w, v, W \rangle$, onde $I = \{1, 2, \dots, n\}$ é um conjunto de n itens sendo que cada $i \in I$ tem um peso w_i e um valor v_i associados, e W é a capacidade total da mochila.

Saída: 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.

Como exemplo, considere a instância em que $n = 3$, $v_1 = 60$, $w_1 = 10$, $v_2 = 100$, $w_2 = 20$, $v_3 = 120$, $w_3 = 30$ e $W = 50$. Note que temos várias soluções viáveis (itens que cabem na mochila). Por exemplo, uma solução possível seria apenas o item 1, o que dá um peso total de $10 \leq W$ e valor total de 60. Outra solução melhor seria 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 de todas no entanto, que é a solução ótima, é escolher os itens 2 e 3, cujo peso é $50 \leq W$ e valor é 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: para cada item, podemos escolhê-lo ou não. Para cada subconjunto, em tempo $O(n)$ verificamos se os itens cabem na mochila e calculamos seu valor. Ou seja, esse algoritmo leva tempo $O(n2^n)$ e, portanto, não é eficiente.

Para facilitar as discussões a seguir, vamos denotar uma instância da mochila inteira pelo par (n, W) , que indica que temos n itens e capacidade W de mochila, deixando os valores e pesos dos itens subentendidos. Podemos tentar uma abordagem recursiva

para construir uma solução S para (n, W) da seguinte forma. Escolha um item $i \in \{1, \dots, n\}$. 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, $S = S'$ já é uma solução para (n, W) . Se você decidir por utilizá-lo, então a capacidade da mochila reduz de w_i unidades, e 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) . Perceba ainda que se $w_i > W$, então a única opção é não utilizar esse item. Por fim, veja que os casos bases aqui são dois: um em que não temos item para escolher e cuja solução é 0, pois independente do tamanho da mochila não é possível obter nenhum valor, e um em que não temos capacidade de mochila e cuja solução também é 0, 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, caso o item i caiba na mochila: colocá-lo na mochila ou não. Considerando o objetivo do problema, podemos tentar ambas e devolver a melhor opção dentre as duas (de maior valor). E qual item i escolher dentre os n disponíveis? 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 $i \in \{1, \dots, n\}$ que quisermos. Por comodidades que facilitam a implementação, vamos escolher $i = n$. O algoritmo recursivo descrito é formalizado no Algoritmo 22.10.

Algoritmo 22.10: MOCHILAINTEIRA(n, W)

```

1 se  $n == 0$  ou  $W == 0$  então
2   └ devolve 0
3 se  $w_n > W$  então
4   └ devolve MOCHILAINTEIRA( $n-1, W$ )
5 senão
6   └  $usa = v_n + \text{MOCHILAINTEIRA}(n-1, W - w_n)$ 
7   └  $naousa = \text{MOCHILAINTEIRA}(n-1, W)$ 
8   └ devolve  $\max\{usa, naousa\}$ 

```

Veja a Figura 22.5 para um exemplo da árvore de recursão da estratégia mencionada com uma instância específica e veja a Figura 22.6 para um trecho da árvore de recursão descrita sobre uma instância genérica. Note como qualquer solução está descrita em algum caminho que vai da raiz até uma folha dessas árvores.

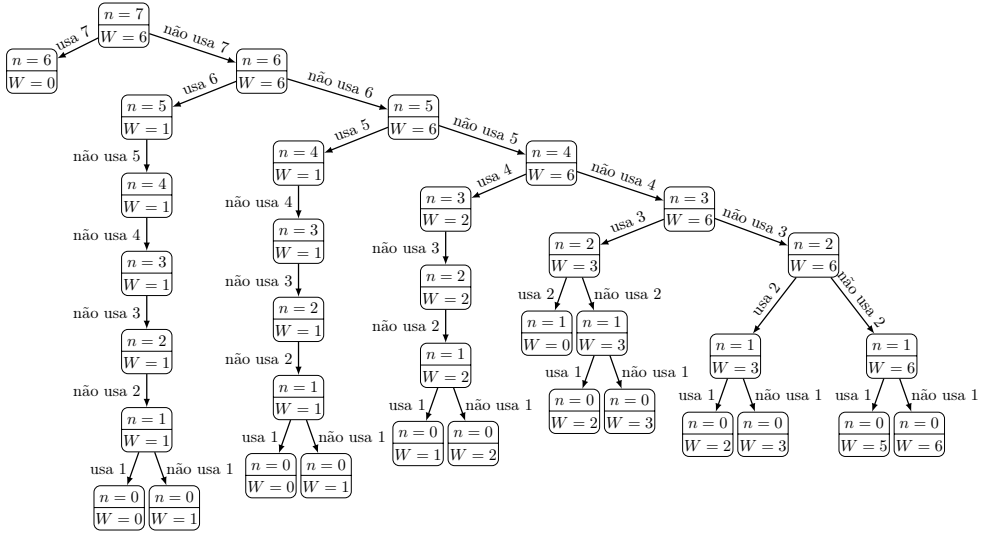


Figura 22.5: Árvore de execução completa de MOCHILAINTEIRA(7, 6) (Algoritmo 22.10), com $v_1 = 60, w_1 = 1, v_2 = 150, w_2 = 3, v_3 = 120, w_3 = 3, v_4 = 160, w_4 = 4, v_5 = 200, w_5 = 5, v_6 = 150, w_6 = 5, v_7 = 60, w_7 = 6$. Cada nó representa uma chamada algoritmo e é rotulado com o tamanho do problema correspondente.

De fato, se $V_{j,X}$ é o valor de uma solução ótima para (j, X) , então

$$V_{j,X} = \begin{cases} \max\{V_{j-1,X}, & V_{j-1,X-w_j} + v_j\} & \text{se } w_j \leq X \\ V_{j-1,X} & \text{se } w_j > X \end{cases} \quad (22.3)$$

Ademais, $V_{0,X} = V_{j,0} = 0$ para qualquer $0 \leq i \leq n$ e $0 \leq X \leq W$. Essa recorrência por sua vez é verdadeira porque uma solução ótima para uma mochila (j, X) contém soluções ótimas para mochilas menores (com menos capacidade e/ou com menos itens), como provado a seguir. 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 viável para $(j-1, X-w_j)$ e, além disso, é ótima para essa instância. Se ela não fosse ótima, então haveria S' ótima para $(j-1, X-w_j)$ com $|S'| > |S^* \setminus \{j\}|$. Mas então $S' \cup \{j\}$ é viável para (j, X) com $|S' \cup \{j\}| = |S'| + 1 > |S^* \setminus \{j\}| = |S^*|$, isto é, $S' \cup \{j\}$ teria valor melhor do que S^* , o que é uma contradição com a escolha de S^* . Se, por outro lado, $j \notin S^*$, então note que S^* já é uma solução viável para $(j-1, X-w_j)$. Também afirmamos que S^* é na verdade ótima para $(j-1, X-w_j)$ neste caso, pois se não fosse, então haveria S' ótima para $(j-1, X-w_j)$, e portanto $|S'| > |S^*|$. Mas note que S' é viável para (j, X) e teria um custo melhor do que S^* , o que também é uma contradição. A expressão acima juntamente com uma prova por indução simples mostra que o algoritmo de fato encontra uma solução ótima.

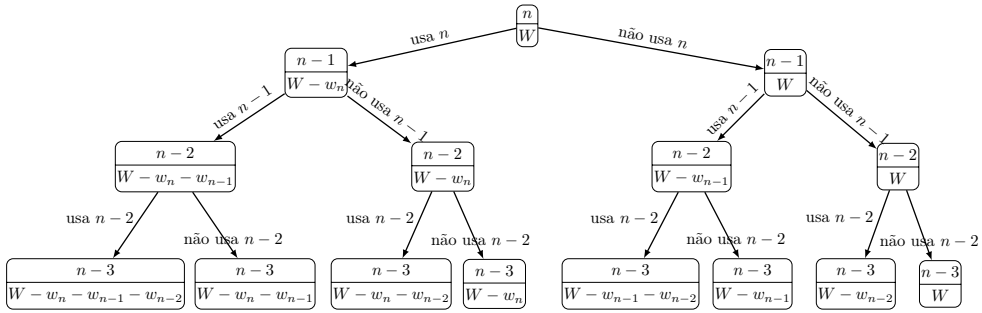


Figura 22.6: Parte da árvore de execução de $\text{MOCHILAINTEIRA}(n, W)$ (Algoritmo 22.10). Cada nó representa uma chamada ao algoritmo e é rotulado com o tamanho do problema correspondente.

O tempo de execução $T(n, W)$ de MOCHILAINTEIRA pode ser descrito pela recorrência $T(n, W) \leq T(n-1, W) + T(n-1, W-w_n) + \Theta(1)$. Ela 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 percebemos que o problema desse algoritmo está no fato de ele realizar as mesmas chamadas recursivas diversas vezes. Veja na Figura 22.5, por exemplo, a repetição dos subproblemas $(4, 1)$ e $(3, 1)$. Já na Figura 22.6, podemos ver que se tivéssemos $w_n = w_{n-1} = 1$ e $w_{n-2} = 2$, então os subproblemas $(n-3, W-w_n-w_{n-1})$ e $(n-3, W-w_{n-2})$ seriam iguais (a $(n-3, W-2)$), bem como os subproblemas $(n-3, W-w_n-w_{n-2})$ e $(n-3, W-w_{n-1}-w_{n-2})$.

Porém, podemos perceber que existem no máximo $(n+1) \times (W+1)$ subproblemas diferentes, já que um subproblema é totalmente descrito por (j, x) , e temos que $0 \leq j \leq n$ 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, porém utilizar uma matriz M de dimensões $(n+1) \times (W+1)$ nos permite um acesso mais intuitivo: $M[j][X]$ armazenará o valor $V_{j,X}$, e nosso objetivo é calcular $M[n][W]$. É importante observar também que temos dois casos bases importantes. Primeiro, se não há itens a se escolher, então não há lucro algum, não importando qual é o tamanho da mochila. Por isso, $M[0][X] = 0$ para todo $0 \leq X \leq W$. Além disso, se a capacidade da mochila é 0, então também não há lucro, não importando quantos itens estejam disponíveis. Por isso, $M[j][0] = 0$ para todo $0 \leq j \leq n$. Com essas observações, o Algoritmo 22.11 formaliza a programação dinâmica com a abordagem *top-down* e utiliza, por sua vez, o Algoritmo 22.12, enquanto que o Algoritmo 22.13 formaliza com a abordagem *bottom-up*.

A tabela a seguir mostra o resultado final da matriz após a execução dos algoritmos

Algoritmo 22.11: MOCHILAINTEIRA-TOPDOWN(n, 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 MOCHILAINTEIRA RECURSIVO-TOPDOWN( $n, W$ )

```

Algoritmo 22.12: MOCHILAINTEIRA RECURSIVO-TOPDOWN(j, X)

```

1  se  $M[j][X] == -1$  então
2      se  $w_j > X$  então
3           $M[j][X] = \text{MOCHILAINTEIRA RECURSIVO-TOPDOWN}(j - 1, X)$ 
4      senão
5           $usa = v_j + \text{MOCHILAINTEIRA RECURSIVO-TOPDOWN}(j - 1, X - w_j)$ 
6           $naousa = \text{MOCHILAINTEIRA RECURSIVO-TOPDOWN}(j - 1, X)$ 
7           $M[j][X] = \max\{usa, naousa\}$ 
8  devolve  $M[j][X]$ 

```

Algoritmo 22.13: MOCHILAINTEIRA-BOTTOMUP(n, 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  $M[n][W]$ 

```

sobre a instância em que $n = 4$, $W = 7$, $w_1 = 1$, $v_1 = 10$, $w_2 = 3$, $v_2 = 40$, $w_3 = 4$, $v_3 = 50$, $w_4 = 5$ e $v_4 = 70$:

item $\downarrow \setminus$ capacidade \rightarrow	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

Perceba que o tempo de execução desses algoritmos de programação dinâmica para o problema Mochila Inteira é $\Theta(nW)$. Agora veja que isso **não é** polinomial no tamanho da entrada! O tamanho da entrada no problema da mochila é descrito por $m = n$ e $s = \lg W$. Assim, o tempo de um algoritmo de tempo polinomial para esse problema deveria ser descrito de forma que m e s ficassem em polinômios. No entanto, $nW = m2^s$, de onde vemos que o tempo da programação dinâmica de fato não é polinomial. Por outro lado, se a entrada fosse codificada em unário, então W seria de fato o tamanho da entrada. Algoritmos cujo tempo de execução é polinomial no tamanho da entrada quando a parte numérica está codificada em unário são chamados de **pseudopolinomiais**. Observe, por fim, que se W for algum polinômio de n , então de fato os algoritmos de programação dinâmica serão polinomiais.

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. Isso é descrito no Algoritmo 22.14, que claramente executa em tempo $O(n)$.

Algoritmo 22.14: CONSTROI MOCHILA(n, W, M)

```

1  $S = \emptyset$ 
2  $X = W$ 
3  $j = n$ 
4 enquanto  $j \geq 1$  e  $X \neq 0$  faça
5     se  $w_j \leq X$  e  $M[j][X] == M[j-1][X - w_j] + v_j$  então
6          $S = S \cup \{j\}$ 
7          $X = X - w_j$ 
8      $j = j - 1$ 
9 devolve  $S$ 
```

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:

$$\begin{array}{cccccc} A & G & G & G & C & T \\ A & G & G & - & C & A \end{array} \quad \text{e} \quad \begin{array}{cccccc} A & G & G & G & C & - & T \\ A & G & G & - & C & A & - \end{array}$$

Dadas duas sequências, note que 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 $\alpha(\text{gap})$ 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 $\alpha(\text{gap}) = -1$, então o alinhamento da esquerda dado acima tem pontuação 3 enquanto que o alinhamento da direita tem pontuação 5. Definimos o problema formalmente a seguir.

Problema 22.4: Alinhamento de Sequências

Entrada: $\langle X, n, Y, m, \alpha \rangle$, onde X e Y são duas sequências sobre um mesmo alfabeto Σ , em que $X = x_1x_2 \cdots x_m$, $Y = y_1y_2 \cdots y_n$ e $x_i, y_j \in \Sigma$, e α é uma função de pontuação.

Saída: Alinhamento entre X e Y de pontuação máxima.

Uma possível abordagem recursiva para esse problema é a seguinte. Para reduzir o tamanho da entrada, podemos remover um ou mais caracteres de X e/ou de Y , pois assim continuamos com duas sequências. 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 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

¹Existem variações onde caracteres diferentes têm penalidades diferentes ao serem alinhados com *gaps*.

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*.

Note que essas três envolvem todas as possibilidades pois a chamada recursiva para resolver o alinhamento de $x_1x_2 \cdots x_m$ com $y_1y_2 \cdots y_{n-1}$ envolve a possibilidade de alinhar x_m com qualquer outro caractere de Y . Note ainda que essa é uma abordagem recursiva válida, pois estamos sempre reduzindo o tamanho de uma das duas sequências. Assim, se uma das sequências não tem nenhum caractere, basta alinhar a outra apenas com *gaps*. Mas qual das três possibilidades escolher? Como sempre, podemos testar as três e escolher a que dá a melhor pontuação.

Observe que essa abordagem considera todas as possibilidades de alinhamento possíveis entre as duas sequências iniciais X e Y . Veja na Figura 22.7 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 melhor pontuação obtida ao alinhar $x_1x_2 \dots x_i$ com $y_1y_2 \dots y_j$, então

$$P_{i,j} = \max\{\alpha(x_i, y_j) + P_{i-1,j-1}, \alpha(\text{gap}) + P_{i-1,j}, \alpha(\text{gap}) + P_{i,j-1}\}. \quad (22.4)$$

Ademais, note que $P_{0,j} = j\alpha(\text{gap})$ e $P_{i,0} = i\alpha(\text{gap})$, pois a única opção é alinhar os caracteres da sequência restante com *gaps*. A recorrência acima é verdadeira pois qualquer solução ótima para alinhar $x_1x_2 \dots x_i$ com $y_1y_2 \dots y_j$ contém alinhamentos ótimos de sequências menores, como provado a seguir. Por conveniência, vamos denotar uma cadeia $a_1a_2 \dots a_m$ por $A_{1..m}$. Seja O^* um alinhamento ótimo para alinhar $X_{1..i}$ com $Y_{1..j}$. 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 última posição, deve ser um alinhamento ótimo para alinhar $X_{1..i-1}$ com $Y_{1..j-1}$: se não fosse esse o caso, então teríamos um alinhamento ótimo O'' para $X_{1..i-1}$ com $Y_{1..j-1}$ que poderia ser combinado ao alinhamento de x_i com y_j e render um alinhamento melhor do que O^* para $X_{1..i}$ com $Y_{1..j}$, uma contradição. Outra possibilidade é O^* ter, na última posição, x_i alinhado com um *gap*. Nesse caso, O' deve ser um alinhamento ótimo para $X_{1..i-1}$ com $Y_{1..j}$: não fosse esse o caso, teríamos um alinhamento ótimo O'' para $X_{1..i-1}$ com $Y_{1..j}$ que poderia ser combinado ao alinhamento de x_i com um *gap* e render um alinhamento melhor do que O^* . Por fim, se a última posição de O^* tiver y_j alinhado com um *gap*, então O' deve ser um alinhamento ótimo para $X_{1..i}$ com $Y_{1..j-1}$, por um argumento semelhante aos anteriores. A expressão acima juntamente com uma prova por indução simples mostra que esse algoritmo devolve uma solução ótima para o problema.

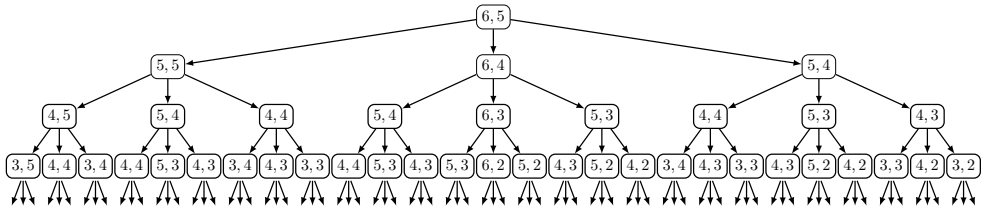


Figura 22.7: Árvore de execução completa do algoritmo recursivo simples para o alinhamento das sequências $X = AGGGCT$ e $Y = AGGCA$ de tamanhos $n = 6$ e $m = 5$, respectivamente. Cada nó representa uma chamada ao algoritmo e é rotulado com um par m, n , referente ao tamanho do problema correspondente.

Também é fácil perceber pela Figura 22.7 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 $0 \leq i \leq m$ e $0 \leq j \leq n$. 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 nosso objetivo é calcular $M[m][n]$. O Algoritmo 22.15 mostra um algoritmo de programação dinâmica na abordagem *bottom-up* para o problema 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     $M[i][0] = i \times \alpha(\text{gap})$ 
4  para  $j = 0$  até  $n$ , incrementando faça
5     $M[0][j] = j \times \alpha(\text{gap})$ 
6  para  $i = 1$  até  $m$ , incrementando faça
7    para  $j = 1$  até  $n$ , incrementando faça
8       $M[i][j] = \max\{M[i-1][j-1] + \alpha(x_i, y_j), M[i-1][j] + \alpha(\text{gap}), M[i][j-1] + \alpha(\text{gap})\}$ 
9  devolve  $M[m][n]$ 

```

Não é difícil perceber que este algoritmo tem tempo $\Theta(nm)$. É um bom exercício escrever sua versão na abordagem *top-down* e encontrar o alinhamento final uma vez que temos a matriz M preenchida.

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 as nove ligações sem que qualquer uma das linhas se cruzem?

Não, e é possível provar isso!

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 dos 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.

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 $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$, $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$, $\psi(e_1) = \{v_5, v_5\}$, $\psi(e_2) = \{v_2, v_3\}$, $\psi(e_3) = \{v_0, v_3\}$, $\psi(e_4) = \{v_4, v_5\}$, $\psi(e_5) = \{v_5, v_1\}$, $\psi(e_6) = \{v_0, v_1\}$, $\psi(e_7) = \{v_0, v_3\}$, $\psi(e_8) = \{v_0, v_5\}$ e $\psi(e_9) = \{v_3, v_4\}$, é um grafo.

Um **digrafo** D também é uma tripla (V, E, ψ) , onde V e E são definidos como acima, com E mais normalmente sendo chamado de um conjunto de **arcos**, e a diferença é que a função de incidência ψ associa um arco a um par *ordenado* de vértices. Por exemplo, $J = (V, E, \psi)$ em que $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$, $E = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9\}$, $\psi(a_1) = (v_5, v_5)$, $\psi(a_2) = (v_2, v_3)$, $\psi(a_3) = (v_0, v_3)$, $\psi(a_4) = (v_4, v_5)$, $\psi(a_5) = (v_5, v_1)$, $\psi(a_6) = (v_0, v_1)$, $\psi(a_7) = (v_0, v_3)$, $\psi(a_8) = (v_0, v_5)$ e $\psi(a_9) = (v_3, v_4)$, 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

¹Alguns 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.

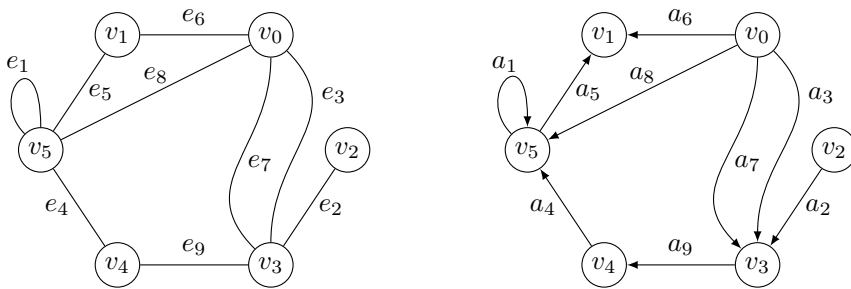


Figura 23.1: Representação gráfica de um grafo à esquerda e de um digrafo à direita. Círculos representam os vértices e linhas ou setas representam as arestas ou arcos, respectivamente.

acima. O rótulo de um vértice aparece dentro do círculo que o representa e o rótulo de uma aresta/arco aparece próximo à sua linha/seta.

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, a partir de agora, definir um (di)grafo sem precisar nomear os elementos da tripla, o que é bastante útil quando estamos lidando com mais de um desses objetos por vez. 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*, respectivamente. Os termos **multigrafo** e **multidigrafo** 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 e J em que $V(J) = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ e $E(J) = \{(v_0, v_1), (v_0, v_2), (v_2, v_0), (v_0, v_3), (v_4, v_0),$

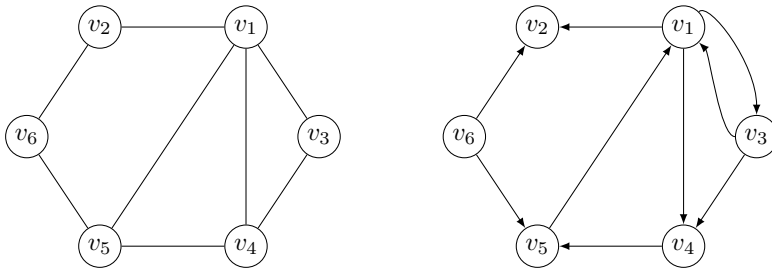


Figura 23.2: Exemplos de um grafo e um digrafo (simples).

$(v_2, v_3), (v_3, v_4), (v_5, v_4), (v_5, v_1)\}$ é um digrafo. Eles são representados na Figura 23.2.

Em geral, vamos denotar uma aresta $e = \{x, y\}$ ou um arco $a = (x, y)$ simplesmente como xy . É importante observar que se xy é uma aresta, então $xy = yx = \{x, y\}$, mas no caso de arcos temos $xy = (x, y) \neq yx = (y, x)$.

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.

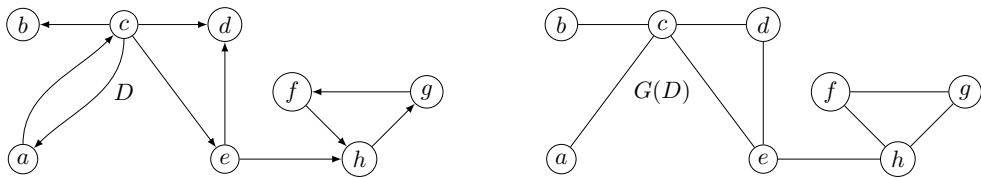


Figura 23.3: Digrafo D e seu grafo subjacente $G(D)$.

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.

Também é possível obter um digrafo \vec{G} a partir de 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** . Veja a Figura 23.4 para um exemplo. Se nos é dado um digrafo D qualquer e ele é uma orientação de algum grafo H , então D é chamado de **grafo orientado**.

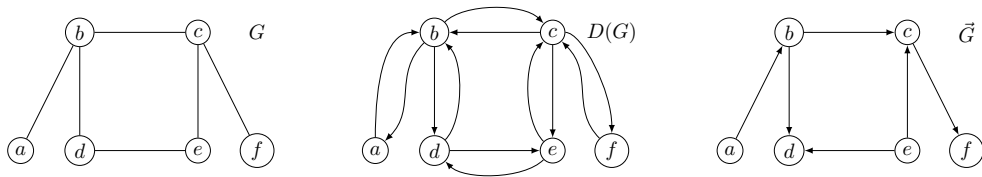


Figura 23.4: Um grafo G , seu digrafo associado $D(G)$ e uma possível orientação de G .

23.2 Adjacências e incidências

Seja $e = xy$ uma aresta de um grafo G . Dizemos que os vértices x e y são **vizinhos** ou **adjacentes**. Os vértices x e y são chamados de **extremos** da aresta xy . Arestas que possuem um extremo em comum também 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 , chamado de **vizinhança** de x , é denotado por $N_G(x)$. Naturalmente, dado um conjunto $X \subseteq V(G)$, definimos a **vizinhança** de X como a união das vizinhanças de cada $x \in X$, isto é, $N_G(X) = \bigcup_{x \in X} N_G(x)$. 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., $\delta(G) = \min\{d(x) : x \in V(G)\}$. O **grau máximo** de G , denotado por $\Delta(G)$, é o maior grau dentre todos os vértices de G , i.e., $\Delta(G) = \max\{d(x) : x \in V(G)\}$. 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, principalmente se estivermos pensando no seu grafo subjacente. Porém, existem conceitos que se aplicam *apenas* a digrafos.

Seja $a = xy$ um arco de um digrafo D . Também dizemos que os vértices x e y são **vizinhos** ou **adjacentes**. Dizemos ainda que x é a **cabeça** de a e que y é a **cauda**, sendo ambos x e y os **extremos** de a . É comum dizer também o arco xy **sai** de x e **entra** em y .

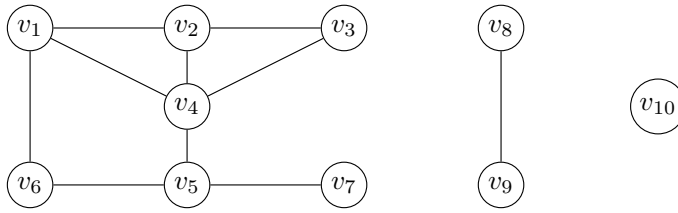


Figura 23.5: Grafo G com $d(v_1) = 3$, $d(v_2) = 3$, $d(v_3) = 2$, $d(v_4) = 4$, $d(v_5) = 3$, $d(v_6) = 2$, $d(v_7) = 1$, $d(v_8) = 1$, $d(v_9) = 1$ e $d(v_{10}) = 0$. Assim, $\delta(G) = 0$, $\Delta(G) = 4$ e $\bar{d}(G) = (3 + 3 + 2 + 4 + 3 + 2 + 1 + 1 + 1 + 0)/10 = 2$. Note que $N(v_1) = \{v_2, v_4, v_6\}$ 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.

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.

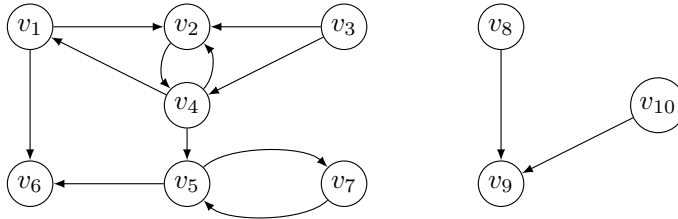


Figura 23.6: Digrafo D com $d^+(v_1) = 2$, $d^-(v_1) = 1$, $d^+(v_2) = 1$, $d^-(v_2) = 3$, $d^+(v_3) = 2$, $d^-(v_3) = 0$, $d^+(v_4) = 2$, $d^-(v_4) = 2$, $d^+(v_5) = 2$, $d^-(v_5) = 2$, $d^+(v_6) = 0$, $d^-(v_6) = 2$, $d^+(v_7) = 1$, $d^-(v_7) = 1$, $d^+(v_8) = 1$, $d^-(v_8) = 0$, $d^+(v_9) = 0$, $d^-(v_9) = 2$, $d^+(v_{10}) = 1$ 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 $N^-(v_{10}) = \emptyset$.

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 23.2

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 $\sum_{x \in V(G)} d_G^+(x) = \sum_{x \in V(G)} d^-(x) = e(G)$.

Demonstração. Um arco uv é contado uma vez no grau de saída de u e uma vez no grau de entrada de v . \square

Uma outra importante relação é a de que $e(G)$ é $O(v(G)^2)$. Mais precisamente, vale que

$$e(G) \leq \frac{v(G)(v(G) - 1)}{2}, \quad (23.1)$$

onde o lado direito da inequação é o máximo de arestas que um grafo pode ter, o que ocorre se cada um dos $v(G)$ vértices for vizinho de todos os outros $v(G) - 1$ vértices. A divisão por 2 é necessária por estarmos considerando uma mesma aresta duas vezes nessa contagem.

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 $c: V(G) \rightarrow \mathcal{N}$ e/ou uma função $w: E(G) \rightarrow \mathcal{N}$, onde \mathcal{N} em geral é algum conjunto numérico, como os números reais (\mathbb{R}) ou os inteiros (\mathbb{Z}).

Graficamente, esses valores são indicados próximos às arestas ou aos vértices. Veja a Figura 23.7 para alguns exemplos.

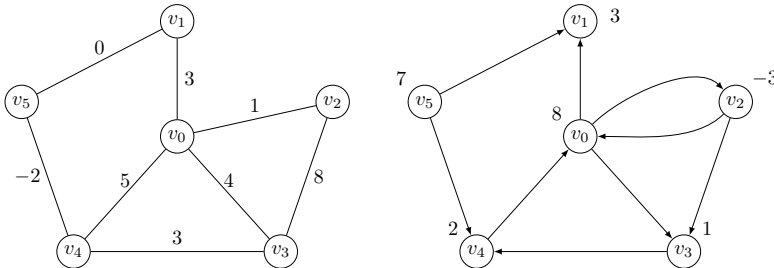


Figura 23.7: Exemplos de ponderação nas arestas e nos vértices.

Se um (di)grafo G é ponderado nas arestas por uma função $w: E(G) \rightarrow \mathcal{N}$, 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 $c: V(G) \rightarrow \mathcal{N}$, 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 em computadores

Certamente podemos representar (di)grafos em um computador simplesmente utilizando conjuntos para vértices e arestas/arcs, ou mesmo desenhos. Porém, é desejável utilizar alguma estrutura de dados que nos permita ganhar em eficiência dependendo da tarefa que necessitamos. Conforme o número de vértices e arestas aumenta, desenhos ou apenas os conjuntos de vértices e arestas não são representações razoáveis. As duas formas mais comuns de representação de (di)grafos em computadores são *listas de adjacências* e *matriz de adjacências*.

A partir desta seção, vamos assumir que qualquer (di)grafo com n vértices tem como conjunto de vértices $\{1, 2, \dots, n\}$. Note que caso você tenha um grafo cujos rótulos dos vértices são diferentes destes números, basta criar uma estrutura de dicionário para realizar a conversão do rótulo para o devido número.

Na representação por *listas de adjacências*, um (di)grafo G é dado por $v(G)$ listas ligadas, uma para cada vértice. A lista de um vértice i contém apenas os vizinhos de i se G é grafo, e apenas os vizinhos de saída de i se G é digrafo. Assim, são necessários $\Theta(v(G))$ ponteiros para as listas e a lista de um vértice i tem $d(i)$ nós. Pelos Teoremas 23.1 e 23.2, podemos concluir que a quantidade de nós em todas as listas é, portanto, $\Theta(e(G))$. 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.

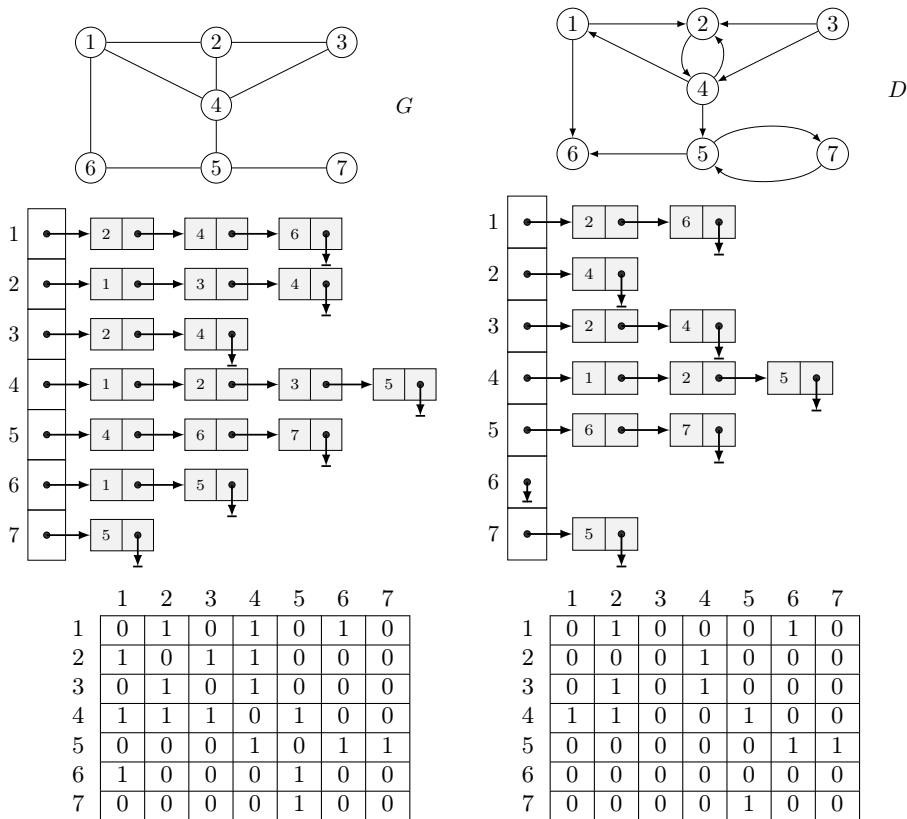


Figura 23.8: Representação gráfica de um grafo G e um digrafo D na primeira linha, suas listas de adjacências na segunda linha e suas matrizes de adjacências na última linha.

Em geral, o uso de listas de adjacências é preferido para representar (di)grafos **esparços**, que são aqueles em que $e(G) = O(v(G))$, pois o espaço $\Theta(v(G)^2)$ necessário pela matriz de adjacências acaba sendo dispendioso. Já a representação por matriz de adjacências é muito boa para representar (di)grafos **densos**, que são aqueles com $\Theta(v(G)^2)$ arestas. Porém, esse não é o único fator importante na escolha da estrutura de dados utilizada na representação, pois determinados algoritmos podem ganhar eficiência a depender das operações que precisam, que podem ser melhores em listas ou

matriz. Por exemplo, se já temos dois dados vértices i e j , podemos verificar se eles são vizinhos em tempo $O(1)$ em matriz e em tempo $O(d(i))$ em listas.

Por fim, se o (di)grafo é ponderado, 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 conter um campo para armazenar esses valores. 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 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 toda vértice  $x \in V(G)$  faça
3   se  $d_G(x) > \text{max}$  então
4     max ←  $d_G(x)$ 
5 devolve max
```

