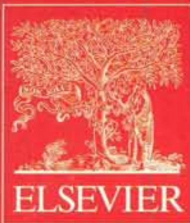


THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

ALGORITMOS

TRADUÇÃO DA 2ª EDIÇÃO AMERICANA

www.therebels.biz

Teoria e Prática

EDITORA CAMPUS

ALGORITMOS

TRADUÇÃO DA 2ª EDIÇÃO AMERICANA

Teoria e Prática

Preencha a **ficha de cadastro** no final deste livro
e receba gratuitamente informações
sobre os lançamentos e as promoções da
Editora Campus.

Consulte também nosso catálogo
completo e últimos lançamentos em
www.campus.com.br

THOMAS H. CORMEN

CHARLES E. LEISERSON

RONALD L. RIVEST

CLIFFORD STEIN

ALGORITMOS

TRADUÇÃO DA 2ª EDIÇÃO AMERICANA

Teoria e Prática

REVISORA TÉCNICA

JUSSARA PIMENTA MATOS

Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da USP e Consultora em Engenharia de Software

TRADUÇÃO

VANDENBERG D. DE SOUZA

4ª Tiragem

Do original
Introduction to algorithms – Second Edition

Tradução autorizada do idioma inglês da edição publicada por The MIT Press

Copyright 2001 by The Massachusetts Institute of Technology

© 2002, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998.
Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Editoração Eletrônica
Estúdio Castellani

Revisão Gráfica
Jane Castellani

Projeto Gráfico
Elsevier Editora Ltda.
A Qualidade da Informação.
Rua Sete de Setembro, 111 – 16º andar
20050-006 Rio de Janeiro RJ Brasil
Telefone: (21) 3970-9300 FAX: (21) 2507-1991
E-mail: *info@elsevier.com.br*
Escritório São Paulo:
Rua Elvira Ferraz, 198
04552-040 Vila Olímpia São Paulo SP
Tel.: (11) 3841-8555

ISBN 85-352-0926-3
(Edição original: ISBN 0-07-013151-1)

CIP-Brasil. Catalogação-na-fonte.
Sindicato Nacional dos Editores de Livros, RJ

A385

Algoritmos : teoria e prática / Thomas H. Cormen... [*et al.*]; tradução da segunda edição [americana] Vandenberg D. de Souza. – Rio de Janeiro : Elsevier, 2002 – 4ª Reimpressão.

Tradução de: Introduction to algorithms
ISBN 85-352-0926-3

1. Programação (Computadores). 2. Algoritmos de computador. I. Cormen. Thomas H.

01-1674 CDD – 005.1
CDU – 004.421

04 05 06 07 7 6 5 4

Sumário

Prefácio .... **XI**

Parte I **Fundamentos**

Introdução **1**

1 **A função dos algoritmos na computação** **3**
1.1 Algoritmos 3
1.2 Algoritmos como uma tecnologia 7

2 **Conceitos básicos** **11**
2.1 Ordenação por inserção 11
2.2 Análise de algoritmos 16
2.3 Projeto de algoritmos 21

3 **Crescimento de funções** **32**
3.1 Notação assintótica 32
3.2 Notações padrão e funções comuns 40

4 **Recorrências** **50**
4.1 O método de substituição 51
4.2 O método de árvore de recursão 54
4.3 O método mestre 59
★ 4.4 Prova do teorema mestre 61

5 **Análise probabilística e algoritmos aleatórios** **73**
5.1 O problema da contratação 73
5.2 Indicadores de variáveis aleatórias 76
5.3 Algoritmos aleatórios 79
★ 5.4 Análise probabilística e usos adicionais de indicadores de variáveis aleatórias 85

Parte II **Ordenação e estatísticas de ordem**

Introdução **99**

6 **Heapsort** **103**
6.1 Heaps 103
6.2 Manutenção da propriedade de heap 105
6.3 A construção de um heap 107
6.4 O algoritmo heapsort 110
6.5 Filas de prioridades 111

7 **Quicksort** **117**
7.1 Descrição do quicksort 117
7.2 O desempenho de quicksort 120
7.3 Uma versão aleatória de quicksort 124
7.4 Análise de quicksort 125

8 Ordenação em tempo linear **133**
8.1 Limites inferiores para ordenação 133
8.2 Ordenação por contagem 135
8.3 Radix sort 137
8.4 Bucket sort 140

9 Medianas e estatísticas de ordem **147**
9.1 Mínimo e máximo 147
9.2 Seleção em tempo esperado linear 149
9.3 Seleção em tempo linear no pior caso 152

Parte III Estruturas de dados

Introdução **159**

10 Estruturas de dados elementares **163**
10.1 Pilhas e filas 163
10.2 Listas ligadas 166
10.3 Implementação de ponteiros e objetos 170
10.4 Representação de árvores enraizadas 173

11 Tabelas hash **179**
11.1 Tabelas de endereço direto 179
11.2 Tabelas hash 181
11.3 Funções hash 185
11.4 Endereçamento aberto 192
★ 11.5 Hash perfeito 198

12 Árvores de pesquisa binária **204**
12.1 O que é uma árvore de pesquisa binária? 204
12.2 Consultas em uma árvore de pesquisa binária 207
12.3 Inserção e eliminação 210
★ 12.4 Árvores de pesquisa binária construídas aleatoriamente 213

13 Árvores vermelho-preto **220**
13.1 Propriedades de árvores vermelho-preto 220
13.2 Rotações 223
13.3 Inserção 225
13.4 Eliminação 231

14 Ampliando estruturas de dados **242**
14.1 Estatísticas de ordem dinâmicas 242
14.2 Como ampliar uma estrutura de dados 247
14.3 Árvores de intervalos 249

Parte IV Técnicas avançadas de projeto e análise

Introdução **257**

15 Programação dinâmica **259**
15.1 Programação de linha de montagem 260
15.2 Multiplicação de cadeias de matrizes 266
15.3 Elementos de programação dinâmica 272
15.3 Subseqüência comum mais longa 281
15.5 Árvores de pesquisa binária ótimas 285

16 Algoritmos gulosos **296**
16.1 Um problema de seleção de atividade 297
16.2 Elementos da estratégia gulosa 303
16.3 Códigos de Huffman 307
★ 16.4 Fundamentos teóricos de métodos gulosos 314
★ 16.5 Um problema de programação de tarefas 319

17 Análise amortizada **324**
17.1 A análise agregada 325
17.2 O método de contabilidade 328
17.3 O método potencial 330
17.4 Tabelas dinâmicas 333

Parte V Estruturas de dados avançadas

Introdução **345**

18 Árvores B **349**
18.1 Definição de árvores B 352
18.2 Operações básicas sobre árvores B 354
18.3 Eliminação de uma chave de uma árvore B 360

19 Heaps binomiais **365**
19.1 Árvores binomiais e heaps binomiais 366
19.2 Operações sobre heaps binomiais 370

20 Heaps de Fibonacci **381**
20.1 Estrutura de heaps de Fibonacci 382
20.2 Operações de heaps intercaláveis 384
20.3 Como decrementar uma chave e eliminar um nó 390
20.4 Como limitar o grau máximo 394

21 Estruturas de dados para conjuntos disjuntos **398**
21.1 Operações de conjuntos disjuntos 398
21.2 Representação de conjuntos disjuntos por listas ligadas 400
21.3 Florestas de conjuntos disjuntos 403
★ 21.4 Análise da união por ordenação com compressão de caminho 406

Parte VI Algoritmos de grafos

Introdução **417**

22 Algoritmos elementares de grafos **419**
22.1 Representações de grafos 419
22.2 Pesquisa primeiro na extensão 422
22.3 Pesquisa primeiro na profundidade 429
22.4 Ordenação topológica 436
22.5 Componentes fortemente conectados 438

23 Árvores de amplitude mínima **445**
23.1 Como aumentar uma árvore de amplitude mínima 446
23.2 Os algoritmos de Kruskal e Prim 450

24 Caminhos mais curtos de única origem **459**
24.1 O algoritmo de Bellman-Ford 465

24.2 Caminhos mais curtos de única origem em grafos acíclicos orientados..... 468
24.3 Algoritmo de Dijkstra 470
24.4 Restrições de diferenças e caminhos mais curtos 475
24.5 Provas de propriedades de caminhos mais curtos..... 480

25 Caminhos mais curtos de todos os pares 490
25.1 Caminhos mais curtos e multiplicação de matrizes..... 492
25.2 O algoritmo de Floyd-Warshall..... 497
25.3 Algoritmo de Johnson para grafos esparsos 503

26 Fluxo máximo 509
26.1 Fluxo em redes 510
26.2 O método de Ford-Fulkerson 515
26.3 Correspondência bipartida máxima..... 525
★ 26.4 Algoritmos de push-relabel 529
★ 26.5 O algoritmo de relabel-to-front..... 538

Parte VII Tópicos selecionados

Introdução 553

27 Redes de ordenação 555
27.1 Redes de comparação 555
27.2 O princípio zero um 559
27.3 Uma rede de ordenação bitônica 561
27.4 Uma rede de intercalação 564
27.5 Uma rede de ordenação..... 566

28 Operações sobre matrizes..... 571
28.1 Propriedades de matrizes..... 571
28.2 Algoritmo de Strassen para multiplicação de matrizes 579
28.3 Resolução de sistemas de equações lineares..... 585
28.4 Inversão de matrizes 597
28.5 Matrizes simétricas definidas como positivas e aproximação de mínimos quadrados 601

29 Programação linear..... 610
29.1 Formas padrão e relaxada 616
29.2 Formulação de problemas como programas lineares 622
29.3 O algoritmo simplex 626
29.4 Dualidade 638
29.5 A solução básica inicial possível..... 643

30 Polinômios e a FFT 651
30.1 Representação de polinômios 653
30.2 A DFT e a FFT 658
30.3 Implementações eficientes de FFT..... 664

31 Algoritmos de teoria dos números 672
31.1 Noções de teoria elementar dos números..... 673
31.2 Máximo divisor comum 678
31.3 Aritmética modular 682
31.4 Resolução de equações lineares modulares 688
31.5 O teorema chinês do resto..... 691
31.6 Potências de um elemento 693

★ 31.7 O sistema de criptografia de chave pública RSA. 697
★ 31.8 Como testar o caráter primo 702
★ 31.9 Fatoração de inteiros. 709

32 Correspondência de cadeias 717
32.1 O algoritmo simples de correspondência de cadeias. 719
32.2 O algoritmo de Rabin-Karp 721
32.3 Correspondência de cadeias com autômatos finitos 725
★ 32.4 O algoritmo de Knuth-Morris-Pratt. 730

33 Geometria computacional. 738
33.1 Propriedades de segmentos de linha 738
33.2 Como determinar se dois segmentos quaisquer se cruzam 743
33.3 Como encontrar a envoltória convexa. 749
33.4 Localização do par de pontos mais próximos. 756

34 Problemas NP-completos 763
34.1 Tempo polinomial. 767
34.2 Verificação de tempo polinomial. 773
34.3 Caráter NP-completo e redutibilidade 776
34.4 Provas do caráter NP-completo 785
34.5 Problemas NP-completos. 791

35 Algoritmos de aproximação 806
35.1 O problema de cobertura de vértices 808
35.2 O problema do caixeiro-viajante 810
35.3 O problema de cobertura de conjuntos 815
35.4 Aleatoriedade e programação linear. 819
35.5 O problema de soma de subconjuntos 823

Parte VIII Apêndice: Fundamentos de matemática

Introdução 833

A Somatórios. 835
A.1 Fórmulas e propriedades de somatórios. 835
A.2 Como limitar somatórios. 838

B Conjuntos e outros temas 845
B.1 Conjuntos 845
B.2 Relações 849
B.3 Funções. 851
B.4 Grafos. 853
B.5 Árvores 856

C Contagem e probabilidade 863
C.1 Contagem 863
C.2 Probabilidade 868
C.3 Variáveis aleatórias discretas 873
C.4 As distribuições geométrica e binomial 878
★ C.5 As extremidades da distribuição binomial 883

Bibliografia. 890

Índice 898

Prefácio

Este livro oferece uma introdução abrangente ao estudo moderno de algoritmos de computador. Ele apresenta muitos algoritmos e os examina com uma profundidade considerável, tornando seu projeto e sua análise acessíveis aos leitores de todos os níveis. Tentamos manter as explicações em um nível elementar sem sacrificar a profundidade do enfoque ou o rigor matemático.

Cada capítulo apresenta um algoritmo, uma técnica de projeto, uma área de aplicação ou um tópico relacionado. Os algoritmos são descritos em linguagem comum e em um "pseudocódigo" projetado para ser legível por qualquer pessoa que tenha um pouco de experiência em programação. O livro contém mais de 230 figuras ilustrando como os algoritmos funcionam. Tendo em vista que enfatizamos a *eficiência* como um critério de projeto, incluímos análises cuidadosas dos tempos de execução de todos os nossos algoritmos.

O texto foi planejado principalmente para uso em graduação e pós-graduação em algoritmos ou estruturas de dados. Pelo fato de discutir questões de engenharia relacionadas ao projeto de algoritmos, bem como aspectos matemáticos, o livro é igualmente adequado para auto-estudo de profissionais técnicos.

Nesta segunda edição, atualizamos o livro inteiro. As mudanças variam da adição de novos capítulos até a reestruturação de frases individuais.

Para o professor

Este livro foi projetado para ser ao mesmo tempo versátil e completo. Você descobrirá sua utilidade para uma variedade de cursos, desde a graduação em estruturas de dados, até a pós-graduação em algoritmos. Pelo fato de fornecermos uma quantidade de material consideravelmente maior do que poderia caber em um curso típico de um período, você deve imaginar o livro como um "bufê" ou "depósito", do qual pode selecionar e extrair o material que melhor atender ao curso que desejar ministrar.

Você achará fácil organizar seu curso apenas em torno dos capítulos de que necessitar. Tornamos os capítulos relativamente autônomos, para que você não precise se preocupar com uma dependência inesperada e desnecessária de um capítulo em relação a outro. Cada capítulo apresenta primeiro o material mais fácil e mostra os assuntos mais difíceis em seguida, com limites de seções assinalando pontos de parada naturais. Na graduação, poderão ser utilizadas apenas as primeiras seções de um capítulo; na pós-graduação, será possível estudar o capítulo inteiro.

Incluímos mais de 920 exercícios e mais de 140 problemas. Cada seção termina com exercícios, e cada capítulo com problemas. Em geral, os exercícios são perguntas curtas que testam o domínio básico do assunto. Alguns são exercícios simples criados para a sala de aula, enquanto outros são mais substanciais e apropriados para uso como dever de casa. Os problemas são estudos de casos mais elaborados que, com freqüência, apresentam novos assuntos; normalmente, eles consistem em várias perguntas que conduzem o aluno por etapas exigidas para chegar a uma solução.

Assinalamos com asteriscos (*) as seções e os exercícios mais adequados para alunos avançados. Uma seção com asteriscos não é necessariamente mais difícil que outra que não tenha asteriscos, mas pode exigir a compreensão de matemática em um nível mais profundo. Da mesma forma, os exercícios com asteriscos podem exigir um conhecimento mais avançado ou criatividade acima da média.

Para o aluno

Esperamos que este livro-texto lhe proporcione uma introdução agradável ao campo dos algoritmos. Tentamos tornar cada algoritmo acessível e interessante. Para ajudá-lo quando você encontrar algoritmos pouco familiares ou difíceis, descrevemos cada um deles passo a passo. Também apresentamos explicações cuidadosas dos conhecimentos matemáticos necessários à compreensão da análise dos algoritmos. Se já tiver alguma familiaridade com um tópico, você perceberá que os capítulos estão organizados de modo que seja possível passar os olhos pelas seções introdutórias e seguir rapidamente para o material mais avançado.

Este é um livro extenso, e sua turma provavelmente só examinará uma parte de seu conteúdo. Porém, procuramos torná-lo útil para você agora como um livro-texto, e também mais tarde em sua carreira, sob a forma de um guia de referência de matemática ou um manual de engenharia.

Quais são os pré-requisitos para a leitura deste livro?

- Você deve ter alguma experiência em programação. Em particular, deve entender procedimentos recursivos e estruturas de dados simples como arranjos e listas ligadas.

- Você deve ter alguma facilidade com a realização de demonstrações por indução matemática. Algumas partes do livro se baseiam no conhecimento de cálculo elementar. Além disso, as Partes I e VIII deste livro ensinam todas as técnicas matemáticas de que você irá necessitar.

Para o profissional

A ampla variedade de tópicos deste livro o torna um excelente manual sobre algoritmos. Como cada capítulo é relativamente autônomo, você pode se concentrar nos tópicos que mais o interessem.

A maioria dos algoritmos que discutimos tem grande utilidade prática. Por essa razão, abordamos conceitos de implementação e outras questões de engenharia. Em geral, oferecemos alternativas práticas para os poucos algoritmos que têm interesse principalmente teórico.

Se desejar implementar algum dos algoritmos, você irá achar a tradução do nosso pseudocódigo em sua linguagem de programação favorita uma tarefa bastante objetiva. O pseudocódigo foi criado para apresentar cada algoritmo de forma clara e sucinta. Conseqüentemente, não nos preocupamos com o tratamento de erros e outras questões ligadas à engenharia de software que exigem suposições específicas sobre o seu ambiente de programação. Tentamos apresentar cada algoritmo de modo simples e direto sem permitir que as idiossincrasias de uma determinada linguagem de programação obscurecessem sua essência.

Para os nossos colegas

Fornecemos uma bibliografia e ponteiros extensivos para a literatura corrente. Cada capítulo termina com um conjunto de "notas do capítulo" que fornecem detalhes e referências históricas. Contudo, as notas dos capítulos não oferecem uma referência completa para o campo inteiro de algoritmos. Porém, pode ser difícil acreditar que, em um livro com este tamanho, muitos algoritmos interessantes não puderam ser incluídos por falta de espaço.

Apesar dos inúmeros pedidos de soluções para problemas e exercícios feitos pelos alunos, escolhemos como norma não fornecer referências para problemas e exercícios, a fim de evitar que os alunos cedessem à tentação de olhar uma solução pronta em lugar de encontrá-la eles mesmos.

Mudanças na segunda edição

O que mudou entre a primeira e a segunda edição deste livro?

Dependendo de como você o encara, o livro pode não ter mudado muito ou ter mudado bastante.

Um rápido exame no sumário mostra que a maior parte dos capítulos e das seções da primeira edição também estão presentes na segunda edição. Removemos dois capítulos e algumas seções, mas adicionamos três novos capítulos e quatro novas seções além desses novos capítulos. Se fosse julgar o escopo das mudanças pelo sumário, você provavelmente concluiria que as mudanças foram modestas.

Porém, as alterações vão muito além do que aparece no sumário. Sem qualquer ordem particular, aqui está um resumo das mudanças mais significativas da segunda edição:

- Cliff Stein foi incluído como co-autor.
- Os erros foram corrigidos. Quantos erros? Vamos dizer apenas que foram vários.
- Há três novos capítulos:
  - O Capítulo 1 discute a função dos algoritmos em informática.
  - O Capítulo 5 abrange a análise probabilística e os algoritmos aleatórios. Como na primeira edição, esses tópicos aparecem em todo o livro.
  - O Capítulo 29 é dedicado à programação linear.
- Dentro dos capítulos que foram trazidos da primeira edição, existem novas seções sobre os seguintes tópicos:
  - Hash perfeito (Seção 11.5).
  - Duas aplicações de programação dinâmica (Seções 15.1 e 15.5).
  - Algoritmos de aproximação que usam técnicas aleatórias e programação linear (Seção 35.4).
- Para permitir que mais algoritmos apareçam mais cedo no livro, três dos capítulos sobre fundamentos matemáticos foram reposicionados, migrando da Parte I para os apêndices, que formam a Parte VIII.
- Há mais de 40 problemas novos e mais de 185 exercícios novos.
- Tornamos explícito o uso de loops invariantes para demonstrar a correção. Nosso primeiro loop invariante aparece no Capítulo 2, e nós os utilizamos algumas dezenas de vezes ao longo do livro.
- Muitas das análises probabilísticas foram reescritas. Em particular, usamos em aproximadamente uma dezena de lugares a técnica de "variáveis indicadoras aleatórias" que simplificam as análises probabilísticas, em especial quando as variáveis aleatórias são dependentes.
- Expandimos e atualizamos as notas dos capítulos e a bibliografia. A bibliografia cresceu mais de 50%, e mencionamos muitos novos resultados algorítmicos que surgiram após a impressão da primeira edição.

Também fizemos as seguintes mudanças:

- O capítulo sobre resolução de recorrências não contém mais o método de iteração. Em vez disso, na Seção 4.2, "promovemos" as árvores de recursão, que passaram a constituir um método por si só. Concluímos que criar árvores de recursão é menos propenso a erros do que fazer a iteração de recorrências. Porém, devemos assinalar que as árvores de recursão são mais bem usadas como um modo de gerar hipóteses que serão então verificadas através do método de substituição.