Note como o algoritmo usa notações que muito provavelmente não estarão disponíveis em qualquer linguagem de programação, como “ $x \in V(G)$ ”, ou não estarão disponíveis em tempo constante sempre, como $d_G(x)$. Se na nossa implementação mantivermos os graus dos vértices já calculados, por exemplo em um vetor indexado pelos vértices, então de fato uma execução da linha 4 leva tempo constante. Porém, vamos considerar que essa facilidade não está disponível. Assim, veja sua reescrita no Algoritmo 23.2 para que fique claro que uma execução da linha 4 não leva tempo constante.

Algoritmo 23.2: GRAUMAXIMO(G)

```

1 max ← 0
2 para toda vértice  $x \in V(G)$  faça
3   graux ← 0
4   para toda vértice  $y \in N(x)$  faça
5     graux ← graux + 1
6   se graux > max então
7     max ← graux
8 devolve max

```

Esse novo algoritmo, para cada vértice, primeiro calcula o seu grau e então verifica se é o maior grau que já foi encontrado até o momento. Agora inserimos outra notação em conjuntos, “ $y \in N(x)$ ”, que indica o acesso à vizinhança de um vértice, e é justamente o que muda quando usamos listas ou em matriz de adjacências. Os Algoritmos 23.3 e 23.4 mostram mais detalhes da representação do grafo.

Algoritmo 23.3: GRAUMAXIMO(M, n)

```

1 max ← 0
2 para  $x \leftarrow 1$  até  $n$ , incrementando faça
3   graux ← 0
4   para  $y \leftarrow 1$  até  $n$ , incrementando faça
5     se  $M[x][y] = 1$  então
6       graux ← graux + 1
7   se graux > max então
8     max ← graux
9 devolve max

```

No que segue, vamos usar $n = v(G)$ e $m = e(G)$ e vamos denotar o conjunto de vértices por $V(G) = \{1, 2, \dots, n\}$.

No Algoritmo 23.3, o tempo $T_M(n, m)$ é dado pela seguinte expressão:

$$\begin{aligned}
T_M(n, m) = & \underbrace{\Theta(1)}_{\text{linha 1}} + \underbrace{\sum_{x=1}^n \Theta(1)}_{\text{linha 2}} + \underbrace{\sum_{x=1}^n \Theta(1)}_{\text{linha 3}} + \underbrace{\sum_{x=1}^n \sum_{y=1}^n \Theta(1)}_{\text{linha 4}} + \underbrace{\sum_{x=1}^n \sum_{y=1}^n \Theta(1)}_{\text{linha 5}} \\
& + \underbrace{\sum_{x=1}^n \Theta(d(x))}_{\text{linha 6}} + \underbrace{\sum_{x=1}^n \Theta(n)}_{\text{linha 7}} + \underbrace{O(n)}_{\text{linha 8}} + \underbrace{\Theta(1)}_{\text{linha 9}},
\end{aligned}$$

Algoritmo 23.4: GRAUMAXIMO(L, n)

```

1 max ← 0
2 para x ← 1 até n, incrementando faça
3   graux ← 0
4   atual ← L[x]
5   enquanto atual ≠ NULL faça
6     graux ← graux + 1
7     atual ← atual.prox
8   se graux > max então
9     max ← graux
10 devolve max

```

onde usamos o fato de que a linha 6 será executada exatamente $d(x)$ vezes. A expressão acima é por sua vez é igual a

$$T_M(n, m) = \underbrace{\Theta(n)}_{\text{linha 2}} + \underbrace{\Theta(n)}_{\text{linha 3}} + \underbrace{\Theta(n^2)}_{\text{linha 4}} + \underbrace{\Theta(n^2)}_{\text{linha 5}} + \underbrace{\Theta(m)}_{\text{linha 6}} + \underbrace{\Theta(n)}_{\text{linha 7}} + \underbrace{O(n)}_{\text{linha 8}} = \Theta(n^2),$$

onde usamos o fato de que m é $O(n^2)$ (Eq. (23.1)) e que $\sum_{v \in V(G)} d(v) = 2m$ (Teorema 23.1) para qualquer grafo. Note que a linha 8 executa no máximo uma vez por vértice do grafo, sendo que uma única execução leva tempo constante, e por isso é $O(n)$. Logo, o tempo do algoritmo com matriz de adjacências é $\Theta(n^2)$.

Já para o Algoritmo 23.4 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 $T_L(n, m)$ é dado por:

$$\begin{aligned}
T_L(n, m) &= \underbrace{\Theta(1)}_{\text{linha 1}} + \underbrace{\sum_{x=1}^n \Theta(1)}_{\text{linha 2}} + \underbrace{\sum_{x=1}^n \Theta(1)}_{\text{linha 3}} + \underbrace{\sum_{x=1}^n \Theta(1)}_{\text{linha 4}} + \underbrace{\sum_{x=1}^n O(n)}_{\text{linha 5}} \\
&\quad + \underbrace{\sum_{x=1}^n O(n)}_{\text{linha 6}} + \underbrace{\sum_{x=1}^n O(n)}_{\text{linha 7}} + \underbrace{\sum_{x=1}^n \Theta(1)}_{\text{linha 8}} + \underbrace{O(n)}_{\text{linha 9}} + \underbrace{\Theta(1)}_{\text{linha 10}} \\
&= \Theta(1) + \Theta(n) + O(n^2) + O(n),
\end{aligned}$$

de onde vemos que $T_L(n, m)$ é $O(n^2)$. Essa análise nos leva a um pior caso assintoticamente igual ao das matrizes de adjacências.

Vejam agora a análise mais justa. Para ela, lembra-se que cada vértice i tem exatamente $d(i)$, seu grau, vértices em sua lista de adjacências. Assim, o tempo $T_L(n, m)$ é dado por:

$$\begin{aligned}
 T_L(n, m) &= \underbrace{\Theta(1)}_{\text{linha 1}} + \underbrace{\sum_{x=1}^n \Theta(1)}_{\text{linha 2}} + \underbrace{\sum_{x=1}^n \Theta(1)}_{\text{linha 3}} + \underbrace{\sum_{x=1}^n \Theta(1)}_{\text{linha 4}} + \underbrace{\sum_{x=1}^n \Theta(d(x))}_{\text{linha 5}} + \\
 &+ \underbrace{\sum_{x=1}^n \Theta(d(x))}_{\text{linha 6}} + \underbrace{\sum_{x=1}^n \Theta(d(x))}_{\text{linha 7}} + \underbrace{\sum_{x=1}^n \Theta(1)}_{\text{linha 8}} + \underbrace{O(n)}_{\text{linha 9}} + \underbrace{\Theta(1)}_{\text{linha 10}} \\
 &= \Theta(1) + \Theta(n) + \Theta(m) + O(n) = \Theta(n + m),
 \end{aligned}$$

pois $\sum_{x=1}^n d(x) = 2m$. Note ainda que esse resultado não invalida o anterior, já que m é de fato $O(n^2)$. No entanto, $\Theta(n + m)$ pode ser bem melhor do que $O(n^2)$, dependendo do grafo dado.

23.6 Subgrafos

Um (di)grafo H é **sub(di)grafo** de um (di)grafo G se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. 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 $G[S]$, é o sub(di)grafo H de G tal que $V(H) = S$ e $E(H)$ é o conjunto de arestas de G com os dois extremos em S , i.e., $E(H) = \{uv : uv \in E(G) \text{ e } u, v \in S\}$. Similarmente, se F é um subconjunto de arestas de G , então o sub(di)grafo de G **induzido por F** , denotado $G[F]$, 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., $V(H) = \{v : \text{existe } u \text{ com } uv \in F\}$.

Denotamos por $G - X$ e $G - F$, respectivamente, os (di)grafos obtidos de G pela remoção de $X \subseteq V(G)$ e $F \subseteq E(G)$, de forma que $G - X = G[V(G) \setminus X]$ e $G - F$ é tal que $V(G - F) = V(G)$ e $E(G - F) = E(G) \setminus F$. Note então que $G - X$ e $G - F$ são sub(di)grafos de G .

Ademais, dado um (di)grafo G , um conjunto de vértices S' que não está em $V(G)$

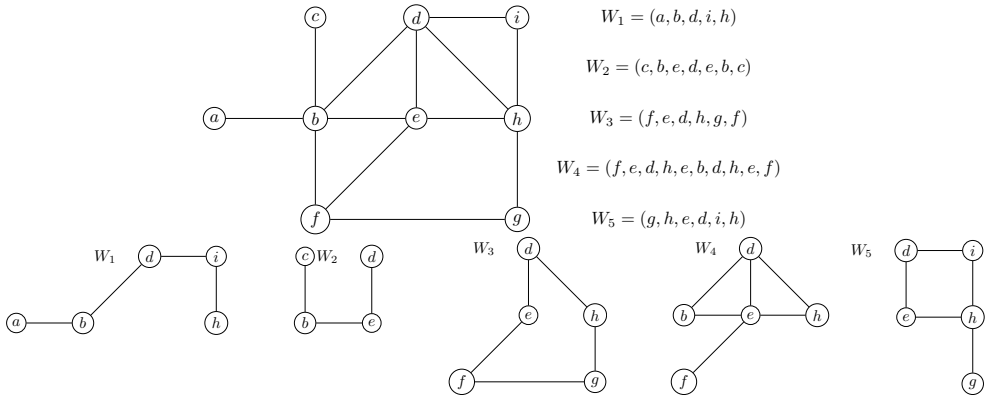


Figura 23.9: Ao topo da figura, um grafo G e passeios W_1, \dots, W_5 sobre ele. Na parte inferior, os subgrafos induzidos pelas arestas desses passeios. Os passeios W_1 e W_5 são abertos enquanto que W_2, W_3 e W_4 são fechados. Veja que W_1 é um caminho e W_3 é um ciclo. Veja que W_2 não é um caminho, apesar do grafo induzido pelas suas arestas ser.

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' , de forma que $G + S'$ é tal que $V(G + S') = V(G) \cup S'$ e $E(G + S') = E(G)$ e $G + F'$ é tal que $V(G + F') = V(G)$ e $E(G + F') = E(G) \cup F'$. Note então que G é sub(di)grafo de $G + S'$ e $G + F'$.

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 $w: E(G) \rightarrow \mathbb{R}$, então o peso de H , denotado $w(H)$, é dado pela soma do 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 $c: V(G) \rightarrow \mathbb{R}$, então o peso de H , denotado $c(H)$, é dado pela soma do 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.

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. Formalmente, $G - F$ é tal que $V(G - F) = V(G)$ e $E(G - F) = E(G) \setminus F$.

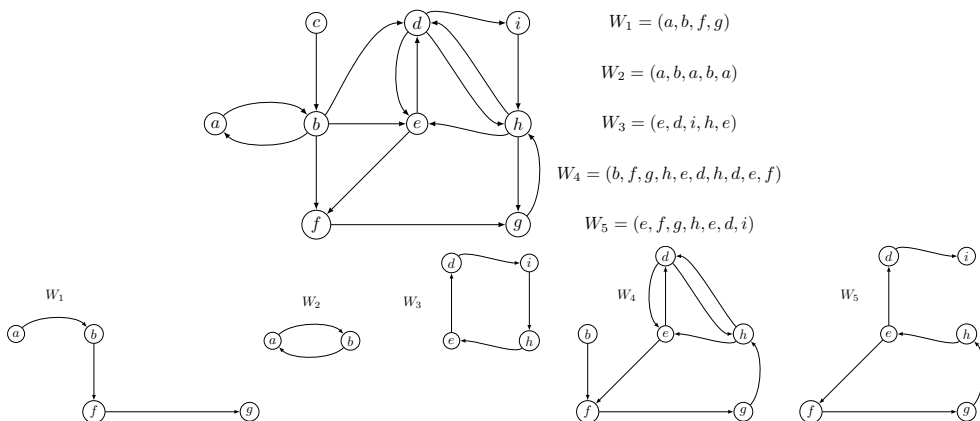


Figura 23.10: Ao topo da figura, um digrafo D e passeios W_1, \dots, W_5 sobre ele. Na parte inferior, os subdigrafos induzidos pelas arestas desses passeios. Os passeios W_1 , W_4 e W_5 são abertos enquanto que W_2 e W_3 são fechados. Veja que W_1 é um caminho e W_3 é um ciclo. Veja que W_2 não é um ciclo, apesar do grafo induzido pelas suas arestas ser. Note que (b, f, e, h) não é um passeio em D .

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 $S \not\subseteq V(G)$, 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 $F \not\subseteq E(G)$ 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 $W = (v_0, v_1, \dots, v_k)$ 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, \dots, v_{k-1} são **vértices internos**. Dizemos ainda que W conecta v_0 a v_k e que ele é um **$v_0 v_k$ -passeio**. As arestas $v_0 v_1, v_1 v_2, \dots, v_{k-1} v_k$ são chamadas de **arestas do passeio**. O **comprimento** do passeio é dado pelo número de arestas do passeio. Um passeio é

dito **fechado** se sua origem e término são iguais. Dizemos que um passeio é **aberto** quando queremos enfatizar o fato de ele não ser fechado.

Quando conveniente, podemos observar o subgrafo induzido pelas arestas de um passeio $W = (v_0, v_1, \dots, v_k)$. Assim, vamos usar a notação $V(W)$ e $E(W)$ para nos referir aos conjuntos $\{v_0, \dots, v_k\}$ e $\{v_i v_{i+1} : 0 \leq i < k\}$, respectivamente. Veja as Figuras 23.9 e 23.10 para exemplos das definições acima.

Seja G um (di)grafo e $W = (v_0, v_1, \dots, v_k)$ um passeio em G com $v_0 \neq v_k$ (aberto). 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** também para denotar (di)grafos que tenham n vértices v_1, v_2, \dots, v_n e $n - 1$ arestas do tipo $v_i v_{i+1}$. 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** também para nos referirmos a (di)grafos que tenham n vértices v_1, v_2, \dots, v_n e n arestas, sendo estas do tipo $v_i v_{i+1}$ para $1 \leq i < n$ e a aresta $v_n v_1$. 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

O termo **maximal** está relacionado com o conceito de *inclusão*: um subgrafo com uma propriedade \mathcal{P} é **maximal** se não está contido em outro subgrafo que tem a mesma propriedade \mathcal{P} . Já o termo **máximo** está relacionado com o conceito de *quantidade*: um subgrafo com uma propriedade \mathcal{P} é **máximo** se não existe nenhum outro subgrafo que tem a mesma propriedade \mathcal{P} e que tenha tamanho maior do que o primeiro.

Note que ambos conceitos dependem do subgrafo ter uma *propriedade*. 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”. Exemplificando especificamente com a propriedade de ser um caminho: um caminho $P = (v_1, \dots, v_k)$ em um grafo é maximal se não

existir nenhum caminho da forma (v, v_1, \dots, v_k) ou (v_1, \dots, v_k, v) em G ; um caminho $P = (v_1, \dots, v_k)$ é máximo se não existe nenhum caminho $(u_1, \dots, u_k, u_{k+1})$ em G . Assim notamos que todo caminho máximo é também maximal.

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 suas componentes conexas. A quantidade de componentes de um grafo G é denotada por $c(G)$. Um grafo conexo possui uma única componente conexa enquanto um grafo desconexo possui mais de uma.

Por exemplo, o grafo G da Figura 23.5 é desconexo e possui 3 componentes conexas (uma possui os vértices v_1, \dots, v_7 , outra possui os vértices v_8, v_9 e a terceira possui o vértice v_{10} apenas). Perceba que o conjunto de vértices $\{v_1, v_2, v_3\}$ sozinhos não formam uma componente conexa pois eles estão contidos no conjunto $\{v_1, v_2, v_3, v_4\}$, que também é conexo. 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 **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). Note que nem todos os arcos de um digrafo farão parte de uma componente fortemente conexa. Veja a Figura 23.11 para um exemplo.

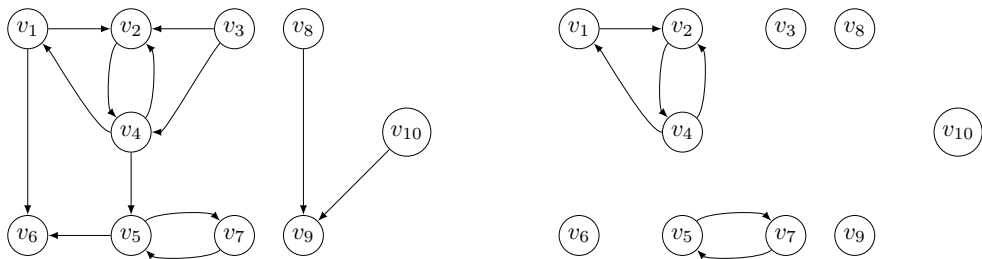


Figura 23.11: Digrafo D à esquerda, que não é fortemente conexo (por exemplo, porque não há caminho entre v_6 e v_3) e possui 7 componentes fortemente conexas, em destaque à direita.

23.9 Distância entre vértices

Seja G um (di)grafo não ponderado. A *distância entre u e v em G* , denotada por $\text{dist}_G(u, v)$, é o menor comprimento de um uv -caminho, dentre todos os uv -caminhos, ou é ∞ , caso não haja uv -caminho. Assim, $\text{dist}_G(u, u) = 0$. Se um uv -caminho tem comprimento igual à distância entre u e v , 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 menor comprimento, no entanto, é (a, b, e, h) . Como não há duas arestas que ligam a a h , temos $\text{dist}_G(a, h) = 3$. Há ainda outro caminho mínimo, que é (a, b, d, h) (caminhos mínimos não são únicos). 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 possíveis dg -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 todo 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 .

Agora seja G um (di)grafo ponderado nas arestas, com $w: E(G) \rightarrow \mathbb{R}$ sendo a função de peso. Lembre-se que o peso de um caminho é igual à soma dos pesos das arestas desse caminho. A *distância (ponderada) entre u e v em G* , denotada por $\text{dist}_G^w(u, v)$, é o menor peso de um uv -caminho, dentre todos os uv -caminhos, ou é ∞ , caso não haja uv -caminho. Assim, $\text{dist}_G^w(u, u) = 0$. Se um uv -caminho tem peso igual à distância entre u e v , então dizemos que ele é um uv -caminho *mínimo*.

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. Na verdade, com o que se sabe até o momento, não é possível existir um algoritmo eficiente que resolva problemas de distância nessas situações².

²Essa afirmação será provada no Capítulo 29.

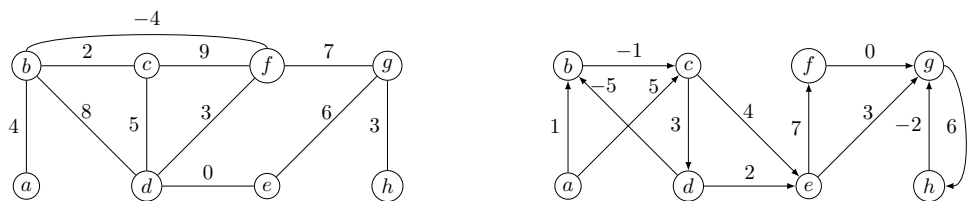


Figura 23.12: O grafo acima tem uma aresta de peso negativo, bf , e o digrafo tem um ciclo de peso negativo, (d, b, c, d) .

23.10 Algumas classes importantes de grafos

Um (di)grafo G 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.

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 exatamente $\binom{n}{2} = \frac{n(n-1)}{2}$ arestas, que é a quantidade total de pares de vértices e o máximo de arestas que qualquer grafo com n vértices pode ter.

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, ou seja, $G[S]$ é um grafo completo. Dizemos que S é um **conjunto independente** se não existe nenhuma aresta entre os pares de vértices de S , ou seja, $E(G[S]) = \emptyset$. Assim, note clique e conjunto independente são definições complementares. Note que o conjunto de vértices de um grafo completo é uma clique, bem como qualquer subconjunto de vértices também. Por isso, o maior conjunto independente em um grafo completo contém somente um vértice.