- O método de particionamento usado para ordenação rápida (Seção 7.1) e no algoritmo de ordem estatística de tempo linear esperado (Seção 9.2) é diferente. Agora usamos o método desenvolvido por Lomuto que, junto com variáveis indicadoras aleatórias, permite uma análise um pouco mais simples. O método da primeira edição, devido a Hoare, é apresentado como um problema no Capítulo 7.
- Modificamos a discussão sobre o hash universal da Seção 11.3.3 de forma que ela se integre à apresentação do hash perfeito.
- Você encontrará na Seção 12.4 uma análise muito mais simples da altura de uma árvore de pesquisa binária construída aleatoriamente.
- As discussões sobre os elementos de programação dinâmica (Seção 15.3) e os elementos de algoritmos gulosos (Seção 16.2) foram significativamente expandidas. A exploração do problema de seleção de atividade, que inicia o capítulo de algoritmos gulosos, ajuda a esclarecer a relação entre programação dinâmica e algoritmos gulosos.
- Substituímos a prova do tempo de execução da estrutura de dados de união de conjuntos disjuntos na Seção 21.4 por uma prova que emprega o método potencial para derivar um limite restrito.
- A prova de correção do algoritmo para componentes fortemente conectados na Seção 22.5 é mais simples, mais clara e mais direta.
- O Capítulo 24, que aborda os caminhos mais curtos de origem única, foi reorganizado com a finalidade de mover as provas das propriedades essenciais para suas próprias seções. A nova organização permite que nos concentremos mais cedo nos algoritmos.
- A Seção 34.5 contém uma visão geral ampliada do caráter NP-completo, e também novas provas de caráter NP-completo para os problemas do ciclo hamiltoniano e da soma de subconjuntos.

Por fim, virtualmente todas as seções foram editadas para corrigir, simplificar e tornar mais claras as explicações e demonstrações.

Agradecimentos da primeira edição

Muitos amigos e colegas contribuíram bastante para a qualidade deste livro. Agradecemos a todos por sua ajuda e suas críticas construtivas.

O laboratório de ciência da computação do MIT proporcionou um ambiente de trabalho ideal. Nossos colegas do grupo de teoria da computação do laboratório foram particularmente incentivadores e tolerantes em relação aos nossos pedidos incessantes de avaliação crítica dos capítulos. Agradecemos especificamente a Baruch Awerbuch, Shafi Goldwasser, Leo Guibas, Tom Leighton, Albert Meyer, David Shmoys e Éva Tardos. Agradecemos a William Ang, Sally Bemus, Ray Hirschfeld e Mark Reinhold por manterem nossas máquinas (equipamentos DEC Microvax, Apple Macintosh e Sun Sparcstation) funcionando e por recompilarem TEX sempre que excedemos um limite de prazo de compilação. A Thinking Machines Corporation forneceu suporte parcial a Charles Leiserson para trabalhar neste livro durante um período de ausência do MIT.

Muitos colegas usaram rascunhos deste texto em cursos realizados em outras faculdades. Eles sugeriram numerosas correções e revisões. Desejamos agradecer em particular a Richard Beigel, Andrew Goldberg, Joan Lucas, Mark Overmars, Alan Sherman e Diane Souvaine.

Muitos assistentes de ensino em nossos cursos apresentaram contribuições significativas para o desenvolvimento deste material. Agradecemos especialmente a Alan Baratz, Bonnie Berger, Aditi Dhagat, Burt Kaliski, Arthur Lent, Andrew Moulton, Marios Papaefthymiou, Cindy Phillips, Mark Reinhold, Phil Rogaway, Flavio Rose, Arie Rudich, Alan Sherman, Cliff Stein, Susmita Sur, Gregory Troxel e Margaret Tuttle.

Uma valiosa assistência técnica adicional foi fornecida por muitas pessoas. Denise Sergent passou muitas horas nas bibliotecas do MIT pesquisando referências bibliográficas. Maria Sensale, a bibliotecária de nossa sala de leitura, foi sempre atenciosa e alegre. O acesso à biblioteca pessoal de Albert Meyer nos poupou muitas horas na biblioteca durante a preparação das anotações dos capítulos. Shlomo Kipnis, Bill Niehaus e David Wilson revisaram os exercícios antigos, desenvolveram novos e escreveram notas sobre suas soluções. Marios Papaefthymiou e Gregory Troxel contribuíram para a indexação. Ao longo dos anos, nossas secretárias Inna Radzihovsky, Denise Sergent, Gayle Sherman e especialmente Be Blackburn proporcionaram apoio infindável a este projeto, e por isso somos gratos a elas.

Muitos erros nos rascunhos iniciais foram relatados por alunos. Agradecemos especificamente a Bobby Blumofe, Bonnie Eisenberg, Raymond Johnson, John Keen, Richard Lethin, Mark Lillibridge, John Pezaris, Steve Ponzio e Margaret Tuttle por sua leitura cuidadosa dos originais.

Nossos colegas também apresentaram resenhas críticas de capítulos específicos, ou informações sobre determinados algoritmos, pelos quais somos gratos. Agradecemos ainda a Bill Aiello, Alok Aggarwal, Eric Bach, Vašek Chvátal, Richard Cole, Johan Hastad, Alex Ishii, David Johnson, Joe Kilian, Dina Kravets, Bruce Maggs, Jim Orlin, James Park, Thane Plambeck, Herschel Safer, Jeff Shallit, Cliff Stein, Gil Strang, Bob Tarjan e Paul Wang. Vários de nossos colegas gentilmente também nos forneceram problemas; agradecemos em particular a Andrew Goldberg, Danny Sleator e Umesh Vazirani.

Foi um prazer trabalhar com The MIT Press e a McGraw-Hill no desenvolvimento deste texto. Agradecemos especialmente a Frank Satlow, Terry Ehling, Larry Cohen e Lorrie Lejeune da The MIT Press, e a David Shapiro da McGraw-Hill por seu encorajamento, apoio e paciência. Somos particularmente agradecidos a Larry Cohen por seu excelente trabalho de edição.

Agradecimentos da segunda edição

Quando pedimos a Julie Sussman, P. P. A., que atuasse como editora técnica da segunda edição, não sabíamos que bom negócio estávamos fazendo. Além de realizar a edição do conteúdo técnico, Julie editou entusiasticamente nosso texto. É humilhante pensar em quantos erros Julie encontrou em nossos esboços antigos; entretanto, considerando a quantidade de erros que ela achou na primeira edição (depois de impressa, infelizmente), isso não surpreende. Além disso, Julie sacrificou sua própria programação para se adaptar à nossa – chegou até a levar capítulos do livro com ela em uma viagem às Ilhas Virgens! Julie, nunca conseguiremos agradecer-lhe o bastante pelo trabalho fantástico que você realizou.

O trabalho para a segunda edição foi feito enquanto os autores eram membros do departamento de ciência da computação no Dartmouth College e no laboratório de ciência da computação do MIT. Ambos foram ambientes de trabalho estimulantes, e agradecemos a nossos colegas por seu apoio.

Amigos e colegas do mundo inteiro ofereceram sugestões e opiniões que orientaram nossa escrita. Muito obrigado a Sanjeev Arora, Javed Aslam, Guy Blelloch, Avrim Blum, Scot Drysdale, Hany Farid, Hal Gabow, Andrew Goldberg, David Johnson, Yanlin Liu, Nicolas Schabanel, Alexander Schrijver, Sasha Shen, David Shmoys, Dan Spielman, Gerald Jay Sussman, Bob Tarjan, Mikkel Thorup e Vijay Vazirani.

Vários professores e colegas nos ensinaram muito sobre algoritmos. Em particular, reconhecemos o esforço e a dedicação de nossos professores Jon L. Bentley, Bob Floyd, Don Knuth, Harold Kuhn, H. T. Kung, Richard Lipton, Arnold Ross, Larry Snyder, Michael I. Shamos, David Shmoys, Ken Steiglitz, Tom Szymanski, Éva Tardos, Bob Tarjan e Jeffrey Ullman.

Reconhecemos o trabalho dos muitos assistentes de ensino dos cursos de algoritmos no MIT e em Dartmouth, inclusive Joseph Adler, Craig Barrack, Bobby Blumofe, Roberto De Prisco, Matteo Frigo, Igal Galperin, David Gupta, Raj D. Iyer, Nabil Kahale, Sarfraz Khurshid, Stavros Kolliopoulos, Alain Leblanc, Yuan Ma, Maria Minkoff, Dimitris Mitsouras, Alin Popescu, Harald Pro-

kop, Sudipta Sengupta, Donna Slonim, Joshua A. Tauber, Sivan Toledo, Elisheva Werner-Reiss, Lea Wittie, Qiang Wu e Michael Zhang.

O suporte de informática foi oferecido por William Ang, Scott Blomquist e Greg Shomo no MIT e por Wayne Cripps, John Konkle e Tim Tregubov em Dartmouth. Agradecemos também a Be Blackburn, Don Dailey, Leigh Deacon, Irene Sebeda e Cheryl Patton Wu no MIT, e a Phyllis Bellmore, Kelly Clark, Delia Mauceli, Sammie Travis, Deb Whiting e Beth Young, de Dartmouth, pelo suporte administrativo. Michael Fromberger, Brian Campbell, Amanda Eubanks, Sung Hoon Kim e Neha Narula também ofereceram apoio oportuno em Dartmouth.

Muitas pessoas fizeram a gentileza de informar sobre erros cometidos na primeira edição. Agradecemos aos leitores mencionados na lista a seguir; cada um deles foi o primeiro a relatar um erro da primeira edição: Len Adleman, Selim Akl, Richard Anderson, Juan Andrade-Cetto, Gregory Bachelis, David Barrington, Paul Beame, Richard Beigel, Margrit Betke, Alex Blakemore, Bobby Blumofe, Alexander Brown, Xavier Cazin, Jack Chan, Richard Chang, Chienhua Chen, Ien Cheng, Hoon Choi, Drue Coles, Christian Collberg, George Collins, Eric Conrad, Peter Csaszar, Paul Dietz, Martin Dietzfelbinger, Scot Drysdale, Patricia Ealy, Yaakov Eisenberg, Michael Ernst, Michael Formann, Nedim Fresko, Hal Gabow, Marek Galecki, Igal Galperin, Luisa Gargano, John Gately, Rosario Genario, Mihaly Gereb, Ronald Greenberg, Jerry Grossman, Stephen Guattery, Alexander Hartemik, Anthony Hill, Thomas Hofmeister, Mathew Hostetter, Yih-Chun Hu, Dick Johnsonbaugh, Marcin Jurdzinki, Nabil Kahale, Fumiaki Kamiya, Anand Kanagala, Mark Kantrowitz, Scott Karlin, Dean Kelley, Sanjay Khanna, Haluk Konuk, Dina Kravets, Jon Kroger, Bradley Kuszmaul, Tim Lambert, Hang Lau, Thomas Lengauer, George Madrid, Bruce Maggs, Victor Miller, Joseph Muskat, Tung Nguyen, Michael Orlov, James Park, Seongbin Park, Ioannis Paschalidis, Boaz Patt-Shamir, Leonid Peshkin, Patricio Poblete, Ira Pohl, Stephen Ponzio, Kjell Post, Todd Poynor, Colin Prepscius, Sholom Rosen, Dale Russell, Hershel Safer, Karen Seidel, Joel Seiferas, Erik Seligman, Stanley Selkow, Jeffrey Shallit, Greg Shannon, Micha Sharir, Sasha Shen, Norman Shulman, Andrew Singer, Daniel Sleator, Bob Sloan, Michael Sofka, Volker Strumpen, Lon Sunshine, Julie Sussman, Asterio Tanaka, Clark Thomborson, Nils Thommesen, Homer Tilton, Martin Tompa, Andrei Toom, Felzer Torsten, Hirendu Vaishnav, M. Veldhorst, Luca Venuti, Jian Wang, Michael Wellman, Gerry Wiener, Ronald Williams, David Wolfe, Jeff Wong, Richard Woundy, Neal Young, Huaiyuan Yu, Tian Yuxing, Joe Zachary, Steve Zhang, Florian Zschoke e Uri Zwick.

Muitos de nossos colegas apresentaram críticas atentas ou preencheram um longo formulário de pesquisa. Agradecemos aos revisores Nancy Amato, Jim Aspnes, Kevin Compton, William Evans, Peter Gacs, Michael Goldwasser, Andrzej Proskurowski, Vijaya Ramachandran e John Reif. Também agradecemos às seguintes pessoas por devolverem a pesquisa: James Abello, Josh Benaloh, Bryan Beresford-Smith, Kenneth Blaha, Hans Bodlaender, Richard Borie, Ted Brown, Domenico Cantone, M. Chen, Robert Cimikowski, William Clocksin, Paul Cull, Rick Decker, Matthew Dickerson, Robert Douglas, Margaret Fleck, Michael Goodrich, Susanne Hambrusch, Dean Hendrix, Richard Johnsonbaugh, Kyriakos Kalorkoti, Srinivas Kankanahalli, Hikyoo Koh, Steven Lindell, Errol Lloyd, Andy Lopez, Dian Rae Lopez, George Lucker, David Maier, Charles Martel, Xiannong Meng, David Mount, Alberto Policriti, Andrzej Proskurowski, Kirk Pruhs, Yves Robert, Guna Seetharaman, Stanley Selkow, Robert Sloan, Charles Steele, Gerard Tel, Murali Varanasi, Bernd Walter e Alden Wright. Gostaríamos que tivesse sido possível implementar todas as suas sugestões. O único problema é que, se isso fosse feito, a segunda edição teria mais ou menos 3.000 páginas!

A segunda edição foi produzida em LaTeX 2$_\varepsilon$. Michael Downes converteu as macros de LaTeX do LaTeX "clássico" para LaTeX 2$_\varepsilon$ e converteu os arquivos de texto para usar essas novas macros. David Jones também forneceu suporte para LaTeX 2$_\varepsilon$. As figuras da segunda edição foram produzidas pelos autores usando o MacDraw Pro. Como na primeira edição, o índice foi compilado com o uso de Windex, um programa em C escrito pelos autores, e a bibliografia foi preparada com a utilização do BibTeX Ayorkor Mills-Tettey e Rob Leathern ajudaram a converter as figuras para MacDraw Pro, e Ayorkor também conferiu nossa bibliografia.

Como também aconteceu na primeira edição, trabalhar com The MIT Press e com a McGraw-Hill foi um prazer. Nossos editores, Bob Prior da MIT Press e Betsy Jones da McGraw-Hill, toleraram nossos gracejos e nos mantiveram no rumo.

Finalmente, agradecemos a nossas esposas – Nicole Cormen, Gail Rivest e Rebecca Ivry – a nossos filhos – Ricky, William e Debby Leiserson, Alex e Christopher Rivest, e Molly, Noah e Benjamin Stein – e a nossos pais – Renee e Perry Cormen, Jean e Mark Leiserson, Shirley e Lloyd Rivest, e Irene e Ira Stein – por seu carinho e apoio durante a elaboração deste livro. O amor, a paciência e o incentivo de nossos familiares tornaram este projeto possível. Dedicamos afetuosamente este livro a eles.

Thomas H. Cormen	*Hanover, New Hampshire*
Charles E. Leiserson	*Cambridge, Massachusetts*
Ronald L. Rivest	*Cambridge, Massachusetts*
Clifford Stein	*Hanover, New Hampshire*

Maio de 2001

Parte I

Fundamentos

Introdução

Esta parte o fará refletir sobre o projeto e a análise de algoritmos. Ela foi planejada para ser uma introdução suave ao modo como especificamos algoritmos, a algumas das estratégias de projeto que usaremos ao longo deste livro e a muitas das idéias fundamentais empregadas na análise de algoritmos. As partes posteriores deste livro serão elaboradas sobre essa base.

O Capítulo 1 é uma visão geral dos algoritmos e de seu lugar nos modernos sistemas de computação. Esse capítulo define o que é um algoritmo e lista alguns exemplos. Ele também apresenta os algoritmos como uma tecnologia, da mesma maneira que um hardware rápido, interfaces gráficas do usuário, sistemas orientados a objetos e redes de computadores.

No Capítulo 2, veremos nossos primeiros algoritmos, que resolvem o problema de ordenar uma seqüência de n números. Eles são escritos em um pseudocódigo que, embora não possa ser traduzido diretamente para qualquer linguagem de programação convencional, transmite a estrutura do algoritmo com clareza suficiente para que um programador competente possa implementá-la na linguagem de sua escolha. Os algoritmos de ordenação que examinaremos são a ordenação por inserção, que utiliza uma abordagem incremental, e a ordenação por intercalação, que usa uma técnica recursiva conhecida como "dividir e conquistar". Embora o tempo exigido por cada uma aumente com o valor n, a taxa de aumento difere entre os dois algoritmos. Determinaremos esses tempos de execução no Capítulo 2 e desenvolveremos uma notação útil para expressá-los.

O Capítulo 3 define com exatidão essa notação, que chamaremos notação assintótica. Ele começa definindo diversas notações assintóticas que utilizaremos para delimitar os tempos de execução dos algoritmos acima e/ou abaixo. O restante do Capítulo 3 é principalmente uma apresentação da notação matemática. Seu propósito maior é o de assegurar que o uso que você fará da notação corresponderá à utilização deste livro, em vez de ensinar-lhe novos conceitos matemáticos.

O Capítulo 4 mergulha mais profundamente no método de dividir e conquistar introduzido no Capítulo 2. Em particular, o Capítulo 4 contém métodos para solução de recorrências que são úteis para descrever os tempos de execução de algoritmos recursivos. Uma técnica eficiente é o "método mestre", que pode ser usado para resolver recorrências que surgem dos algoritmos de dividir e conquistar. Grande parte do Capítulo 4 é dedicada a demonstrar a correção do método mestre, embora essa demonstração possa ser ignorada sem problemas.

O Capítulo 5 introduz a análise probabilística e os algoritmos aleatórios. Em geral, usaremos a análise probabilística para determinar o tempo de execução de um algoritmo nos casos em que, devido à presença de uma distribuição de probabilidades inerente, o tempo de execução pode diferir em diversas entradas do mesmo tamanho. Em alguns casos, vamos supor que as entradas obedecem a uma distribuição de probabilidades conhecida, e assim calcularemos o tempo de execução médio sobre todas as entradas possíveis. Em outros casos, a distribuição de probabilidades não vem das entradas, mas sim das escolhas aleatórias feitas durante o curso do algoritmo. Um algoritmo cujo comportamento é determinado não apenas por sua entrada, mas também pelos valores produzidos por um gerador de números aleatórios, é um algoritmo aleatório. Podemos usar algoritmos aleatórios para impor uma distribuição de probabilidade sobre as entradas – assegurando assim que nenhuma entrada específica sempre causará um fraco desempenho – ou mesmo para limitar a taxa de erros de algoritmos que têm permissão para produzir resultados incorretos de forma limitada.

Os Apêndices A, B e C contêm outros materiais matemáticos que você irá considerar úteis à medida que ler este livro. É provável que você tenha visto grande parte do material dos apêndices antes de encontrá-los neste livro (embora as convenções específicas de notação que usamos possam diferir em alguns casos daquelas que você já viu), e assim você deve considerar os apêndices um guia de referência. Por outro lado, é provável que você ainda não tenha visto a maior parte do material contido na Parte I. Todos os capítulos da Parte I e os apêndices foram escritos com um "toque" de tutorial.

Capítulo 1

A função dos algoritmos na computação

O que são algoritmos? Por que o estudo dos algoritmos vale a pena? Qual é a função dos algoritmos em relação a outras tecnologias usadas em computadores? Neste capítulo, responderemos a essas perguntas.

1.1 Algoritmos

Informalmente, um ***algoritmo*** é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como ***entrada*** e produz algum valor ou conjunto de valores como ***saída***. Portanto, um algoritmo é uma seqüência de passos computacionais que transformam a entrada na saída.

Também podemos visualizar um algoritmo como uma ferramenta para resolver um ***problema computacional*** bem especificado. O enunciado do problema especifica em termos gerais o relacionamento entre a entrada e a saída desejada. O algoritmo descreve um procedimento computacional específico para se alcançar esse relacionamento da entrada com a saída.

Por exemplo, poderia ser necessário ordenar uma seqüência de números em ordem não decrescente. Esse problema surge com freqüência na prática e oferece um solo fértil para a introdução de muitas técnicas de projeto padrão e ferramentas de análise. Vejamos como definir formalmente o ***problema de ordenação***:

Entrada: Uma seqüência de n números $\langle a_1, a_2, ..., a_n \rangle$.

Saída: Uma permutação (reordenação) $\langle a'_1, a'_2, ..., a'_n \rangle$ da seqüência de entrada, tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Dada uma seqüência de entrada como $\langle 31,41,59,26,41,58 \rangle$, um algoritmo de ordenação retorna como saída a seqüência $\langle 26,31,41,41,58,59 \rangle$. Uma seqüência de entrada como essa é chamada uma ***instância*** do problema de ordenação. Em geral, uma ***instância de um problema*** consiste na entrada (que satisfaz a quaisquer restrições impostas no enunciado do problema) necessária para se calcular uma solução para o problema.

A ordenação é uma operação fundamental em ciência da computação (muitos programas a utilizam como uma etapa intermediária) e, como resultado, um grande número de bons algoritmos de ordenação tem sido desenvolvido. O melhor algoritmo para uma determinada aplicação

depende – entre outros fatores – do número de itens a serem ordenados, da extensão em que os itens já estão ordenados de algum modo, de possíveis restrições sobre os valores de itens e da espécie de dispositivo de armazenamento a ser usado: memória principal, discos ou fitas.

Um algoritmo é dito ***correto*** se, para cada instância de entrada, ele pára com a saída correta. Dizemos que um algoritmo correto ***resolve*** o problema computacional dado. Um algoritmo incorreto pode não parar em algumas instâncias de entrada, ou então pode parar com outra resposta que não a desejada. Ao contrário do que se poderia esperar, às vezes os algoritmos incorretos podem ser úteis, se sua taxa de erros pode ser controlada. Veremos um exemplo desse fato no Capítulo 31, quando estudarmos algoritmos para localizar grandes números primos. Porém, em situações comuns, iremos nos concentrar apenas no estudo de algoritmos corretos.

Um algoritmo pode ser especificado em linguagem comum, como um programa de computador, ou mesmo como um projeto de hardware. O único requisito é que a especificação deve fornecer uma descrição precisa do procedimento computacional a ser seguido.

Que tipos de problemas são resolvidos por algoritmos?

A ordenação não é de modo algum o único problema computacional para o qual foram desenvolvidos algoritmos. (Você provavelmente suspeitou disso quando viu o tamanho deste livro.) As aplicações práticas de algoritmos são onipresentes e incluem os exemplos a seguir:

- O Projeto Genoma Humano tem como objetivos identificar todos os 100.000 genes do DNA humano, determinar as seqüências dos 3 bilhões de pares de bases químicas que constituem o DNA humano, armazenar essas informações em bancos de dados e desenvolver ferramentas para análise de dados. Cada uma dessas etapas exige algoritmos sofisticados. Embora as soluções para os vários problemas envolvidos estejam além do escopo deste livro, idéias de muitos capítulos do livro são usadas na solução desses problemas biológicos, permitindo assim aos cientistas realizarem tarefas ao mesmo tempo que utilizam com eficiência os recursos. As economias são de tempo, tanto humano quanto da máquina, e de dinheiro, à medida que mais informações podem ser extraídas de técnicas de laboratório.

- A Internet permite que pessoas espalhadas por todo o mundo acessem e obtenham com rapidez grandes quantidades de informações. Para isso, são empregados algoritmos inteligentes com a finalidade de gerenciar e manipular esse grande volume de dados. Os exemplos de problemas que devem ser resolvidos incluem a localização de boas rotas pelas quais os dados viajarão (as técnicas para resolver tais problemas são apresentadas no Capítulo 24) e o uso de um mecanismo de pesquisa para encontrar com rapidez páginas em que residem informações específicas (as técnicas relacionadas estão nos Capítulos 11 e 32).

- O comércio eletrônico permite que mercadorias e serviços sejam negociados e trocados eletronicamente. A capacidade de manter privativas informações como números de cartão de crédito, senhas e extratos bancários é essencial para a ampla utilização do comércio eletrônico. A criptografia de chave pública e as assinaturas digitais (estudadas no Capítulo 31) estão entre as tecnologias centrais utilizadas e se baseiam em algoritmos numéricos e na teoria dos números.

- Na indústria e em outras instalações comerciais, muitas vezes é importante alocar recursos escassos da maneira mais benéfica. Uma empresa petrolífera talvez deseje saber onde localizar seus poços para tornar máximo o lucro esperado. Um candidato à presidência da República talvez queira determinar onde gastar dinheiro em publicidade de campanha com a finalidade de ampliar as chances de vencer a eleição. Uma empresa de transporte aéreo pode designar as tripulações para os vôos da forma menos dispendiosa possível, certificando-se de que cada vôo será atendido e que as regulamentações do governo relativas à escala

das tripulações serão obedecidas. Um provedor de serviços da Internet talvez queira definir onde instalar recursos adicionais para servir de modo mais eficiente a seus clientes. Todos esses são exemplos de problemas que podem ser resolvidos com o uso da programação linear, que estudaremos no Capítulo 29.

Embora alguns dos detalhes desses exemplos estejam além do escopo deste livro, forneceremos técnicas básicas que se aplicam a esses problemas e a essas áreas de problemas. Também mostraremos neste livro como resolver muitos problemas concretos, inclusive os seguintes:

- Temos um mapa rodoviário no qual a distância entre cada par de interseções adjacentes é marcada, e nossa meta é determinar a menor rota de uma interseção até outra. O número de rotas possíveis pode ser enorme, ainda que sejam descartadas as rotas que cruzam sobre si mesmas. Como escolher qual de todas as rotas possíveis é a mais curta? Aqui, modelamos o mapa rodoviário (que é ele próprio um modelo das estradas reais) como um grafo (o que veremos no Capítulo 10 e no Apêndice B) e desejamos encontrar o caminho mais curto de um vértice até outro no grafo. Veremos como resolver esse problema de forma eficiente no Capítulo 24.

- Temos uma seqüência $\langle A_1, A_2, ..., A_n \rangle$ de n matrizes e desejamos determinar seu produto $A_1 A_2 ... A_n$. Como a multiplicação de matrizes é associativa, existem várias ordens de multiplicação válidas. Por exemplo, se $n = 4$, podemos executar as multiplicações de matrizes como se o produto estivesse entre parênteses em qualquer das seguintes ordens: $(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1(A_2A_3))A_4)$ ou $(((A_1A_2)A_3)A_4)$. Se essas matrizes forem todas quadradas (e portanto tiverem o mesmo tamanho), a ordem de multiplicação não afetará o tempo de duração das multiplicações de matrizes. Porém, se essas matrizes forem de tamanhos diferentes (ainda que seus tamanhos sejam compatíveis para a multiplicação de matrizes), então a ordem de multiplicação pode fazer uma diferença muito grande. O número de ordens de multiplicação possíveis é exponencial em n, e assim tentar todas as ordens possíveis pode levar um tempo muito longo. Veremos no Capítulo 15 como usar uma técnica geral conhecida como programação dinâmica para resolver esse problema de modo muito mais eficiente.

- Temos uma equação $ax \equiv b \pmod{n}$, onde a, b e n são inteiros, e desejamos encontrar todos os inteiros x, módulo n, que satisfazem à equação. Pode haver zero, uma ou mais de uma solução. Podemos simplesmente experimentar $x = 0, 1, ..., n - 1$ em ordem, mas o Capítulo 31 mostra um método mais eficiente.

- Temos n pontos no plano e desejamos encontrar a envoltória convexa desses pontos. A envoltória convexa é o menor polígono convexo que contém os pontos. Intuitivamente, podemos imaginar que cada ponto é representado por um prego fixado a uma tábua. A envoltória convexa seria representada por um elástico apertado que cercasse todos os pregos. Cada prego pelo qual o elástico passa é um vértice da envoltória convexa. (Veja um exemplo na Figura 33.6.) Quaisquer dos 2^n subconjuntos dos pontos poderiam ser os vértices da envoltória convexa. Saber quais pontos são vértices da envoltória convexa não é suficiente, pois também precisamos conhecer a ordem em que eles aparecem. Portanto, há muitas escolhas para os vértices da envoltória convexa. O Capítulo 33 apresenta dois bons métodos para se encontrar a envoltória convexa.

Essas listas estão longe de esgotar os exemplos (como você novamente já deve ter imaginado pelo peso deste livro), mas exibem duas características comuns a muitos algoritmos interessantes.

1. Existem muitas soluções candidatas, a maioria das quais não é aquilo que desejamos. Encontrar a solução que queremos pode representar um desafio.

2. Existem aplicações práticas. Dos problemas da lista anterior, o caminho mais curto fornece os exemplos mais fáceis. Uma empresa de transportes que utiliza caminhões ou vagões ferroviários tem interesse financeiro em encontrar os caminhos mais curtos em uma rede ferroviária ou rodoviária, porque percursos menores resultam em menor trabalho e menor consumo de combustível. Ou então, um nó de roteamento na Internet pode precisar encontrar o caminho mais curto através da rede, a fim de rotear uma mensagem com rapidez.

Estruturas de dados

Este livro também contém várias estruturas de dados. Uma ***estrutura de dados*** é um meio para armazenar e organizar dados com o objetivo de facilitar o acesso e as modificações. Nenhuma estrutura de dados única funciona bem para todos os propósitos, e assim é importante conhecer os pontos fortes e as limitações de várias delas.

Técnica

Embora possa usar este livro como um "livro de receitas" para algoritmos, algum dia você poderá encontrar um problema para o qual não seja possível descobrir prontamente um algoritmo publicado (muitos dos exercícios e problemas deste livro, por exemplo!). Este livro lhe ensinará técnicas de projeto e análise de algoritmos, de forma que você possa desenvolver algoritmos por conta própria, mostrar que eles fornecem a resposta correta e entender sua eficiência.

Problemas difíceis

A maior parte deste livro trata de algoritmos eficientes. Nossa medida habitual de eficiência é a velocidade, isto é, quanto tempo um algoritmo demora para produzir seu resultado. Porém, existem alguns problemas para os quais não se conhece nenhuma solução eficiente. O Capítulo 34 estuda um subconjunto interessante desses problemas, conhecidos como NP-completos.

Por que os problemas NP-completos são interessantes? Primeiro, embora ainda não tenha sido encontrado nenhum algoritmo eficiente para um problema NP-completo, ninguém jamais provou que não é possível existir um algoritmo eficiente para esse fim. Em outras palavras, desconhecemos se existem ou não algoritmos eficientes para problemas NP-completos. Em segundo lugar, o conjunto de problemas NP-completos tem a propriedade notável de que, se existe um algoritmo eficiente para qualquer um deles, então existem algoritmos eficientes para todos. Esse relacionamento entre os problemas NP-completos torna a falta de soluções eficientes ainda mais torturante. Em terceiro lugar, vários problemas NP-completos são semelhantes, mas não idênticos, a problemas para os quais conhecemos algoritmos eficientes. Uma pequena mudança no enunciado do problema pode provocar uma grande alteração na eficiência do melhor algoritmo conhecido.

É valioso conhecer os problemas NP-completos, porque alguns deles surgem com freqüência surpreendente em aplicações reais. Se for chamado a produzir um algoritmo eficiente para um problema NP-completo, é provável que você perca muito tempo em uma busca infrutífera. Por outro lado, se conseguir mostrar que o problema é NP-completo, você poderá em vez disso dedicar seu tempo ao desenvolvimento de um algoritmo eficiente que ofereça uma solução boa, embora não seja a melhor possível.

Como um exemplo concreto, considere uma empresa de transporte por caminhão com um armazém central. A cada dia, ela carrega o caminhão no armazém e o envia a diversos locais para efetuar entregas. No final do dia, o caminhão tem de estar de volta ao armazém, a fim de ser preparado para receber a carga do dia seguinte. Para reduzir custos, a empresa deve selecionar uma ordem de paradas de entrega que represente a menor distância total a ser percorrida pelo caminhão. Esse problema é o famoso "problema do caixeiro-viajante", e é NP-completo. Ele não

tem nenhum algoritmo eficiente conhecido. Contudo, sob certas hipóteses, há algoritmos eficientes que fornecem uma distância total não muito acima da menor possível. O Capítulo 35 discute esses "algoritmos de aproximação".

Exercícios

1.1-1
Forneça um exemplo real no qual apareça um dos problemas computacionais a seguir: ordenação, determinação da melhor ordem para multiplicação de matrizes ou localização da envoltória convexa.

1.1-2
Além da velocidade, que outras medidas de eficiência poderiam ser usadas em uma configuração real?

1.1-3
Selecione uma estrutura de dados que você já tenha visto antes e discuta seus pontos fortes e suas limitações.

1.1-4
Em que aspectos os problemas do caminho mais curto e do caixeiro-viajante anteriores são semelhantes? Em que aspectos eles são diferentes?

1.1-5
Mostre um problema real no qual apenas a melhor solução servirá. Em seguida, apresente um problema em que baste uma solução que seja "aproximadamente" a melhor.

1.2 Algoritmos como uma tecnologia

Suponha que os computadores fossem infinitamente rápidos e que a memória do computador fosse livre. Você teria alguma razão para estudar algoritmos? A resposta é sim, se não por outra razão, pelo menos porque você ainda gostaria de demonstrar que o método da sua solução termina, e o faz com a resposta correta.

Se os computadores fossem infinitamente rápidos, qualquer método correto para resolver um problema serviria. É provável que você quisesse que sua implementação estivesse dentro dos limites da boa prática de engenharia de software (isto é, que ela fosse bem documentada e projetada) mas, com maior freqüência, você utilizaria o método que fosse o mais fácil de implementar.

É claro que os computadores podem ser rápidos, mas não são infinitamente rápidos. A memória pode ser de baixo custo, mas não é gratuita. Assim, o tempo de computação é um recurso limitado, bem como o espaço na memória. Esses recursos devem ser usados de forma sensata, e algoritmos eficientes em termos de tempo ou espaço ajudarão você a usá-los.

Eficiência

Algoritmos criados para resolver o mesmo problema muitas vezes diferem de forma drástica em sua eficiência. Essas diferenças podem ser muito mais significativas que as diferenças relativas a hardware e software.

Veremos no Capítulo 2, como exemplo, dois algoritmos para ordenação. O primeiro, conhecido como ***ordenação por inserção***, leva um tempo aproximadamente igual a c_1n^2 para ordenar n itens, onde c_1 é uma constante que não depende de n. Isto é, ela demora um tempo aproximadamente proporcional a n^2. O segundo, de ***ordenação por intercalação***, leva um tempo aproximadamente igual a $c_2n \lg n$, onde $\lg n$ representa $\log_2 n$ e c_2 é outra constante que tam-

bém não depende de n. A ordenação por inserção normalmente tem um fator constante menor que a ordenação por intercalação; e assim, $c_1 < c_2$. Veremos que os fatores constantes podem ser muito menos significativos no tempo de execução que a dependência do tamanho da entrada n. Onde a ordenação por intercalação tem um fator $\lg n$ em seu tempo de execução, a ordenação por inserção tem um fator n, que é muito maior. Embora a ordenação por inserção em geral seja mais rápida que a ordenação por intercalação para pequenos tamanhos de entradas, uma vez que o tamanho da entrada n se tornar grande o suficiente, a vantagem da ordenação por intercalação de $\lg n$ contra n compensará com sobras a diferença em fatores constantes. Independente do quanto c_1 seja menor que c_2, sempre haverá um ponto de passagem além do qual a ordenação por intercalação será mais rápida.

Como um exemplo concreto, vamos comparar um computador mais rápido (computador A) que executa a ordenação por inserção com um computador mais lento (computador B) que executa a ordenação por intercalação. Cada um deles deve ordenar um arranjo de um milhão de números.

Suponha que o computador A execute um bilhão de instruções por segundo e o computador B execute apenas dez milhões de instruções por segundo; assim, o computador A será 100 vezes mais rápido que o computador B em capacidade bruta de computação. Para tornar a diferença ainda mais drástica, suponha que o programador mais astucioso do mundo codifique a ordenação por inserção em linguagem de máquina para o computador A, e que o código resultante exija $2n^2$ instruções para ordenar n números. (Aqui, $c_1 = 2$.) Por outro lado, a ordenação por intercalação é programada para o computador B por um programador médio que utiliza uma linguagem de alto nível com um compilador ineficiente, com o código resultante totalizando $50n \lg n$ instruções (de forma que $c_2 = 50$). Para ordenar um milhão de números, o computador A demora

$$\frac{2 \cdot (10^6)^2 \text{ instruções}}{10^9 \text{ instruções/segundo}} = 2000 \text{ segundos} ,$$

enquanto o computador B demora

$$\frac{50 \cdot 10^6 \ \lg 10^6 \ \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 100 \text{ segundos} .$$

Usando um algoritmo cujo tempo de execução cresce mais lentamente, até mesmo com um compilador fraco, o computador B funciona 20 vezes mais rápido que o computador A! A vantagem da ordenação por intercalação é ainda mais pronunciada quando ordenamos dez milhões de números: onde a ordenação por inserção demora aproximadamente 2,3 dias, a ordenação por intercalação demora menos de 20 minutos. Em geral, à medida que o tamanho do problema aumenta, também aumenta a vantagem relativa da ordenação por intercalação.

Algoritmos e outras tecnologias

O exemplo anterior mostra que os algoritmos, como o hardware de computadores, constituem uma ***tecnologia***. O desempenho total do sistema depende da escolha de algoritmos eficientes tanto quanto da escolha de hardware rápido. Da mesma maneira que estão havendo rápidos avanços em outras tecnologias computacionais, eles também estão sendo obtidos em algoritmos.

Você poderia indagar se os algoritmos são verdadeiramente tão importantes nos computadores contemporâneos em comparação com outras tecnologias avançadas, como:

- Hardware com altas taxas de clock, pipelines e arquiteturas superescalares.
- Interfaces gráficas do usuário (GUIs) intuitivas e fáceis de usar.

- Sistemas orientados a objetos.
- Redes locais e remotas.

A resposta é sim. Embora existam algumas aplicações que não exigem explicitamente conteúdo algorítmico no nível da aplicação (por exemplo, algumas aplicações simples baseadas na Web), a maioria também requer um certo grau de conteúdo algorítmico por si só. Por exemplo, considere um serviço da Web que determina como viajar de um local para outro. (Havia diversos serviços desse tipo no momento em que este livro foi escrito.) Sua implementação dependeria de hardware rápido, de uma interface gráfica do usuário, de redes remotas e também, possivelmente, de orientação a objetos. Contudo, ele também exigiria algoritmos para certas operações, como localização de rotas (talvez empregando um algoritmo de caminho mais curto), interpretação de mapas e interpolação de endereços.

Além disso, até mesmo uma aplicação que não exige conteúdo algorítmico no nível da aplicação depende muito de algoritmos. Será que a aplicação depende de hardware rápido? O projeto de hardware utilizou algoritmos. A aplicação depende de interfaces gráficas do usuário? O projeto de qualquer GUI depende de algoritmos. A aplicação depende de rede? O roteamento em redes depende muito de algoritmos. A aplicação foi escrita em uma linguagem diferente do código de máquina? Então, ela foi processada por um compilador, um interpretador ou um assembler, e todos fazem uso extensivo de algoritmos. Os algoritmos formam o núcleo da maioria das tecnologias usadas em computadores contemporâneos.

Além disso, com a capacidade cada vez maior dos computadores, nós os utilizamos para resolver problemas maiores do que nunca. Como vimos na comparação anterior entre ordenação por inserção e ordenação por intercalação, em problemas de tamanhos maiores, as diferenças na eficiência dos algoritmos se tornam particularmente importantes.

Uma sólida base de conhecimento e técnica de algoritmos é uma característica que separa os programadores verdadeiramente qualificados dos novatos. Com a moderna tecnologia computacional, você pode executar algumas tarefas sem saber muito sobre algoritmos; porém, com uma boa base em algoritmos, é possível fazer muito, muito mais.

Exercícios

1.2-1

Forneça um exemplo de aplicação que exige conteúdo algorítmico no nível da aplicação e discuta a função dos algoritmos envolvidos.

1.2-2

Vamos supor que estamos comparando implementações de ordenação por inserção e ordenação por intercalação na mesma máquina. Para entradas de tamanho n, a ordenação por inserção é executada em $8n^2$ etapas, enquanto a ordenação por intercalação é executada em $64n \lg n$ etapas. Para que valores de n a ordenação por inserção supera a ordenação por intercalação?

1.2-3

Qual é o menor valor de n tal que um algoritmo cujo tempo de execução é $100n^2$ funciona mais rápido que um algoritmo cujo tempo de execução é 2^n na mesma máquina?

Problemas

1-1 Comparação entre tempos de execução

Para cada função $f(n)$ e cada tempo t na tabela a seguir, determine o maior tamanho n de um problema que pode ser resolvido no tempo t, supondo-se que o algoritmo para resolver o problema demore $f(n)$ microssegundos.