Um grafo G é **bipartido** se $V(G)$ pode ser particionado em dois conjuntos independentes X e Y . Em outras palavras, G é bipartido 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 . Se X e Y são os conjuntos da bipartição, dizemos também que G é (X, Y) -bipartido. 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 $N(u) = Y$. Denotamos G (X, Y) -bipartido completo com $p = |X|$ e $q = |Y|$ por $K_{p,q}$.

A Figura 23.13 exemplifica as terminologias discutidas acima.

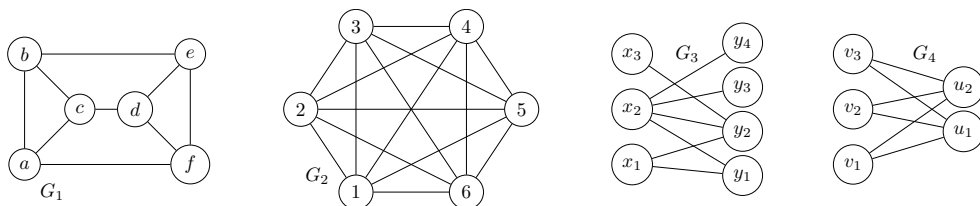


Figura 23.13: G_1 é 3-regular, G_2 é um K_6 (e 5-regular), G_3 é bipartido e G_4 é um $K_{3,2}$.

Exemplos de cliques são os conjuntos $\{a, b, c\}$, $\{e, f\}$, $\{1, 2, 3, 4\}$, $\{x_2, y_4\}$, $\{v_1, u_1\}$. Os conjuntos $\{b, f\}$, $\{3\}$, $\{x_1, x_3, y_3, y_4\}$, $\{v_1, v_2, v_3\}$ são exemplos de conjuntos independentes.

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 perceber 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 acíclico ou, alternativamente, se T é conexo e tem $v(T) - 1$ arestas. Note que todo caminho é uma árvore. Também vale que toda árvore é um grafo bipartido. A Figura 23.14 mostra exemplos de árvores.

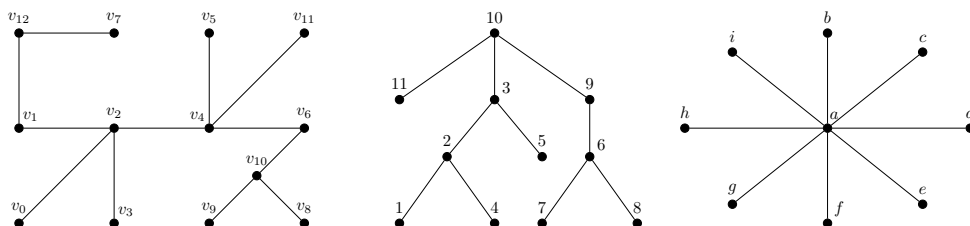


Figura 23.14: Exemplos de árvores.

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 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 *florestas*. Note que toda árvore é uma floresta.

O Teorema 23.3 a seguir apresenta várias características importantes sobre árvores.

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.

Pelo 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 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.

Seja T uma x -árvore (uma árvore enraizada em um vértice 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 D . Se T é uma arborescência tal que $V(T) = V(D)$, 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.

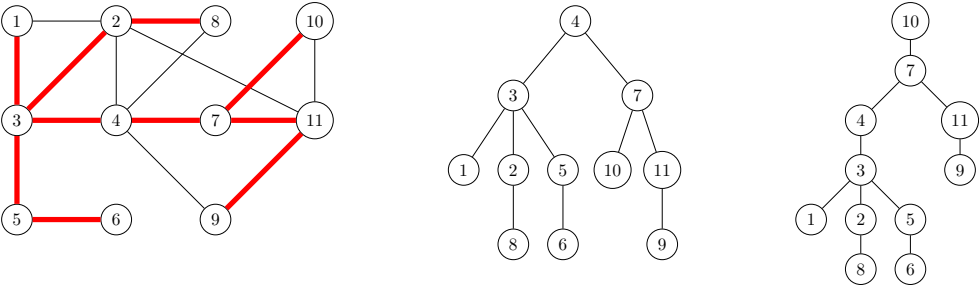


Figura 23.15: Exemplo de uma árvore geradora do grafo G , com destaque em vermelho sobre o próprio G . À direita, mostramos a mesma árvore geradora, mas como uma 4-árvore e como uma 10-árvore.

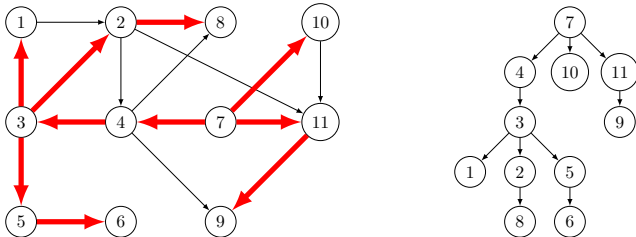


Figura 23.16: Exemplo de uma arborescência geradora do digrafo D , com destaque em vermelho sobre o próprio D . À direita, mostramos a mesma arborescência geradora, mas como uma 7-arborescência.

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, como 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 um algoritmo eficiente para testar a conexidade de qualquer grafo: basta escolher uma árvore inicial! A árvore mais simples que certamente é

subgrafo de G é uma árvore trivial (um único vértice s). Note que aumentar essa árvore conforme descrito acima resultará em 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*, e ela é considerada *enraizada* no vértice inicial s . 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   |  $V(T) \leftarrow V(T) \cup \{y\}$ 
5   |  $E(T) \leftarrow E(T) \cup \{xy\}$ 
6 devolve  $T$ 
```

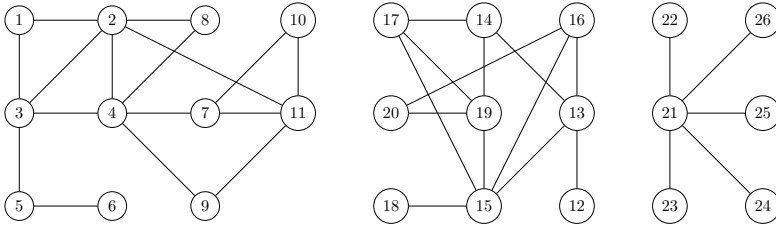
Na prática, nem sempre se constrói uma árvore explicitamente, mas se faz uso da terminologia comum. Seja G um grafo e s um vértice qualquer de G . Seja T a s -árvore devolvida por BUSCA(G, s). Se $v \in V(G)$, com $v \neq s$, então todo vértice no sv -caminho é um *ancestral* de v . O ancestral imediato de $v \neq s$ é seu *predecessor*, que será armazenado em $\text{pred}[v]$. Com isso, temos que $E(T) = \{\{\text{pred}[v], v\} : v \in V(T) \setminus \{s\}\}$, motivo pelo qual não é necessário construir a árvore explicitamente, bastando armazenar corretamente os predecessores. Cada vértice terá ainda um indicador $\text{visitado}[v]$, 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 vetores visitado e pred , 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 que estão na mesma componente conexa de s .

Algoritmo 24.2: CHAMABUSCA(G)

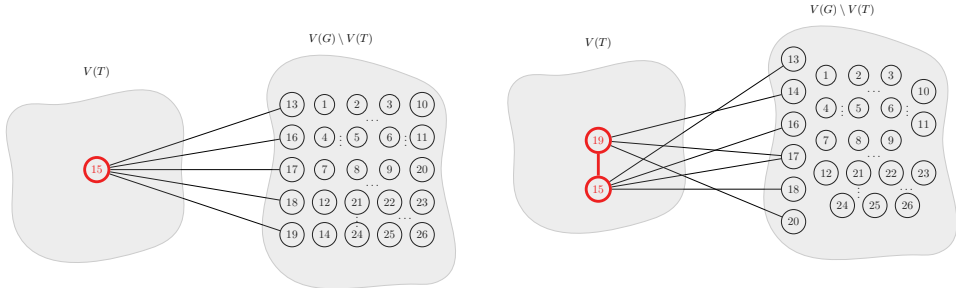
```

1 para toda vértice  $v \in V(G)$  faça
2   |  $\text{visitado}[v] \leftarrow 0$ 
3   |  $\text{pred}[v] \leftarrow -1$ 
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 simplesmente $(s, \dots, \text{pred}[\text{pred}[v]], \text{pred}[v], v)$. O Algoritmo 24.4, CONS-

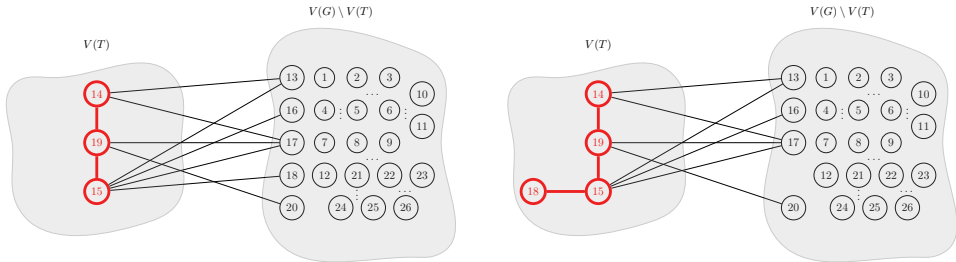


(a) Grafo G de entrada (desconexo, com 3 componentes).



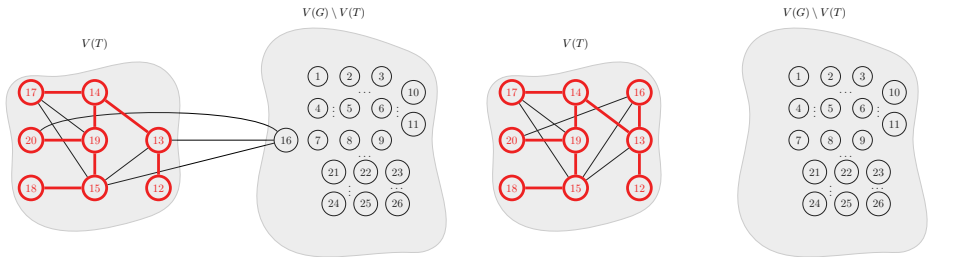
(b) O vértice 15 inicia T . Existem 5 arestas entre $V(T)$ e $V(G) \setminus V(T)$.

(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 $\text{BUSCA}(G, 15)$. 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 visitado[ $s$ ]  $\leftarrow$  1
2 pred[ $s$ ]  $\leftarrow$   $s$ 
3 enquanto houver aresta com um extremo visitado e outro não faça
4   | Seja  $xy$  uma aresta com visitado[ $x$ ] = 1 e visitado[ $y$ ] = 0
5   | visitado[ $y$ ]  $\leftarrow$  1
6   | pred[ $y$ ]  $\leftarrow$   $x$ 

```

TROICAMINHO(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).

Algoritmo 24.4: CONSTROICAMINHO(G, s, v)

```

1 seja  $L$  uma lista vazia
2 se visitado[ $v$ ] = 0 então
3   | devolve  $L$ 
4 atual  $\leftarrow$   $v$ 
5 enquanto atual  $\neq$   $s$  faça
6   | INSERENOINICIOLISTA( $L, \text{atual}$ )
7   | atual  $\leftarrow$  pred[atual]
8 INSERENOINICIOLISTA( $L, s$ )
9 devolve  $L$ 

```

O lema a seguir nos permite concluir que BUSCA(G, s) visita todos os vértices que estão na mesma componente conexa de s . Em outras palavras, a árvore construída (mesmo que implicitamente) é geradora da componente conexa que contém s .

Lema 24.1

Sejam G um grafo e $s \in V(G)$ um vértice qualquer. Um vértice $v \in V(G)$ está visitado no fim da execução de BUSCA(G, s) se e somente se existe um sv -caminho em G .

Demonstração. Vamos provar primeiro que se um vértice v está visitado no final da execução, então existe um sv -caminho em G . Note que uma vez marcado como visitado, um vértice não é desmarcado até o final do algoritmo. Por isso, basta mostrar que um sv -caminho existe no momento em que o vértice v é visitado. Faremos isso por indução no número i de iterações do algoritmo. Quando $i = 0$, nenhuma iteração executou. De fato, neste momento temos apenas s visitado e, de fato, existe ss -caminho. Agora suponha que $i > 0$ e suponha que para todos os vértices visitados até o momento existe um caminho a partir de s . Seja xy a aresta escolhida nesta

iteração. Como x já está visitado, por hipótese de indução existe sx -caminho. Tal caminho seguido da aresta xy certamente contém um sy -caminho.

Agora vamos provar que se existe um sv -caminho, então v está visitado ao final da execução. Suponha, para fins de contradição, que v não está visitado ao final. Como existe sv -caminho $P = (s = v_1, v_2, \dots, v_k = v)$ e s claramente está visitado, podemos concluir que existe um vértice $v_i \in V(P)$ tal que v_1, \dots, v_i estão visitados e v_{i+1} não está. Mas então a aresta $v_i v_{i+1}$ é uma aresta que satisfaz a condição do laço **enquanto**, contradizendo o fato do algoritmo ter terminado. \square

Note que encontrar uma aresta xy na linha 3 do Algoritmo 24.3, BUSCA, envolve percorrer 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 aplicações 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 desenfileirado. Note que, após s , os próximos vértices inseridos na fila, em ordem, são exatamente os vizinhos de s .

Também manteremos, para cada vértice u , os atributos `pred[u]` e `visitado[u]`. Em `pred[u]` indicaremos 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, os predecessores nos auxiliam a descrever um su -caminho. Já `visitado[u]` 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 vetores `visitado` e `pred`, uma vez que essa tarefa não faz parte da busca.

O algoritmo BUSCALARGURA 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 `pred[v]` com u e enfileira-se v . Perceba que BUSCALARGURA é uma implementação de BUSCA, de forma que o enunciado do Lema 24.1 vale para BUSCALARGURA também.

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 $n = v(G)$ e $m = e(G)$. A inicialização (linhas 1

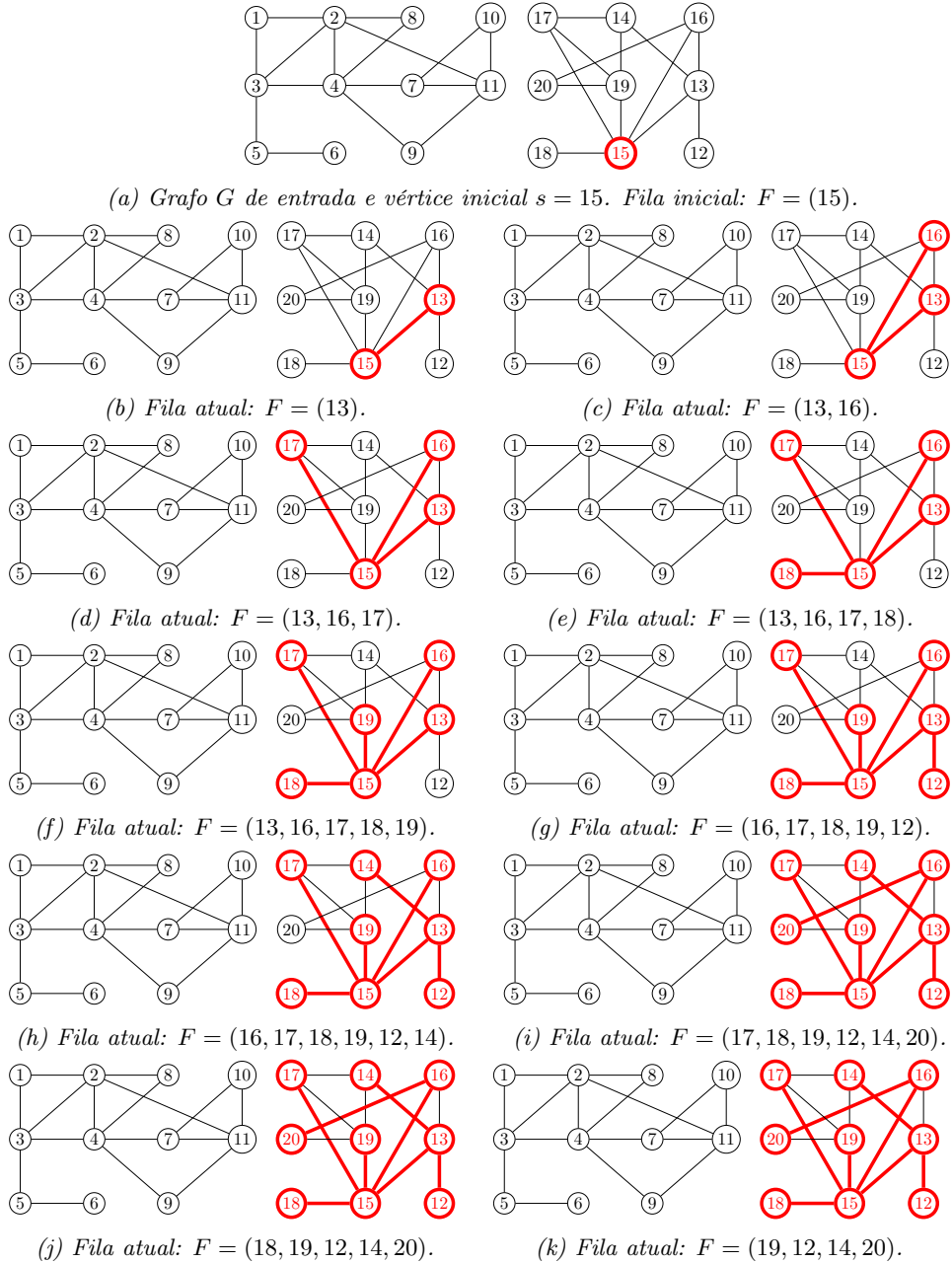


Figura 24.2: Execução de $\text{BUSCALARGURA}(G, 15)$. Visitamos os vizinhos dos vértices em ordem numérica crescente. Vértices visitados estão em vermelho. A árvore construída de forma indireta pelos predecessores está em vermelho. Após 24.2k, a fila é esvaziada e nenhum outro vértice é marcado.

Algoritmo 24.5: BUSCALARGURA(G, s)

```

1 visitado[ $s$ ]  $\leftarrow$  1
2 cria fila vazia  $F$ 
3 ENFILEIRA( $F, s$ )
4 enquanto  $F.tamanho > 0$  faça
5    $u \leftarrow$  DESENFILEIRA( $F$ )
6   para toda  $v \in N(u)$  faça
7     se visitado[ $v$ ] = 0 então
8       visitado[ $v$ ]  $\leftarrow$  1
9       pred[ $v$ ]  $\leftarrow$   $u$ 
10      ENFILEIRA( $F, v$ )

```

a 3) leva tempo total $\Theta(1)$. Note que antes de enfileirar um vértice v , atualizamos **visitado**[v] de 0 para 1 (linha 8) e não modificamos mais esse atributo. Assim, 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 no máximo $n + 1$ vezes e a chamada a DESENFILEIRA (linha 5) é executada no máximo n vezes.

Resta então analisar a quantidade de vezes que o conteúdo do laço **para** da linha 6 é executado. 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 $O(n) + O(n^2) = O(n^2)$. Porém, se utilizarmos listas de adjacências, então o laço **para** é executado apenas $|N(u)|$ vezes, de modo que, no total, ele é executado no máximo $\sum_{u \in V(G)} |N(u)| = 2m$ vezes, e então o tempo total de execução do algoritmo é $O(n) + O(m) = O(n + m)$. 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, a *árvore de busca em largura* é a árvore T tal que

$$V(T) = \{v \in V(G) : \text{pred}[v] \neq -1\} \cup \{s\}$$

$$E(T) = \{\{\text{pred}[v], v\} : v \in V(T) \setminus \{s\}\}.$$

Ela é uma árvore geradora da componente que contém s em G e contém um único sv -caminho para qualquer $v \in V(T)$. 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 em `est_dist[v]`. Essa acaba sendo, portanto, uma estimativa da distância entre s e v (posteriormente mostraremos que ela é de fato a distância). O Algoritmo 24.6 contém apenas duas diferenças com relação ao algoritmo que havíamos apresentado anteriormente: as linhas 3 e 11. Novamente considere que um outro algoritmo, como por exemplo o CHAMABUSCA, inicializou os vetores `visitado` com 0, `pred` com -1 , e `est_dist` com ∞ .

Algoritmo 24.6: BUSCALARGURADISTANCIA(G, s)

```

1 visitado[s] ← 1
2 pred[s] ← s
3 est_dist[s] ← 0
4 cria fila vazia F
5 ENFILEIRA(F, s)
6 enquanto F.tamanho > 0 faça
7   u ← DESENFILEIRA(F)
8   para todo vértice v ∈ N(u) faça
9     se visitado[v] = 0 então
10      visitado[v] ← 1
11      est_dist[v] ← est_dist[u] + 1
12      pred[v] ← u
13      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 `est_dist[v]` arestas. A seguir mostramos que, ao fim de BUSCALARGURADISTANCIA(G, s), o valor `est_dist[v]` 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 24.2, que garante que os valores em `est_dist` dos 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 `est_dist[u] ≤ est_dist[v]`.

Lema 24.2

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

$$\text{est_dist}[u] \leq \text{est_dist}[v] \leq \text{est_dist}[u] + 1 .$$

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 $F = (x_1, \dots, x_k)$ e que vale $\text{est_dist}[x_i] \leq \text{est_dist}[x_j] \leq \text{est_dist}[x_i] + 1$ 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 $F = (x_1, \dots, x_k)$ no início dessa iteração. Durante a iteração, o algoritmo remove x_1 de F e adiciona seus vizinhos não visitados, digamos w_1, \dots, w_h a F , de modo que agora temos $F = (x_2, \dots, x_k, w_1, \dots, w_h)$. Ademais, o algoritmo fez $\text{est_dist}[w_j] = \text{est_dist}[x_1] + 1$ 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

$$\text{est_dist}[x_1] \leq \text{est_dist}[x_i] \leq \text{est_dist}[x_1] + 1 .$$

Assim, para qualquer vizinho w_j de x_1 temos, pela desigualdade acima, que, para todo $2 \leq i \leq k$,

$$\text{est_dist}[x_i] \leq \text{est_dist}[x_1] + 1 = \text{est_dist}[w_j] = \text{est_dist}[x_1] + 1 \leq \text{est_dist}[x_i] + 1 .$$

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 ($\text{est_dist}[x_1] + 1$). Portanto, todos os pares de vértices em $\{x_2, \dots, x_k, w_1, \dots, w_h\}$ satisfazem a conclusão do lema. \square

Note que, como um vértice não tem seu valor em `est_dist` alterado mais de uma vez pelo algoritmo, a conclusão do Lema 24.2 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 24.3 a seguir prova que `BUSCALARGURADISTANCIA`(G, s) calcula corretamente os caminhos mais curtos entre s e todos os vértices de G .

Teorema 24.3

Sejam G um grafo e $s \in V(G)$. Ao fim de `BUSCALARGURADISTANCIA`(G, s), para todo $v \in V(G)$ vale que $\text{est_dist}[v] = \text{dist}_G(s, v)$.

Demonstração. Começemos mostrando que $\text{est_dist}[v] \geq \text{dist}_G(s, v)$ para todo $v \in V(G)$. Como o valor em $\text{est_dist}[v]$ muda apenas quando v é visitado, e um vértice nunca deixa de ser visitado, basta mostrar que a desigualdade vale exatamente neste momento, que é quando v entra na fila. Vamos mostrar então que a desigualdade vale fazendo 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 $\text{est_dist}[s] = 0 \geq \text{dist}_G(s, s) = 0$ e $\text{est_dist}[v] = \infty \geq \text{dist}_G(s, v)$ para todo $v \neq s$, e portanto o resultado é válido.

Seja então $k > 1$ e suponha que se x é um dos primeiros $k - 1$ vértices inseridos na fila, então $\text{est_dist}[x] \geq \text{dist}_G(s, x)$. 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 8 por ser vizinho de algum vértice u que foi removido da fila. Então u foi um dos $k - 1$ primeiros vértices a serem inseridos na fila o que, usando a hipótese de indução, nos diz que $\text{est_dist}[u] \geq \text{dist}_G(s, u)$. 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 11, obtemos

$$\text{est_dist}[v] = \text{est_dist}[u] + 1 \geq \text{dist}_G(s, u) + 1 \geq \text{dist}_G(s, v) .$$

Como um vértice entra na fila somente uma vez, o valor em $\text{est_dist}[v]$ não muda mais durante a execução do algoritmo. Logo, $\text{est_dist}[v] \geq \text{dist}_G(s, v)$ para todo $v \in V(G)$.

Agora mostraremos que $\text{est_dist}[v] \leq \text{dist}_G(s, v)$ para todo $v \in V(G)$, de onde só poderemos concluir que $\text{est_dist}[v] = \text{dist}_G(s, v)$.

Suponha, para fins de contradição, que ao fim da execução de `BUSCALARGURADISTANCIA`(G, s) existe ao menos um $x \in V(G)$ com $\text{est_dist}[x] > \text{dist}_G(s, x)$. Seja v o vértice com menor valor $\text{dist}_G(s, v)$ para o qual isso acontece, isto é, tal que $\text{est_dist}[v] > \text{dist}_G(s, v)$.

Considere um sv -caminho mínimo (s, \dots, 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 que $\text{est_dist}[u] = \text{dist}_G(s, u)$. Assim,

$$\text{est_dist}[v] > \text{dist}_G(s, v) = \text{dist}_G(s, u) + 1 = \text{est_dist}[u] + 1 . \quad (24.1)$$

Considere agora 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 $\text{est_dist}[v] = \text{est_dist}[w] + 1$ e, pelo Lema 24.2, $\text{est_dist}[w] \leq \text{est_dist}[u]$, de forma que $\text{est_dist}[v] \leq \text{est_dist}[u] + 1$, uma contradição com (24.1). Assim, assuma que v não havia sido visitado neste momento. Nesse caso, quando v entrar na fila (certamente entra, pois é vizinho de u), teremos $\text{est_dist}[v] = \text{est_dist}[u] + 1$, que é também uma contradição com (24.1). \square

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 manteremos, para cada vértice u , os valores `pred`[u] e `visitado`[u]. O valor em `pred`[u] 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 `visitado`[u]

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 $\text{CONSULTA}(P)$ 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 vetores **visitado** e **pred**, uma vez que essa tarefa não faz parte da busca.

Algoritmo 24.7: $\text{BUSCAPROFITERATIVA}(G, s)$

```

1 visitado[ $s$ ]  $\leftarrow$  1
2 pred[ $s$ ]  $\leftarrow$   $s$ 
3 cria pilha vazia  $P$ 
4  $\text{EMPILHA}(P, s)$ 
5 enquanto  $P.\text{tamanho} > 0$  faça
6    $u \leftarrow \text{CONSULTA}(P)$ 
7   se existe  $uv \in E(G)$  com  $v.\text{visitado} = 0$  então
8     visitado[ $v$ ]  $\leftarrow$  1
9     pred[ $v$ ]  $\leftarrow$   $u$ 
10     $\text{EMPILHA}(P, v)$ 
11  senão
12     $u \leftarrow \text{DESEMPILHA}(P)$ 

```

Note que a diferença de BUSCALARGURA para $\text{BUSCAPROFITERATIVA}$ é a estrutura de dados 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 $\text{BUSCAPROFITERATIVA}$, 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 $\text{BUSCAPROFITERATIVA}$ 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 **pred**[v] 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.

Perceba que $\text{BUSCAPROFITERATIVA}$ é uma implementação de BUSCA , de forma que o enunciado do Lema 24.1 vale para $\text{BUSCAPROFITERATIVA}$ também. Na Figura 24.3 simulamos uma execução da busca em profundidade.