	1 segundo	1 minuto	1 hora	1 dia	1 mês	1 ano	1 século
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Notas do capítulo

Existem muitos textos excelentes sobre o tópico geral de algoritmos, inclusive os de Aho, Hopcroft e Ullman [5, 6], Baase e Van Gelder [26], Brassard e Bratley [46, 47], Goodrich e Tamassia [128], Horowitz, Sahni e Rajasekaran [158], Kingston [179], Knuth [182, 183, 185], Kozen [193], Manber [210], Mehlhorn [217, 218, 219], Purdom e Brown [252], Reingold, Nievergelt e Deo [257], Sedgewick [269], Skiena [280] e Wilf [315]. Alguns dos aspectos mais práticos do projeto de algoritmos são discutidos por Bentley [39, 40] e Gonnet [126]. Pesquisas sobre o campo dos algoritmos também podem ser encontradas no Handbook of Theoretical Computer Science, Volume A [302] e no CRC Handbook on Algorithms and Theory of Computation [24]. Avaliações dos algoritmos usados em biologia computacional podem ser encontradas em livros-texto de Gusfield [136], Pevzner [240], Setubal e Medinas [272], e Waterman [309].

Capítulo 2

Conceitos básicos

Este capítulo tem o objetivo de familiarizá-lo com a estrutura que usaremos em todo o livro para refletir sobre o projeto e a análise de algoritmos. Ele é autônomo, mas inclui diversas referências ao material que será apresentado nos Capítulos 3 e 4. (E também contém diversos somatórios, que o Apêndice A mostra como resolver.)

Começaremos examinando o problema do algoritmo de ordenação por inserção para resolver o problema de ordenação apresentado no Capítulo 1. Definiremos um "pseudocódigo" que deverá ser familiar aos leitores que tenham estudado programação de computadores, e o empregaremos com a finalidade de mostrar como serão especificados nossos algoritmos. Tendo especificado o algoritmo, demonstraremos então que ele efetua a ordenação corretamente e analisaremos seu tempo de execução. A análise introduzirá uma notação centrada no modo como o tempo aumenta com o número de itens a serem ordenados. Seguindo nossa discussão da ordenação por inserção, introduziremos a abordagem de dividir e conquistar para o projeto de algoritmos e a utilizaremos com a finalidade de desenvolver um algoritmo chamado ordenação por intercalação. Terminaremos com uma análise do tempo de execução da ordenação por intercalação.

2.1 Ordenação por inserção

Nosso primeiro algoritmo, o de ordenação por inserção, resolve o ***problema de ordenação*** introduzido no Capítulo 1:

Entrada: Uma seqüência de n números $\langle a_1, a_2, ..., a_n \rangle$.

Saída: Uma permutação (reordenação) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da seqüência de entrada, tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Os números que desejamos ordenar também são conhecidos como ***chaves***.

Neste livro, descreveremos tipicamente algoritmos como programas escritos em um ***pseudocódigo*** muito semelhante em vários aspectos a C, Pascal ou Java. Se já conhece qualquer dessas linguagens, você deverá ter pouca dificuldade para ler nossos algoritmos. O que separa o pseudocódigo do código "real" é que, no pseudocódigo, empregamos qualquer método expressivo para especificar de forma mais clara e concisa um dado algoritmo. Às vezes, o método mais claro é a linguagem comum; assim, não se surpreenda se encontrar uma frase ou sentença em nosso idioma (ou em inglês) embutida no interior de uma seção de código "real". Outra diferen-

ça entre o pseudocódigo e o código real é que o pseudocódigo em geral não se relaciona com questões de engenharia de software. As questões de abstração de dados, modularidade e tratamento de erros são freqüentemente ignoradas, com a finalidade de transmitir a essência do algoritmo de modo mais conciso.

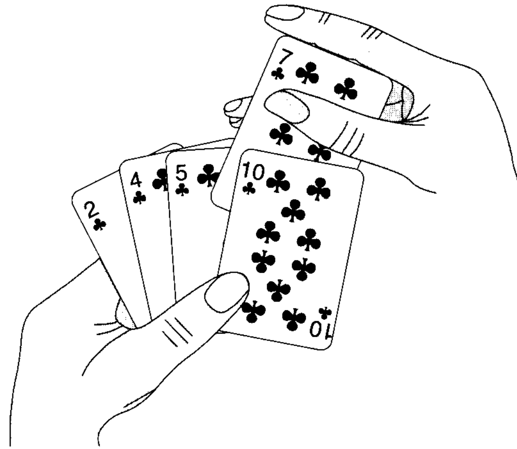


FIGURA 2.1 Ordenando cartas com o uso da ordenação por inserção

Começaremos com a ***ordenação por inserção***, um algoritmo eficiente para ordenar um número pequeno de elementos. A ordenação por inserção funciona da maneira como muitas pessoas ordenam as cartas em um jogo de bridge ou pôquer. Iniciaremos com a mão esquerda vazia e as cartas viradas com a face para baixo na mesa. Em seguida, removeremos uma carta de cada vez da mesa, inserindo-a na posição correta na mão esquerda. Para encontrar a posição correta de uma carta, vamos compará-la a cada uma das cartas que já estão na mão, da direita para a esquerda, como ilustra a Figura 2.1. Em cada instante, as cartas seguras na mão esquerda são ordenadas; essas cartas eram originalmente as cartas superiores da pilha na mesa.

Nosso pseudocódigo para ordenação por inserção é apresentado como um procedimento chamado INSERTION-SORT, que toma como parâmetro um arranjo $A[1 \,..\, n]$ contendo uma seqüência de comprimento n que deverá ser ordenada. (No código, o número n de elementos em A é denotado por *comprimento*$[A]$.) Os números da entrada são ***ordenados no local***: os números são reorganizados dentro do arranjo A, com no máximo um número constante deles armazenado fora do arranjo em qualquer instante. O arranjo de entrada A conterá a seqüência de saída ordenada quando INSERTION-SORT terminar.

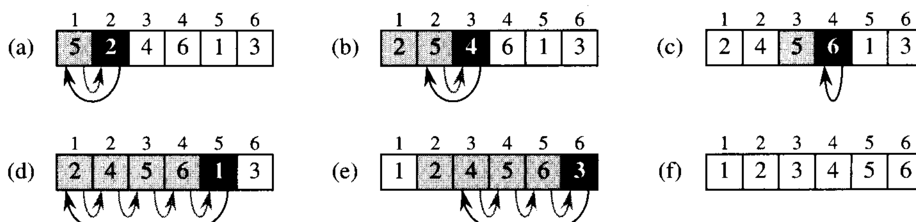


FIGURA 2.2 A operação de INSERTION-SORT sobre o arranjo $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Os índices do arranjo aparecem acima dos retângulos e os valores armazenados nas posições do arranjo aparecem dentro dos retângulos. **(a)–(e)** As iterações do loop **for** das linhas 1 a 8. Em cada iteração, o retângulo preto contém a chave obtida de $A[j]$, que é comparada aos valores contidos nos retângulos sombreados à sua esquerda, no teste da linha 5. Setas sombreadas mostram os valores do arranjo deslocados uma posição à direita na linha 6, e setas pretas indicam para onde a chave é deslocada na linha 8. **(f)** O arranjo ordenado final

```
INSERTION-SORT(A)
1 for j ← 2 to comprimento[A]
2     do chave ← A[j]
3        ▷ Inserir A[j] na seqüência ordenada A[1..j – 1].
4        i ← j – 1
5        while i > 0 e A[i] > chave
6           do A[i + 1] ← A[i]
7              i ← i – 1
8        A[i + 1] ← chave
```

Loops invariantes e a correção da ordenação por inserção

A Figura 2.2 mostra como esse algoritmo funciona para $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. O índice j indica a "carta atual" sendo inserida na mão. No início de cada iteração do loop **for** "externo", indexado por j, o subarranjo que consiste nos elementos $A[1 \, .. \, j-1]$ constitui a mão atualmente ordenada, e os elementos $A[j+1 \, .. \, n]$ correspondem à pilha de cartas ainda na mesa. Na verdade, os elementos $A[1 \, .. \, j-1]$ são os elementos que estavam ***originalmente*** nas posições de 1 a $j-1$, mas agora em seqüência ordenada. Enunciamos formalmente essas propriedades de $A[1 \, .. \, j-1]$ como um ***loop invariante***:

> No começo de cada iteração do loop ***for*** das linhas 1 a 8, o subarranjo $A[1 \, .. \, j-1]$ consiste nos elementos contidos originalmente em $A[1 \, .. \, j-1]$, mas em seqüência ordenada.

Usamos loops invariantes para nos ajudar a entender por que um algoritmo é correto. Devemos mostrar três detalhes sobre um loop invariante:

Inicialização: Ele é verdadeiro antes da primeira iteração do loop.

Manutenção: Se for verdadeiro antes de uma iteração do loop, ele permanecerá verdadeiro antes da próxima iteração.

Término: Quando o loop termina, o invariante nos fornece uma propriedade útil que ajuda a mostrar que o algoritmo é correto.

Quando as duas primeiras propriedades são válidas, o loop invariante é verdadeiro antes de toda iteração do loop. Note a semelhança em relação à indução matemática; nesta última, para provar que uma propriedade é válida, você demonstra um caso básico e uma etapa indutiva. Aqui, mostrar que o invariante é válido antes da primeira iteração é equivalente ao caso básico, e mostrar que o invariante é válido de uma iteração para outra equivale à etapa indutiva.

A terceira propriedade talvez seja a mais importante, pois estamos usando o loop invariante para mostrar a correção. Ela também difere do uso habitual da indução matemática, em que a etapa indutiva é usada indefinidamente; aqui, paramos a "indução" quando o loop termina.

Vamos ver como essas propriedades são válidas para ordenação por inserção:

Inicialização: Começamos mostrando que o loop invariante é válido antes da primeira iteração do loop, quando $j = 2$.[1] Então, o subarranjo $A[1 \, .. \, j-1]$ consiste apenas no único elemento $A[1]$, que é de fato o elemento original em $A[1]$. Além disso, esse subarranjo é ordenado (de forma trivial, é claro), e isso mostra que o loop invariante é válido antes da primeira iteração do loop.

---

[1] Quando o loop é um loop **for**, o momento em que verificamos o loop invariante imediatamente antes da primeira iteração ocorre logo após a atribuição inicial à variável do contador de loop e imediatamente antes do primeiro teste no cabeçalho do loop. No caso de INSERTION-SORT, esse instante ocorre após a atribuição de 2 à variável j, mas antes do primeiro teste para verificar se $j \leq comprimento[A]$.

Manutenção: Em seguida, examinamos a segunda propriedade: a demonstração de que cada iteração mantém o loop invariante. Informalmente, o corpo do loop **for** exterior funciona deslocando-se $A[j-1]$, $A[j-2]$, $A[j-3]$ e daí por diante uma posição à direita, até ser encontrada a posição adequada para $A[j]$ (linhas 4 a 7), e nesse ponto o valor de $A[j]$ é inserido (linha 8). Um tratamento mais formal da segunda propriedade nos obrigaria a estabelecer e mostrar um loop invariante para o loop **while** "interno". Porém, nesse momento, preferimos não nos prender a tal formalismo, e assim contamos com nossa análise informal para mostrar que a segunda propriedade é válida para o loop exterior.

Término: Finalmente, examinamos o que ocorre quando o loop termina. No caso da ordenação por inserção, o loop **for** externo termina quando j excede n, isto é, quando $j = n + 1$. Substituindo j por $n + 1$ no enunciado do loop invariante, temos que o subarranjo $A[1 \,..\, n]$ consiste nos elementos originalmente contidos em $A[1 \,..\, n]$, mas em seqüência ordenada. Contudo, o subarranjo $A[1 \,..\, n]$ é o arranjo inteiro! Desse modo, o arranjo inteiro é ordenado, o que significa que o algoritmo é correto.

Empregaremos esse método de loops invariantes para mostrar a correção mais adiante neste capítulo e também em outros capítulos.

Convenções de pseudocódigo

Utilizaremos as convenções a seguir em nosso pseudocódigo.

1. O recuo (ou endentação) indica uma estrutura de blocos. Por exemplo, o corpo do loop **for** que começa na linha 1 consiste nas linhas 2 a 8, e o corpo do loop **while*** que começa na linha 5 contém as linhas 6 e 7, mas não a linha 8. Nosso estilo de recuo também se aplica a instruções **if-then-else**. O uso de recuo em lugar de indicadores convencionais de estrutura de blocos, como instruções **begin** e **end**, reduz bastante a desordem ao mesmo tempo que preserva, ou até mesmo aumenta, a clareza.[2]

2. As construções de loops **while**, **for** e **repeat** e as construções condicionais **if**, **then** e **else** têm interpretações semelhantes às que apresentam em Pascal.[3] Porém, existe uma diferença sutil com respeito a loops **for**: em Pascal, o valor da variável do contador de loop é indefinido na saída do loop mas, neste livro, o contador do loop retém seu valor após a saída do loop. Desse modo, logo depois de um loop **for**, o valor do contador de loop é o valor que primeiro excedeu o limite do loop **for**. Usamos essa propriedade em nosso argumento de correção para a ordenação por inserção. O cabeçalho do loop **for** na linha 1 é **for** $j \leftarrow 2$ **to** $comprimento[A]$, e assim, quando esse loop termina, $j = comprimento[A] + 1$ (ou, de forma equivalente, $j = n + 1$, pois $n = comprimento[A]$).

3. O símbolo "▷" indica que o restante da linha é um comentário.

4. Uma atribuição múltipla da forma $i \leftarrow j \leftarrow e$ atribui às variáveis i e j o valor da expressão e; ela deve ser tratada como equivalente à atribuição $j \leftarrow e$ seguida pela atribuição $i \leftarrow j$.

5. Variáveis (como i, j e *chave*) são locais para o procedimento dado. Não usaremos variáveis globais sem indicação explícita.

---

[2]Em linguagens de programação reais, em geral não é aconselhável usar o recuo sozinho para indicar a estrutura de blocos, pois os níveis de recuo são difíceis de descobrir quando o código se estende por várias páginas.

[3]A maioria das linguagens estruturadas em blocos tem construções equivalentes, embora a sintaxe exata possa diferir da sintaxe de Pascal.

* Manteremos na edição brasileira os nomes das instruções e dos comandos de programação (destacados em negrito) em inglês, bem como os títulos dos algoritmos, conforme a edição original americana, a fim de facilitar o processo de conversão para uma linguagem de programação qualquer, caso necessário. Por exemplo, usaremos **while** em vez de **enquanto**. (N.T.)

6. Elementos de arranjos são acessados especificando-se o nome do arranjo seguido pelo índice entre colchetes. Por exemplo, $A[i]$ indica o i-ésimo elemento do arranjo A. A notação ".." é usada para indicar um intervalo de valores dentro de um arranjo. Desse modo, $A[1 \,..\, j]$ indica o subarranjo de A que consiste nos j elementos $A[1], A[2], ..., A[j]$.

7. Dados compostos estão organizados tipicamente em ***objetos***, os quais são constituídos por ***atributos*** ou ***campos***. Um determinado campo é acessado usando-se o nome do campo seguido pelo nome de seu objeto entre colchetes. Por exemplo, tratamos um arranjo como um objeto com o atributo *comprimento* indicando quantos elementos ele contém. Para especificar o número de elementos em um arranjo A, escrevemos *comprimento*$[A]$. Embora sejam utilizados colchetes para indexação de arranjos e atributos de objetos, normalmente ficará claro a partir do contexto qual a interpretação pretendida.

   Uma variável que representa um arranjo ou um objeto é tratada como um ponteiro para os dados que representam o arranjo ou objeto. Para todos os campos f de um objeto x, a definição de $y \leftarrow x$ causa $f[y] = f[x]$. Além disso, se definirmos agora $f[x] \leftarrow 3$, então daí em diante não apenas $f[x] = 3$, mas também $f[y] = 3$. Em outras palavras, x e y apontarão para ("serão") o mesmo objeto após a atribuição $y \leftarrow x$.

   Às vezes, um ponteiro não fará referência a nenhum objeto. Nesse caso, daremos a ele o valor especial NIL.

8. Parâmetros são passados a um procedimento ***por valor***: o procedimento chamado recebe sua própria cópia dos parâmetros e, se ele atribuir um valor a um parâmetro, a mudança *não* será vista pela rotina de chamada. Quando objetos são passados, o ponteiro para os dados que representam o objeto é copiado, mas os campos do objeto não o são. Por exemplo, se x é um parâmetro de um procedimento chamado, a atribuição $x \leftarrow y$ dentro do procedimento chamado não será visível para o procedimento de chamada. Contudo, a atribuição $f[x] \leftarrow 3$ será visível.

9. Os operadores booleanos "e" e "ou" são operadores de ***curto-circuito***. Isto é, quando avaliamos a expressão "x e y", avaliamos primeiro x. Se x for avaliado como FALSE, então a expressão inteira não poderá ser avaliada como TRUE, e assim não avaliaremos y. Se, por outro lado, x for avaliado como TRUE, teremos de avaliar y para determinar o valor da expressão inteira. De forma semelhante, na expressão "x ou y", avaliamos a expressão y somente se x for avaliado como FALSE. Os operadores de curto-circuito nos permitem escrever expressões booleanas como "x ... NIL e $f[x] = y$" sem nos preocuparmos com o que acontece ao tentarmos avaliar $f[x]$ quando x é NIL.

Exercícios

2.1-1

Usando a Figura 2.2 como modelo, ilustre a operação de INSERTION-SORT no arranjo $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

Reescreva o procedimento INSERTION-SORT para ordenar em ordem não crescente, em vez da ordem não decrescente.

2.1-3

Considere o ***problema de pesquisa***:

Entrada: Uma seqüência de n números $A = \langle a_1, a_2, ..., a_n \rangle$ e um valor v.

Saída: Um índice i tal que $v = A[i]$ ou o valor especial NIL, se v não aparecer em A.

Escreva o pseudocódigo para ***pesquisa linear***, que faça a varredura da seqüência, procurando por v. Usando um loop invariante, prove que seu algoritmo é correto. Certifique-se de que seu loop invariante satisfaz às três propriedades necessárias.

2.1-4

Considere o problema de somar dois inteiros binários de n bits, armazenados em dois arranjos de n elementos A e B. A soma dos dois inteiros deve ser armazenada em forma binária em um arranjo de $(n + 1)$ elementos C. Enuncie o problema de modo formal e escreva o pseudocódigo para somar os dois inteiros.

2.2 Análise de algoritmos

Analisar um algoritmo significa prever os recursos de que o algoritmo necessitará. Ocasionalmente, recursos como memória, largura de banda de comunicação ou hardware de computador são a principal preocupação, mas com freqüência é o tempo de computação que desejamos medir. Em geral, pela análise de vários algoritmos candidatos para um problema, pode-se identificar facilmente um algoritmo mais eficiente. Essa análise pode indicar mais de um candidato viável, mas vários algoritmos de qualidade inferior em geral são descartados no processo.

Antes de podermos analisar um algoritmo, devemos ter um modelo da tecnologia de implementação que será usada, inclusive um modelo dos recursos dessa tecnologia e seus custos. Na maior parte deste livro, faremos a suposição de um modelo de computação genérico com um único processador, a ***RAM*** (***random-access machine*** – máquina de acesso aleatório), como nossa tecnologia de implementação e entenderemos que nossos algoritmos serão implementados sob a forma de programas de computador. No modelo de RAM, as instruções são executadas uma após outra, sem operações concorrentes (ou simultâneas). Porém, em capítulos posteriores teremos oportunidade de investigar modelos de hardware digital.

No sentido estrito, devemos definir com precisão as instruções do modelo de RAM e seus custos. Porém, isso seria tedioso e daria pouco percepção do projeto e da análise de algoritmos. Também devemos ter cuidado para não abusar do modelo de RAM. Por exemplo, e se uma RAM tivesse uma instrução de ordenação? Então, poderíamos ordenar com apenas uma instrução. Tal RAM seria irreal, pois os computadores reais não têm tais instruções. Portanto, nosso guia é o modo como os computadores reais são projetados. O modelo de RAM contém instruções comumente encontradas em computadores reais: instruções aritméticas (soma, subtração, multiplicação, divisão, resto, piso, teto), de movimentação de dados (carregar, armazenar, copiar) e de controle (desvio condicional e incondicional, chamada e retorno de sub-rotinas). Cada uma dessas instruções demora um período constante.

Os tipos de dados no modelo de RAM são inteiros e de ponto flutuante. Embora normalmente não nos preocupemos com a precisão neste livro, em algumas aplicações a precisão é crucial. Também supomos um limite sobre o tamanho de cada palavra de dados. Por exemplo, ao trabalharmos com entradas de tamanho n, em geral supomos que os inteiros são representados por $c \lg n$ bits para alguma constante $c \geq 1$. Exigimos $c \geq 1$ para que cada palavra possa conter o valor de n, permitindo-nos indexar os elementos de entradas individuais, e limitamos c a uma constante para que o tamanho da palavra não cresça arbitrariamente. (Se o tamanho da palavra pudesse crescer arbitrariamente, seria possível armazenar enormes quantidades de dados em uma única palavra e operar sobre toda ela em tempo constante – claramente um cenário impraticável.)

Computadores reais contêm instruções não listadas anteriormente, e tais instruções representam uma área cinza no modelo de RAM. Por exemplo, a exponenciação é uma instrução de tempo constante? No caso geral, não; são necessárias várias instruções para calcular x^y quando x e y são números reais. Porém, em situações restritas, a exponenciação é uma operação de tempo constante. Muitos computadores têm uma instrução "deslocar à esquerda" que desloca em tempo constante os bits de um inteiro k posições à esquerda. Na maioria dos computadores, deslocar os bits de um inteiro uma posição à esquerda é equivalente a efetuar a multiplicação por 2.

Deslocar os bits k posições à esquerda é equivalente a multiplicar por 2^k. Portanto, tais computadores podem calcular 2^k em uma única instrução de tempo constante, deslocando o inteiro 1 k posições à esquerda, desde que k não seja maior que o número de bits em uma palavra de computador. Procuraremos evitar essas áreas cinza no modelo de RAM, mas trataremos a computação de 2^k como uma operação de tempo constante quando k for um inteiro positivo suficientemente pequeno.

No modelo de RAM, não tentaremos modelar a hierarquia da memória que é comum em computadores contemporâneos. Isto é, não modelaremos caches ou memória virtual (que é implementada com maior freqüência com paginação por demanda). Vários modelos computacionais tentam levar em conta os efeitos da hierarquia de memória, que às vezes são significativos em programas reais de máquinas reais. Alguns problemas neste livro examinam os efeitos da hierarquia de memória mas, em sua maioria, as análises neste livro não irão considerá-los.

Os modelos que incluem a hierarquia de memória são bem mais complexos que o modelo de RAM, de forma que pode ser difícil utilizá-los. Além disso, as análises do modelo de RAM em geral permitem previsões excelentes do desempenho em máquinas reais.

Até mesmo a análise de um algoritmo simples no modelo de RAM pode ser um desafio. As ferramentas matemáticas exigidas podem incluir análise combinatória, teoria das probabilidades, destreza em álgebra e a capacidade de identificar os termos mais significativos em uma fórmula. Tendo em vista que o comportamento de um algoritmo pode ser diferente para cada entrada possível, precisamos de um meio para resumir esse comportamento em fórmulas simples, de fácil compreensão.

Embora normalmente selecionemos apenas um único modelo de máquina para analisar um determinado algoritmo, ainda estaremos diante de muitas opções na hora de decidir como expressar nossa análise. Um objetivo imediato é encontrar um meio de expressão que seja simples de escrever e manipular, que mostre as características importantes de requisitos de recurso de um algoritmo e que suprima os detalhes tediosos.

Análise da ordenação por inserção

O tempo despendido pelo procedimento INSERTION-SORT depende da entrada: a ordenação de mil números demora mais que a ordenação de três números. Além disso, INSERTION-SORT pode demorar períodos diferentes para ordenar duas seqüências de entrada do mesmo tamanho, dependendo do quanto elas já estejam ordenadas. Em geral, o tempo de duração de um algoritmo cresce com o tamanho da entrada; assim, é tradicional descrever o tempo de execução de um programa como uma função do tamanho de sua entrada. Para isso, precisamos definir os termos "tempo de execução" e "tamanho da entrada" com mais cuidado.

A melhor noção de ***tamanho da entrada*** depende do problema que está sendo estudado. No caso de muitos problemas, como a ordenação ou o cálculo de transformações discretas de Fourier, a medida mais natural é o *número de itens na entrada* – por exemplo, o tamanho do arranjo n para ordenação. Para muitos outros problemas, como a multiplicação de dois inteiros, a melhor medida do tamanho da entrada é o *número total de bits* necessários para representar a entrada em notação binária comum. Às vezes, é mais apropriado descrever o tamanho da entrada com dois números em lugar de um. Por exemplo, se a entrada para um algoritmo é um grafo, o tamanho da entrada pode ser descrito pelos números de vértices e arestas no grafo. Indicaremos qual medida de tamanho da entrada está sendo usada com cada problema que estudarmos.

O ***tempo de execução*** de um algoritmo em uma determinada entrada é o número de operações primitivas ou "etapas" executadas. É conveniente definir a noção de etapa (ou passo) de forma que ela seja tão independente da máquina quanto possível. Por enquanto, vamos adotar a visão a seguir. Um período constante de tempo é exigido para executar cada linha do nosso pseudocódigo. Uma única linha pode demorar um período diferente de outra linha, mas vamos considerar que cada execução da i-ésima linha leva um tempo c_i, onde c_i é uma constante. Esse pon-

to de vista está de acordo com o modelo de RAM, e também reflete o modo como o pseudocódigo seria implementado na maioria dos computadores reais.[4]

Na discussão a seguir, nossa expressão para o tempo de execução de INSERTION-SORT evoluirá desde uma fórmula confusa que utiliza todos os custos da instrução c_i até uma notação mais simples, mais concisa e mais facilmente manipulada. Essa notação mais simples também facilitará a tarefa de descobrir se um algoritmo é mais eficiente que outro.

Começaremos apresentando o procedimento INSERTION-SORT com o "custo" de tempo de cada instrução e o número de vezes que cada instrução é executada. Para cada $j = 2, 3, \ldots, n$, onde $n = comprimento[A]$, seja t_j o número de vezes que o teste do loop **while** na linha 5 é executado para esse valor de j. Quando um loop **for** ou **while** termina da maneira usual (isto é, devido ao teste no cabeçalho loop), o teste é executado uma vez além do corpo do loop. Supomos que comentários não são instruções executáveis e, portanto, não demandam nenhum tempo.

INSERTION-SORT(A)	*custo*	*vezes*
1 **for** $j \leftarrow 2$ **to** *comprimento*$[A]$	c_1	n
2 **do** *chave* $\leftarrow A[j]$	c_2	$n-1$
3 ▷ Inserir $A[j]$ na seqüência ordenada $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ e $A[i] >$ *chave*	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow$ *chave*	c_8	$n-1$

O tempo de execução do algoritmo é a soma dos tempos de execução para cada instrução executada; uma instrução que demanda c_i passos para ser executada e é executada n vezes, contribuirá com $c_i n$ para o tempo de execução total.[5] Para calcular $T(n)$, o tempo de execução de INSERTION-SORT, somamos os produtos das *colunas custo* e *vezes*, obtendo

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) - c_8(n-1).$$

Mesmo para entradas de um dado tamanho, o tempo de execução de um algoritmo pode depender de *qual* entrada desse tamanho é dada. Por exemplo, em INSERTION-SORT, o melhor caso ocorre se o arranjo já está ordenado. Para cada $j = 2, 3, \ldots, n$, descobrimos então que $A[i] \leq$ *chave* na linha 5 quando i tem seu valor inicial $j - 1$. Portanto, $t_j = 1$ para $j = 2, 3, \ldots, n$, e o tempo de execução do melhor caso é

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

[4]Há algumas sutilezas aqui. As etapas computacionais que especificamos em linguagem comum freqüentemente são variantes de um procedimento que exige mais que apenas uma quantidade constante de tempo. Por exemplo, mais adiante neste livro, poderíamos dizer "ordene os pontos pela coordenada x" que, como veremos, demora mais que uma quantidade constante de tempo. Além disso, observe que uma instrução que chama uma sub-rotina demora um tempo constante, embora a sub-rotina, uma vez invocada, possa durar mais. Ou seja, separamos o processo de ***chamar*** a sub-rotina – passar parâmetros a ela etc. – do processo de ***executar*** a sub-rotina.

[5]Essa característica não se mantém necessariamente para um recurso como a memória. Uma instrução que referencia m palavras de memória e é executada n vezes não consome necessariamente mn palavras de memória no total.

Esse tempo de execução pode ser expresso como $an + b$ para *constantes* a e b que dependem dos custos de instrução c_i; assim, ele é uma ***função linear*** de n.

Se o arranjo estiver ordenado em ordem inversa – ou seja, em ordem decrescente –, resulta o pior caso. Devemos comparar cada elemento $A[j]$ com cada elemento do subarranjo ordenado inteiro, $A[1 \,..\, j-1]$, e então $t_j = j$ para 2, 3, ..., n. Observando que

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

e

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

(veremos no Apêndice A como resolver esses somatórios), descobrimos que, no pior caso, o tempo de execução de INSERTION-SORT é

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n-1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Esse tempo de execução no pior caso pode ser expresso como $an^2 + bn + c$ para constantes a, b e c que, mais uma vez, dependem dos custos de instrução c_i; portanto, ele é uma ***função quadrática*** de n.

Em geral, como na ordenação por inserção, o tempo de execução de um algoritmo é fixo para uma determinada entrada, embora em capítulos posteriores devamos ver alguns algoritmos "aleatórios" interessantes, cujo comportamento pode variar até mesmo para uma entrada fixa.

Análise do pior caso e do caso médio

Em nossa análise da ordenação por inserção, observamos tanto o melhor caso, no qual o arranjo de entrada já estava ordenado, quanto o pior caso, no qual o arranjo de entrada estava ordenado em ordem inversa. Porém, no restante deste livro, em geral nos concentraremos apenas na descoberta do ***tempo de execução do pior caso***; ou seja, o tempo de execução mais longo para *qualquer* entrada de tamanho n. Apresentaremos três razões para essa orientação.

- O tempo de execução do pior caso de um algoritmo é um limite superior sobre o tempo de execução para qualquer entrada. Conhecê-lo nos dá uma garantia de que o algoritmo nunca irá demorar mais tempo. Não precisamos fazer nenhuma suposição baseada em fatos sobre o tempo de execução, e temos a esperança de que ele nunca seja muito pior.
- Para alguns algoritmos, o pior caso ocorre com bastante freqüência. Por exemplo, na pesquisa de um banco de dados em busca de um determinado fragmento de informação, o pior caso do algoritmo de pesquisa ocorrerá freqüentemente quando a informação não estiver presente no banco de dados. Em algumas aplicações de pesquisa, a busca de informações ausentes pode ser freqüente.

- Muitas vezes, o "caso médio" é quase tão ruim quanto o pior caso. Suponha que sejam escolhidos aleatoriamente n números e que se aplique a eles a ordenação por inserção. Quanto tempo irá demorar para se descobrir o lugar no subarranjo $A[1..j-1]$ em que se deve inserir o elemento $A[j]$? Em média, metade dos elementos em $A[1..j-1]$ são menores que $A[j]$, e metade dos elementos são maiores. Assim, em média, verificamos que metade do subarranjo $A[1..j-1]$, e então $t_j = j/2$. Se desenvolvermos o tempo de execução do caso médio resultante, ele será uma função quadrática do tamanho da entrada, exatamente como o tempo de execução do pior caso.

Em alguns casos particulares, estaremos interessados no tempo de execução do ***caso médio*** ou ***esperado*** de um algoritmo. Contudo, um problema na realização de uma análise do caso médio é que pode não ser aparente, o que constitui uma entrada "média" para um determinado problema. Freqüentemente, iremos supor que todas as entradas de um dado tamanho são igualmente prováveis. Na prática, é possível que essa suposição seja violada, mas às vezes podemos utilizar um ***algoritmo aleatório***, que efetua escolhas ao acaso, a fim de permitir uma análise probabilística.

Ordem de crescimento

Usamos algumas abstrações simplificadoras para facilitar nossa análise do procedimento INSERTION-SORT. Primeiro, ignoramos o custo real de cada instrução, usando as constantes c_i para representar esses custos. Em seguida, observamos que até mesmo essas constantes nos oferecem mais detalhes do que realmente necessitamos: o tempo de execução do pior caso é $an^2 + bn + c$ para constantes a, b e c que dependem dos custos de instrução c_i. Desse modo, ignoramos não apenas os custos reais de instrução, mas também os custos abstratos c_i.

Agora, faremos mais uma abstração simplificadora. É a ***taxa de crescimento***, ou ***ordem de crescimento***, do tempo de execução que realmente nos interessa. Assim, consideramos apenas o termo inicial de uma fórmula (por exemplo, an^2), pois os termos de mais baixa ordem são relativamente insignificantes para grandes valores de n. Também ignoramos o coeficiente constante do termo inicial, tendo em vista que fatores constantes são menos significativos que a taxa de crescimento na determinação da eficiência computacional para grandes entradas. Portanto, escrevemos que a ordenação por inserção, por exemplo, tem um tempo de execução do pior caso igual a $\Theta(n^2)$ (lido como "theta de n ao quadrado"). Usaremos informalmente neste capítulo a notação Θ; ela será definida com precisão no Capítulo 3.

Em geral, consideramos um algoritmo mais eficiente que outro se o tempo de execução do seu pior caso apresenta uma ordem de crescimento mais baixa. Essa avaliação pode ser incorreta para entradas pequenas; porém, para entradas suficientemente grandes, um algoritmo $\Theta(n^2)$, por exemplo, será executado mais rapidamente no pior caso que um algoritmo $\Theta(n^3)$.

Exercícios

2.2-1

Expresse a função $n^3/1000 - 100n^2 - 100n + 3$ em termos da notação Θ.

2.2-2

Considere a ordenação de n números armazenados no arranjo A, localizando primeiro o menor elemento de A e permutando esse elemento com o elemento contido em $A[1]$. Em seguida, encontre o segundo menor elemento de A e o troque pelo elemento $A[2]$. Continue dessa maneira para os primeiros $n-1$ elementos de A. Escreva o pseudocódigo para esse algoritmo, conhecido como ***ordenação por seleção***. Que loop invariante esse algoritmo mantém? Por que ele só precisa ser executado para os primeiros $n-1$ elementos, e não para todos os n elementos? Forneça os tempos de execução do melhor caso e do pior caso da ordenação por seleção em notação Θ.

2.2-3
Considere mais uma vez a pesquisa linear (ver Exercício 2.1-3). Quantos elementos da seqüência de entrada precisam ser verificados em média, supondo-se que o elemento que está sendo procurado tenha a mesma probabilidade de ser qualquer elemento no arranjo? E no pior caso? Quais são os tempos de execução do caso médio e do pior caso da pesquisa linear em notação Θ? Justifique suas respostas.

2.2-4
Como podemos modificar praticamente qualquer algoritmo para ter um bom tempo de execução no melhor caso?

2.3 Projeto de algoritmos

Existem muitas maneiras de projetar algoritmos. A ordenação por inserção utiliza uma abordagem ***incremental***: tendo ordenado o subarranjo $A[1 \mathinner{..} j-1]$, inserimos o elemento isolado $A[j]$ em seu lugar apropriado, formando o subarranjo ordenado $A[1 \mathinner{..} j]$.

Nesta seção, examinaremos uma abordagem de projeto alternativa, conhecida como "dividir e conquistar". Usaremos o enfoque de dividir e conquistar para projetar um algoritmo de ordenação cujo tempo de execução do pior caso é muito menor que o da ordenação por inserção. Uma vantagem dos algoritmos de dividir e conquistar é que seus tempos de execução são freqüentemente fáceis de determinar com a utilização de técnicas que serão introduzidas no Capítulo 4.

2.3.1 A abordagem de dividir e conquistar

Muitos algoritmos úteis são ***recursivos*** em sua estrutura: para resolver um dado problema, eles chamam a si mesmos recursivamente uma ou mais vezes para lidar com subproblemas intimamente relacionados. Em geral, esses algoritmos seguem uma abordagem de ***dividir e conquistar***: eles desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

O paradigma de dividir e conquistar envolve três passos em cada nível da recursão:

Dividir o problema em um determinado número de subproblemas.

Conquistar os subproblemas, resolvendo-os recursivamente. Porém, se os tamanhos dos subproblemas forem pequenos o bastante, basta resolver os subproblemas de maneira direta.

Combinar as soluções dadas aos subproblemas, a fim de formar a solução para o problema original.

O algoritmo de ***ordenação por intercalação*** a seguir obedece ao paradigma de dividir e conquistar. Intuitivamente, ele opera do modo ilustrado a seguir.

Dividir: Divide a seqüência de n elementos a serem ordenados em duas subseqüências de $n/2$ elementos cada uma.

Conquistar: Classifica as duas subseqüências recursivamente, utilizando a ordenação por intercalação.

Combinar: Faz a intercalação das duas seqüências ordenadas, de modo a produzir a resposta ordenada.

A recursão "não funciona" quando a seqüência a ser ordenada tem comprimento 1, pois nesse caso não há nenhum trabalho a ser feito, tendo em vista que toda seqüência de comprimento 1 já está ordenada.

A operação chave do algoritmo de ordenação por intercalação é a intercalação de duas seqüências ordenadas, no passo de "combinação". Para executar a intercalação, usamos um procedimento auxiliar MERGE(A, p, q, r), onde A é um arranjo e p, q e r são índices de enumeração dos elementos do arranjo, tais que $p \leq q < r$. O procedimento pressupõe que os subarranjos $A[p .. q]$ e $A[q + 1 .. r]$ estão em seqüência ordenada. Ele os ***intercala*** (ou ***mescla***) para formar um único subarranjo ordenado que substitui o subarranjo atual $A[p .. r]$.

Nosso procedimento MERGE leva o tempo $\Theta(n)$, onde $n = r - p + 1$ é o número de elementos que estão sendo intercalados, e funciona como a seguir. Retornando ao nosso exemplo de motivação do jogo de cartas, vamos supor que temos duas pilhas de cartas com a face para cima sobre uma mesa. Cada pilha está ordenada, com as cartas de menor valor em cima. Desejamos juntar as duas pilhas (fazendo a intercalação) em uma única pilha de saída ordenada, que ficará com a face para baixo na mesa. Nosso passo básico consiste em escolher a menor das duas cartas superiores nas duas pilhas viradas para cima, removê-la de sua pilha (o que irá expor uma nova carta superior) e colocar essa carta com a face voltada para baixo sobre a pilha de saída. Repetimos esse passo até uma pilha de entrada se esvaziar e, nesse momento, simplesmente pegamos a pilha de entrada restante e a colocamos virada para baixo sobre a pilha de saída. Em termos computacionais, cada passo básico demanda um tempo constante, pois estamos verificando apenas duas cartas superiores. Tendo em vista que executamos no máximo n passos básicos, a intercalação demorará um tempo $\Theta(n)$.