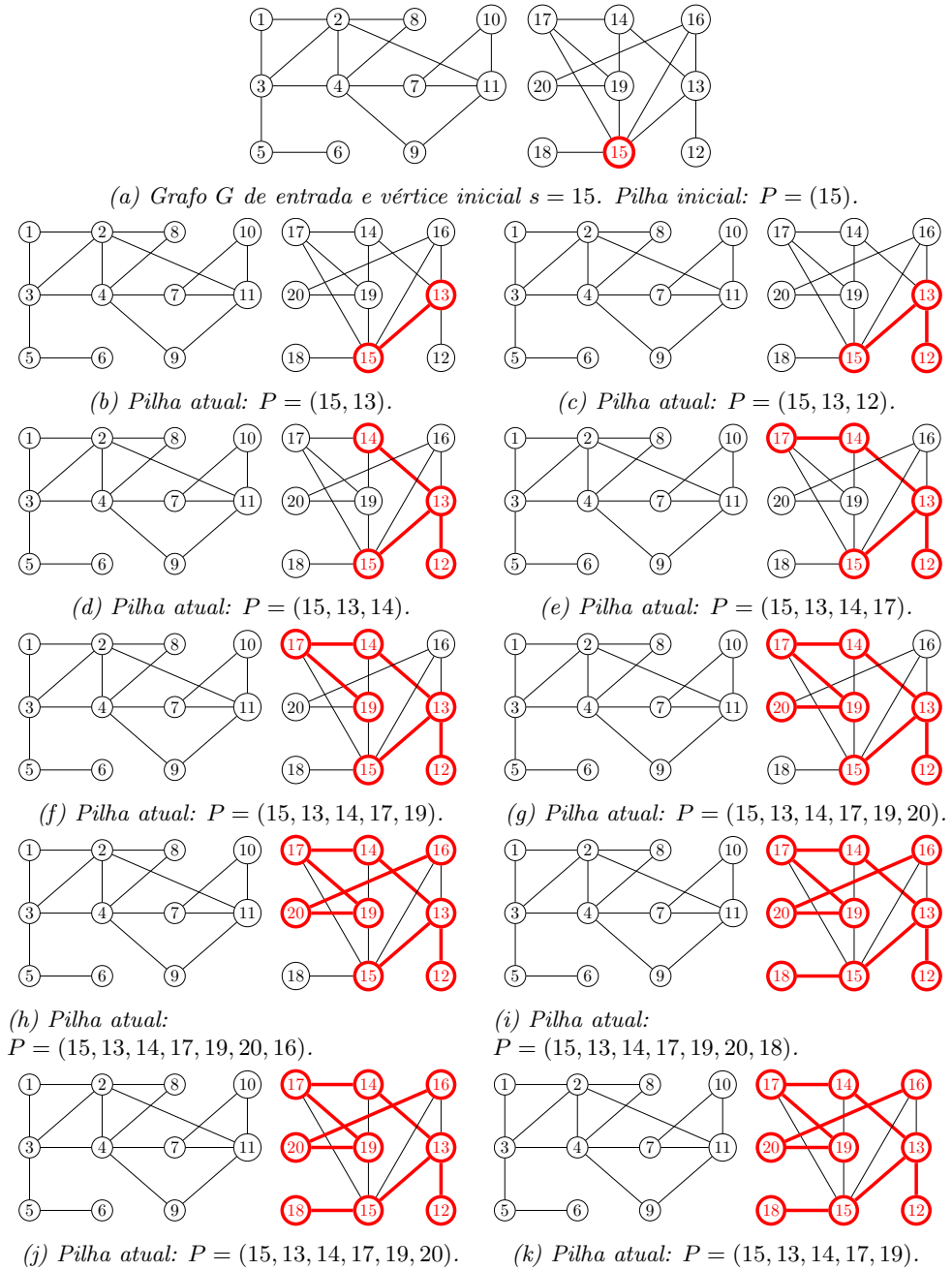


Figura 24.3: Execução de $BUSCAPROFITERATIVA(G, 15)$. Visitamos os vizinhos dos vértices ordem numérica crescente. Vértices visitados estão em vermelho. A árvore construída de forma indireta pelos predecessores está em vermelho. Após 24.3k, a pilha é esvaziada e nenhum outro vértice é marcado.

Seja G um grafo e $s \in V(G)$ qualquer. Vamos analisar o tempo de execução de `BUSCAPROFITERATIVA(G, s)`. Sejam $n = v(G)$ e $m = e(G)$. Na inicialização (linhas 1 a 4) é gasto tempo total $\Theta(1)$. Note que antes de um vértice v ser empilhado, atualizamos `visitado[v]` de 0 para 1 (linha 8) e tal atributo não é modificado novamente. Assim, uma vez que um vértice entra na pilha, ele nunca mais passará no teste da linha 7. Portanto, todo vértice para o qual existe um sv -caminho entra somente uma vez na pilha. Assim, as linhas 8, 9, 10 e 12 são executadas no máximo n vezes cada. Resta então analisar a quantidade de vezes que as outras linhas do laço **enquanto** são executadas. Note que as linhas 5 e 6 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, essas linhas executam no máximo $\sum_{u \in V(G)} |N(u)| = 2m$ vezes ao todo.

Por fim, a linha 7 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 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 que ao usar matriz de adjacências, o tempo é no máximo $O(n) + O(m) + \sum_{u \in V(G)} n = O(n) + O(m) + O(n^2) = O(n^2)$, onde a última igualdade vale porque $m = O(n^2)$ em qualquer grafo. Ao usar listas de adjacências, o tempo é no máximo $O(n) + O(m) + \sum_{u \in V(G)} |N(u)| = O(n) + O(m) + O(m) = O(n+m)$. Veja que a implementação com listas de adjacência é mais eficiente quando m for pequeno em relação a n .

Note que a **árvore de busca em profundidade** é árvore T tal que

$$\begin{aligned} V(T) &= \{v \in V(G) : v.\text{pred} \neq \text{NULL}\} \cup \{s\} \\ E(T) &= \{\{v.\text{pred}, v\} : v \in V(T) \setminus \{s\}\}. \end{aligned}$$

Ela é uma árvore geradora da componente que contém s em G e contém um único

sv -caminho para qualquer $v \in V(T)$. 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 (pelo contrário, facilita sua análise). 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 visitado[ $s$ ]  $\leftarrow$  1
2 pred[ $s$ ]  $\leftarrow$   $s$ 
3 para todo vértice  $v \in N(s)$  faça
4   se visitado[ $v$ ] = 0 então
5     pred[ $v$ ]  $\leftarrow$   $s$ 
6     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 informações sobre a ordem em que os vértices são visitados e deixam de ser explorados. Tradicionalmente, esses valores são chamados de pré-ordem (momento em que o vértice é visitado) e pós-ordem (momento em que ele deixa de ter a vizinhança explorada). Para isso, teremos também uma variável auxiliar que conta o “tempo” do algoritmo. Essas informações serão úteis em diversas aplicações da busca em profundidade (veja Seções 24.4.2 e 24.4.1), porém nem todas as aplicações precisam de ambos os valores.

O Algoritmo 24.9 apresenta BUSCAPROFUNDIDADE alterada para manter os valores discutidos acima, onde assumimos que o algoritmo que chama BUSCAPROFUNDIDADE inicializa **tempo** com o valor 0. Note ainda que essas mudanças não alteram o tempo de execução do algoritmo, uma vez que são alterações em tempo constante feitas em vetores.

Algoritmo 24.9: BUSCAPROFUNDIDADE(G, s)

```

1 visitado[s] ← 1
2 pred[s] ← s
3 tempo ← tempo + 1
4 PREORDEM[s] ← tempo
5 para toda vértice  $v \in N(s)$  faça
6   se visitado[v] = 0 então
7     pred[v] ← s
8     BUSCAPROFUNDIDADE( $G, v$ )
9 tempo ← tempo + 1
10 POSORDEM[s] ← tempo

```

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. Se não, existirão vértices não visitados ao fim de uma execução desses algoritmos. Assim, para encontrar todas as componentes do grafo, podemos fazer várias buscas, cada uma iniciando 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 uma busca 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 $v.pred = u$), faça $v.componente = u.componente$, uma vez que u e v estão na mesma componente.

Vamos analisar o tempo de execução de BUSCACOMPONENTES(G). Seja $n = v(G)$ e

Algoritmo 24.10: BUSCAComponentes(G)

```

1 para todo vértice  $v \in V(G)$  faça
2    $v.\text{visitado} = 0$ 
3    $v.\text{pred} = \text{NULL}$ 
4  $\text{qtdComponentes} = 0$ 
5 para todo vértice  $s \in V(G)$  faça
6   se  $s.\text{visitado} == 0$  então
7      $s.\text{visitado} = 1$ 
8      $s.\text{componente} = s$ 
9      $\text{qtdComponentes} = \text{qtdComponentes} + 1$ 
10     $\text{BUSCA}(G, s)$ 
11 devolve  $\text{qtdComponentes}$ 

```

$m = e(G)$. 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 componente conexa de G . Dessas, note que apenas a última não leva tempo constante. Como visto, tanto $\text{BUSCALARGURA}(G, s)$ quanto $\text{BUSCAPROFUNDIDADE}(G, s)$ 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 $\text{BUSCAComponentes}(G)$ 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.

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    $u = \text{DESENFILEIRA}(F)$ 
6   para todo vértice  $v \in N^+(u)$  faça
7     se  $v.visitado == 0$  então
8        $v.visitado = 1$ 
9        $v.pred = u$ 
10      ENFILEIRA( $F, v$ )
  
```

Algoritmo 24.12: BUSCAPROFUNDIDADE(D, s)

```

1 INSERENOFIMLISTA( $D.PREORDEM, s$ )
2  $s.visitado = 1$ 
3 para todo vértice  $v \in N^+(s)$  faça
4   se  $v.visitado == 0$  então
5      $v.pred = s$ 
6     BUSCAPROFUNDIDADE( $D, v$ )
7 INSERENOFIMLISTA( $D.POSORDEM, s$ )
8 INSERENOINICIOLISTA( $D.POSORDEMREV, s$ )
  
```

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.

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, formando 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 *quando sabemos a ordem dos vértices iniciais* a partir dos quais podemos 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, \dots, 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 arcos em apenas uma direção, pois caso contrário a união de D_i com D_j formaria uma componente maior contendo D_i e 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 arco saindo de D_i em direção a nenhuma outra componente, apenas arco entrando em D_i .

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 $\overleftarrow{D}.\text{PosOrdemRev}$.
2. Execute BUSCAProfundidade em D , com uma pequena alteração para que seja feito `v.componente = s.componente` logo após `v.pred = s`, e visite os vértices de acordo com a ordem da lista $\overleftarrow{D}.\text{PosOrdemRev}$.

Esse procedimento está descrito formalmente no Algoritmo 24.13. Durante o segundo passo, cada chamada recursiva a $\text{BUSCAPROFUNDIDADE}(D, u)$ (linha 9) identifica os vértices de uma das componentes fortemente conexas. As Figuras 24.6 e 24.7 exemplificam uma execução do algoritmo.

A intuição por trás desse procedimento é a seguinte. Após a execução de $\text{BUSCAComponentes}(\tilde{D})$, o primeiro vértice de $\tilde{D}.\text{PosOrdemRev}$ 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: $\text{ComponentesFortementeConexas}(D)$

```

1 para todo vértice  $v \in V(D)$  faça
2    $v.\text{visitado} = 0$ 
3    $v.\text{pred} = \text{NULL}$ 
4  $\text{BUSCAComponentes}(\tilde{D})$  /* Com  $\text{BUSCAPROFUNDIDADE}$  no lugar de  $\text{BUSCA}$  */
5  $u = \tilde{D}.\text{PosOrdemRev}.\text{cabeca}$ 
6 enquanto  $u \neq \text{NULL}$  faça
7   se  $u.\text{visitado} == 0$  então
8      $u.\text{componente} = u$ 
9      $\text{BUSCAPROFUNDIDADE}(D, u)$ 
10     $u = u.\text{prox}$ 
```

Se o digrafo estiver representado com lista de adjacências, então $\text{ComponentesFortementeConexas}(D)$ tem tempo $\Theta(v(D) + e(D))$. No Teorema 24.4 a seguir mostramos que esse algoritmo identifica corretamente as componentes fortemente conexas de D .

Teorema 24.4

Seja D um digrafo. Ao fim da execução de $\text{ComponentesFortementeConexas}(D)$ temos que, para quaisquer $u, v \in V(D)$, os vértices u e v estão na mesma componente fortemente conexa se e somente se $u.\text{componente} = v.\text{componente}$.

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:

$$\begin{aligned} v \in V(D) \text{ é visitado durante a chamada } \text{BUSCAPROFUNDIDADE}(D, u) \\ \text{se e somente se } v \in V(D_u). \end{aligned} \quad (24.2)$$

De fato, se (24.2) é válida, então após a execução de `BUSCAPROFUNDIDADE(D, u)` teremos que os únicos vértices com $v.\text{componente} = u$ são os vértices que estão em D_u . Assim, para um vértice v ter $v.\text{componente} = u$ ele precisa ser visitado durante a chamada `BUSCAPROFUNDIDADE(D, u)`. Como o algoritmo `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 24.5

Se $v \in V(D_u)$, então v é visitado na chamada `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 `BUSCAPROFUNDIDADE(D, u)` é executada, então como existe vu -caminho, certamente o vértice u seria visitado antes de `BUSCAPROFUNDIDADE(D, u)`, de modo que a chamada a `BUSCAPROFUNDIDADE(D, u)` nunca seria executada, levando a um absurdo. Portanto, sabemos que no início da execução de `BUSCAPROFUNDIDADE(D, u)`, o vértice v ainda não foi visitado. Logo, como existe um uv -caminho, o vértice v é visitado durante essa chamada. \diamond

Para completar a prova, resta mostrar a seguinte afirmação.

Afirmação 24.6

Se $v \in V(D)$ foi visitado na chamada `BUSCAPROFUNDIDADE(D, u)`, então $v \in V(D_u)$.

Demonstração. Seja $v \in V(D)$ um vértice que foi visitado durante a chamada

$\text{BUSCAPROFUNDIDADE}(D, u)$. 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 $\overleftarrow{D}.\text{POSORDEMREV}$ e v foi visitado durante a chamada $\text{BUSCAPROFUNDIDADE}(D, u)$, isso significa que u aparece antes de v nessa lista. Portanto, quando executamos $\text{BUSCAComponentes}(\overleftarrow{D})$ na linha 4, vale que

$$\begin{aligned} &\text{a chamada } \text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, v) \text{ termina antes} \\ &\text{do fim da execução de } \text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, u). \end{aligned} \quad (24.3)$$

Considere o momento do *início* da execução de $\text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, u)$. Como existe vu -caminho em \overleftarrow{D} (pois existe uv -caminho em D), sabemos que $\text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, u)$ não pode ter sido iniciada após o término da execução de $\text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, v)$, pois nesse caso u teria sido visitado durante a execução de $\text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, v)$ e, por conseguinte, $\text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, v)$ terminaria depois do fim da execução de $\text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, u)$, contrariando (24.3).

Assim, a chamada $\text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, u)$ teve início antes de $\text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, v)$ e, por (24.3), terminou depois do fim de $\text{BUSCAPROFUNDIDADE}(\overleftarrow{D}, v)$, o que significa que existe um uv -caminho em \overleftarrow{D} . Portanto, existe um vu -caminho em D . \diamond

As Afirmações 24.5 e 24.6 juntamente com (24.2) provam o resultado do teorema. \square

24.4.2 Ordenação topológica

Uma **ordenação topológica** de um digrafo D é, formalmente, rotulação $f: V(D) \rightarrow \{1, 2, \dots, v(D)\}$ 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 sendo que o vértice u que tiver $f(u) = 1$ deve ficar mais à esquerda, tendo o vértice u que tem $f(u) = 2$ à sua direita, e assim sucessivamente, até o vértice u que tem $f(u) = v(D)$. Com isso, para todo arco uv vale que 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.

Começemos observando que um digrafo admite ordenação topológica se, e somente se, ele não tem ciclos. Isto é, se não existe uma sequência de vértices $(v_1, v_2, \dots, v_k, v_1)$ 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 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 $D.POSORDEMREV$. 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 $D.POSORDEMREV$ 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 24.7 prova que esse algoritmo de fato está correto.

Lema 24.7

Dado um digrafo acíclico D , a rotulação f que é devolvida pelo algoritmo $ORDENACAO_TOPOLOGICA(D)$ é uma ordenação topológica.

Demonstração. Por construção, note que temos $f(u) \neq f(v)$ para todo $u \neq v$ e

Algoritmo 24.14: ORDENACAO_TOPOLOGICA(D)

```

1 BUSCA_COMPONENTES( $D$ ) /* Usando BUSCA_PROFUNDIDADE no lugar de BUSCA */

2 atual =  $D.POSORDEMREV.cabeca$ 
3  $i = 1$ 
4 enquanto atual  $\neq$  NULL faça
5    $f(atual) = i$ 
6    $i = i + 1$ 
7   atual = atual.prox
8 devolve  $f$ 

```

também que $f(u) \in \{1, \dots, v(D)\}$. Resta então provar que para qualquer $uv \in E(D)$, 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 $BUSCA_PROFUNDIDADE(D, v)$ termina sua execução antes de $BUSCA_PROFUNDIDADE(D, u)$. Dessa forma, v é incluído no início de $D.POSORDEMREV$ 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 $BUSCA_PROFUNDIDADE(D, v)$ não visita o vértice u e termina sua execução antes mesmo de $BUSCA_PROFUNDIDADE(D, u)$ começar. Dessa forma, v é incluído no início de $D.POSORDEMREV$ antes de u ser incluído, caso em que $f(u) < f(v)$ também. \square

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.

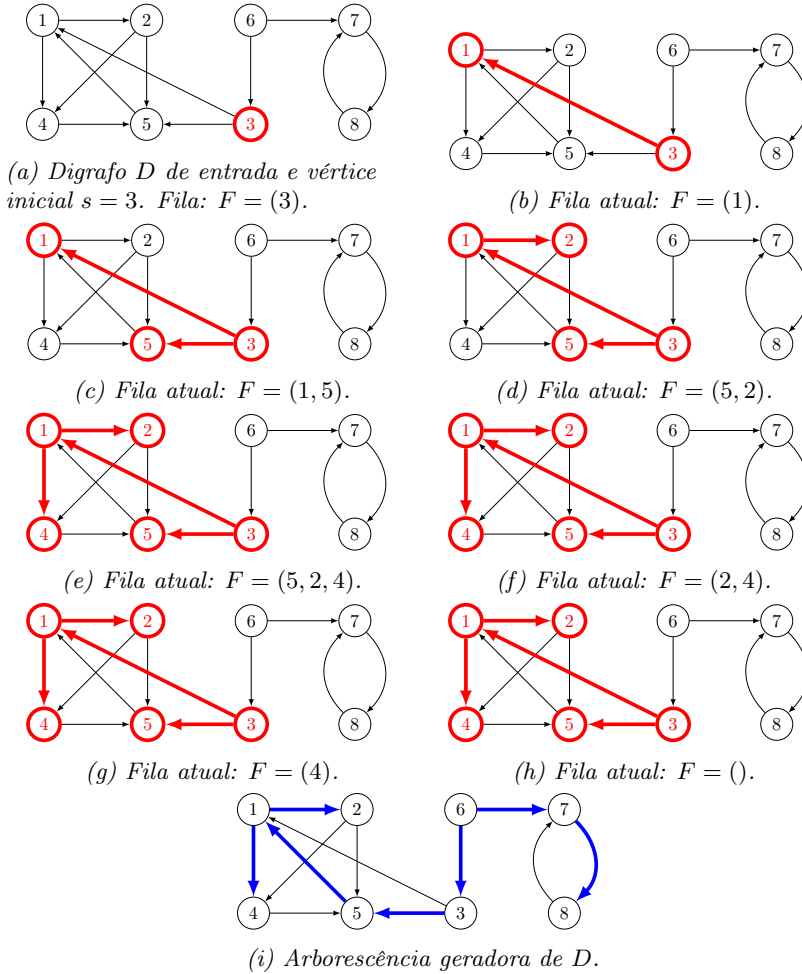


Figura 24.4: Execução de $\text{BUSCALARGURA}(D, 3)$. Os vértices visitados estão em vermelho, bem como a arborescência construída de forma indireta pelos predecessores. Note que no fim não temos uma arborescência geradora (mas uma existe, como mostra 24.4i).

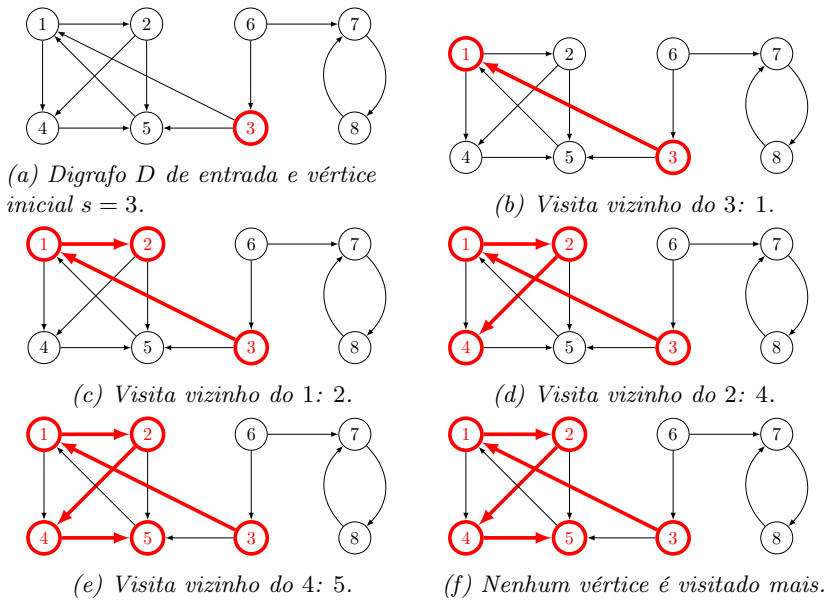


Figura 24.5: Execução de $BUSCAPROFUNDIDADE(D, 3)$. Os vértices visitados estão em vermelho. A arborescência construída de forma indireta pelos predecessores está em vermelho. Note que no fim não temos uma arborescência geradora.

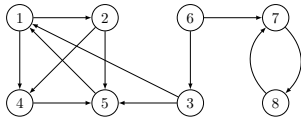
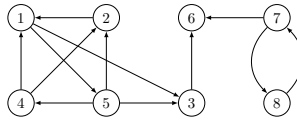
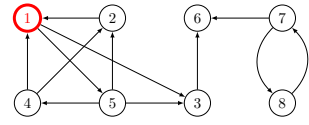
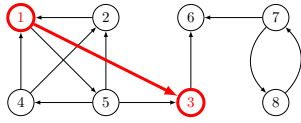
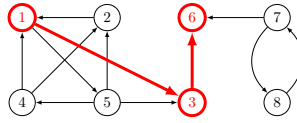
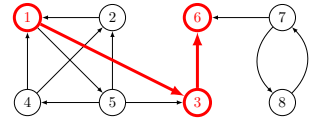
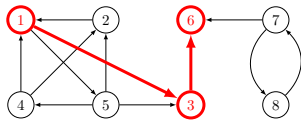
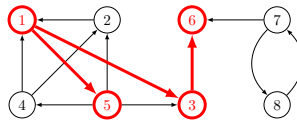
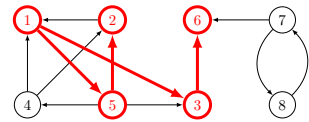
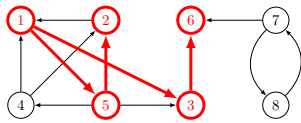
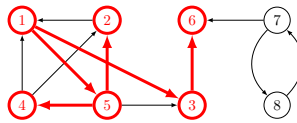
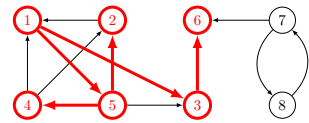
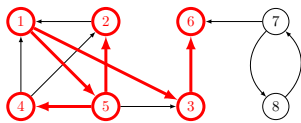
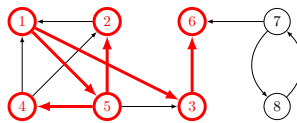
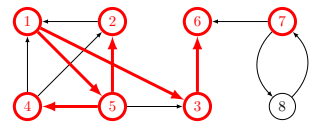
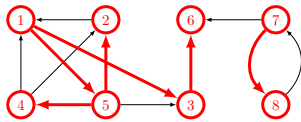
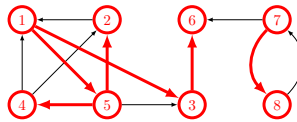
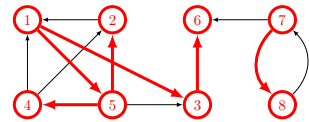
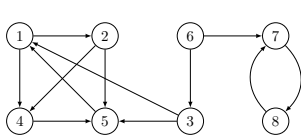
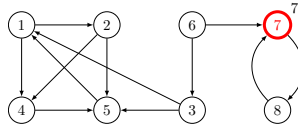
(a) Digrafo D de entrada.(b) Digrafo \overleftarrow{D} . Chama $\text{BUSCAComponentes}(\overleftarrow{D})$.(c) Nenhum visitado. Chama $\text{BUSCAProfundidade}(\overleftarrow{D}, 1)$.(d) Visita um vizinho do 1, o 3. $\overleftarrow{D}.\text{PosOrdemRev}$: $()$.(e) Visita um vizinho do 3, o 6. $\overleftarrow{D}.\text{PosOrdemRev}$: $()$.(f) Não há vizinhos do 6. $\overleftarrow{D}.\text{PosOrdemRev}$: (6) .(g) Não há vizinhos do 3. $\overleftarrow{D}.\text{PosOrdemRev}$: $(3, 6)$.(h) Visita um vizinho do 1, o 5. $\overleftarrow{D}.\text{PosOrdemRev}$: $(3, 6)$.(i) Visita um vizinho do 5, o 2. $\overleftarrow{D}.\text{PosOrdemRev}$: $(3, 6)$.(j) Não há vizinhos do 2. $\overleftarrow{D}.\text{PosOrdemRev}$: $(2, 3, 6)$.(k) Visita um vizinho do 5, o 4. $\overleftarrow{D}.\text{PosOrdemRev}$: $(2, 3, 6)$.(l) Não há vizinhos do 4. $\overleftarrow{D}.\text{PosOrdemRev}$: $(4, 2, 3, 6)$.(m) Não há vizinhos do 5. $\overleftarrow{D}.\text{PosOrdemRev}$: $(5, 4, 2, 3, 6)$.(n) Não há vizinhos do 1. $\overleftarrow{D}.\text{PosOrdemRev}$: $(1, 5, 4, 2, 3, 6)$.(o) Chama $\text{BUSCAProfundidade}(\overleftarrow{D}, 7)$. $\overleftarrow{D}.\text{PosOrdemRev}$: $(1, 5, 4, 2, 3, 6)$.(p) Visita um vizinho do 7, o 8. $\overleftarrow{D}.\text{PosOrdemRev}$: $(1, 5, 4, 2, 3, 6)$.(q) Não há vizinhos do 8. $\overleftarrow{D}.\text{PosOrdemRev}$: $(8, 1, 5, 4, 2, 3, 6)$.(r) Não há vizinhos do 7. $\overleftarrow{D}.\text{PosOrdemRev}$: $(7, 8, 1, 5, 4, 2, 3, 6)$.

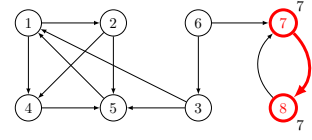
Figura 24.6: Primeira parte da execução de $\text{ComponentesForteConexas}(D)$: execução de $\text{BUSCAComponentes}(\overleftarrow{D})$.



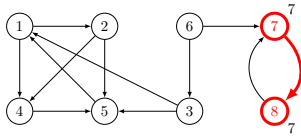
(a) Digrafo D de entrada.
 $\overleftarrow{D}.POSORDEMREV$:
 $(\underline{7}, 8, 1, 5, 4, 2, 3, 6)$.



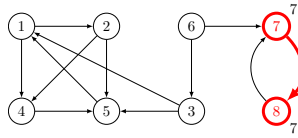
(b) Faz 7. componente = 7.
 Chama
 $BUSCAPROFUNDIDADE(D, 7)$.



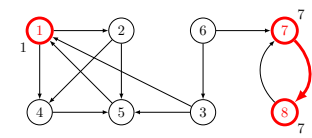
(c) Visita um vizinho do 7,
 o 8. Faz 8. componente = 7.



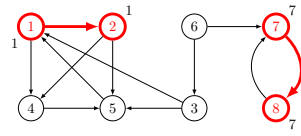
(d) Não há vizinhos do 8.



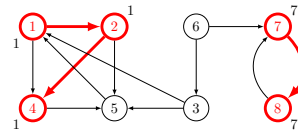
(e) Não há vizinhos do 7.
 $\overleftarrow{D}.POSORDEMREV$:
 $(7, 8, \underline{1}, 5, 4, 2, 3, 6)$.



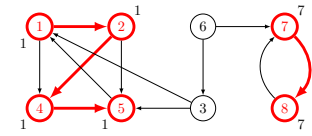
(f) Faz 1. componente = 1.
 Chama
 $BUSCAPROFUNDIDADE(D, 1)$.



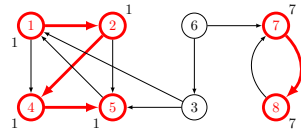
(g) Visita um vizinho do 1,
 o 2. Faz 2. componente = 1.



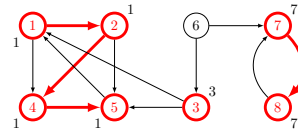
(h) Visita um vizinho do 2,
 o 4. Faz 4. componente = 1.



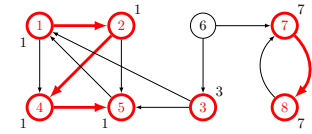
(i) Visita um vizinho do 4,
 o 5. Faz 5. componente = 1.



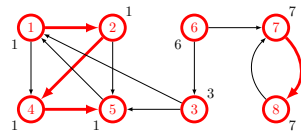
(j) Não há vizinhos do 5.
 $\overleftarrow{D}.POSORDEMREV$:
 $(7, 8, 1, 5, 4, 2, \underline{3}, 6)$.



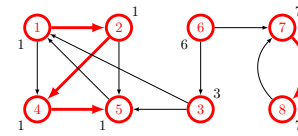
(k) Chama
 $BUSCAPROFUNDIDADE(D, 3)$.
 Faz 3. componente = 3.



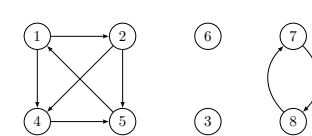
(l) Não há vizinhos do 3.
 $\overleftarrow{D}.POSORDEMREV$:
 $(7, 8, 1, 5, 4, 2, 3, \underline{6})$.



(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 $\overleftarrow{D}.POSORDEMREV$, encontrando as componentes. Os números ao redor dos vértices indicam seus atributos **componente**.

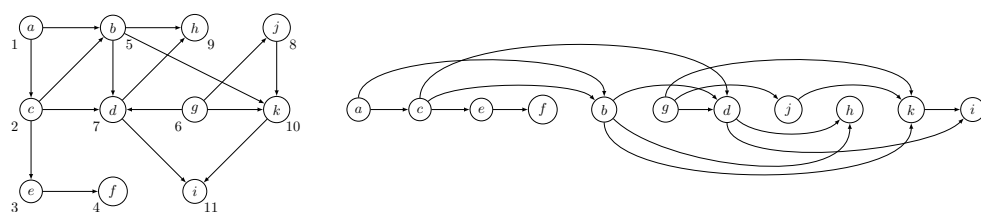


Figura 24.8: Digrafo D à esquerda e uma possível ordenação topológica sua à direita. Os números ao redor dos vértices indicam a rotulação dos mesmos.

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 uma árvore geradora passa a ter um peso, que é a soma dos pesos de suas arestas, e diferentes árvores geradoras possuem pesos diferentes.

Formalmente, dado um grafo G e uma função $w: E(G) \rightarrow \mathbb{R}$ 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

$$w(T) = \min\{w(T'): T' \text{ é uma árvore geradora de } G\}.$$

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

Entrada: $\langle G, w \rangle$, onde G é um grafo conexo e $w: E(G) \rightarrow \mathbb{R}$.

Saída: Árvore geradora T de G cujo peso $w(T) = \sum_{e \in E(T)} w(e)$ é mínimo.

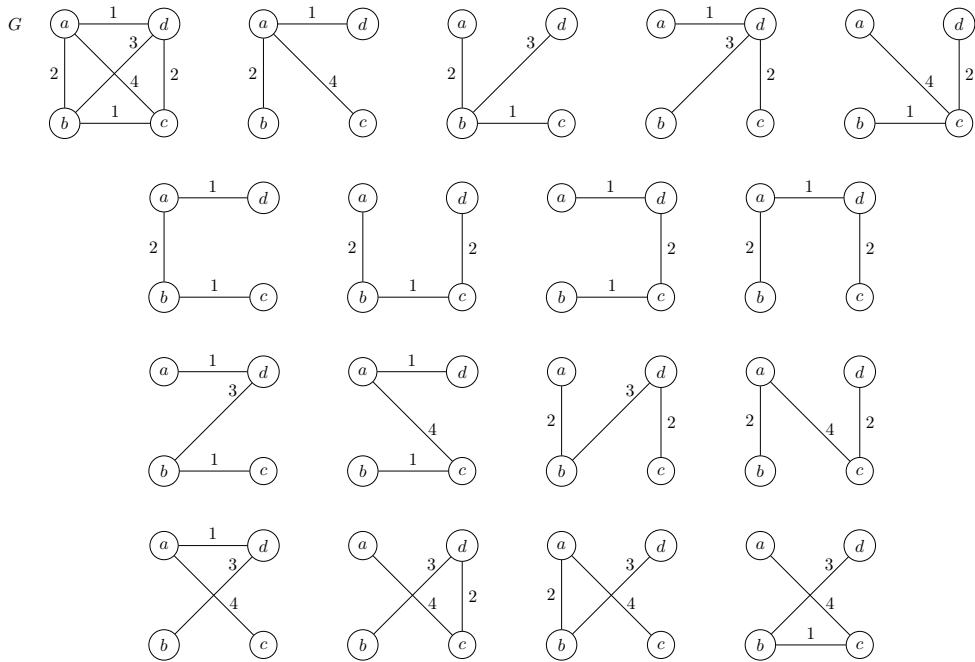


Figura 25.1: Exemplo de um grafo G ao topo e todas as suas 16 árvores geradoras. A primeira e a terceira árvores da segunda linha são árvores geradoras mínimas.

Note que podemos considerar que G é um grafo conexo pois, caso não seja, a árvore geradora mínima de uma componente conexa de G não tem influência sobre as árvores geradoras mínimas das outras componentes. Assim, o problema de encontrar uma floresta geradora mínima em um grafo desconexo G se reduziria a encontrar uma árvore geradora mínima para cada componente conexa de G .

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 fazer 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 . O conjunto $E[S, R] = \{xy \in E(G) : x \in S \text{ e } y \in R\}$ contém as **arestas do corte** (S, R) . Uma aresta $uv \in E(G)$ **cruza** o corte (S, R) se $u \in S$ e $v \in R$, ou seja, se $uv \in E[S, R]$. 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 o corte $(S, V(G) \setminus S)$ 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 $v \in V(H) \setminus S$ para algum $S \subseteq V(H)$. Assim, podemos escrever $C = (u, v, x_1, \dots, x_k, u)$. 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, \dots, 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. Dado um grafo G , dizemos que um corte $(S, V(G) \setminus S)$ **respeita** um conjunto $F \subseteq E(G)$ de arestas se nenhuma aresta de F cruza o corte, isto é, se $F \cap \partial_G(S) = \emptyset$.

Teorema 25.3

Sejam G um grafo conexo e $w: E(G) \rightarrow \mathbb{R}$ uma função de pesos. Seja F um conjunto arestas que está contido em uma árvore geradora mínima de G . Sejam $(S, V(G) \setminus S)$ um corte que respeita F e $e \in \partial_G(S)$ uma aresta mínima desse corte. Então $F \cup \{e\}$ está contido em alguma árvore geradora mínima de G .

Demonstração. Sejam G um grafo conexo e $w: E(G) \rightarrow \mathbb{R}$ uma função de pesos. Considere uma árvore geradora mínima T de G tal que $F \subseteq E(T)$ e seja $S \subseteq V(G)$ um conjunto tal que o corte $(S, V(G) \setminus S)$ respeita F .

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 $F \cup \{e\}$ não está em nenhuma árvore geradora mínima de G .

Como T é uma árvore geradora, então existe caminho de u a v em T , o que significa que $T + e$ contém exatamente um ciclo. Assim, pelo Lema 25.2, 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 também uma árvore geradora. Ademais, como $(S, V(G) \setminus S)$ respeita F , vale que $f \notin F$.

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 $w(T') = w(T)$, isto é, T' é árvore geradora mínima também. Mas note que como $f \notin F$, teremos $F \cup \{e\} \subseteq E(T')$, o que é uma contradição. \square

O Teorema 25.3 nos permite a construção de um algoritmo iterativo simples que começa com um conjunto de arestas F que faz parte de uma árvore geradora mínima e, a cada iteração, adiciona uma nova aresta a esse conjunto (que satisfaça o enunciado do lema), mantendo a invariante de que F sempre fará parte de uma árvore geradora mínima. Com isso, ao final do procedimento, teremos de fato todas as arestas de uma árvore geradora mínima. A dúvida que deve surgir, então, é: como conseguir um tal conjunto F inicial? Felizmente, existe um conjunto de arestas bastante fácil de se obter e que certamente fará parte de qualquer árvore geradora mínima do grafo: o conjunto vazio!

Os algoritmos de Kruskal e Prim, que serão apresentados com detalhes nas seções a seguir, funcionam exatamente dessa forma para resolver o problema Árvore Geradora Mínima. No algoritmo de Kruskal, o conjunto F sempre fará parte de uma floresta que já contém todos os vértices do grafo. Como F começa vazio, no início temos $v(G)$ componentes triviais (um vértice sozinho em cada componente). Já no algoritmo de Prim, o conjunto F sempre fará parte de uma única árvore. Como F começa vazio, no início teremos um único vértice de G , escolhido arbitrariamente. É interessante notar que o algoritmo de Prim é 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 . Ele é considerado um algoritmo guloso (veja Capítulo 21), por ter como escolha gulosa arestas 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, inicialmente vazio. Inicialmente, ele ordena as arestas de forma não-decrescente nos pesos. Em seguida, percorre 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 $\{3, 4\}$ poderiam ter sido escolhidas. Fizemos a escolha da aresta $\{1, 3\}$ 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 KRUSKAL(G, w) de fato gera uma árvore geradora mínima para G .

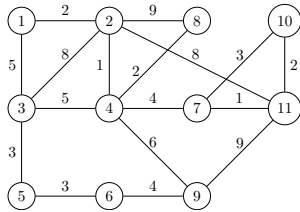
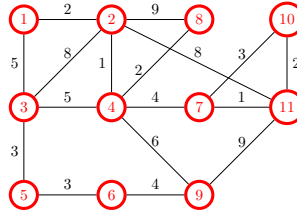
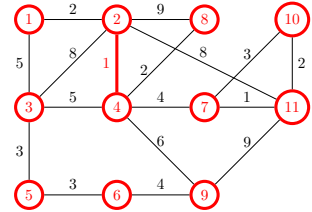
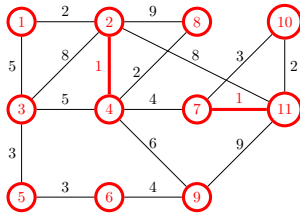
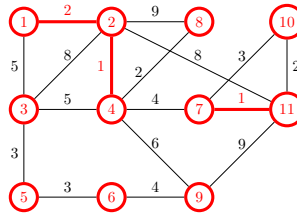
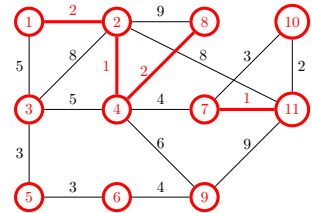
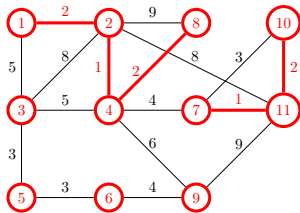
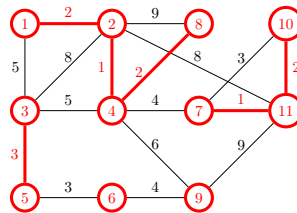
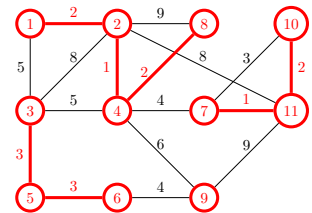
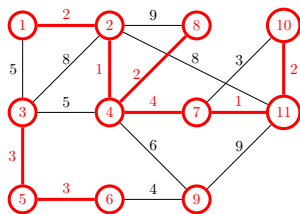
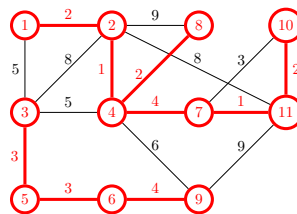
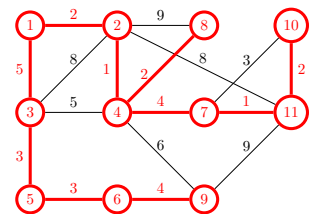
(a) Grafo G de entrada.(b) $F = \emptyset$.(c) $F = \{24\}$.(d) $F = \{24, 711\}$.(e) $F = \{24, 711, 12\}$.(f) $F = \{24, 711, 12, 48\}$.(g) $F = \{24, 711, 12, 48, 1011\}$.(h) $F = \{24, 711, 12, 48, 1011, 35\}$.(i) $F = \{24, 711, 12, 48, 1011, 35, 56\}$.(j) $F = \{24, 711, 12, 48, 1011, 35, 56, 47\}$.(k) $F = \{24, 711, 12, 48, 1011, 35, 56, 47, 69\}$.(l) $F = \{24, 711, 12, 48, 1011, 35, 56, 47, 69, 13\}$.

Figura 25.2: Execução de $\text{KRUSKAL}(G, w)$. As componentes estão destacadas em vermelho.

Lema 25.4

Seja G um grafo conexo e w uma função de pesos nas arestas. O conjunto F de arestas devolvido por $\text{KRUSKAL}(G, w)$ é tal que $G[F]$ é árvore geradora mínima de G .

Demonstração. Para mostrar o resultado, usaremos a seguinte invariante de laço. Para facilitar a discussão, vamos denotar por F_t o valor do conjunto F logo no início da t -ésima iteração.

Invariante: Laço para – KRUSKAL

$P(t)$ = “Antes da t -ésima iteração começar, temos $i = t$ e F_t está contido em uma árvore geradora mínima de G .”

Demonstração. Vamos provar que P é invariante por indução no número t de vezes que o teste da condição de parada é executado.

Caso base. Quando $t = 1$, o teste executou uma única vez. Claramente temos $i = 1$ e $F = \emptyset$ trivialmente está contido em uma árvore geradora mínima de G .

Passo. Considere então que o teste executou $t \geq 1$ vezes e que a t -ésima iteração vai começar. Suponha que $P(t)$ vale e vamos provar que $P(t+1)$ vale. Como $P(t)$ vale, F_t está contido em uma árvore geradora mínima de G . Seja $C[t] = xy$ e seja S o conjunto de vértices da componente conexa de $G[F_t]$ que contém x . Note que $\partial_{G[F_t]}(S)$ é vazio (e que não necessariamente $\partial_G(S)$ é vazio). Note ainda que o corte $(S, V(G) \setminus S)$, pela construção de S , respeita F_t .

Vamos mostrar agora que nenhuma aresta de $\{C[1], \dots, C[t-1]\}$ está em $\partial_G(S)$. De fato, suponha que esse não é o caso e que $C[i] \in \partial_G(S)$, para algum $i < t$. Note que $\partial_{G[F_t]}(S)$ é vazio (caso contrário $\partial_{G[F_t]}(S)$ não seria), o que significa que $\partial_{G[F_t \cup \{C[i]\}]}(S)$ contém apenas $C[i]$. Sendo sozinha no corte em $G[F_t \cup \{C[i]\}]$, pelo Lema 25.2, temos que não existem ciclos em $G[F_t \cup \{C[i]\}]$. Mas então $C[i]$ teria sido escolhida e teríamos $C[i] \in F_t$, e $\partial_{G[F_t]}(S)$ não seria vazio.

Se $G[F_t \cup \{xy\}]$ não contém ciclos, então xy deve cruzar o corte $(S, V(G) \setminus S)$, já que F_t é floresta. Como $w(C[t]) \leq w(C[j])$ para todo $j > t$, e como $\{C[1], \dots, C[t-1]\} \cap \partial_G(S) = \emptyset$, vale que a aresta $C[t]$ é uma aresta mínima em $\partial_G(S)$. O algoritmo então faz $F_{t+1} = F_t \cup \{C[t]\}$ que, pelo Teorema 25.3, está contido em uma árvore geradora mínima de G , o que prova $P(t+1)$. Se $G[F_t \cup \{xy\}]$ contém ciclos, então teremos $F_{t+1} = F_t$, e nesse caso $P(t+1)$ vale diretamente. \diamond

Como P é invariante e o laço executa $e(G)$ vezes, temos que, ao final da execução, F está contido em uma árvore geradora mínima G . Resta mostrar que $G[F]$ é conexo e gerador.

Para isso, basta mostrar que para qualquer corte não vazio do grafo existe uma aresta em F que o cruza. Seja $S \subset V(G)$ 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 para qualquer corte do grafo G existe uma aresta em F que o cruza, de forma que $G[F]$ é conexo e gerador. \square

Com relação ao tempo de execução, 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 $O(mn)$. Assim, 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 $\text{FINDSET}(x)$, que devolve o representante do conjunto que contém o objeto x , e $\text{UNION}(x, y)$, 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 $\text{MAKESET}(x)$ cria um conjunto novo contendo somente o elemento x .

Algoritmo 25.2: $\text{KRUSKALUNIONFIND}(G, w)$

```

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  $F = \emptyset$ 
4 para todo vértice  $v \in V(G)$  faça
5    $\text{MAKESET}(v)$ 
6 para  $i = 1$  até  $e(G)$ , incrementando faça
7   Seja  $uv$  a aresta em  $C[i]$ 
8   se  $\text{FINDSET}(u) \neq \text{FINDSET}(v)$  então
9      $F = F \cup \{C[i]\}$ 
10     $\text{UNION}(u, v)$ 
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).

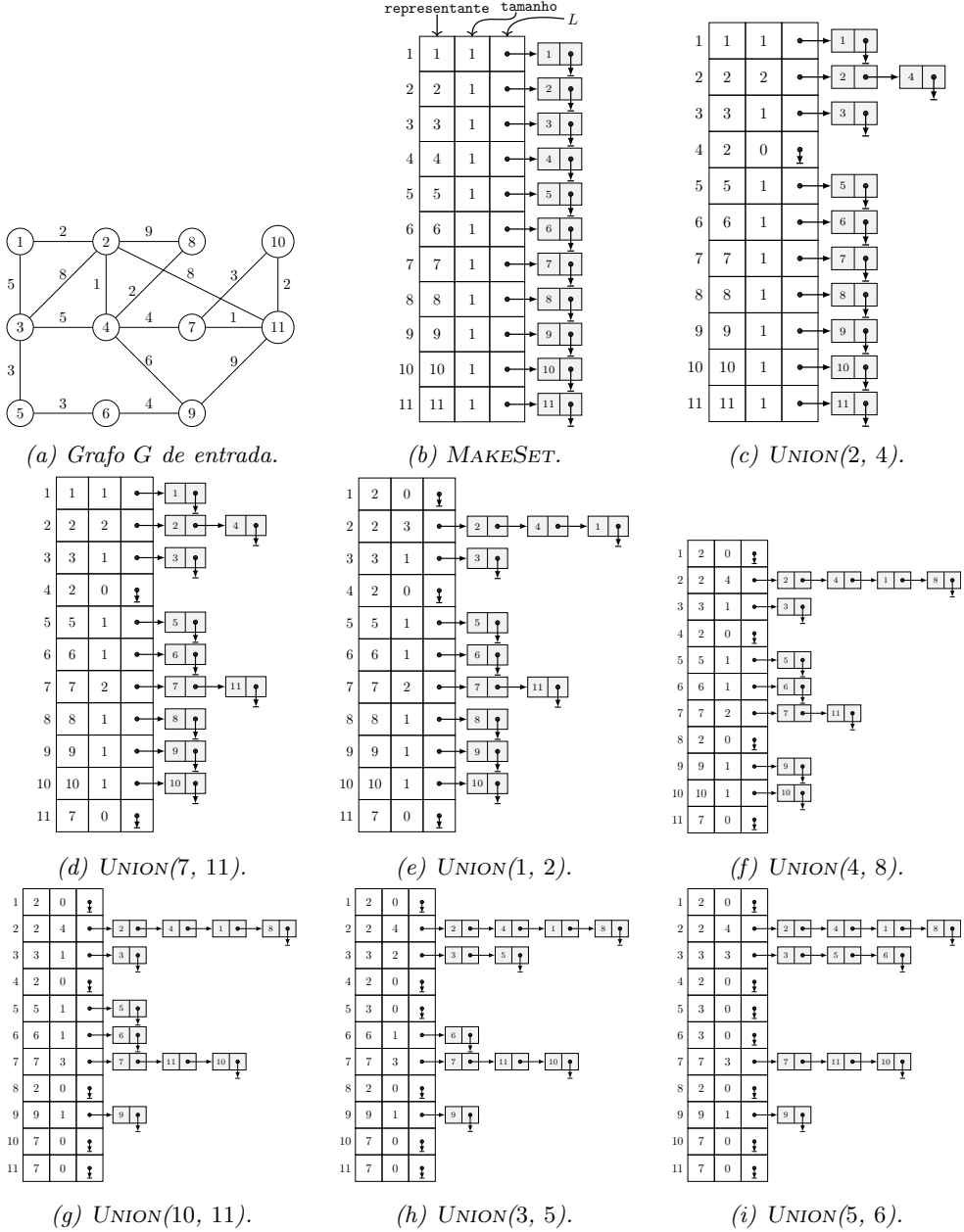
Aqui vamos considerar uma implementação simples do *union-find*, como mencionada na Seção 13.1. Cada conjunto tem como representante um vértice que seja membro do mesmo. Cada vértice x tem um atributo $x.\text{representante}$, que armazena o vértice representante do seu conjunto. Um vértice x também tem um atributo $x.\text{tamanho}$, 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

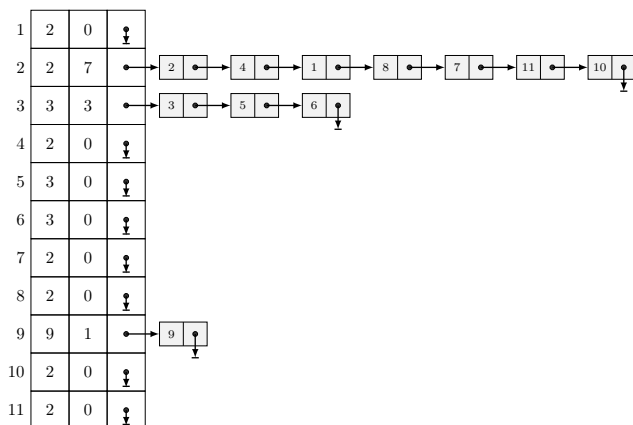
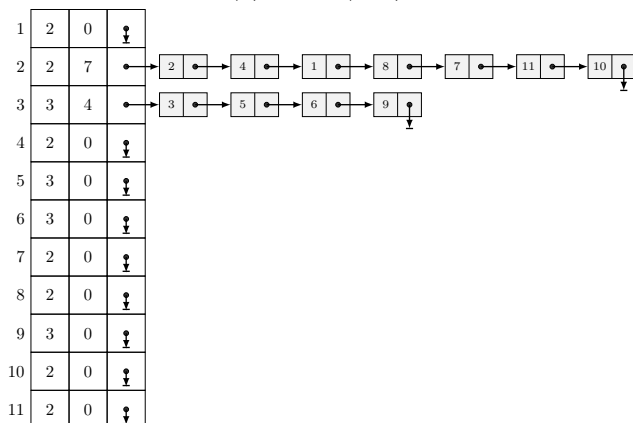
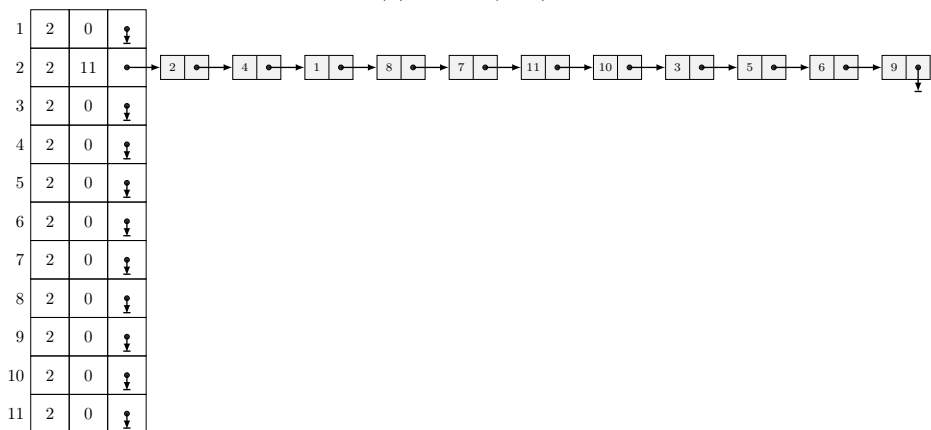
que estão no conjunto representado por x . Assim, $\text{FINDSET}(u)$ leva tempo $\Theta(1)$. Toda vez que dois conjuntos forem unidos, deve-se atualizar os representantes do menor deles com o representante do maior. Assim, $\text{UNION}(u, v)$ percorre a lista $L[u.\text{representante}]$, se $u.\text{tamanho} < v.\text{tamanho}$, ou $L[v.\text{representante}]$, caso contrário, para atualizar os representantes dos vértices presentes nela. Leva, portanto, tempo $\Theta(\min\{u.\text{tamanho}, v.\text{tamanho}\})$. As Figuras 25.3 e 25.4 mostram 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 $O(mn)$. Assim, se $T(n, m)$ é o tempo de execução de $\text{KRUSKALUNIONFIND}(G, w)$, então vale que

$$\begin{aligned} T(n, m) &= \Theta(n + m) + O(m \log m) + \Theta(m) + O(n \log n) \\ &= O(m) + O(m \log n) + \Theta(m) + O(m \log n) \\ &= O(m \log n) . \end{aligned}$$

Figura 25.3: Parte 1 da execução de $\text{KRUSKALUNIONFIND}(G, w)$.

(a) $UNION(4, 7)$.(b) $UNION(6, 9)$.(c) $UNION(1, 3)$.Figura 25.4: Parte 2 da execução de $KRUSKALUNIONFIND(G, w)$.

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 uma árvore trivial, contendo um 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 . 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 marca os vértices como visitados ou não e indica os predecessores de cada vértice. Por isso, cada vértice u tem atributos $u.\text{visitado}$ e $u.\text{pred}$. O atributo $u.\text{pred}$ indica qual vértice levou u a ser visitado. Já o atributo $u.\text{visitado}$ tem valor 1 se o vértice u já foi visitado pelo algoritmo e 0 caso contrário. O algoritmo termina quando não há mais vértices não visitados, o que indica que alcançamos todos os vértices do grafo.

Seja S o conjunto de vértices visitados em um dado momento qualquer do algoritmo. O algoritmo de Prim é um algoritmo guloso (veja Capítulo 21) pois 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, a BUSCA. A Figura 25.5 apresenta um exemplo de sua execução.