O pseudocódigo a seguir implementa a idéia anterior, mas tem uma alteração adicional que evita a necessidade de verificar se uma das pilhas está vazia em cada etapa básica. A idéia é colocar na parte inferior de cada pilha uma carta ***sentinela***, que contém um valor especial que empregamos para simplificar nosso código. Aqui, usamos ∞ como valor de sentinela de forma que, sempre que uma carta com ∞ for exposta, ela não poderá ser a carta menor, a menos que ambas as pilhas tenham suas cartas sentinela expostas. Porém, uma vez que isso acontecer, todas as cartas que não são sentinelas já terão sido colocadas sobre a pilha de saída. Como sabemos com antecedência que exatamente $r - p + 1$ cartas serão colocadas sobre a pilha de saída, podemos parar após a execução dessas muitas etapas básicas.

```
MERGE(A, P, q, r)
 1 n1 ← q - p + 1
 2 n2 ← r - q
 3 criar arranjos L[1..n1 + 1] e R[1..n2 + 1]
 4 for i ← 1 to n1
 5       do L[i] ← A[p + i - 1]
 6 for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8 L[n1 + 1] ← ∞
 9 R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13        do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else A[k] ← R[j]
17                   j ← j + 1
```

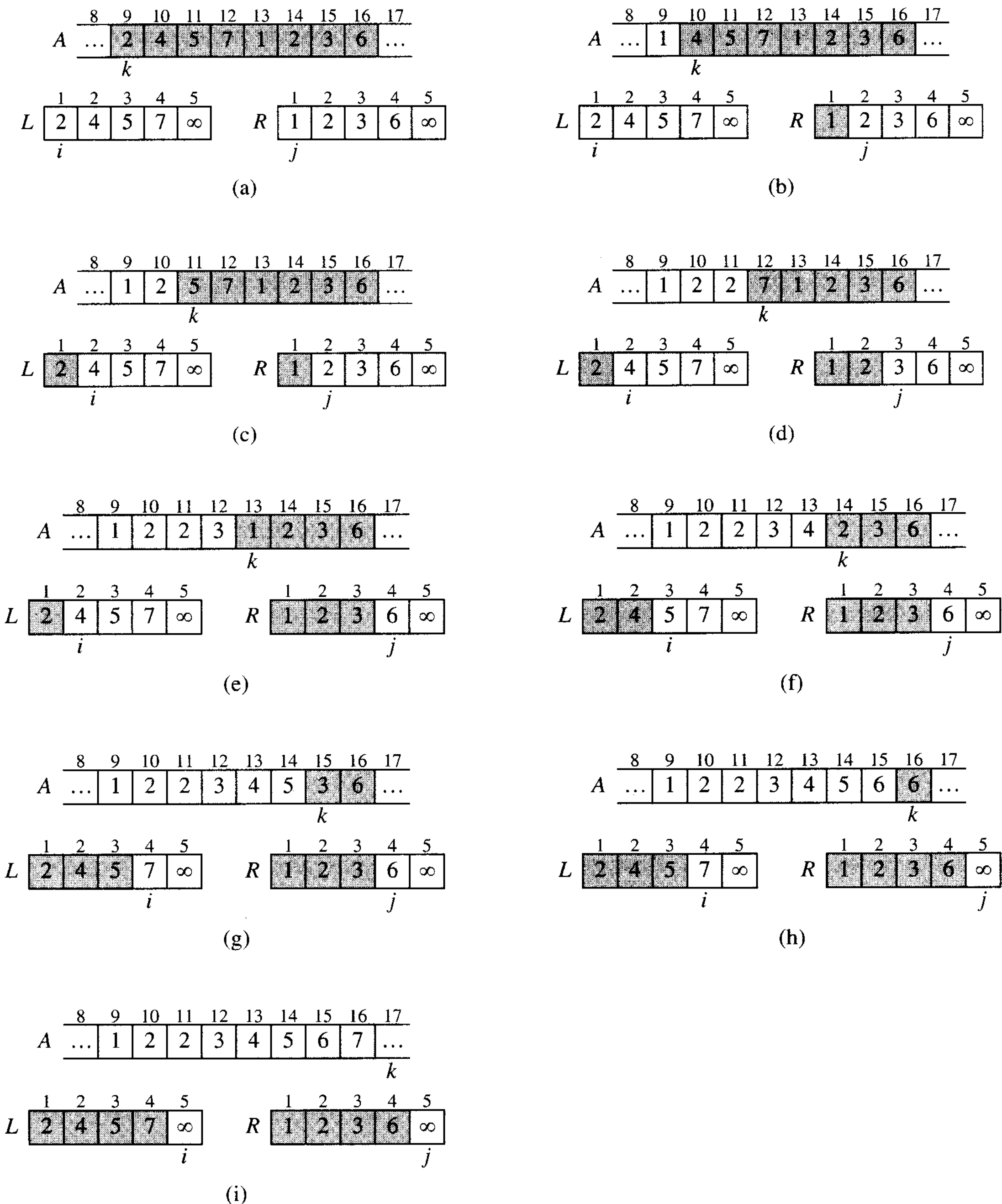


FIGURA 2.3 A operação das linhas 10 a 17 na chamada MERGE(A, 9, 12, 16) quando o subarranjo $A[9 .. 16]$ contém a seqüência $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. Depois de copiar e inserir sentinelas, o arranjo L contém $\langle 2, 4, 5, 7, \infty \rangle$, e o arranjo R contém $\langle 1, 2, 3, 6, \infty \rangle$. Posições levemente sombreadas em A contêm seus valores finais , e posições levemente sombreadas em L e R contêm valores que ainda têm de ser copiados de volta em A. Juntas, as posições levemente sombreadas sempre incluem os valores contidos originalmente em A [9 .. 16], além das duas sentinelas. Posições fortemente sombreadas em A contêm valores que serão copiados, e posições fortemente sombreadas em L e R contêm valores que já foram copiados de volta em A. **(a)-(h)** Os arranjos A, L e R e seus respectivos índices k, i, e j antes de cada iteração do loop das linhas 12 a 17. **(i)** Os arranjos e índices no final. Nesse momento, o subarranjo em $A[9 .. 16]$ está ordenado, e as duas sentinelas em L e R são os dois únicos elementos nesses arranjos que não foram copiados em A

Em detalhes, o procedimento MERGE funciona da maneira ilustrada a seguir. A linha 1 calcula o *comprimento* n_1 do subarranjo $A[p .. q]$, e a linha 2 calcula o *comprimento* n_2 do subarranjo $A[q + 1..r]$. Criamos os arranjos L e R (de "left" e "right", ou "direita" e "esquerda" em inglês) de comprimentos $n_1 + 1$ e $n_2 + 1$, respectivamente, na linha 3. O loop **for** das linhas 4 e 5 copia o

subarranjo $A[p \,..\, q]$ em $L[1 \,..\, n_1]$, e o loop **for** das linhas 6 e 7 copia o subarranjo $A[q + 1 \,..\, r]$ em $R[1 \,..\, n_2]$. As linhas 8 e 9 colocam as sentinelas nas extremidades dos arranjos L e R. As linhas 10 a 17, ilustradas na Figura 2.3, executam as $r - p + 1$ etapas básicas, mantendo o loop invariante a seguir:

> No início de cada iteração do loop **for** das linhas 12 a 17, o subarranjo $A[p \,..\, k - 1]$ contém os $k - p$ menores elementos de $L[1 \,..\, n_1 + 1]$ e $R[1 \,..\, n_2 + 1]$, em seqüência ordenada.

Além disso, $L[i]$ e $R[j]$ são os menores elementos de seus arranjos que não foram copiados de volta em A.

Devemos mostrar que esse loop invariante é válido antes da primeira iteração do loop **for** das linhas 12 a 17, que cada iteração do loop mantém o invariante, e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

Inicialização: Antes da primeira iteração do loop, temos $k = p$, de forma que o subarranjo $A[p \,..\, k - 1]$ está vazio. Esse subarranjo vazio contém os $k - p = 0$ menores elementos de L e R e, desde que $i = j = 1$, tanto $L[i]$ quanto $R[j]$ são os menores elementos de seus arranjos que não foram copiados de volta em A.

Manutenção: Para ver que cada iteração mantém o loop invariante, vamos supor primeiro que $L[i] \leq R[j]$. Então $L[i]$ é o menor elemento ainda não copiado de volta em A. Como $A[p \,..\, k - 1]$ contém os $k - p$ menores elementos, depois da linha 14 copiar $L[i]$ em $A[k]$, o subarranjo $A[p \,..\, k]$ conterá os $k - p + 1$ menores elementos. O incremento de k (na atualização do loop **for**) e de i (na linha 15) restabelece o loop invariante para a próxima iteração. Se, em vez disso, $L[i] > R[j]$, então as linhas 16 e 17 executam a ação apropriada para manter o loop invariante.

Término: No término, $k = r + 1$. Pelo loop invariante, o subarranjo $A[p \,..\, k - 1]$, que é $A[p \,..\, r]$, contém os $k - p = r - p + 1$ menores elementos de $L[1 \,..\, n_1 + 1]$ e $R[1 \,..\, n_2 + 1]$ em seqüência ordenada. Os arranjos L e R contêm juntos $n_1 + n_2 + 2 = r - p + 3$ elementos. Todos os elementos, exceto os dois maiores, foram copiados de volta em A, e esses dois maiores elementos são as sentinelas.

Para ver que o procedimento MERGE é executado no tempo $\Theta(n)$, onde $n = r - p + 1$, observe que cada uma das linhas 1 a 3 e 8 a 11 demora um tempo constante, que os loops **for** das linhas 4 a 7 demoram o tempo $\Theta(n_1 + n_2) = \Theta(n)$,[6] e que há n iterações do loop **for** das linhas 12 a 17, cada uma demorando um tempo constante.

Agora podemos usar o procedimento MERGE como uma sub-rotina no algoritmo de ordenação por intercalação. O procedimento MERGE-SORT(A, p, r) ordena os elementos do subarranjo $A[p \,..\, r]$. Se $p \geq r$, o subarranjo tem no máximo um elemento e, portanto, já está ordenado. Caso contrário, a etapa de divisão simplesmente calcula um índice q que particiona $A[p \,..\, r]$ em dois subarranjos: $A[p \,..\, q]$, contendo $\lceil n/2 \rceil$ elementos, e $A[q + 1 \,..\, r]$, contendo $\lfloor n/2 \rfloor$ elementos.[7]

```
MERGE-SORT(A, p, r)
1 if p < r
2    then q ← ⌊(p + r)/2⌋
3         MERGE-SORT(A, p, q)
4         MERGE-SORT(A, q + 1, r)
5         MERGE(A, p, q, r)
```

---

[6] Veremos no Capítulo 3 como interpretar formalmente equações contendo notação Θ.

[7] A expressão $\lceil x \rceil$ denota o menor inteiro maior que ou igual a x, e $\lfloor x \rfloor$ denota o maior inteiro menor que ou igual a x. Essas notações são definidas no Capítulo 3. O caminho mais fácil para verificar que definir q como $\lfloor (p + r)/2 \rfloor$ produz os subarranjos $A[p \,..\, q]$ e $A[q + 1 \,..\, r]$ de tamanhos $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$, respectivamente, é examinar os quatro casos que surgem dependendo do fato de cada valor de p e r ser ímpar ou par.

Para ordenar a seqüência $A = \langle A[1], A[2], ..., A[n]\rangle$ inteira, efetuamos a chamada inicial MERGE-SORT(A, 1, *comprimento*[A]), onde, mais uma vez, *comprimento*[A] $= n$. A Figura 2.4 ilustra a operação do procedimento de baixo para cima quando n é uma potência de 2. O algoritmo consiste em intercalar pares de seqüências de um item para formar seqüências ordenadas de comprimento 2, intercalar pares de seqüências de comprimento 2 para formar seqüências ordenadas de comprimento 4 e assim por diante, até duas seqüências de comprimento $n/2$ serem intercaladas para formar a seqüência ordenada final de comprimento n.

2.3.2 Análise de algoritmos de dividir e conquistar

Quando um algoritmo contém uma chamada recursiva a si próprio, seu tempo de execução freqüentemente pode ser descrito por uma ***equação de recorrência*** ou ***recorrência***, que descreve o tempo de execução global sobre um problema de tamanho n em termos do tempo de execução sobre entradas menores. Então, podemos usar ferramentas matemáticas para resolver a recorrência e estabelecer limites sobre o desempenho do algoritmo.

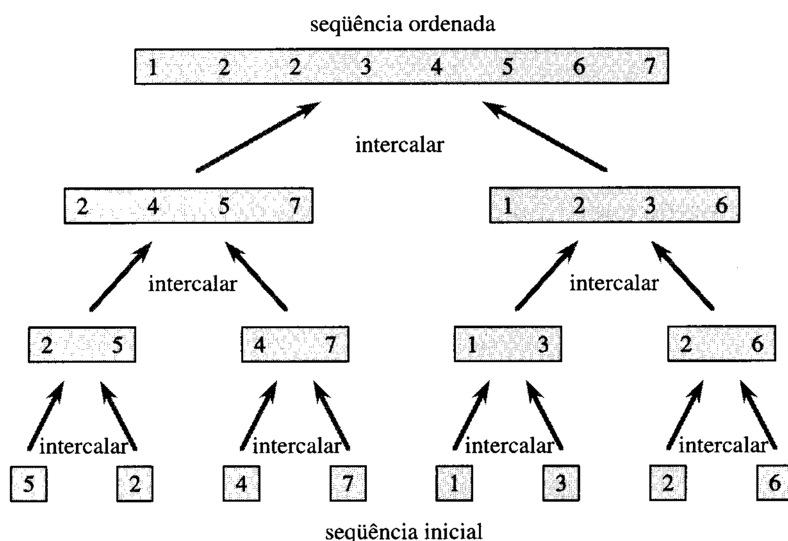


FIGURA 2.4 A operação de ordenação por intercalação sobre o arranjo $A = \langle 5, 2, 4, 7, 1, 3, 2, 6\rangle$. Os comprimentos das seqüências ordenadas que estão sendo intercaladas aumentam com a progressão do algoritmo da parte inferior até a parte superior

Uma recorrência para o tempo de execução de um algoritmo de dividir e conquistar se baseia nos três passos do paradigma básico. Como antes, consideramos $T(n)$ o tempo de execução sobre um problema de tamanho n. Se o tamanho do problema for pequeno o bastante, digamos $n \le c$ para alguma constante c, a solução direta demorará um tempo constante, que consideraremos $\Theta(1)$. Vamos supor que o problema seja dividido em a subproblemas, cada um dos quais com $1/b$ do tamanho do problema original. (No caso da ordenação por intercalação, tanto a quanto b são iguais a 2, mas veremos muitos algoritmos de dividir e conquistar nos quais a ... b.) Se levarmos o tempo $D(n)$ para dividir o problema em subproblemas e o tempo $C(n)$ para combinar as soluções dadas aos subproblemas na solução para o problema original, obteremos a recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \le c, \\ aT(n/b) + D(n) + C(n) & \text{em caso contrário.} \end{cases}$$

No Capítulo 4, veremos como resolver recorrências comuns dessa maneira.

Análise da ordenação por intercalação

Embora o pseudocódigo para MERGE-SORT funcione corretamente quando o número de elementos não é par, nossa análise baseada na recorrência será simplificada se fizermos a suposição de que o tamanho do problema original é uma potência de dois. Então, cada passo de dividir produzirá duas subseqüências de tamanho exatamente $n/2$. No Capítulo 4, veremos que essa premissa não afeta a ordem de crescimento da solução para a recorrência.

Apresentaremos em seguida nossas razões para configurar a recorrência correspondente a $T(n)$, o tempo de execução do pior caso da ordenação por intercalação sobre n números. A ordenação por intercalação sobre um único elemento demora um tempo constante. Quando temos $n > 1$ elementos, desmembramos o tempo de execução do modo explicado a seguir.

Dividir: A etapa de dividir simplesmente calcula o ponto médio do subarranjo, o que demora um tempo constante. Portanto, $D(n) = \Theta(1)$.

Conquistar: Resolvemos recursivamente dois subproblemas; cada um tem o tamanho $n/2$ e contribui com $2T(n/2)$ para o tempo de execução.

Combinar: Já observamos que o procedimento MERGE em um subarranjo de n elementos leva o tempo $\Theta(n)$; assim, $C(n) = \Theta(n)$.

Quando somamos as funções $D(n)$ e $C(n)$ para a análise da ordenação por intercalação, estamos somando uma função que é $\Theta(n)$ a uma função que é $\Theta(1)$. Essa soma é uma função linear de n, ou seja, $\Theta(n)$. A adição dessa função ao termo $2T(n/2)$ da etapa de "conquistar" fornece a recorrência para o tempo de execução do pior caso $T(n)$ da ordenação por intercalação:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1, \\ 2T(n/2) + \Theta(n) & \text{se } n > 1. \end{cases} \tag{2.1}$$

No Capítulo 4, mostraremos o "teorema mestre", que podemos utilizar para demonstrar que $T(n)$ é $\Theta(n \lg n)$, onde $\lg n$ significa $\log_2 n$. Para entradas suficientemente grandes, a ordenação por intercalação, com seu tempo de execução $\Theta(n \lg n)$, supera a ordenação por inserção, cujo tempo de execução é $\Theta(n^2)$, no pior caso.

Não precisamos do teorema mestre para entender intuitivamente por que a solução para a recorrência (2.1) é $T(n) = \Theta(n \lg n)$. Vamos reescrever a recorrência (2.1) como

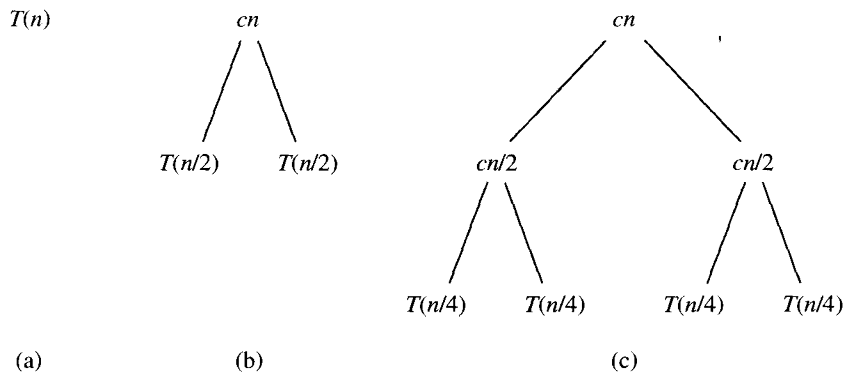
$$T(n) = \begin{cases} c & \text{se } n = 1, \\ 2T(n/2) + cn & \text{se } n > 1. \end{cases} \tag{2.2}$$

onde a constante c representa o tempo exigido para resolver problemas de tamanho 1, como também o tempo por elemento do arranjo para as etapas de dividir e combinar.[8]

A Figura 2.5 mostra como podemos resolver a recorrência (2.2). Por conveniência, supomos que n é uma potência exata de 2. A parte (a) da figura mostra $T(n)$ que, na parte (b), foi expandida em uma árvore equivalente representando a recorrência. O termo cn é a raiz (o custo no nível superior da recursão), e as duas subárvores da raiz são as duas recorrências menores $T(n/2)$. A parte (c) mostra esse processo levado uma etapa adiante pela expansão de $T(n/2)$. O custo para cada um dos dois subnós no segundo nível de recursão é $cn/2$. Continuamos a expandir cada nó na árvore, desmembrando-o em suas partes constituintes como determina a recorrência, até os tamanhos de problemas se reduzirem a 1, cada qual com o custo c. A parte (d) mostra a árvore resultante.

---

[8] É improvável que a mesma constante represente exatamente o tempo para resolver problemas de tamanho 1 e também o tempo por elemento do arranjo para as etapas de dividir e combinar. Podemos contornar esse problema permitindo que c seja o maior desses tempos e reconhecendo que nossa recorrência fornece um limite superior sobre o tempo de execução, ou permitindo a c ser o menor desses tempos e reconhecendo que nossa recorrência fornece um limite inferior sobre o tempo de execução. Ambos os limites serão estabelecidos sobre a ordem de $n \lg n$ e, tomados juntos, corresponderão ao tempo de execução $\Theta(n \lg n)$.



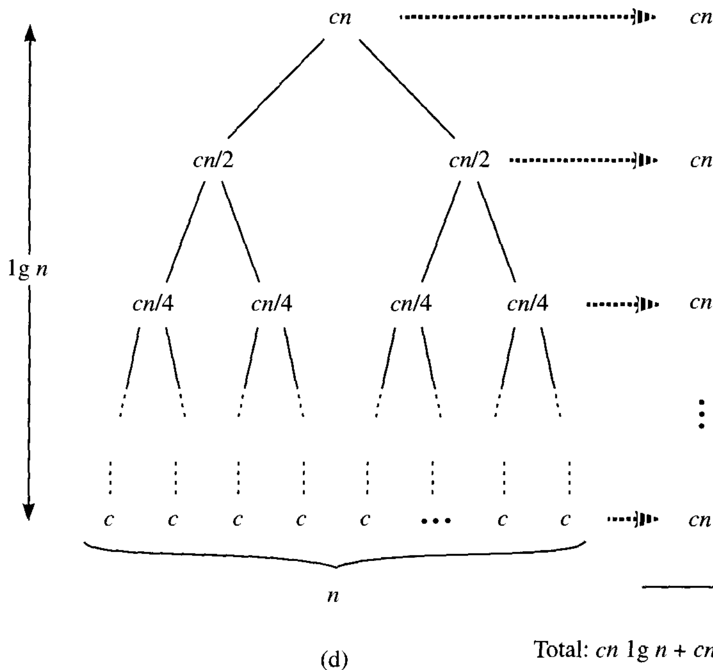


FIGURA 2.5 A construção de uma árvore de recursão para a recorrência $T(n) = 2T(n/2) + cn$. A parte **(a)** mostra $T(n)$, que é progressivamente expandido em **(b)–(d)** para formar a árvore de recursão. A árvore completamente expandida da parte (d) tem $\lg n + 1$ níveis (isto é, tem altura $\lg n$, como indicamos), e cada nível contribui com o custo total cn. Então, o custo total é $cn \lg n + cn$, que é $\Theta(n \lg n)$

Em seguida, somamos os custos através de cada nível da árvore. O nível superior tem o custo total cn, o próximo nível abaixo tem custo total $c(n/2) + c(n/2) = cn$, o nível após esse tem custo total $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$ e assim por diante. Em geral, o nível i abaixo do topo tem 2^i nós, cada qual contribuindo com um custo $c(n/2^i)$, de forma que o i-ésimo nível abaixo do topo tem custo total $2^i\, c(n/2^i) = cn$.

No nível inferior existem n nós, cada um contribuindo com um custo c, para um custo total cn.

O número total de níveis da "árvore de recursão" da Figura 2.5 é $\lg n + 1$. Esse fato é facilmente visto por um argumento indutivo informal. O caso básico ocorre quando $n = 1$, e nesse caso só há um nível. Como $\lg 1 = 0$, temos que $\lg n + 1$ fornece o número correto de níveis.

Agora suponha, como uma hipótese indutiva, que o número de níveis de uma árvore de recursão para 2^i nós seja $\lg 2^i + 1 = i + 1$ (pois, para qualquer valor de i, temos $\lg 2^i = i$). Como estamos supondo que o tamanho da entrada original é uma potência de 2, o tamanho da próxima entrada a considerar é 2^{i+1}. Uma árvore com 2^{i+1} nós tem um nível a mais em relação a uma árvore de 2^i nós, e então o número total de níveis é $(i + 1) + 1 = \lg 2^{i+1} + 1$.

Para calcular o custo total representado pela recorrência (2.2), simplesmente somamos os custos de todos os níveis. Há $\lg n + 1$ níveis, cada um com o custo cn, o que nos dá o custo total $cn(\lg n + 1) = cn \lg n + cn$. Ignorando o termo de baixa ordem e a constante c, obtemos o resultado desejado, $\Theta(n \lg n)$.