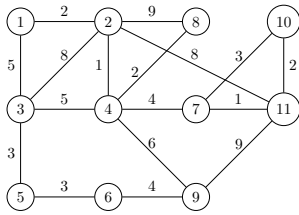
Algoritmo 25.3: PRIM(G, w)

```

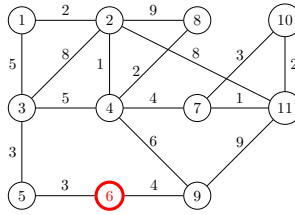
1 para todo vértice  $v \in V(G)$  faça
2    $v.\text{visitado} = 0$ 
3    $v.\text{pred} = \text{NULL}$ 
4 Seja  $s \in V(G)$  qualquer
5  $s.\text{visitado} = 1$ 
6 enquanto houver vértice não visitado faça
7   Seja  $xy$  uma aresta de menor peso com  $x.\text{visitado} == 1$  e
8      $y.\text{visitado} == 0$ 
9    $y.\text{visitado} = 1$ 
10   $y.\text{pred} = x$ 

```

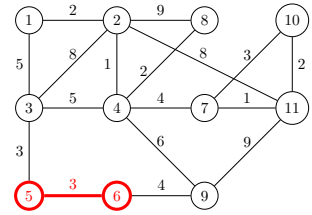
O Lema 25.5 a seguir mostra que PRIM(G, w) de fato gera uma árvore geradora mínima para G .



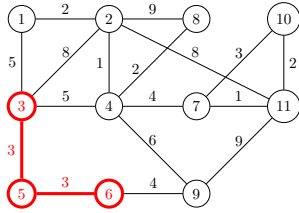
(a) Grafo G de entrada.
Vértice inicial arbitrário:
 $s = 6$.



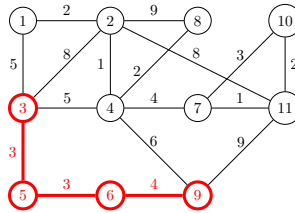
(b) Vértices visitados:
 $S = \{6\}$. $\partial_G(S) = \{56, 69\}$.



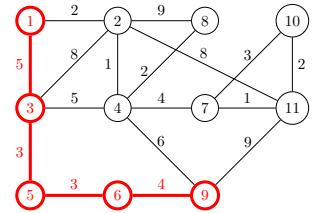
(c) Aresta mínima: 56.
Visitados: $S = \{5, 6\}$.
 $\partial_G(S) = \{35, 69\}$.



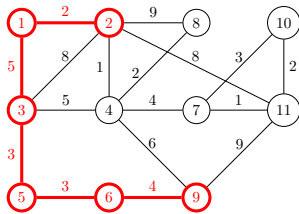
(d) Aresta mínima: 35.
Visitados: $S = \{3, 5, 6\}$.
 $\partial_G(S) = \{13, 23, 34, 69\}$.



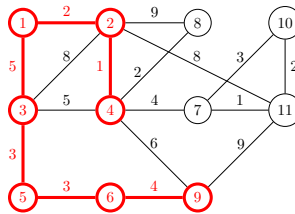
(e) Aresta mínima: 69.
Visitados: $S = \{3, 5, 6, 9\}$.
 $\partial_G(S) = \{13, 23, 34, 49, 911\}$.



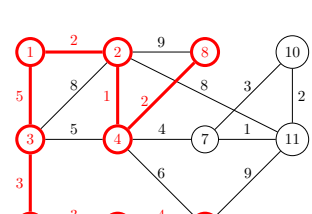
(f) Aresta mínima: 13.
Visitados: $S = \{1, 3, 5, 6, 9\}$.
 $\partial_G(S) = \{12, 23, 34, 49, 911\}$.



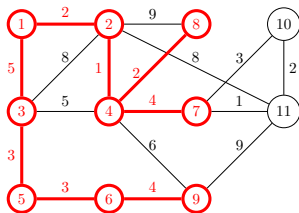
(g) Aresta mínima: 12.
Visitados:
 $S = \{1, 2, 3, 5, 6, 9\}$.
 $\partial_G(S) = \{24, 28, 211, 34, 49, 911\}$.



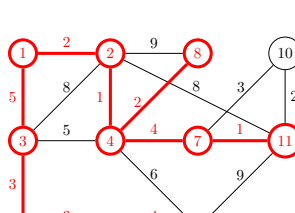
(h) Aresta mínima: 24.
Visitados:
 $S = \{1, 2, 3, 4, 5, 6, 9\}$.
 $\partial_G(S) = \{28, 211, 47, 48, 911\}$.



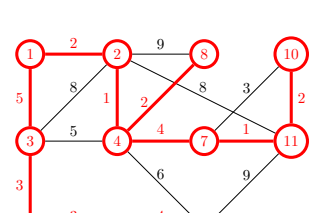
(i) Aresta mínima: 48.
Visitados:
 $S = \{1, 2, 3, 4, 5, 6, 8, 9\}$.
 $\partial_G(S) = \{211, 47, 911\}$.



(j) Aresta mínima: 47.
Visitados:
 $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
 $\partial_G(S) = \{211, 710, 711, 911\}$.



(k) Aresta mínima: 711.
Visitados: $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11\}$.
 $\partial_G(S) = \{710, 1011\}$.



(l) Aresta mínima: 1011.
Visitados: $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$.
 $\partial_G(S) = \emptyset$.

Figura 25.5: Execução de $\text{PRIM}(G, w)$. Os vértices visitados estão em vermelho. A árvore construída de forma indireta pelos predecessores está em vermelho.

Lema 25.5

Seja G um grafo conexo e w uma função de peso nas arestas. Após a execução de $\text{PRIM}(G, w)$, sendo $s \in V(G)$ escolhido na linha 4, o subgrafo T com $V(T) = \{v \in V(G) : v.\text{pred} \neq \text{NULL}\} \cup \{s\}$ e $E(T) = \{\{v.\text{pred}, v\} : v \in V(T) \setminus \{s\}\}$ é uma árvore geradora mínima para G .

Demonstração. Seja S_i o conjunto de vértices visitados no início da i -ésima iteração. Seja $F_i = \{\{v.\text{pred}, v\} : v \in S_i \text{ e } v.\text{pred} \neq \text{NULL}\}$. É fácil perceber, usando o Teorema 25.3, que a seguinte invariante de laço é válida para o algoritmo.

Invariante: Laço enquanto – PRIM

$P(t)$ = “Antes da t -ésima iteração começar, $|S_t| = t$, $G[F_k]$ é conexo e F_t é está contido em uma árvore geradora mínima de G .”

Assim, se o laço executa k vezes, ao final de sua execução temos que $|S_k| = k$, $G[F_k]$ é conexo e F_k está contido em uma árvore geradora mínima de G . Resta então mostrar que $k = |V(G)|$.

De fato, $k = |V(G)|$ diretamente, pois o grafo G é conexo e, portanto, sempre haverá uma escolha possível a ser feita na linha 7, o que faz com que um novo vértice seja visitado a cada iteração e que todos os vértices em $V(G)$ sejam visitados. \square

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 $x.\text{visitado} = 1$ e $y.\text{visitado} = 0$ 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. 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 $S = \{v: v.\text{visitado} = 1\}$ e, para cada $x \in V(G) \setminus S$, $x.\text{prioridade}$ 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 $V(G) = \{1, \dots, v(G)\}$ 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 **pred**, 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 $\text{INSERENAHEAP}(H, v)$ insere o elemento v em H , $\text{REMOVEDAHEAP}(H)$ remove e devolve o elemento de maior prioridade de H e $\text{ALTERAHEAP}(H, v.\text{indice}, x)$ atualiza o valor em $v.\text{prioridade}$ para x . Todas essas operações mantêm a propriedade de heap em H . As Figuras 25.6 e 25.7 mostram a execução de PRIMHEAP sobre o mesmo grafo da Figura 25.5, 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.

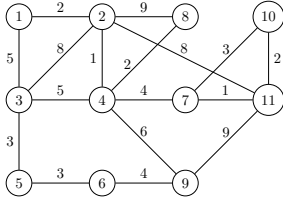
No laço **para** da linha 5 indicamos que os vizinhos de s , ainda não visitados, têm prioridade dada pelo custo da aresta que os conecta a s (lembre-se que o custo é negativo pois maior prioridade é indicada por quem tem menor valor indicador de prioridade). Eles são os únicos que têm aresta para um 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,

Algoritmo 25.4: PRIMHEAP(G, w)

```

1  Seja  $s \in V(G)$  qualquer
2   $s.\text{visitado} = 1$ 
3   $s.\text{pred} = \text{NULL}$ 
4  Seja  $H[1..v(G) - 1]$  um vetor vazio
5  para todo  $v \in N(s)$  faça
6       $v.\text{prioridade} = -w(sv)$ 
7       $v.\text{visitado} = 0$ 
8       $v.\text{pred} = s$ 
9      INSERENAHEAP( $H, v$ )
10 para todo  $v \notin N(s)$  faça
11      $v.\text{prioridade} = -\infty$ 
12      $v.\text{visitado} = 0$ 
13      $v.\text{pred} = \text{NULL}$ 
14     INSERENAHEAP( $H, v$ )
15 enquanto  $H.\text{tamanho} > 0$  faça
16      $v = \text{REMOVEDAHEAP}(H)$ 
17      $v.\text{visitado} = 1$ 
18     para todo  $x \in N(v)$  faça
19         se  $x.\text{visitado} == 0$  e  $x.\text{prioridade} < -w(vx)$  então
20              $x.\text{pred} = v$ 
21             ALTERAHEAP( $H, x.\text{indice}, -w(vx)$ )
  
```



(a) Grafo G de entrada. Vértice inicial arbitrário: $s = 6$.

H	5	9									
	1	2									
pred					6	null			6		
visitado					0	1			0		
índice					1				2		
prioridade					-3				-4		
	1	2	3	4	5	6	7	8	9	10	11

(c) Atualiza prioridade, visitado e pred dos vizinhos de s . Insere-os no heap.

H	3	9	1	2	11	4	7	8	10		
	1	2	3	4	5	6	7	8	9		
pred	null	null	5	null	6	null	null	null	6	null	null
visitado	0	0	0	0	1	1	0	0	0	0	0
índice	3	4	1	6			7	8	2	9	5
prioridade	$-\infty$	$-\infty$	-3	$-\infty$	-3		$-\infty$	$-\infty$	-4	$-\infty$	$-\infty$
	1	2	3	4	5	6	7	8	9	10	11

(e) Remove 5 do heap. Atualiza prioridade de 3.

H	1	2	4	10	11	8	7				
	1	2	3	4	5	6	7				
pred	3	3	5	3	6	null	null	null	6	null	9
visitado	0	0	1	0	1	1	0	0	1	0	0
índice	1	2		3			7	6		4	5
prioridade	-5	-8	-3	-5	-3		$-\infty$	$-\infty$	-4	$-\infty$	-9
	1	2	3	4	5	6	7	8	9	10	11

(g) Remove 9 do heap. Atualiza prioridade de 11.

H											
pred						null					
visitado						1					
índice											
prioridade											
	1	2	3	4	5	6	7	8	9	10	11

(b) Visita s e atualiza seu pred.

H	5	9	1	2	3	4	7	8	10	11	
	1	2	3	4	5	6	7	8	9	10	
pred	null	null	null	null	6	null	null	null	6	null	null
visitado	0	0	0	0	0	1	0	0	0	0	0
índice	3	4	5	6	1		7	8	2	9	10
prioridade	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-3		$-\infty$	$-\infty$	-4	$-\infty$	$-\infty$
	1	2	3	4	5	6	7	8	9	10	11

(d) Atualiza prioridade, visitado e pred dos não vizinhos de s . Insere-os no heap.

H	9	2	1	10	11	4	7	8			
	1	2	3	4	5	6	7	8			
pred	3	3	5	3	6	null	null	null	6	null	null
visitado	0	0	1	0	1	1	0	0	0	0	0
índice	3	2		6			7	8	1	4	5
prioridade	-5	-8	-3	-5	-3		$-\infty$	$-\infty$	-4	$-\infty$	$-\infty$
	1	2	3	4	5	6	7	8	9	10	11

(f) Remove 3 do heap. Atualiza prioridade de 1, 2, 4.

H	2	4	7	10	11	8					
	1	2	3	4	5	6					
pred	3	1	5	3	6	null	null	null	6	null	9
visitado	1	0	1	0	1	1	0	0	1	0	0
índice		1		2			3	6		4	5
prioridade	-5	-2	-3	-5	-3		$-\infty$	$-\infty$	-4	$-\infty$	-9
	1	2	3	4	5	6	7	8	9	10	11

(h) Remove 1 do heap. Atualiza prioridade de 2.

Figura 25.6: Parte 1 da execução de $\text{PRIMHEAP}(G, w)$ em que G e w são dados em 25.6a. 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						
pred	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
	1	2	3	4	5	6	7	8	9	10	11

(a) Remove 2 do heap. Atualiza prioridade de 4, 8, 11.

H	8	7	11	10							
	1	2	3	4							
pred	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
	1	2	3	4	5	6	7	8	9	10	11

(b) Remove 4 do heap. Atualiza prioridade de 7, 8.

H	7	10	11								
	1	2	3								
pred	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
	1	2	3	4	5	6	7	8	9	10	11

(c) Remove 8 do heap. Sem vizinhos não visitados.

H	11	10									
	1	2									
pred	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
	1	2	3	4	5	6	7	8	9	10	11

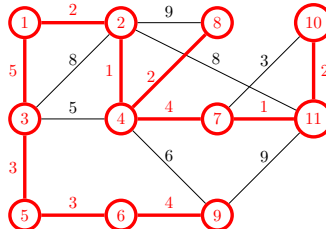
(d) Remove 7 do heap. Atualiza prioridade de 10, 11.

H	10										
	1										
pred	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
	1	2	3	4	5	6	7	8	9	10	11

(e) Remove 11 do heap. Atualiza prioridade de 10.

H											
pred	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
	1	2	3	4	5	6	7	8	9	10	11

(f) Remove 10 do heap. Sem vizinhos não visitados.



(g) Heap vazio. Árvore gerada com os predecessores.

Figura 25.7: Parte 2 da execução de $\text{PRIMHEAP}(G, w)$ em que G e w são dados em 25.6a. Note que em H estamos mostrando os rótulos dos vértices, e não suas prioridades.

as prioridades de alguns vértices podem mudar, pois os conjuntos de vértices visitados e não 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 $\Theta(mn)$.

Trilhas Eulerianas

Lembre-se que uma *trilha* em um (di)grafo G é uma sequência de vértices (v_0, v_1, \dots, 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, \dots, 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 26.3

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 26.4

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.

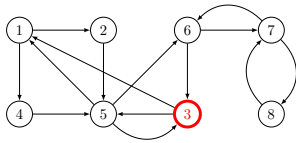
O algoritmo de Fleury, descrito no Algoritmo 26.1, recebe um digrafo fortemente conexo D do tipo fechado ou aberto e um vértice inicial s . Ele devolve um vetor W tal que $(W[1], W[2], \dots, W[e(G)+1])$ é uma trilha Euleriana fechada ou aberta em D . Se D é do tipo fechado, então s pode ser um vértice 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)

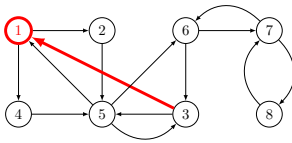
```

1  Seja  $W[1..e(D)+1]$  um vetor vazio
2   $W[1] = s$ 
3   $i = 1$ 
4  enquanto  $d_D^+(W[i]) \geq 1$  faça
5      se existe arco  $(W[i], y)$  que é seguro  $D$  então
6           $W[i+1] = y$ 
7      senão
8           $W[i+1] = y$ , onde  $(W[i], y)$  não é seguro em  $D$ 
9           $D = D - (W[i], W[i+1])$  /* Removendo o arco visitado pela trilha */
10          $i = i + 1$ 
11 devolve  $W$ 
```

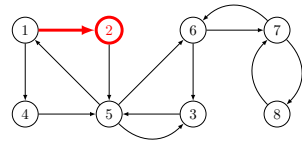
Uma **decomposição** de um (di)grafo G é uma coleção $\mathcal{D} = \{G_1, \dots, G_k\}$ de subgrafos de G tal que $E(G_i) \cap E(G_j) = \emptyset$ para todo $1 \leq i < j \leq k$ e $\bigcup_{i=1}^k E(G_i) = E(G)$. Em geral, fala-se em decomposição em um mesmo tipo de subgrafos. Por exemplo, se todos 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.



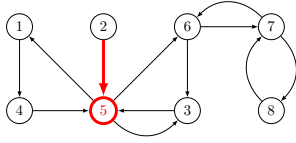
(a) Digrafo D de entrada (do tipo fechado) e vértice inicial $s = 3$. Trilha: (3) .



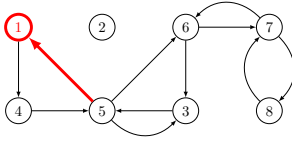
(b) Pode seguir qualquer arco que sai de 3. Trilha: $(3, 1)$.



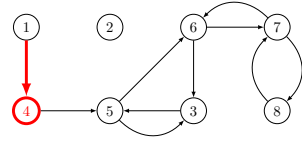
(c) Pode seguir qualquer arco que sai de 1. Trilha: $(3, 1, 2)$.



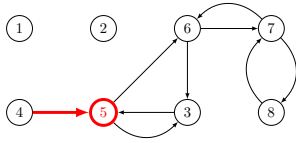
(d) Saem apenas arcos não seguros de 2. Trilha: $(3, 1, 2, 5)$.



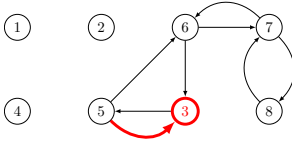
(e) Pode seguir qualquer arco que sai de 5. Trilha: $(3, 1, 2, 5, 1)$.



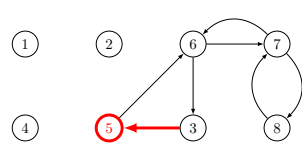
(f) Saem apenas arcos não seguros de 1. Trilha: $(3, 1, 2, 5, 1, 4)$.



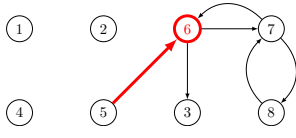
(g) Saem apenas arcos não seguros de 4. Trilha: $(3, 1, 2, 5, 1, 4, 5)$.



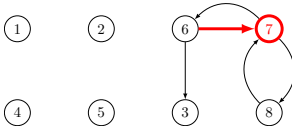
(h) Pode seguir qualquer arco que sai de 5. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3)$.



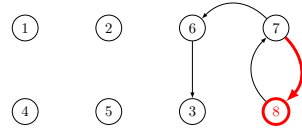
(i) Pode seguir qualquer arco que sai de 3. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3, 5)$.



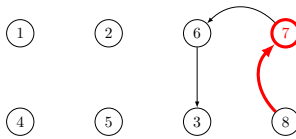
(j) Saem apenas arcos não seguros de 5. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3, 5, 6)$.



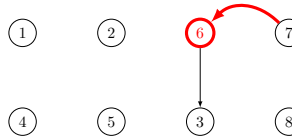
(k) O arco $(6, 3)$ não é seguro mas $(6, 7)$ é. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3, 5, 6, 7)$.



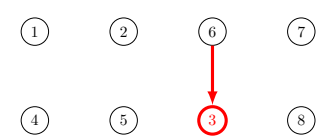
(l) O arco $(7, 6)$ não é seguro mas $(7, 8)$ é. Trilha: $(3, 1, 2, 5, 1, 4, 5, 3, 5, 6, 7, 8)$.



(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

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.

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 $T = (W[1], \dots, W[j])$ 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 $E(H) = E(D) \setminus \{W[i]W[i+1] : 1 \leq i < j\}$. 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. \square

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 $v.\text{visitado} == 0$, 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.

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 $w: E(G) \rightarrow \mathbb{R}$ sendo a função de peso. Lembre-se que o peso de um caminho $P = (v_0, v_1, \dots, v_k)$ é igual à soma dos pesos das arestas ou arcos desse caminho, isto é, $w(P) = \sum_{i=0}^{k-1} w(v_i v_{i+1})$. A **distância entre u e v em G** , denotada por $\text{dist}_G^w(u, v)$, é o menor peso de um uv -caminho, dentre todos os uv -caminhos, ou é ∞ , caso não haja uv -caminho. Assim, $\text{dist}_G^w(u, u) = 0$. 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 situações¹.

Uma técnica 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

¹Essa afirmação será provada no Capítulo 29.

$w'(uv) = w'(vu) = w(uv)$. 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 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

Entrada: $\langle D, w, s \rangle$, onde D é um digrafo, w é uma função de peso nos arcos e $s \in V(D)$.

Saída: Valor $\text{dist}_D^w(s, v)$ para todo $v \in V(D)$.

Problema 27.2: Caminhos Mínimos Entre Todos os Pares

Entrada: $\langle D, w \rangle$, onde D é um digrafo e w é uma função de peso nos arcos.

Saída: Valor $\text{dist}_D^w(u, v)$ 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 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 $w: E(D) \rightarrow \mathbb{Z}_+$ 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 $P = (v_1, \dots, v_k)$ é um $v_1 v_k$ -caminho mínimo, então qualquer subcaminho (v_i, \dots, v_j) de P também é mínimo. Suponha que existam pares $1 \leq i < j \leq k$ tais que $(v_i, v_{i+1}, \dots, v_{j-1}, v_j)$ não é mínimo. Seja então $(v_i, u_1, \dots, u_q, v_j)$ um $v_i v_j$ -caminho mínimo. Ao substituir esse trecho em P , temos que $(v_1, \dots, v_i, u_1, \dots, u_q, v_j, v_{j+1}, \dots, v_k)$ é um $v_1 v_k$ -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, 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 $v.\text{dist}$, 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 $v.\text{dist} = \text{dist}_D^w(s, v)$. Cada vértice possui também um atributo $v.\text{pred}$, que contém o predecessor de v no sv -caminho que foi encontrado pelo algoritmo, e um atributo $v.\text{visitado}$, que indica se v já foi alcançado a partir de s ou não. Já vimos que com o atributo $v.\text{pred}$ 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 `dist` dos vértices por meio de relaxações. Dizemos que um arco uv é **relaxado** quando verificamos se $v.\text{dist} > u.\text{dist} + w(uv)$, e atualizamos, em caso positivo, o valor de $v.\text{dist}$ para $u.\text{dist} + w(uv)$. 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 $P = (v_0, v_1, v_2, \dots, v_k)$ em um digrafo, dizemos que uma sequência de relaxações de arcos é **P -ordenada** se os arcos $v_0v_1, v_1v_2, \dots, v_{k-1}v_k$ forem relaxadas nesta ordem, não importando se outros arcos forem relaxadas entre quaisquer das relaxações $v_0v_1, v_1v_2, \dots, v_{k-1}v_k$. 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 $s.\text{dist} = 0$ e $v.\text{dist} = \infty$ 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 $v.\text{dist} \geq \text{dist}_D^w(s, v)$.
2. Se $P = (s, v_1, v_2, \dots, v_k)$ é um sv_k -caminho mínimo e A realiza uma sequência P -ordenada de relaxações, então ao fim da execução de A teremos $v_k.\text{dist} = \text{dist}_D^w(s, v_k)$.

Demonstração. Começemos mostrando o item 1. Suponha, para fins de contradição, que em algum momento da execução de A temos um vértice v com $v.\text{dist} < \text{dist}_D^w(s, v)$. Seja v o *primeiro* vértice a ficar com $v.\text{dist} < \text{dist}_D^w(s, v)$ durante a sequência de relaxações. Pela natureza do algoritmo A , o vértice v teve $v.\text{dist}$ alterado quando algum arco uv foi relaxado. Pela escolha de v , sabemos que $u.\text{dist} \geq \text{dist}_D^w(s, u)$. Portanto, ao relaxar o arco uv , obtivemos $v.\text{dist} = u.\text{dist} + w(uv) \geq \text{dist}_D^w(s, u) + w(uv) \geq \text{dist}_D^w(s, v)$, 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 $P = (s, v_1, v_2, \dots, v_k)$. 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 $s.\text{dist} = 0 = \text{dist}_D^w(s, s)$.

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 $P = (s, v_1, v_2, \dots, v_k)$ de s a v_k com k arcos e suponha que os arcos $sv_1, v_1v_2, \dots, v_{k-1}v_k$ foram relaxados nessa ordem. Note que como $P' = (s, v_1, v_2, \dots, v_{k-1})$ 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 $sv_1, v_1v_2, \dots, v_{k-2}v_{k-1}$, foram relaxadas nessa ordem e P' tem $k - 1$ arcos, concluímos por hipótese de indução que $v_{k-1}.\text{dist} = \text{dist}_D^w(s, v_{k-1})$. Caso $v_k.\text{dist} = \text{dist}_D^w(s, v_k)$, então a prova está concluída. Assim, podemos assumir que

$$v_k.\text{dist} \neq \text{dist}_D^w(s, v_k) . \quad (27.1)$$

Não é o caso de $v_k.\text{dist} < \text{dist}_D^w(s, v_k)$, como já vimos no item 1. Assim, $v_k.\text{dist} > \text{dist}_D^w(s, v_k)$. Ao relaxar o arco $v_{k-1}v_k$, o algoritmo vai verificar que $v_k.\text{dist} > \text{dist}_D^w(s, v_k) = \text{dist}_D^w(s, v_{k-1}) + w(v_{k-1}v_k) = v_{k-1}.\text{dist} + w(v_{k-1}v_k)$, atualizando o valor de $v_k.\text{dist}$, portanto, para $v_{k-1}.\text{dist} + w(v_{k-1}v_k)$, que é igual a $\text{dist}_D^w(s, v_k)$, uma contradição com (27.1). Logo, $v_k.\text{dist} = \text{dist}_D^w(s, v_k)$. \square

27.1.1 Algoritmo de Dijkstra

Um dos algoritmos mais clássicos na Ciência da Computação é o *algoritmo de Dijkstra*, que resolve o problema 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 $s.\text{dist} = 0$, e todos os outros vértices têm estimativa de distância $v.\text{dist} = \infty$. A cada iteração, um vértice não visitado x é escolhido para

ser visitado (e, portanto, o valor em $x.\text{dist}$ 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 $x.\text{dist}$) seguido do peso do arco xy pode ser melhor do que a estimativa atual de y (que está em $y.\text{dist}$). 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 $\text{dist} \neq \infty$, 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 $v.\text{pred}$ e $v.\text{visitado}$, além do atributo $v.\text{dist}$ já mencionado. O atributo $v.\text{pred}$ deve conter o predecessor de v no sv -caminho que está sendo construído pelo algoritmo. Já o atributo $v.\text{visitado}$ deve ter valor 1 se v foi visitado e 0 caso contrário.

O Algoritmo 27.1 formaliza o algoritmo de Dijkstra e 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     $v.\text{pred} = \text{NULL}$ 
3     $v.\text{dist} = \infty$ 
4     $v.\text{visitado} = 0$ 
5   $s.\text{dist} = 0$ 
6  enquanto houver vértice  $u$  com  $u.\text{visitado} == 0$  e  $u.\text{dist} \neq \infty$  faça
7    seja  $x$  um vértice não visitado com menor valor  $x.\text{dist}$ 
8     $x.\text{visitado} = 1$ 
9    para todo vértice  $y \in N^+(x)$  faça
10   se  $y.\text{visitado} == 0$  então
11     se  $x.\text{dist} + w(xy) < y.\text{dist}$  então
12        $y.\text{dist} = x.\text{dist} + w(xy)$ 
13        $y.\text{pred} = x$ 

```

Note que o digrafo T tal que

$$V(T) = \{v \in V(D) : v.\text{pred} \neq \text{NULL}\} \cup \{s\}$$

$$E(T) = \{(v.\text{pred}, v) : v \in V(T) \setminus \{s\}\}$$

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 $v.\text{dist}$,

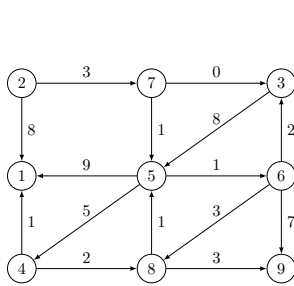
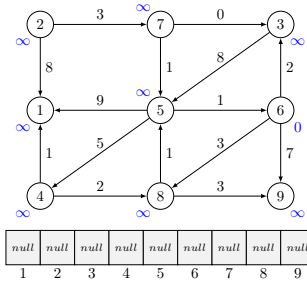
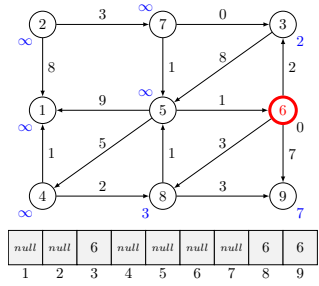
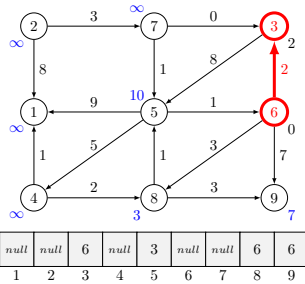
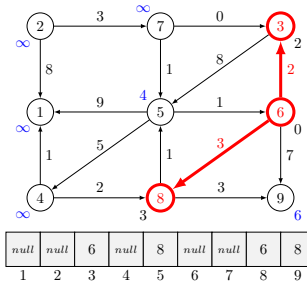
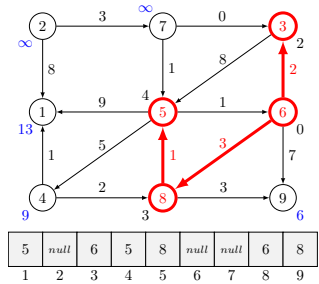
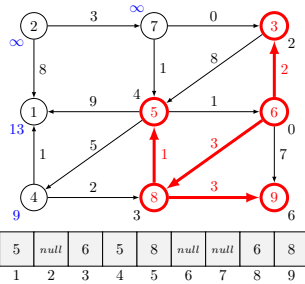
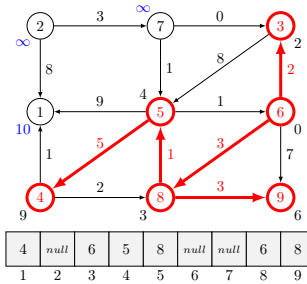
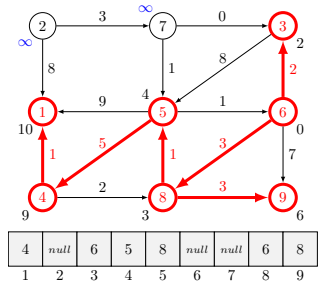
(a) Digrafo D de entrada.(b) Vértice inicial: $s = 6$.
Inicializa atributos.(c) Menor estimativa:
visita 6. Relaxa arcos 63, 68
e 69.(d) Menor estimativa:
visita 3. Relaxa arco 35.(e) Menor estimativa:
visita 8. Relaxa arcos 85 e
89.(f) Menor estimativa:
visita 5. Relaxa arcos 51 e
54.(g) Menor estimativa:
visita 9. Sem arcos para
relaxar.(h) Menor estimativa:
visita 4. Relaxa arco 41.(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 $\text{dist} \neq \infty$.

pode ser construído pelo Algoritmo 24.4, CONSTROICAMINHO.

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

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 $\text{DIJKSTRA}(G, w, s)$ temos $v.\text{dist} = \text{dist}_D^w(s, v)$ para todo $v \in V(D)$.

Demonstração. Considere uma execução de $\text{DIJKSTRA}(G, w, s)$. 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 $v.\text{dist} = \text{dist}_D^w(s, v)$ para todo $v \in V(D)$. Inicialmente, fazemos $v.\text{dist} = \infty$ e $s.\text{dist} = 0$. A partir daí, o algoritmo só modifica as estimativas $v.\text{dist}$ através de relaxações. Assim, pelo Lema 27.3, sabemos que $v.\text{dist} \geq \text{dist}_D^w(s, v)$. Uma vez que o algoritmo nunca modifica o atributo $v.\text{dist}$ depois que v é visitado, basta provarmos que $v.\text{dist} \leq \text{dist}_D^w(s, v)$ no momento em que v é visitado.

Para finalizar a prova, usaremos a seguinte invariante de laço.

Invariante: Laço enquanto – DIJKSTRA

$R(t)$ = “Antes da t -ésima iteração começar, temos $v.\text{dist} \leq \text{dist}_D^w(s, v)$ para todo $v \in V(D)$ tal que $v.\text{visitado} = 1$.”

Demonstração. Provaremos que R é invariante por indução no número t de vezes que o teste da condição de parada é executado.

Caso base. Quando $t = 1$, o teste executou uma única vez. Antes da primeira iteração começar, nenhum vértice está visitado, de forma que $R(1)$ vale trivialmente. Também note que apenas $s.\text{dist} = 0$ e $v.\text{dist} = \infty$ para todo outro

$v \in V(D) \setminus \{s\}$. Assim, durante a primeira iteração, o vértice escolhido para ser visitado na linha 7 é s e, de fato, $\text{dist}_D^w(s, s) = 0$. Assim, provamos também que $R(2)$ vale.

Passo. Considere então que o teste executou t vezes, com $t \geq 2$, e que a t -ésima iteração vai começar. Suponha que $R(t)$ vale e vamos mostrar $R(t+1)$.

Seja x o vértice escolhido na t -ésima iteração, na linha 7. Como $t \geq 2$, sabemos que $x \neq s$. Como x será o único vértice com $x.\text{visitado}$ alterado neste momento, basta mostrar que $x.\text{dist} \leq \text{dist}_D^w(s, x)$ para garantir que $R(t+1)$ é verdadeira.

Seja P um sx -caminho mínimo qualquer e seja q o primeiro vértice não visitado de P , dentre os vértices que não estão visitados na iteração atual (deve haver um tal q , já que $x \neq s$, s está visitado e x não está). Ademais, seja z o vértice que precede q no caminho P . Pela escolha de q , sabemos que z foi visitado em iterações anteriores. Assim, como $R(t)$ vale, sabemos que $z.\text{dist} \leq \text{dist}_D^w(s, z)$, o que significa que $z.\text{dist} = \text{dist}_D^w(s, z)$. Note que $w(P) = \text{dist}_D^w(s, x)$.

Note que podemos descrever o peso de P como $\text{dist}_D^w(s, x) = w(P) = w(P') + w(zq) + w(P'')$, em que P' é o subcaminho que vai de s a z e P'' é o subcaminho que vai de q 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, $\text{dist}_D^w(s, x) \geq w(P') + w(zq) = \text{dist}_D^w(s, z) + w(zq) = z.\text{dist} + w(zq)$. No momento em que z foi visitado, os arcos que saem de z para vértices não visitados foram relaxados, incluindo o arco zq . Portanto, $q.\text{dist} \leq z.\text{dist} + w(zq)$. Juntando com o resultado anterior, temos $t.\text{dist} \leq \text{dist}_D^w(s, x)$.

Por fim, note que no momento em que x é escolhido para ser visitado, o vértice q 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 $x.\text{dist} \leq q.\text{dist}$, o que junto com o resultado anterior nos diz que $x.\text{dist} \leq \text{dist}_D^w(s, x)$, provando assim $R(t+1)$. \diamond

Como R é invariante, ao final do laço vale que todo vértice v visitado tem $v.\text{dist} = \text{dist}_D^w(s, v)$. Se o laço terminou porque não há vértice não visitado, então o resultado da invariante prova o teorema diretamente. Então todo vértice u com $u.\text{visitado} = 0$ tem $u.\text{dist} = \infty$. Vamos provar que não há su -caminho para um tal u , o que, junto com o resultado da invariante termina a prova do teorema.

Suponha que há su -caminho P e $u.\text{visitado} = 0$. Seja q o primeiro vértice não visitado em P e z seu predecessor. Então z é visitado e $q.\text{dist} = \infty$. Mas quando z

foi visitado, o arco zq foi relaxado, o que é uma contradição com $q.\text{dist} = \infty$. \square

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 $\text{DIJKSTRA}(D, w, s)$. 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 **dist** 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 **REMOVEFROMHEAP**, 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 $-v.\text{dist}$. Note que vamos manter um valor negativo pois tem maior prioridade o vértice que tiver menor valor em $v.\text{dist}$. Assim, o próximo vértice a ser visitado deve ser o vértice removido do heap.

Assuma que $V(D) = \{1, \dots, v(D)\}$ e que cada vértice v ainda possui os atribui-

tos **prioridade**, para armazenar seu valor de prioridade, e **indice**, para indicar a posição do heap em que v está armazenado. O Algoritmo 27.2 rerepresenta o algoritmo de Dijkstra utilizando explicitamente a estrutura heap. Lembre-se que $\text{INSERENAHEAP}(H, v)$ insere o elemento v em H , $\text{REMOVEDAHEAP}(H)$ remove e devolve o elemento de maior prioridade de H e $\text{ALTERAHEAP}(H, v.\text{indice}, x)$ atualiza o valor em $v.\text{prioridade}$ 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)

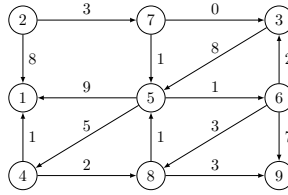
```

1  Seja  $H[1..v(D)]$  um vetor vazio
2  para todo vértice  $v \in V(D)$  faça
3       $v.\text{pred} = \text{NULL}$ 
4       $v.\text{prioridade} = -\infty$ 
5       $v.\text{visitado} = 0$ 
6       $\text{INSERENAHEAP}(H, v)$ 
7   $\text{ALTERAHEAP}(H, s.\text{indice}, 0)$ 
8  enquanto  $u = \text{CONSULTAHEAP}(H)$  e  $u.\text{prioridade} \neq -\infty$  faça
9       $x = \text{REMOVEDAHEAP}(H)$ 
10      $x.\text{visitado} = 1$ 
11     para todo vértice  $y \in N^+(x)$  faça
12         se  $y.\text{visitado} == 0$  então
13             se  $x.\text{prioridade} + (-w(xy)) > y.\text{prioridade}$  então
14                  $\text{ALTERAHEAP}(H, y.\text{indice}, x.\text{prioridade} + (-w(xy)))$ 
15                  $y.\text{pred} = x$ 

```

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 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,

(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
pred	null	null	null	null	null	null	null	null	null
visitado	0	0	0	0	0	0	0	0	0
índice	3	2	6	4	5	1	7	8	9
prioridade	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$
	1	2	3	4	5	6	7	8	9

(b) Inicializa heap. Atribui
6.prioridade = 0.

H	8	5	9	2	4	1	7
	1	2	3	4	5	6	7
pred	null	null	6	null	3	null	6
visitado	0	0	1	0	0	1	0
índice	6	4		5	2		7
prioridade	$-\infty$	$-\infty$	-2	$-\infty$	-10	0	$-\infty$
	1	2	3	4	5	6	7

(d) $\text{REMOVEDAHEAP}(H) = 3$. Relaxa arco
35.

H	9	4	7	2	1
	1	2	3	4	5
pred	5	null	6	5	8
visitado	0	0	1	0	1
índice	5	4		2	
prioridade	-13	$-\infty$	-2	-9	-4
	1	2	3	4	5

(f) $\text{REMOVEDAHEAP}(H) = 5$. Relaxa arcos
51 e 54.

H	1	2	7
	1	2	3
pred	4	null	6
visitado	0	0	1
índice	1	2	
prioridade	-10	$-\infty$	-2
	1	2	3

(h) $\text{REMOVEDAHEAP}(H) = 4$. Relaxa arco
41.

H	3	8	9	2	5	1	7	4
	1	2	3	4	5	6	7	8
pred	null	null	6	null	null	null	6	6
visitado	0	0	0	0	0	1	0	0
índice	6	4	1	8	5		7	2
prioridade	$-\infty$	$-\infty$	-2	$-\infty$	$-\infty$	0	$-\infty$	-3
	1	2	3	4	5	6	7	8

(c) $\text{REMOVEDAHEAP}(H) = 6$. Relaxa
arcos 63, 68 e 69.

H	5	9	7	2	4	1
	1	2	3	4	5	6
pred	null	null	6	null	8	null
visitado	0	0	1	0	0	1
índice	6	4		5	1	
prioridade	$-\infty$	$-\infty$	-2	$-\infty$	-4	0
	1	2	3	4	5	6

(e) $\text{REMOVEDAHEAP}(H) = 8$. Relaxa
arcos 85 e 89.

H	4	1	7	2
	1	2	3	4
pred	5	null	6	5
visitado	0	0	1	0
índice	2	4		1
prioridade	-13	$-\infty$	-2	-9
	1	2	3	4

(g) $\text{REMOVEDAHEAP}(H) = 9$. Sem arcos
para relaxar.

H	7	2
	1	2
pred	4	null
visitado	1	0
índice		2
prioridade	-10	$-\infty$
	1	2

(i) $\text{REMOVEDAHEAP}(H) = 1$. Sem arcos
para relaxar.

Figura 27.2: Execução de $\text{DIJKSTRA-HEAP}(D, w, 6)$. Note que em H estamos mostrando os rótulos dos vértices, e não suas prioridades.

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 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 $V(D) = \{1, \dots, v(D)\}$.

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 $v.\text{pred}$, além do atributo $v.\text{dist}$ já mencionado. O atributo $v.\text{pred}$ 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 $D.\text{cicloNeg}$, 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

$$\begin{aligned} V(T) &= \{v \in V(D) : v.\text{pred} \neq \text{NULL}\} \cup \{s\} \\ E(T) &= \{(v.\text{pred}, v) : v \in V(T) \setminus \{s\}\} \end{aligned}$$

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, CONSTROICAMINHO. As Figuras 27.3 e 27.4 mostram dois exemplos de execução.

Algoritmo 27.3: BELLMAN-FORD(D, w, s)

```

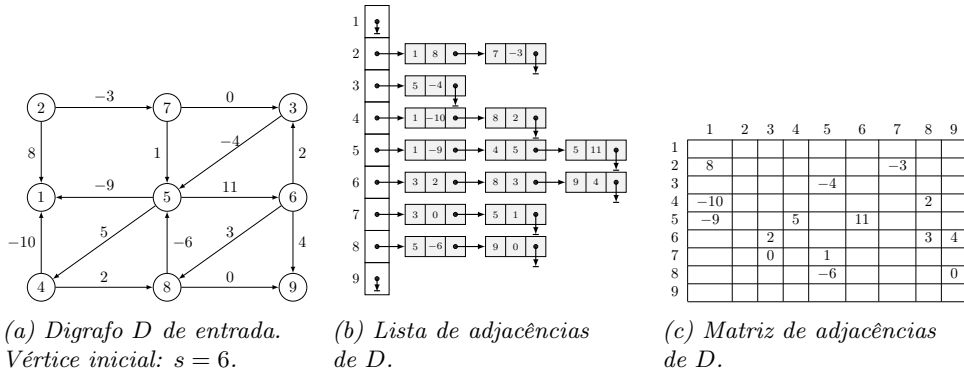
1  $D.\text{cicloNeg} = 0$ 
2 para todo vértice  $v \in V(D)$  faça
3    $v.\text{dist} = \infty$ 
4    $v.\text{pred} = \text{NULL}$ 
5  $s.\text{dist} = 0$ 
6 para  $i = 1$  até  $v(D) - 1$ , incrementando faça
7   para todo arco  $xy \in E(D)$  faça
8     se  $y.\text{dist} > x.\text{dist} + w(xy)$  então
9        $y.\text{dist} = x.\text{dist} + w(xy)$ 
10       $y.\text{pred} = x$ 
11 para todo arco  $xy \in E(D)$  faça
12   se  $y.\text{dist} > x.\text{dist} + w(xy)$  então
13      $D.\text{cicloNeg} = 1$ 

```

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 $P = (s, v_1, v_2, \dots, v_k, v)$, 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 peso 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 $v.\text{dist} = \text{dist}_D^w(s, v)$ para todo vértice $v \in V(D)$. Ademais, temos $D.\text{cicloNeg} = 0$.



$((2, 1), (2, 7), (3, 5), (4, 1), (4, 8), (5, 1), (5, 4), (5, 6), (6, 3), (6, 8), (6, 9), (7, 3), (7, 5), (8, 5), (8, 9))$

(d) Sequência dos arcos seguida pelo algoritmo.

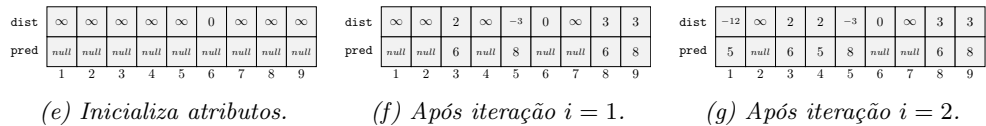


Figura 27.3: Execução de $BELLMAN-FORD(D, w, 6)$. Após a iteração $i = 2$, não há mudanças.

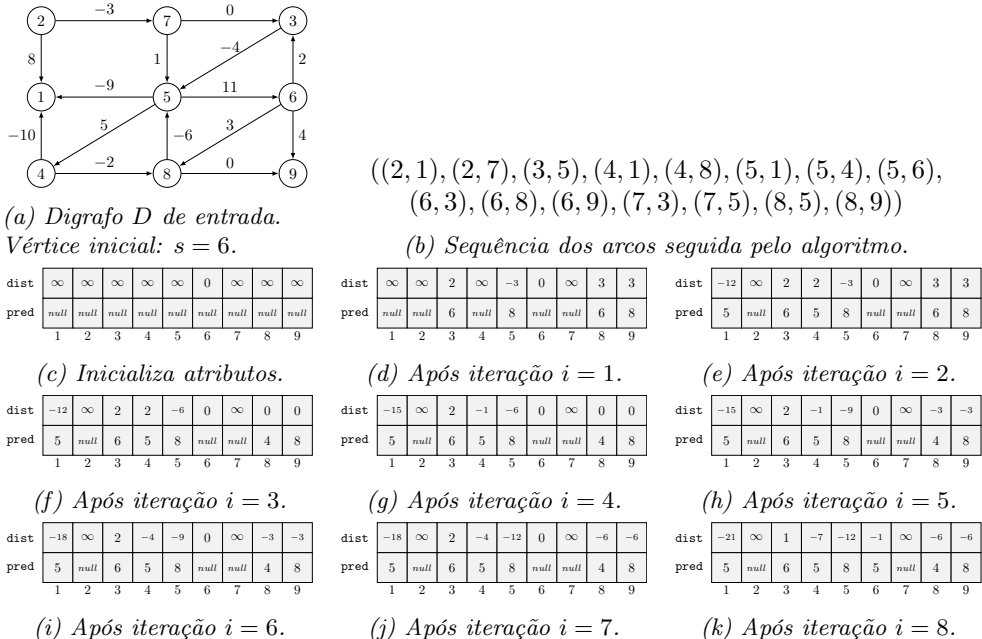


Figura 27.4: Execução de $BELLMAN-FORD(D, w, 6)$. O algoritmo detecta ciclo negativo.

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 $v.\text{dist}$. Assim, para esse tipo de vértice vale que $v.\text{dist} = \infty = \text{dist}_D^w(s, v)$.

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 $P = (s, v_1, v_2, \dots, v_k, v)$ 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 $v_0v_1, v_1v_2, \dots, v_kv_{k+1}$ são relaxados nessa ordem pelo algoritmo. Pelo Lema 27.3, temos $v_{k+1}.\text{dist} = \text{dist}_D^w(s, v_{k+1})$.

Uma vez que $y.\text{dist} = \text{dist}_D^w(s, y)$ 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. \square

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 peso em seus arcos e seja $s \in V(D)$. Se D contém ciclos de peso negativo, então após a execução de $\text{BELLMAN-FORD}(D, w, s)$ temos $D.\text{cicloNeg} = 1$.

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 $v.\text{dist} > u.\text{dist} + w(uv)$, de modo que a linha 13 é executada, fazendo $D.\text{cicloNeg} = 1$. \square

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 $\text{BELLMAN-FORD}(D, w, s)$. 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, $\text{BELLMAN-FORD}(D, w, s)$ 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 Caminhos Mínimos Entre Todos os Pares sobre 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 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 $\text{DIJKSTRA}(G, w, s)$ 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 $\text{BELLMAN-FORD}(D, w, s)$ é $\Theta(nm)$, então n execuções dele levam $\Theta(n^2m)$. 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 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 $V(D) = \{1, \dots, n\}$, 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  $ij \in E(D)$  então
3     devolve  $w(ij)$ 
4   devolve  $\infty$ 
5 Seja  $k \in X$ 
6 nao_usa_k = CAMINHO( $D, w, i, j, X \setminus \{k\}$ )
7 usa_k = CAMINHO( $D, w, i, k, X \setminus \{k\}$ ) + CAMINHO( $D, w, k, j, X \setminus \{k\}$ )
8 devolve  $\min\{\text{nao\_usa\_k}, \text{usa\_k}\}$ 

```

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 $(i, \dots, x, \dots, k, \dots, x, \dots, j)$, 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 $\min\{\text{nao_usa_}k, \text{usa_}k\}$ 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, escolhamos k como sendo o vértice com maior rótulo disponível. Assim, inicialmente $k = n$.

Finalmente, para resolver o problema 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 forma 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 $P_{i,j}^k$ como o peso de um ij -caminho mínimo que contém vértices internos pertencentes ao conjunto $V_k = \{1, 2, \dots, k\}$, 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

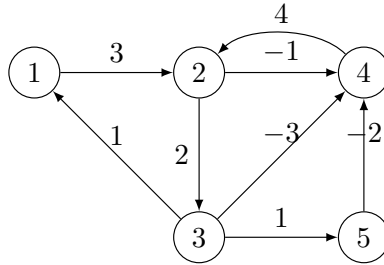
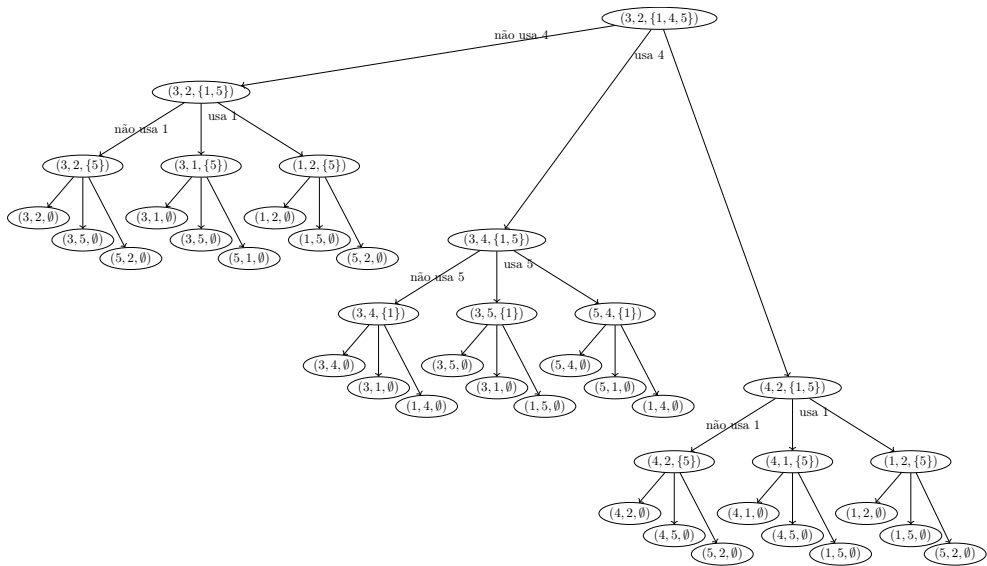
(a) Digrafo D de entrada.(b) Árvore de recursão para $\text{CAMINHO}(D, w, 3, 2, \{1, 4, 5\})$.

Figura 27.5: Construindo 32-caminho no digrafo D . Note como todos os caminhos entre 3 e 2 estão sendo testados 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”.

base, temos que

$$P_{i,j}^0 = \begin{cases} 0 & \text{se } i = j \\ w(ij) & \text{se } ij \in E(D) \text{ e } i \neq j \\ \infty & \text{se } i \neq j \text{ e } ij \notin E(D) \end{cases} . \quad (27.2)$$

E, de modo geral, para $1 \leq k \leq n$, temos

$$P_{i,j}^k = \min \left\{ P_{i,j}^{k-1}, P_{i,k}^{k-1} + P_{k,j}^{k-1} \right\} . \quad (27.3)$$

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 $j.\text{pred}$ que será um vetor de tamanho n tal que $j.\text{pred}[i]$ 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        $W[i][j][k] = \infty$ 
5 para  $i = 1$  até  $n$ , incrementando faça
6   para  $j = 1$  até  $n$ , incrementando faça
7      $W[i][j][n] = \text{FLOYD-WARSHALLREC-TOPDOWN}(D, w, n, i, j)$ 
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.