Exercícios

2.3-1
Usando a Figura 2.4 como modelo, ilustre a operação de ordenação por intercalação sobre o arranjo $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

2.3-2
Reescreva o procedimento MERGE de modo que ele não utilize sentinelas e, em vez disso, se interrompa depois que o arranjo L ou R tiver todos os seus elementos copiados de volta para A, e então copie o restante do outro arranjo de volta em A.

2.3-3
Use indução matemática para mostrar que, quando n é uma potência exata de 2, a solução da recorrência

$$T(n) = \begin{cases} 2 & \text{se } n = 2, \\ 2T(n/2) + n & \text{se } n > 2^k, \text{ para } k > 1 \end{cases}$$

é $T(n) = n \lg n$.

2.3-4
A ordenação por inserção pode ser expressa sob a forma de um procedimento recursivo como a seguir. Para ordenar $A[1 \,..\, n]$, ordenamos recursivamente $A[1 \,..\, n-1]$ e depois inserimos $A[n]$ no arranjo ordenado $A[1 \,..\, n-1]$. Escreva uma recorrência para o tempo de execução dessa versão recursiva da ordenação por inserção.

2.3-5
Voltando ao problema da pesquisa (ver Exercício 2.1-3) observe que, se a seqüência A estiver ordenada, poderemos comparar o ponto médio da seqüência com v e eliminar metade da seqüência de consideração posterior. A ***pesquisa binária*** é um algoritmo que repete esse procedimento, dividindo ao meio o tamanho da porção restante da seqüência a cada vez. Escreva pseudocódigo, sendo ele iterativo ou recursivo, para pesquisa binária. Demonstre que o tempo de execução do pior caso da pesquisa binária é $\Theta(\lg n)$.

2.3-6
Observe que o loop **while** das linhas 5 a 7 do procedimento INSERTION-SORT na Seção 2.1 utiliza uma pesquisa linear para varrer (no sentido inverso) o subarranjo ordenado $A[1 \,..\, j-1]$. Podemos usar em vez disso uma pesquisa binária (ver Exercício 2.3-5) para melhorar o tempo de execução global do pior caso da ordenação por inserção para $\Theta(n \lg n)$?

2.3-7 *
Descreva um algoritmo de tempo $\Theta(n \lg n)$ que, dado um conjunto S de n inteiros e outro inteiro x, determine se existem ou não dois elementos em S cuja soma seja exatamente x.

Problemas

2-1 Ordenação por inserção sobre arranjos pequenos na ordenação por intercalação

Embora a ordenação por intercalação funcione no tempo de pior caso $\Theta(n \lg n)$ e a ordenação por inserção funcione no tempo de pior caso $\Theta(n^2)$, os fatores constantes na ordenação por inserção a tornam mais rápida para n pequeno. Assim, faz sentido usar a ordenação por inserção dentro da ordenação por intercalação quando os subproblemas se tornam suficientemente pequenos. Considere uma modificação na ordenação por intercalação, na qual n/k sublistas de comprimento k são ordenadas usando-se a ordenação por inserção, e depois intercaladas com o uso do mecanismo padrão de intercalação, onde k é um valor a ser determinado.

a. Mostre que as n/k sublistas, cada uma de comprimento k, podem ser ordenadas através da ordenação por inserção no tempo de pior caso $\Theta(nk)$.

b. Mostre que as sublistas podem ser intercaladas no tempo de pior caso $\Theta(n \lg(n/k))$.

c. Dado que o algoritmo modificado é executado no tempo de pior caso $\Theta(nk + n \lg(n/k))$, qual é o maior valor assintótico (notação Θ) de k como uma função de n para a qual o algoritmo modificado tem o mesmo tempo de execução assintótico que a ordenação por intercalação padrão?

d. Como k deve ser escolhido na prática?

2-2 Correção do bubblesort

O bubblesort é um algoritmo de ordenação popular. Ele funciona permutando repetidamente elementos adjacentes que estão fora de ordem.

```
BUBBLESORT(A)
1 for i ← 1 to comprimento[A]
2     do for j ← comprimento[A] downto i + 1
3            do if A[j] < A[j – 1]
4                  then trocar A[j] ↔ A[j – 1]
```

a. Seja A' um valor que denota a saída de BUBBLESORT(A). Para provar que BUBBLESORT é correto, precisamos provar que ele termina e que

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n] , \tag{2.3}$$

onde n = *comprimento*[A]. O que mais deve ser provado para mostrar que BUBBLESORT realmente realiza a ordenação?

As duas próximas partes provarão a desigualdade (2.3).

b. Enuncie com precisão um loop invariante para o loop **for** das linhas 2 a 4 e prove que esse loop invariante é válido. Sua prova deve usar a estrutura da prova do loop invariante apresentada neste capítulo.

c. Usando a condição de término do loop invariante demonstrado na parte (b), enuncie um loop invariante para o loop **for** das linhas 1 a 4 que lhe permita provar a desigualdade (2.3). Sua prova deve empregar a estrutura da prova do loop invariante apresentada neste capítulo.

d. Qual é o tempo de execução do pior caso de bubblesort? Como ele se compara ao tempo de execução da ordenação por inserção?

2-3 Correção da regra de Horner

O fragmento de código a seguir implementa a regra de Horner para avaliar um polinômio

$$P(x) = \sum_{k=0}^{n} a_k x^k$$

$$= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots)) ,$$

dados os coeficientes $a_0, a_1, ..., a_n$ e um valor para x:

```
1 y ← 0
2 i ← n
3 while i ≥ 0
4     do y ← a_i + x · y
5        i ← i - 1
```

a. Qual é o tempo de execução assintótico desse fragmento de código para a regra de Horner?

b. Escreva pseudocódigo para implementar o algoritmo ingênuo de avaliação polinomial que calcula cada termo do polinômio desde o início. Qual é o tempo de execução desse algoritmo? Como ele se compara à regra de Horner?

c. Prove que a expressão a seguir é um loop invariante para o loop **while** das linhas 3 a 5.

No início de cada iteração do loop **while** das linhas 3 a 5,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k .$$

Interprete um somatório sem termos como igual a 0. Sua prova deve seguir a estrutura da prova do loop invariante apresentada neste capítulo e deve mostrar que, no término, $y = \sum_{k=0}^{n} a_k x^k$.

d. Conclua demonstrando que o fragmento de código dado avalia corretamente um polinômio caracterizado pelos coeficientes $a_0, a_1, ..., a_n$.

2-4 *Inversões*

Seja $A[1..n]$ um arranjo de n números distintos. Se $i < j$ e $A[i] > A[j]$, então o par (i,j) é chamado uma ***inversão*** de A.

a. Liste as cinco inversões do arranjo $\langle 2, 3, 8, 6, 1 \rangle$.

b. Qual arranjo com elementos do conjunto $\{1, 2, ..., n\}$ tem o número máximo de inversões? Quantas inversões ele tem?

c. Qual é o relacionamento entre o tempo de execução da ordenação por inserção e o número de inversões no arranjo de entrada? Justifique sua resposta.

d. Dê um algoritmo que determine o número de inversões em qualquer permutação sobre n elementos no tempo do pior caso de $\Theta(n \lg n)$. (*Sugestão*: modifique a ordenação por intercalação.)

Notas do capítulo

Em 1968, Knuth publicou o primeiro de três volumes com o título geral *The Art of Computer Programming* [182, 183, 185]. O primeiro volume iniciou o estudo moderno de algoritmos de computador com um foco na análise do tempo de execução, e a série inteira continua a ser uma referência valiosa e interessante para muitos dos tópicos apresentados aqui. De acordo com Knuth, a palavra "algoritmo" é derivada do nome "al-Khowârizmî", um matemático persa do século IX.

Aho, Hopcroft e Ullman [5] defenderam a análise assintótica dos algoritmos como um meio de comparar o desempenho relativo. Eles também popularizaram o uso de relações de recorrência para descrever os tempos de execução de algoritmos recursivos.

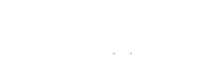
Knuth [185] oferece um tratamento enciclopédico de muitos algoritmos de ordenação. Sua comparação dos algoritmos de ordenação (página 381) inclui análises exatas de contagem de passos, como a que realizamos aqui para a ordenação por inserção. A discussão por Knuth da ordenação por inserção engloba diversas variações do algoritmo. A mais importante delas é a ordenação de Shell, introduzida por D. L. Shell, que utiliza a ordenação por inserção sobre subseqüências periódicas da entrada para produzir um algoritmo de ordenação mais rápido.

A ordenação por intercalação também é descrita por Knuth. Ele menciona que um intercalador mecânico capaz de mesclar dois decks de cartões perfurados em uma única passagem foi inventado em 1938. J. von Neumann, um dos pioneiros da informática, aparentemente escreveu um programa para fazer a ordenação por intercalação no computador EDVAC em 1945.

A primeira referência à demonstração de programas corretos é descrita por Gries [133], que credita a P. Naur o primeiro artigo nesse campo. Gries atribui loops invariantes a R. W. Floyd. O livro-texto de Mitchell [222] descreve o progresso mais recente na demonstração de programas corretos.

Capítulo 3

Crescimento de funções



A ordem de crescimento do tempo de execução de um algoritmo, definida no Capítulo 2, fornece uma caracterização simples da eficiência do algoritmo, e também nos permite comparar o desempenho relativo de algoritmos alternativos. Uma vez que o tamanho da entrada n se torna grande o suficiente, a ordenação por intercalação, com seu tempo de execução do pior caso $\Theta(n \lg n)$, vence a ordenação por inserção, cujo tempo de execução do pior caso é $\Theta(n^2)$. Embora às vezes seja possível determinar o tempo exato de execução de um algoritmo, como fizemos no caso da ordenação por inserção no Capítulo 2, a precisão extra em geral não vale o esforço de calculá-la. Para entradas grandes o bastante, as constantes multiplicativas e os termos de mais baixa ordem de um tempo de execução exato são dominados pelos efeitos do próprio tamanho da entrada.

Quando observamos tamanhos de entrada grandes o suficiente para tornar relevante apenas a ordem de crescimento do tempo de execução, estamos estudando a eficiência ***assintótica*** dos algoritmos. Ou seja, estamos preocupados com a maneira como o tempo de execução de um algoritmo aumenta com o tamanho da entrada *no limite*, à medida que o tamanho da entrada aumenta indefinidamente (sem limitação). Em geral, um algoritmo que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, exceto as muito pequenas.

Este capítulo oferece vários métodos padrão para simplificar a análise assintótica de algoritmos. A próxima seção começa definindo diversos tipos de "notação assintótica", da qual já vimos um exemplo na notação Θ. Várias convenções de notação usadas em todo este livro serão então apresentadas, e finalmente faremos uma revisão do comportamento de funções que surgem comumente na análise de algoritmos.

3.1 Notação assintótica

As notações que usamos para descrever o tempo de execução assintótica de um algoritmo são definidas em termos de funções cujos domínios são o conjunto dos números naturais $\mathbf{N} = \{0, 1, 2, ...\}$. Tais notações são convenientes para descrever a função do tempo de execução do pior caso $T(n)$, que em geral é definida somente sobre tamanhos de entrada inteiros. Contudo, às vezes é conveniente ***abusar*** da notação assintótica de várias maneiras. Por exemplo, a notação é estendida com facilidade ao domínio dos números reais ou, de modo alternativo, limitado a um subconjunto dos números naturais. Porém, é importante entender o significado preciso da notação para que, quando houver um abuso em seu uso, ela não seja ***mal utilizada***. Esta seção define as notações assintóticas básicas e também apresenta alguns abusos comuns.

Notação Θ

No Capítulo 2, descobrimos que o tempo de execução do pior caso da ordenação por inserção é $T(n) = \Theta(n^2)$. Vamos definir o que significa essa notação. Para uma dada função $g(n)$, denotamos por $\Theta(g(n))$ o *conjunto de funções*

$$\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ para todo } n \ge n_0\}.^1$$

Uma função $f(n)$ pertence ao conjunto $\Theta(g(n))$ se existem constantes positivas c_1 e c_2 tais que ela possa ser "imprensada" entre $c_1 g(n)$ e $c_2 g(n)$, para um valor de n suficientemente grande. Como $\Theta(g(n))$ é um conjunto, poderíamos escrever "$f(n) \in \Theta(g(n))$" para indicar que $f(n)$ é um membro de (ou pertence a) $\Theta(g(n))$. Em vez disso, em geral escreveremos "$f(n) = \Theta(g(n))$" para expressar a mesma noção. Esse abuso da igualdade para denotar a condição de membro de um conjunto (pertinência) pode a princípio parecer confuso, mas veremos adiante nesta seção que ele tem suas vantagens.

A Figura 3.1(a) apresenta um quadro intuitivo das funções $f(n)$ e $g(n)$, onde $f(n) = \Theta(g(n))$. Para todos os valores de n à direita de n_0, o valor de $f(n)$ reside em $c_1 g(n)$ ou acima dele, e em $c_2 g(n)$ ou abaixo desse valor. Em outras palavras, para todo $n \ge n_0$, a função $f(n)$ é igual a $g(n)$ dentro de um fator constante. Dizemos que $g(n)$ é um ***limite assintoticamente restrito*** para $f(n)$.

A definição de $\Theta(g(n))$ exige que todo membro $f(n) \in \Theta(g(n))$ seja ***assintoticamente não negativo***, isto é, que $f(n)$ seja não negativo sempre que n for suficientemente grande. (Uma função ***assintoticamente positiva*** é uma função positiva para todo n suficientemente grande.) Em conseqüência disso, a própria função $g(n)$ deve ser assintoticamente não negativa, ou então o conjunto $\Theta(g(n))$ é vazio. Por essa razão, vamos supor que toda função usada dentro da notação Θ é assintoticamente não negativa. Essa premissa também se mantém para as outras notações assintóticas definidas neste capítulo.

No Capítulo 2, introduzimos uma noção informal da notação Θ que consistia em descartar os termos de mais baixa ordem e ignorar o coeficiente inicial do termo de mais alta ordem. Vamos justificar brevemente essa intuição, usando a definição formal para mostrar que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Para isso, devemos definir constantes positivas c_1, c_2 e n_0 tais que

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

para todo $n \ge n_0$. A divisão por n^2 produz

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

A desigualdade do lado direito pode ser considerada válida para qualquer valor de $n \ge 1$, escolhendo-se $c_2 \ge 1/2$. Do mesmo modo, a desigualdade da esquerda pode ser considerada válida para qualquer valor de $n \ge 7$, escolhendo-se $c_1 \le 1/14$. Assim, escolhendo $c_1 = 1/14$, $c_2 = 1/2$ e $n_0 = 7$, podemos verificar que $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Certamente, existem outras opções para as constantes, mas o fato importante é que existe *alguma* opção. Observe que essas constantes dependem da função $\frac{1}{2}n^2 - 3n$; uma função diferente pertencente a $\Theta(n^2)$ normalmente exigiria constantes distintas.

---

[1] Na notação de conjuntos, um sinal de dois-pontos deve ser lido como "tal que".

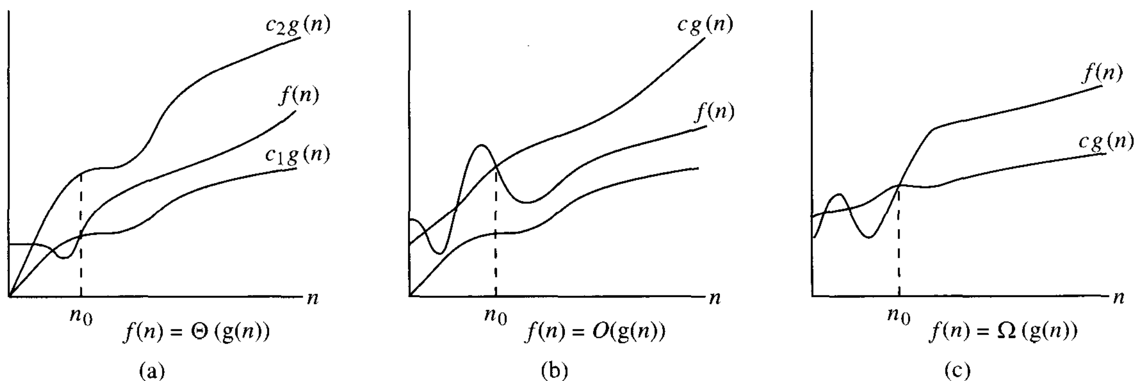


FIGURA 3.1 Exemplos gráficos das notações Θ, O e Ω. Em cada parte, o valor de n_0 mostrado é o valor mínimo possível; qualquer valor maior também funcionaria. **(a)** A notação Θ limita uma função ao intervalo entre fatores constantes. Escrevemos $f(n) = \Theta(g(n))$ se existem constantes positivas n_0, c_1 e c_2 tais que, à direita de n_0, o valor de $f(n)$ sempre reside entre $c_1g(n)$ e $c_2g(n)$ inclusive. **(b)** A notação O dá um limite superior para uma função dentro de um fator constante. Escrevemos $f(n) = O(g(n))$ se existem constantes positivas n_0 e c tais que, à direita de n_0, o valor de $f(n)$ sempre reside em ou abaixo de $cg(n)$. **(c)** A notação Ω dá um limite inferior para uma função dentro de um fator constante. Escrevemos $f(n) = \Omega(g(n))$ se existem constantes positivas $n0$ e c tais que, à direita de n_0, o valor de $f(n)$ sempre reside em ou acima de $cg(n)$

Também podemos usar a definição formal para verificar que $6n^3 \neq \Theta(n^2)$. Vamos supor, a título de contradição, que existam c_2 e n_0 tais que $6n^3 \leq c_2n_2$ para todo $n \geq n_0$. Mas então $n \leq c_2/6$, o que não pode ser válido para um valor de n arbitrariamente grande, pois c_2 é constante.

Intuitivamente, os termos de mais baixa ordem de uma função assintoticamente positiva podem ser ignorados na determinação de limites assintoticamente restritos, porque eles são insignificantes para grandes valores de n. Uma minúscula fração do termo de mais alta ordem é suficiente para dominar os termos de mais baixa ordem. Desse modo, a definição de c_1 com um valor ligeiramente menor que o coeficiente do termo de mais alta ordem e a definição de c_2 com um valor ligeiramente maior permite que as desigualdades na definição da notação Θ sejam satisfeitas. O coeficiente do termo de mais alta ordem pode do mesmo modo ser ignorado, pois ele só muda $c1$ e c_2 por um fator constante igual ao coeficiente.

Como exemplo, considere qualquer função quadrática $f(n) = an^2 + bn + c$, onde a, b e c são constantes e $a > 0$. Descartando os termos de mais baixa ordem e ignorando a constante, produzimos $f(n) = \Theta(n^2)$. Formalmente, para mostrar a mesma coisa, tomamos as constantes $c_1 = a/4$, $c_2 = 7a/4$ e $n_0 = 2.\max((|b|/a), \sqrt{(|c|/a)})$. O leitor poderá verificar que $0 \leq c_1n^2 \leq an^2 + bn + c \leq c_2n^2$ para todo $n \geq n_0$. Em geral, para qualquer polinômio $p(n) = \sum_{i=0}^{d} a_i n^i$, onde a_i são constantes e $a_d > 0$, temos $p(n) = \Theta(n^d)$ (ver Problema 3-1).

Tendo em vista que qualquer constante é um polinômio de grau 0, podemos expressar qualquer função constante como $\Theta(n^0)$, ou $\Theta(1)$. Porém, essa última notação é um abuso secundário, porque não está claro qual variável está tendendo a infinito.[2] Usaremos com freqüência a notação $\Theta(1)$ para indicar uma constante ou uma função constante em relação a alguma variável.

[2] O problema real é que nossa notação comum para funções não distingue funções de valores. No cálculo de λ, os parâmetros para uma função estão claramente especificados: a função n^2 poderia ser escrita como $\lambda n.n^2$, ou até mesmo $\lambda r.r^2$. Porém, a adoção de uma notação mais rigorosa complicaria manipulações algébricas, e assim optamos por tolerar o abuso.

Notação O

A notação Θ limita assintoticamente uma função acima e abaixo. Quando temos apenas um ***limite assintótico superior***, usamos a notação O. Para uma dada função $g(n)$, denotamos por $O(g(n))$ (lê-se "O maiúsculo de g de n" ou, às vezes, apenas "o de g de n") o conjunto de funções

$$O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \le f(n) \le cg(n) \text{ para todo } n \ge n_0\}.$$

Usamos a notação O para dar um limite superior sobre uma função, dentro de um fator constante. A Figura 3.1(b) mostra a intuição por trás da notação O. Para todos os valores n à direita de n_0, o valor da função $f(n)$ está em ou abaixo de $g(n)$.