Algoritmo 27.6: FLOYD-WARSHALLREC-TOPDOWN(D, w, k, i, j)

```

1 se  $W[i][j][k] == \infty$  então
2   se  $k == 0$  então
3     se  $i == j$  então
4        $W[i][j][0] = 0$ 
5        $j.\text{pred}[i] = i$ 
6     senão se  $ij \in E(D)$  então
7        $W[i][j][0] = w(ij)$ 
8        $j.\text{pred}[i] = i$ 
9     senão
10       $W[i][j][0] = \infty$ 
11       $j.\text{pred}[i] = \text{NULL}$ 
12   senão
13      $\text{nao\_usa\_k} = \text{FLOYD-WARSHALLREC-TOPDOWN}(D, w, k-1, i, j)$ 
14      $\text{usa\_k} = \text{FLOYD-WARSHALLREC-TOPDOWN}(D, w, k-1, i, k) +$ 
         $\text{FLOYD-WARSHALLREC-TOPDOWN}(D, w, k-1, k, j)$ 
15     se  $\text{nao\_usa\_k} < \text{usa\_k}$  então
16        $W[i][j][k] = \text{nao\_usa\_k}$ 
17     senão
18        $W[i][j][k] = \text{usa\_k}$ 
19        $j.\text{pred}[i] = j.\text{pred}[k]$ 
20 devolve  $W[i][j][k]$ 

```

Algoritmo 27.7: FLOYD-WARSHALL-BOTTOMUP(D, w)

```

1  Seja  $W[1..n][1..n][0..n]$  uma matriz
2  para  $i = 1$  até  $n$ , incrementando faça
3      para  $j = 1$  até  $n$ , incrementando faça
4          se  $i == j$  então
5               $W[i][j][0] = 0$ 
6               $j.\text{pred}[i] = i$ 
7          senão se  $ij \in E(D)$  então
8               $W[i][j][0] = w(ij)$ 
9               $j.\text{pred}[i] = i$ 
10         senão
11              $W[i][j][0] = \infty$ 
12              $j.\text{pred}[i] = \text{NULL}$ 
13  para  $k = 1$  até  $n$ , incrementando faça
14      para  $i = 1$  até  $n$ , incrementando faça
15          para  $j = 1$  até  $n$ , incrementando faça
16               $\text{nao\_usa\_k} = W[i][j][k-1]$ 
17               $\text{usa\_k} = W[i][k][k-1] + W[k][j][k-1]$ 
18              se  $\text{nao\_usa\_k} < \text{usa\_k}$  então
19                   $W[i][j][k] = \text{nao\_usa\_k}$ 
20              senão
21                   $W[i][j][k] = \text{usa\_k}$ 
22                   $j.\text{pred}[i] = j.\text{pred}[k]$ 
23  devolve  $W$ 

```

Algoritmo 27.8: FLOYD-WARSHALL-MELHORADO(D, w)

```

1  Seja  $W[1..n][1..n]$  uma matriz
2  para  $i = 1$  até  $n$ , incrementando faça
3      para  $j = 1$  até  $n$ , incrementando faça
4          se  $i == j$  então
5               $W[i][j] = 0$ 
6               $j.\text{pred}[i] = i$ 
7          senão se  $ij \in E(D)$  então
8               $W[i][j] = w(ij)$ 
9               $j.\text{pred}[i] = i$ 
10         senão
11              $W[i][j] = \infty$ 
12              $j.\text{pred}[i] = \text{NULL}$ 
13 para  $k = 1$  até  $n$ , incrementando faça
14     para  $i = 1$  até  $n$ , incrementando faça
15         para  $j = 1$  até  $n$ , incrementando faça
16             se  $W[i][j] > W[i][k] + W[k][j]$  então
17                  $W[i][j] = W[i][k] + W[k][j]$ 
18                  $j.\text{pred}[i] = j.\text{pred}[k]$ 
19 devolve  $W$ 

```

Por causa dos três laços aninhados, independente da economia de espaço ou não, claramente o tempo de execução de $\text{FLOYD-WARSHALL-BOTTOMUP}(D, w)$ 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 o resultado do próprio $\text{FLOYD-WARSHALL-BOTTOMUP}$. 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: $\text{RESOLVECAMINHOSENTRETODOSPARES}(D, w)$

```

1  $W = \text{FLOYD-WARSHALL-BOTTOMUP}(D, w)$ 
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 $\text{RESOLVECAMINHOSENTRETODOSPARES}$. 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: $\text{CONSTROICAMINHO}(D, i, j)$

```

1 Seja  $L$  uma lista vazia
2  $\text{atual} = j$ 
3 enquanto  $\text{atual} \neq i$  faça
4   |  $\text{INSERENOINICIOLISTA}(L, \text{atual})$ 
5   |  $\text{atual} = \text{atual.pred}[i]$ 
6  $\text{INSERENOINICIOLISTA}(L, i)$ 
7 devolve  $L$ 
```

27.2.2 Algoritmo de Johnson

O algoritmo de Johnson também é um algoritmo para tratar do problema Caminhos Mínimos Entre Todos os Pares (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 $w: E(D) \rightarrow \mathbb{R}$ nos arcos sobre o qual queremos resolver o problema Caminhos Mínimos Entre Todos os Pares. Crie um novo digrafo \bar{D} com $V(\bar{D}) = V(D) \cup \{s\}$ e $E(\bar{D}) = E(D) \cup \{sv: v \in V(D)\}$, isto é, existe um novo vértice s que se conecta a todos os outros vértices. Estenda a função w para uma função \bar{w} tal que $\bar{w}(sv) = 0$ para todo $v \in V(D)$ e $\bar{w}(uv) = w(uv)$ 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

$$\bar{w}(xy) = (\text{dist}_{\bar{D}}^{\bar{w}}(s, x) + \bar{w}(xy)) - \text{dist}_{\bar{D}}^{\bar{w}}(s, y)$$

para todo $xy \in E(\bar{D})$. Observe que $\bar{w}(xy) \geq 0$ para todo $xy \in E(\bar{D})$, pois $\text{dist}_{\bar{D}}^{\bar{w}}(s, x) + \bar{w}(xy) \geq \text{dist}_{\bar{D}}^{\bar{w}}(s, y)$, 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 $\text{DIJKSTRA}(\bar{D}, \bar{w}, u)$ 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 $\text{DIJKSTRA}(\bar{D}, \bar{w})$ equivale a um uv -caminho mínimo em D com função w , para todo $u, v \in V(D)$. Seja $P = (u, x_1, \dots, x_k, v)$ 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 \bar{w} ,

em uma expressão que dependa apenas de w da seguinte forma:

$$\begin{aligned}
 \bar{w}(P) &= \bar{w}(ux_1) + \bar{w}(x_1x_2) + \cdots + \bar{w}(x_{k-1}x_k) + \bar{w}(x_kv) \\
 &= (\text{dist}_{\bar{D}}^{\bar{w}}(s, u) + \bar{w}(ux_1) - \text{dist}_{\bar{D}}^{\bar{w}}(s, x_1)) + \\
 &\quad (\text{dist}_{\bar{D}}^{\bar{w}}(s, x_1) + \bar{w}(x_1x_2) - \text{dist}_{\bar{D}}^{\bar{w}}(s, x_2)) + \cdots + \\
 &\quad (\text{dist}_{\bar{D}}^{\bar{w}}(s, x_{k-1}) + \bar{w}(x_{k-1}x_k) - \text{dist}_{\bar{D}}^{\bar{w}}(s, x_k)) + \\
 &\quad (\text{dist}_{\bar{D}}^{\bar{w}}(s, v) + \bar{w}(x_kv) - \text{dist}_{\bar{D}}^{\bar{w}}(s, v)) \\
 &= \bar{w}(ux_1) + \bar{w}(x_1x_2) + \cdots + \bar{w}(x_{k-1}x_k) + \bar{w}(x_kv) + (\text{dist}_{\bar{D}}^{\bar{w}}(s, u) - \text{dist}_{\bar{D}}^{\bar{w}}(s, v)) \\
 &= w(ux_1) + w(x_1x_2) + \cdots + w(x_{k-1}x_k) + w(x_kv) + (\text{dist}_{\bar{D}}^{\bar{w}}(s, u) - \text{dist}_{\bar{D}}^{\bar{w}}(s, v)) .
 \end{aligned}$$

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 de mesmo uv -caminho em D com função w somado ao mesmo valor fixo $\text{dist}_{\bar{D}}^{\bar{w}}(s, u) - \text{dist}_{\bar{D}}^{\bar{w}}(s, v)$. 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 $\text{dist}_{\bar{D}}^{\bar{w}}(s, v)$ para todo $v \in V(D)$? Essa é justamente a definição do problema Caminhos Mínimos de Única Fonte. Note que \bar{D} com função de peso \bar{w} pode conter arcos de pesos negativos, de forma que calcular distâncias em \bar{D} é algo que não pode ser feito por Dijkstra, por exemplo. Porém, aqui podemos utilizar o algoritmo de Bellman-Ford. Inclusive, se em \bar{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 $u.\text{dists}$, que é um vetor com $v(D)$ posições e deverá armazenar em $u.\text{dists}[v]$ o peso de um uv -caminho mínimo. Para facilitar o entendimento do algoritmo, cada vértice u possui ainda um campo $u.\text{distOrig}$ numérico, que armazenará a distância de s a u calculada por Bellman-Ford sobre \bar{D} com função \bar{w} , e um campo $u.\text{dist}$ numérico, que armazenará a su -distância calculada pelo algoritmo de Dijkstra sobre \bar{D} , \bar{w} e s .

Note que o tempo de execução de $\text{JOHNSON}(D, w)$ 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 $O((mn + n^2) \log n)$.

Algoritmo 27.11: JOHNSON(D, w)

```

1 Seja  $\ddot{D}$  um digrafo com  $V(\ddot{D}) = V(D) \cup \{s\}$  e
    $E(\ddot{D}) = E(D) \cup \{sv : v \in V(D)\}$ 
2 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  $uv \in E(D)$ 
3 BELLMAN-FORD( $\ddot{D}, \ddot{w}, s$ )
4 se  $\ddot{D}.\text{cicloNeg} == 1$  então
5   └─ devolve “O digrafo  $D$  contém ciclo de peso negativo”
6 Seja  $\bar{D}$  um digrafo com  $V(\bar{D}) = V(D)$  e  $E(\bar{D}) = E(D)$ 
7 Seja  $\bar{w}$  função nos arcos de  $\bar{D}$  com
    $\bar{w}(uv) = u.\text{distOrig} + \ddot{w}(uv) - v.\text{distOrig}$ , para todo  $uv \in E(D)$ 
8 para toda vértice  $u \in V(D)$  faça
9   └─ DIJKSTRA( $\bar{D}, \bar{w}, u$ ) /* Assim,  $v.\text{dist} = \text{dist}_{\bar{D}}^{\bar{w}}(u, v) \ \forall v \in V(D)$  */
10  └─ para toda vértice  $v \in V(D)$  faça
11    └─  $u.\text{dists}[v] = v.\text{dist} + (v.\text{distOrig} - u.\text{distOrig})$ 

```

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 e k é um inteiro positivo que não depende de n (constante).

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.

Na verdade, muitos problemas interessantes e com fortes motivações e aplicações práticas ainda não são tratáveis, 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.

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 problema¹.

Começemos com um exemplo. Considere o seguinte problema, da Seleção.

Problema 28.1: Seleção

Entrada: $\langle V, n, k \rangle$, onde $V[1..n]$ é um vetor de tamanho n e $k \in \{1, \dots, n\}$.

Saída: 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 simples de resolver esse problema é ordenando V . Com isso, teremos o vetor $(3, 6, 7, 8, 9, 12, 45, 78, 234)$ e fica 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 Seleção para o problema Ordenação! Com isso, temos um algoritmo para o problema Seleção, mostrado no Algoritmo 28.1.

Vamos observar o que está acontecendo nessa redução com detalhes. Nós recebemos uma instância $\langle V, n, k \rangle$ para o problema Seleção. Então nós a transformamos em uma instância válida para o problema Ordenação. No caso desses problemas, não

¹Não 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.

Algoritmo 28.1: $\text{ALG_SELECAO}(V, n, k)$

```

1  $\text{ALG\_ORDENACAO}(V, n)$ 
2 devolve  $V[k]$ 

```

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 QUICKSORT. Dado o resultado do problema Ordenação, que é o vetor modificado, nós o transformamos em um resultado para o problema 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 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

Entrada: $\langle x \rangle$, onde x é um inteiro.

Saída: Valor x^2 .

Problema 28.3: Multiplicação de inteiros

Entrada: $\langle x, y \rangle$, onde x e y são inteiros.

Saída: Produto $x \times y$.

Não é difícil notar que o problema Quadrado se reduz ao problema Multiplicação. Dada uma instância $\langle a \rangle$ para o problema Quadrado, podemos transformá-la na instância $\langle a, a \rangle$ para o problema Multiplicação e, agora, qualquer algoritmo de multiplicação pode ser utilizado. Como $a \times a = a^2$, temos diretamente a solução para o problema original.

Outra possibilidade é a redução na direção inversa: do problema Multiplicação para o problema 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.

Algoritmo 28.2: ALG_MULTIPLICACAO(x, y)

```

1  $a \leftarrow \text{ALG\_QUADRADO}(x + y)$ 
2  $b \leftarrow \text{ALG\_QUADRADO}(x)$ 
3  $c \leftarrow \text{ALG\_QUADRADO}(y)$ 
4 devolve  $(a - b - c)/2$ 
```

Considere agora os problemas de caminhos mínimos, reescritos a seguir, e que já haviam sido definidos no Capítulo 27.

Problema 28.4: *Caminhos Mínimos de Única Fonte*

Entrada: $\langle D, w, s \rangle$, onde D é um digrafo, w é uma função de peso nos arcos e $s \in V(D)$.

Saída: Valor $\text{dist}_D^w(s, v)$ para todo $v \in V(D)$.

Problema 28.5: *Caminhos Mínimos Entre Todos os Pares*

Entrada: $\langle D, w \rangle$, onde D é um digrafo e w é uma função de peso nos arcos.

Saída: Valor $\text{dist}_D^w(u, v)$ para todo par $u, v \in V(D)$.

Observe que é possível reduzir do problema Caminhos Mínimos Entre Todos os Pares para o problema 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. Também é possível fazer a redução na direção inversa: do problema Caminhos Mínimos de Única Fonte para o problema Caminhos Mínimos Entre Todos os Pares. Se sabemos os custos dos caminhos entre todos os pares, então também sabemos os custos a partir de um vértice específico.

Esses exemplos talvez possam nos levar a (erroneamente) achar que só é possível reduzir problemas que tenham entradas parecidas (números com números, vetores com vetores ou grafos com grafos). O exemplo a seguir nos mostrará que é possível reduzir entre problemas que aparentemente não teriam relação. O problema Escalonamento de Tarefas Compatíveis já foi visto na Seção 21.1 e encontra-se replicado a seguir.

Problema 28.6: Escalonamento de Tarefas Compatíveis

Entrada: $\langle T, n, s, f \rangle$, onde $T = \{t_1, \dots, t_n\}$ é um conjunto com n tarefas onde cada $t_i \in T$ tem tempo inicial s_i e tempo final f_i .

Saída: Maior subconjunto $S \subseteq T$ de tarefas mutuamente compatíveis.

Um **conjunto independente** em um grafo G é um conjunto de vértices $S \subseteq V(G)$ tal que não há arestas entre qualquer par de vértices em S . Formalmente, para todo par $u, v \in S$ vale que $uv \notin E(G)$. Vamos mostrar que ele pode ser reduzido ao problema Conjunto Independente Máximo, definido a seguir.

Problema 28.7: Conjunto independente máximo

Entrada: $\langle G \rangle$, onde G é um grafo.

Saída: Conjunto independente $S \subseteq V(G)$ de tamanho máximo.

Queremos então reduzir do problema Escalonamento de Tarefas Compatíveis para o problema Conjunto Independente Máximo. Isso significa que queremos usar um algoritmo que resolve o segundo para resolver 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: $\text{ALG_TAREFAS}(T, n, s, f)$

- 1 Crie um grafo G vazio
- 2 Faça $V(G) = T$
- 3 Faça $E(G) = \{t_i t_j : s_i < f_j \text{ e } s_j < f_i\}$
- 4 $S \leftarrow \text{ALG_CONJ_INDEPENDENTE}(G)$
- 5 devolve S

Se ALG é um algoritmo qualquer para um problema, I é uma instância para tal problema, e $S(I)$ é a solução que ALG cria para I , representaremos graficamente essa situação da seguinte forma:

$$I \longrightarrow \boxed{ALG} \longrightarrow S(I),$$

Reduzir de um problema A para um problema B significa que podemos criar um algoritmo para o problema A usando um algoritmo para o problema B . Especificamente, nós queremos transformar qualquer instância I_A do problema A , por meio de alguma função de transformação f , em uma instância específica I_B para B , usar um algoritmo para B , e depois transformar a solução devolvida para I_B em uma solução válida para I_A , por meio de alguma outra função de transformação g :

$$I_A \xrightarrow{f} I_B \longrightarrow \boxed{ALG_B} \longrightarrow S(I_B) \xrightarrow{g} S(I_A).$$

Observe como isso nos dá um algoritmo para o problema A :

$$I_A \longrightarrow \boxed{\xrightarrow{f} I_B \longrightarrow \boxed{ALG_B} \longrightarrow S(I_B) \xrightarrow{g}} \longrightarrow S(I_A).$$

Perceba que isso não impede que outros algoritmos existam para o problema A !

28.1 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*

Entrada: $\langle G, w, u, v, k \rangle$, onde G é um grafo, w é uma função de peso nas arestas, $u, v \in V(G)$ são dois vértices e k é um valor.

Pergunta: 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**, que é um conjunto de dados extras, bem relacionado com o que chamaríamos de solução nos problemas de otimização. 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 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**.

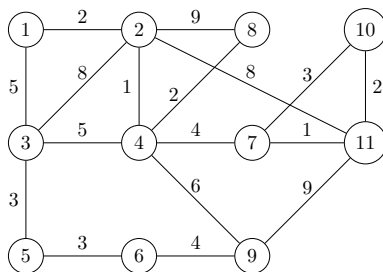


Figura 28.1: Parte de uma instância válida para o problema Caminho: um grafo com pesos nas arestas.

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 a 9. E note como isso é mais difícil de ser verificado: precisamos observar que qualquer caminho entre 3 e 10 tem peso maior do que isso para nos convenceremos 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

Entrada: $\langle G, w, u, v \rangle$, onde G é um grafo, w é uma função de peso nas arestas e $u, v \in V(G)$.

Saída: Um uv -caminho de peso mínimo.

O problema 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 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 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 Caminho (sabemos dizer **sim** ou **não** para qualquer grafo G e valor 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 $k \in \{0, 1, 2, \dots, nC\}$ e, para o menor valor cuja solução for **sim**, temos a resposta para o problema Caminho Mínimo. Veja a Figura 28.2 para um esquema dessa discussão.

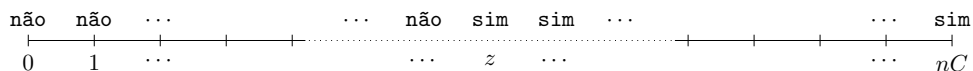


Figura 28.2: Exemplificação da discussão sobre a relação entre problemas de decisão e problemas de otimização.

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 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 **reduzível polinomialmente** para B se existe algoritmo eficiente f tal que $f(I_A) = I_B$ e

I_A é **sim** se e somente se I_B é **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 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*

Entrada: $\langle D \rangle$, onde D é um digrafo.

Pergunta: Existe caminho hamiltoniano em D ?

Problema 28.14: *Ciclo Hamiltoniano*

Entrada: $\langle D \rangle$, onde D é um digrafo.

Pergunta: 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 Caminho Hamiltoniano, o que pode ser certificado pelo caminho (a, d, b, c, e) . Veja que ele é uma instância **não** para o problema 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 problema 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.

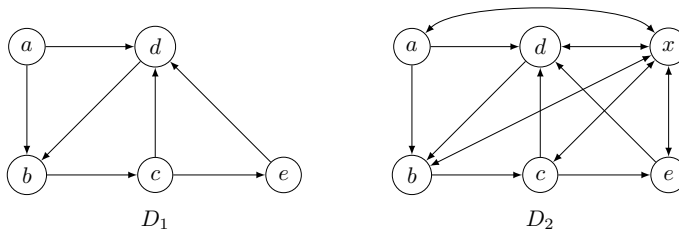


Figura 28.3: Digrafo D_1 entrada para o problema do Caminho hamiltoniano e digrafo D_2 criado a partir de D_1 via redução para o problema Ciclo hamiltoniano.

Vamos mostrar que o problema Caminho Hamiltoniano é redutível em tempo polinomial ao problema 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 então 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 $E(D') = E(D) \cup \{xu, ux : u \in V(D)\}$. Note que isso leva tempo polinomial no tamanho de D . Essa é a transformação mostrada na Figura 28.3. Resta provar que $\langle D \rangle$ é instância **sim** para Caminho Hamiltoniano se e somente se $\langle D' \rangle$ é instância **sim** para Ciclo Hamiltoniano.

Para provar a ida, suponha primeiro que $\langle D \rangle$ é **sim** para Caminho Hamiltoniano. Isso significa que existe um certificado positivo para essa instância, isto é, que existe um caminho $C = (v_1, v_2, \dots, v_n)$ que é hamiltoniano em D . Mas então $C' = (v_1, v_2, \dots, v_n, x, v_1)$ é um ciclo hamiltoniano em D' , ou seja, um certificado positivo. Logo, $\langle D' \rangle$ é **sim** para Ciclo Hamiltoniano.

Para provar a volta, suponha agora que $\langle D' \rangle$ é **sim** para Ciclo Hamiltoniano. Isso significa que existe um ciclo $C = (v_1, \dots, v_i, x, v_{i+1}, \dots, v_n, v_1)$ que é hamiltoniano em D' . Mas então $C' = (v_{i+1}, \dots, v_n, v_1, \dots, v_i)$ é um caminho hamiltoniano em D . Logo, $\langle D \rangle$ é **sim** para Caminho Hamiltoniano.

Note que reduzimos aqui em apenas uma direção: do Caminho Hamiltoniano para o Ciclo Hamiltoniano. É muito comum haver confusão neste ponto, achando que a redução na ordem inversa também foi garantida, já que provamos duas coisas acima. No entanto, é importante entender a diferença: precisamos mostrar duas coisas (a ida e a volta de uma frase “se e somente se”) porque a definição da redução exige isso. Isso está totalmente não relacionado com a redução do segundo problema para o primeiro (se fossemos provar a redução do lado inverso, também precisaríamos provar o “se e somente se” desta nova redução).

Exemplo 2. Considere os dois problemas a seguir, que são as versões de decisão dos problemas Corte de Barras e Mochila Inteira.

Problema 28.15: *BARRA*

Entrada: $\langle n, p, k \rangle$, onde n é um inteiro positivo indicando o tamanho da barra, sendo que cada tamanho i de barra, com $1 \leq i \leq p$, tem um preço p_i de venda, e k é um inteiro.

Pergunta: É possível cortar uma barra de tamanho n e vender os pedaços obtendo lucro pelo menos k ?

Problema 28.16: *MOCHILA*

Entrada: $\langle I, n, w, v, W, V \rangle$, onde $I = \{1, 2, \dots, n\}$ é um conjunto de n itens sendo que cada $i \in I$ tem um peso w_i e um valor v_i associados, W é a capacidade total da mochila e V é um valor.

Saída: 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 $\langle I, m, w, v, W, V \rangle$ 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 $I = \{1, \dots, m\}$;

- $W = n$;
- $V = k$.

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 $\langle I, m, w, v, W, V \rangle$ é **sim** para MOCHILA.

Para a ida, suponha que $\langle n, p, k \rangle$ é **sim** para BARRA. Então existem pedaços (c_1, c_2, \dots, 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 $\langle I, m, w, v, W, V \rangle$ é **sim** para MOCHILA.

Para a volta, suponha agora que $\langle I, m, w, v, W, V \rangle$ 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 $(c_1, c_2, \dots, c_{|S|})$ 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 MOCHILA novamente e o novo problema dado a seguir.

Problema 28.17: SUBSETSUM

Entrada: $\langle A, n, B \rangle$, onde $A = \{s_1, \dots, s_n\}$ é um conjunto que contém n inteiros e B é um inteiro.

Saída: 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 $\langle I, m, w, v, W, V \rangle$ 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 $I = \{1, \dots, n\}$;
- $W = B$;
- $V = B$.

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 $\langle I, m, w, v, W, V \rangle$ é **sim** para MOCHILA.

para a ida, 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 $\langle I, m, w, v, W, V \rangle$ é **sim** para MOCHILA.

Para a volta, suponha que $\langle I, m, w, v, W, V \rangle$ 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_j 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 reduz para B em tempo polinomial, 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.

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.

Considere agora o problema a seguir, um dos clássicos em computação.

Problema 29.2: *TSP*

Entrada: $\langle D, w, k \rangle$, onde D é um digrafo completo, w é uma função de peso nos arcos de D e k é um número.

Pergunta: Existe um ciclo hamiltoniano em D de custo no máximo k ?

TSP é uma sigla para *Travelling Salesman problema*, nome em inglês de um famoso problema 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 provavelmente 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

Entrada: Um algoritmo e uma instância.

Pergunta: A execução do algoritmo dado sobre a instância dada 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: um certificado para eles pode ser facilmente reconhecido. 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 em tempo polinomial. 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**;
2. para toda instância I_T que é **não**, qualquer conjunto de dados D faz $\mathcal{A}(I_T, D)$ devolver **não**;
3. \mathcal{A} executa em tempo polinomial no tamanho de I_T .

O conjunto de dados D que satisfaz o primeiro item 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 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 .

Vejam os seguintes outros problemas que pertencem à classe NP. Dado um grafo G ,

dizemos que um conjunto $S \subseteq V(G)$ de vértices é uma **clique** se há aresta entre todo par de vértices em S . Formalmente, para todo par $u, v \in S$ vale que $uv \in E(G)$. 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 .

Problema 29.6: CLIQUE

Entrada: $\langle G, k \rangle$, onde G é um grafo e k é um inteiro positivo.

Pergunta: Existe clique $S \subseteq V(G)$ tal que $|S| \geq k$?

O problema CLIQUE está em NP pois, dada $\langle G, k \rangle$ e dado 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

Entrada: $\langle G \rangle$, onde G é um grafo.

Pergunta: G é bipartido?

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 verificador para um problema em P pode simplesmente chamar o algoritmo resolvidor de tal problema (não há necessidade de certificado). Ou seja, claramente temos $P \subseteq NP$. A grande questão é, será que $NP \subseteq P$?

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 US\$ 1.000.000,00 para quem conseguir resolvê-lo¹.

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 $A \in \text{NP}$ e todo outro problema de NP é redutível a A em tempo polinomial.

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. $P = \text{NP}$ 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 de cada um deles para o nosso problema. Essa segunda parte não parece nada possível: esse conjunto não é nem um pouco pequeno! Felizmente, 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, \dots, x_n , i.e., que só recebem valores 0 ou 1 (falso ou verdadeiro), 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 a seguir 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

Entrada: $\langle \phi, x, n, m \rangle$, onde ϕ é uma fórmula 3-CNF contendo literais de n variáveis booleanas x_1, \dots, x_n em m cláusulas.

Saída: Existe uma atribuição de valores a x_1, \dots, x_n tal que ϕ é *satisfatível*, i.e., ϕ tem valor 1?

Por exemplo, a fórmula $(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_4)$ é satisfatível pois basta fazer $x_1 = 1$ e $x_2 = 1$, sem nem importar os valores de x_3 e x_4 .

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 29.14

3-SAT é redutível para CLIQUE em tempo polinomial.

Demonstração. Precisamos exibir um algoritmo eficiente que converte uma instância $\langle \phi, x, n, m \rangle$ do 3-SAT em uma instância $\langle G, k \rangle$ do CLIQUE de tal forma que ϕ é satisfatível se e somente se G contém uma clique com pelo menos k vértices.

Seja então $\langle \phi, x, n, m \rangle$ uma instância do 3-SAT. 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 \bar{x} . Veja a Figura 29.1 para um exemplo de construção de G .

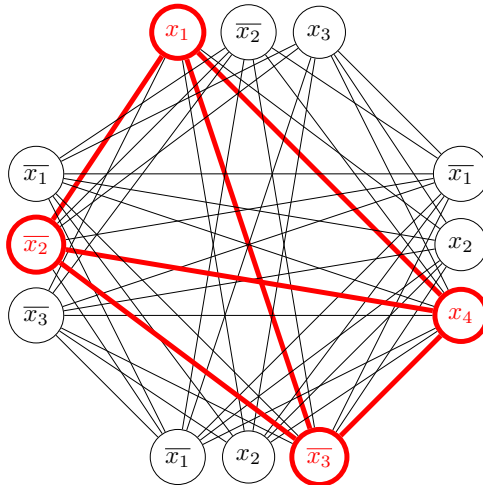


Figura 29.1: Grafo G gerado a partir da instância $\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$ do 3-SAT. A clique em destaque corresponde à solução $x_1 = 1, x_2 = 0, x_3 = 0$ e $x_4 = 1$.

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 pelo menos $k = m$ 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 $x \neq \bar{y}$. 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 é NP-completo.

Considere agora o seguinte problema. Dado um grafo G , um conjunto de vértices $S \subseteq V(G)$ é uma **cobertura por vértices** se toda aresta do grafo tem ao menos um extremo em S . Formalmente, para toda aresta $uv \in E(G)$, temos $u \in S$ ou $v \in S$.

Problema 29.15: VERTEXCOVER

Entrada: $\langle G, k \rangle$, onde G é um grafo e k é um inteiro.

Pergunta: Existe uma cobertura por vértices $S \subseteq V(G)$ tal que $|S| \leq k$?

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 29.17

CLIQUE é redutível para VERTEXCOVER em tempo polinomial.

Demonstração. Precisamos exibir um algoritmo eficiente que converte uma entrada $\langle G, k \rangle$ de CLIQUE em uma entrada $\langle G', k' \rangle$ de VERTEXCOVER 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 $k' = v(G) - k$.

Para a ida, suponha que $\langle G, k \rangle$ é **sim** para CLIQUE, isto é, 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} . Além disso, como $|S| \geq k$, temos $|V(G) \setminus S| = |V(G)| - |S| \leq |V(G)| - k = k'$. Então $\langle \overline{G}, k' \rangle$ é **sim** para VERTEXCOVER.

Para a volta, suponha que $\langle \overline{G}, k' \rangle$ é **sim** para VERTEXCOVER. Isso significa que existe em \overline{G} uma cobertura por vértices S de tamanho no máximo k' , isto é, 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 . Além disso, como $S \leq k' = |V(G)| - k$, temos $|V(G) \setminus S| = |V(G)| - |S| \geq |V(G)| - (|V(G)| - k) = k$. Então $\langle G, k \rangle$ é **sim** para CLIQUE. \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 em tempo polinomial.

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 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, $\text{NP-completo} \subset \text{NP-difícil}$.

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**.

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.