Para indicar que uma função $f(n)$ é um membro de $O(g(n))$, escrevemos $f(n) = O(g(n))$. Observe que $f(n) = \Theta(g(n))$ implica $f(n) = O(g(n))$, pois a notação Θ é uma noção mais forte que a notação O. Em termos da teoria de conjuntos, temos $\Theta(g(n)) \subseteq O(g(n))$. Desse modo, nossa prova de que qualquer função quadrática $an^2 + bn + c$, onde $a > 0$, está em $\Theta(n^2)$ também mostra que qualquer função quadrática está em $O(n^2)$. O que pode ser mais surpreendente é o fato de que qualquer função *linear* $an + b$ está em $O(n^2)$, o que é facilmente verificado fazendo-se $c = a + |b|$ e $n_0 = 1$.

Alguns leitores que viram antes a notação O podem achar estranho que devamos escrever, por exemplo, $n = O(n^2)$. Na literatura, a notação O é usada às vezes de modo informal para descrever limites assintoticamente restritos, ou seja, o que definimos usando a notação Θ. Contudo, neste livro, quando escrevermos $f(n) = O(g(n))$, estamos simplesmente afirmando que algum múltiplo constante de $g(n)$ é um limite assintótico superior sobre $f(n)$, sem qualquer menção sobre o quanto um limite superior é restrito. A distinção entre limites assintóticos superiores e limites assintoticamente restritos agora se tornou padrão na literatura de algoritmos.

Usando a notação O, podemos descrever freqüentemente o tempo de execução de um algoritmo apenas inspecionando a estrutura global do algoritmo. Por exemplo, a estrutura de loop duplamente aninhado do algoritmo de ordenação por inserção vista no Capítulo 2 produz imediatamente um limite superior $O(n^2)$ sobre o tempo de execução do pior caso: o custo do loop interno é limitado na parte superior por $O(1)$ (constante), os índices i e j são ambos no máximo n, e o loop interno é executado no máximo uma vez para cada um dos n^2 pares de valores correspondentes a i e j.

Tendo em vista que a notação O descreve um limite superior, quando a empregamos para limitar o tempo de execução do pior caso de um algoritmo, temos um limite sobre o tempo de execução do algoritmo em cada entrada. Desse modo, o limite $O(n^2)$ no tempo de execução do pior caso da ordenação por inserção também se aplica a seu tempo de execução sobre toda entrada. Porém, o limite $\Theta(n^2)$ no tempo de execução do pior caso da ordenação por inserção não implica um limite $\Theta(n^2)$ no tempo de execução da ordenação por inserção em *toda* entrada. Por exemplo, vimos no Capítulo 2 que, quando a entrada já está ordenada, a ordenação por inserção funciona no tempo $\Theta(n)$.

Tecnicamente, é um abuso dizer que o tempo de execução da ordenação por inserção é $O(n^2)$, pois, para um dado n, o tempo de execução real varia, dependendo da entrada específica de tamanho n. Quando afirmamos que "o tempo de execução é $O(n^2)$", queremos dizer que existe uma função $f(n)$ que é $O(n^2)$ tal que, para qualquer valor de n, não importando que entrada específica de tamanho n seja escolhida, o tempo de execução sobre essa entrada tem um limite superior determinado pelo valor $f(n)$. De modo equivalente, dizemos que o tempo de execução do pior caso é $O(n^2)$.

Notação Ω

Da mesma maneira que a notação O fornece um limite assintótico *superior* sobre uma função, a notação Ω fornece um ***limite assintótico inferior***. Para uma determinada função $g(n)$, denotamos por $\Omega(g(n))$ (lê-se "ômega maiúsculo de g de n" ou, às vezes, "ômega de g de n") o conjunto de funções

$$\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \le cg(n) \le f(n) \text{ para todo } n \ge n_0\}.$$

A intuição por trás da notação Ω é mostrada na Figura 3.1(c). Para todos os valores n à direita de n_0, o valor de $f(n)$ está em ou acima de $g(n)$.

A partir das definições das notações assintóticas que vimos até agora, é fácil demonstrar o importante teorema a seguir (ver Exercício 3.1-5).

Teorema 3.1

Para duas funções quaisquer $f(n)$ e $g(n)$, temos $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$. ■

Como exemplo de aplicação desse teorema, nossa demonstração de que $an^2 + bn + c = \Theta(n^2)$ para quaisquer constantes a, b e c, onde $a > 0$, implica imediatamente que $an^2 + bn + c = \Omega(n^2)$ e $an^2 + bn + c = O(n^2)$. Na prática, em lugar de usar o Teorema 3.1 para obter limites assintóticos superiores e inferiores a partir de limites assintoticamente restritos, como fizemos nesse exemplo, nós o utilizamos normalmente para demonstrar limites assintoticamente restritos a partir de limites assintóticos superiores e inferiores.

Considerando-se que a notação Ω descreve um limite inferior, quando a usamos para limitar o tempo de execução do melhor caso de um algoritmo, por implicação também limitamos o tempo de execução do algoritmo sobre entradas arbitrárias. Por exemplo, o tempo de execução no melhor caso da ordenação por inserção é $\Omega(n)$, o que implica que o tempo de execução da ordenação por inserção é $\Omega(n)$.

Assim, o tempo de execução da ordenação por inserção recai entre $\Omega(n)$ e $O(n^2)$, pois ele fica em qualquer lugar entre uma função linear de n e uma função quadrática de n. Além disso, esses limites são assintoticamente tão restritos quanto possível: por exemplo, o tempo de execução da ordenação por inserção não é $\Omega(n^2)$, pois existe uma entrada para a qual a ordenação por inserção é executada no tempo $\Theta(n)$ (por exemplo, quando a entrada já está ordenada). Contudo, não é contraditório dizer que o tempo de execução do *pior caso* da ordenação por inserção é $\Omega(n^2)$, tendo em vista que existe uma entrada que faz o algoritmo demorar o tempo $\Omega(n^2)$. Quando afirmamos que o *tempo de execução* (sem modificador) de um algoritmo é $\Omega(g(n))$, queremos dizer que, *independentemente da entrada específica de tamanho n escolhida para cada valor de n*, o tempo de execução sobre essa entrada é pelo menos uma constante vezes $g(n)$, para um valor de n suficientemente grande.

Notação assintótica em equações e desigualdades

Já vimos como a notação assintótica pode ser usada dentro de fórmulas matemáticas. Por exemplo, na introdução da notação O, escrevemos "$n = O(n^2)$". Também poderíamos escrever $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Como interpretaremos tais fórmulas?

Quando a notação assintótica está sozinha no lado direito de uma equação (ou desigualdade), como em $n = O(n^2)$, já definimos o sinal de igualdade como símbolo de pertinência a um conjunto: $n \in O(n^2)$. Porém, em geral, quando a notação assintótica aparecer em uma fórmula, nós a interpretaremos com o significado de alguma função anônima que não nos preocupare-

mos em nomear. Por exemplo, a fórmula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ significa que $2n^2 + 3n + 1 = 2n^2 + f(n)$, onde $f(n)$ é alguma função no conjunto $\Theta(n)$. Nesse caso, $f(n) = 3n + 1$, que de fato está em $\Theta(n)$.

O uso da notação assintótica dessa maneira pode ajudar a eliminar detalhes não essenciais e a desordem em uma equação. Por exemplo, expressamos no Capítulo 2 o tempo de execução do pior caso da ordenação por intercalação como a recorrência

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

Se estivermos interessados apenas no comportamento assintótico de $T(n)$, não haverá sentido em especificar exatamente todos os termos de mais baixa ordem; todos eles serão considerados incluídos na função anônima denotada pelo termo $\Theta(n)$.

O número de funções anônimas em uma expressão é entendido como igual ao número de vezes que a notação assintótica aparece. Por exemplo, na expressão

$$\sum_{i=1}^{n} O(i).$$

existe apenas uma única função anônima (uma função de i). Portanto, essa expressão *não* é o mesmo que $O(1) + O(2) + \cdots + O(n)$ que, na realidade, não tem uma interpretação clara.

Em alguns casos, a notação assintótica aparece no lado esquerdo de uma equação, como em

$$2n^2 + \Theta(n) = \Theta(n^2).$$

Interpretamos tais equações usando a seguinte regra: *Independentemente de como as funções anônimas são escolhidas no lado esquerdo do sinal de igualdade, existe um modo de escolher as funções anônimas no lado direito do sinal de igualdade para tornar a equação válida.* Desse modo, o significado de nosso exemplo é que, para *qualquer* função $f(n) \in \Theta(n)$, existe *alguma* função $g(n) \in \Theta(n^2)$, tal que $2n^2 + f(n) = g(n)$ para todo n. Em outras palavras, o lado direito de uma equação fornece um nível mais grosseiro de detalhes que o lado esquerdo.

Vários desses relacionamentos podem ser encadeados, como em

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2). \end{aligned}$$

Podemos interpretar cada equação separadamente pela regra anterior. A primeira equação diz que existe *alguma* função $f(n) \in \Theta(n)$ tal que $2n^2 + 3n + 1 = 2n^2 + f(n)$ para todo n. A segunda equação afirma que, para *qualquer* função $g(n) \in \Theta(n)$ (como a função $f(n)$ que acabamos de mencionar), existe *alguma* função $h(n) \in \Theta(n^2)$ tal que $2n^2 + g(n) = h(n)$ para todo n. Observe que essa interpretação implica que $2n^2 + 3n + 1 = \Theta(n^2)$, que é aquilo que o encadeamento de equações nos fornece intuitivamente.

Notação *o*

O limite assintótico superior fornecido pela notação O pode ser ou não assintoticamente restrito. O limite $2n^2 = O(n^2)$ é assintoticamente restrito, mas o limite $2n = O(n^2)$ não o é. Usamos a notação o para denotar um limite superior que não é assintoticamente restrito. Definimos formalmente $o(g(n))$ (lê-se "*o* minúsculo de *g* de *n*") como o conjunto

$o(g(n)) = \{f(n)$: para qualquer constante positiva $c > 0$, existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$ para todo $n \geq n_0\}$.

Por exemplo, $2n = o(n^2)$, mas $2n^2 \neq o(n^2)$.

As definições da notação O e da notação o são semelhantes. A principal diferença é que em $f(n) = O(g(n))$, o limite $0 \leq f(n) \leq cg(n)$ se mantém válido para *alguma* constante $c > 0$ mas, em $f(n) = o(g(n))$, o limite $0 \leq f(n) < cg(n)$ é válido para *todas* as constantes $c > 0$. Intuitivamente, na notação o, a função $f(n)$ se torna insignificante em relação a $g(n)$ à medida que n se aproxima do infinito; isto é,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 . \qquad (3.1)$$

Alguns autores usam esse limite como uma definição da notação o; a definição neste livro também restringe as funções anônimas a serem assintoticamente não negativas.

Notação ω

Por analogia, a notação ω está para a notação Ω como a notação o está para a notação O. Usamos a notação ω para denotar um limite inferior que não é assintoticamente restrito. Um modo de defini-la é

$$f(n) \in \omega(g(n)) \text{ se e somente se } g(n) \in o(f(n)) .$$

Porém, formalmente, definimos $\omega(g(n))$ (lê-se "ômega minúsculo de g de n") como o conjunto

$\omega(g(n)) = \{f(n)$: para qualquer constante positiva $c > 0$, existe uma constante $n_0 > 0$ tal que $0 \leq cg(n) < f(n)$ para todo $n \geq n_0\}$.

Por exemplo, $n^2/2 = \omega(n)$, mas $n^2/2 \neq \omega(n^2)$. A relação $f(n) = \omega(g(n))$ implica que

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty ,$$

se o limite existe. Isto é, $f(n)$ se torna arbitrariamente grande em relação a $g(n)$ à medida que n se aproxima do infinito.

Comparação de funções

Muitas das propriedades relacionais de números reais também se aplicam a comparações assintóticas. No caso das propriedades seguintes, suponha que $f(n)$ e $g(n)$ sejam assintoticamente positivas.

Transitividade:

$f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$ implicam $f(n) = \Theta(h(n))$,
$f(n) = O(g(n))$ e $g(n) = O(h(n))$ implicam $f(n) = O(h(n))$,
$f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$ implicam $f(n) = \Omega(h(n))$,
$f(n) = o(g(n))$ e $g(n) = o(h(n))$ implicam $f(n) = o(h(n))$,
$f(n) = \omega(g(n))$ e $g(n) = \omega(h(n))$ implicam $f(n) = \omega(h(n))$.

Reflexividade:

$f(n) = \Theta(f(n))$
$f(n) = O(f(n))$
$f(n) = \Omega(f(n))$

Simetria:

$f(n) = \Theta(g(n))$ se e somente se $g(n) = \Theta(f(n))$

Simetria de transposição:

$f(n) = O(g(n))$ se e somente se $g(n) = \Omega(f(n))$
$f(n) = o(g(n))$ se e somente se $g(n) = \omega(f(n))$

Pelo fato dessas propriedades se manterem válidas para notações assintóticas, é possível traçar uma analogia entre a comparação assintótica de duas funções f e g e a comparação de dois números reais a e b:

$f(n) = O(g(n)) \quad \approx \quad a \leq b,$
$f(n) = \Omega(g(n)) \quad \approx \quad a \geq b,$
$f(n) = \Theta(g(n)) \quad \approx \quad a = b,$
$f(n) = o(g(n)) \quad \approx \quad a < b,$
$f(n) = \omega(g(n)) \quad \approx \quad a > b.$

Dizemos que $f(n)$ é ***assintoticamente menor*** que $g(n)$ se $f(n) = o(g(n))$, e que $f(n)$ é ***assintoticamente maior*** que $g(n)$ se $f(n) = \omega(g(n))$.

Contudo, uma propriedade de números reais não é transportada para a notação assintótica:

Tricotomia: Para dois números reais quaisquer a e b, exatamente uma das propriedades a seguir deve ser válida: $a < b$, $a = b$ ou $a > b$.

Embora dois números reais quaisquer possam ser comparados, nem todas as funções são assintoticamente comparáveis. Ou seja, para duas funções $f(n)$ e $g(n)$, pode acontecer que nem $f(n) = O(g(n))$, nem $f(n) = \Omega(g(n))$ seja válida. Por exemplo, as funções n e $n^{1 + \text{sen}\, n}$ não podem ser comparadas utilizando-se a notação assintótica, pois o valor do expoente em $n^{1 + \text{sen}\, n}$ oscila entre 0 e 2, assumindo todos os valores intermediários.

Exercícios

3.1-1

Sejam $f(n)$ e $g(n)$ funções assintoticamente não negativas. Usando a definição básica da notação Θ, prove que $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

3.1-2

Mostre que, para quaisquer constantes reais a e b, onde $b > 0$,

$$(n + a)^b = \Theta(n^b) \ . \qquad (3.2)$$

3.1-3

Explique por que a declaração "O tempo de execução no algoritmo A é no mínimo $O(n^2)$" é isenta de significado.

3.1-4
É verdade que $2^{n+1} = O(2^n)$? É verdade que $2^{2n} = O(2^n)$?

3.1-5
Demonstre o Teorema 3.1.

3.1-6
Prove que o tempo de execução de um algoritmo é $\Theta(g(n))$ se e somente se seu tempo de execução do pior caso é $O(g(n))$ e seu tempo de execução do melhor caso é $\Omega(g(n))$.

3.1-7
Prove que $o(g(n)) \cap \omega(g(n))$ é o conjunto vazio.

3.1-8
Podemos estender nossa notação ao caso de dois parâmetros n e m que podem tender a infinito independentemente a taxas distintas. Para uma dada função $g(n, m)$, denotamos por $O(g(n, m))$ o conjunto de funções

$$O(g(n, m)) = \{f(n, m) : \text{existem constantes positivas } c, n_0 \text{ e } m_0 \text{ tais que } 0 \leq f(n, m) \leq cg(n, m) \text{ para todo } n \geq n_0 \text{ e } m \geq m_0\} .$$

Forneça definições correspondentes para $\Omega(g(n, m))$ e $\Theta(g(n, m))$.

3.2 Notações padrão e funções comuns

Esta seção revê algumas funções e notações matemáticas padrão e explora os relacionamentos entre elas. A seção também ilustra o uso das notações assintóticas.

Monotonicidade

Uma função $f(n)$ é ***monotonicamente crescente*** (ou ***monotonamente crescente***) se $m \leq n$ implica $f(m) \leq f(n)$. De modo semelhante, ela é ***monotonicamente decrescente*** (ou ***monotonamente decrescente***) se $m \leq n$ implica $f(m) \geq f(n)$. Uma função $f(n)$ é ***estritamente crescente*** se $m < n$ implica $f(m) < f(n)$ e ***estritamente decrescente*** se $m < n$ implica $f(m) > f(n)$.

Pisos e tetos

Para qualquer número real x, denotamos o maior inteiro menor que ou igual a x por $\lfloor x \rfloor$ (lê-se "o piso de x") e o menor inteiro maior que ou igual a x por $\lceil x \rceil$ (lê-se "o teto de x"). Para todo x real,

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 . \quad (3.3)$$

Para qualquer inteiro n,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$$

e, para qualquer número real $n \geq 0$ e inteiros $a, b > 0$,

$$\lceil \lceil n/a \rceil /b \rceil = \lceil n/ab \rceil , \quad (3.4)$$

$$\lfloor \lfloor n/a \rfloor /b \rfloor = \lfloor n/ab \rfloor , \quad (3.5)$$

$$\lceil a/b \rceil \leq (a + (b-1))/b, \tag{3.6}$$

$$\lfloor a/b \rfloor \leq (a - (b-1))/b. \tag{3.7}$$

A função piso $f(x) = \lfloor x \rfloor$ é monotonicamente crescente, como também a função teto $f(x) = \lceil x \rceil$.

Aritmética modular

Para qualquer inteiro a e qualquer inteiro positivo n, o valor $a \bmod n$ é o ***resto*** (ou ***resíduo***) do quociente a/n:

$$a \bmod n = a - \lfloor a/n \rfloor n. \tag{3.8}$$

Dada uma noção bem definida do resto da divisão de um inteiro por outro, é conveniente fornecer uma notação especial para indicar a igualdade de restos. Se $(a \bmod n) = (b \bmod n)$, escrevemos $a \equiv b \pmod n$ e dizemos que a é ***equivalente*** a b, módulo n. Em outras palavras, $a \equiv b \pmod n$ se a e b têm o mesmo resto quando divididos por n. De modo equivalente, $a \equiv b \pmod n$ se e somente se n é um divisor de $b - a$. Escrevemos $a \not\equiv b \pmod n$ se a não é equivalente a b, módulo n.

Polinômios

Dado um inteiro não negativo d, um ***polinômio em n de grau d*** é uma função $p(n)$ da forma

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

onde as constantes $a_0, a_1, \ldots, a_d$ são os ***coeficientes*** do polinômio e $a_d \neq 0$. Um polinômio é assintoticamente positivo se e somente se $a_d > 0$. No caso de um polinômio assintoticamente positivo $p(n)$ de grau d, temos $p(n) = \Theta(n^d)$. Para qualquer constante real $a \geq 0$, a função n^a é monotonicamente crescente, e para qualquer constante real $a \leq 0$, a função n^a é monotonicamente decrescente. Dizemos que uma função $f(n)$ é ***polinomialmente limitada*** se $f(n) = O(n^k)$ para alguma constante k.

Exponenciais

Para todos os valores $a \neq 0$, m e n reais, temos as seguintes identidades:

$$\begin{aligned} a^0 &= 1, \\ a^1 &= a, \\ a^{-1} &= 1/a, \\ (a^m)^n &= a^{mn}, \\ (a^m)^n &= (a^n)^m, \\ a^m a^n &= a^{m+n}. \end{aligned}$$

Para todo n e $a \geq 1$, a função a^n é monotonicamente crescente em n. Quando conveniente, consideraremos $0^0 = 1$.

As taxas de crescimento de polinômios e exponenciais podem ser relacionadas pelo fato a seguir. Para todas as constantes reais a e b tais que $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0 , \tag{3.9}$$

da qual podemos concluir que

$$n^b = o(a^n) .$$

Portanto, qualquer função exponencial com uma base estritamente maior que 1 cresce mais rapidamente que qualquer função polinomial.

Usando e para denotar 2,71828 ..., a base da função logaritmo natural, temos para todo x real,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!} , \tag{3.10}$$

onde "!" denota a função fatorial, definida mais adiante nesta seção. Para todo x real, temos a desigualdade

$$e^x \geq 1 + x , \tag{3.11}$$

onde a igualdade se mantém válida somente quando $x = 0$. Quando $|x| \leq 1$, temos a aproximação

$$1 + x \leq e^x \leq 1 + x + x^2 . \tag{3.12}$$

Quando $x \to 0$, a aproximação de e^x por $1 + x$ é bastante boa:

$$e^x = 1 + x + \Theta(x^2) .$$

(Nessa equação, a notação assintótica é usada para descrever o comportamento de limite como $x \to 0$ em lugar de $x \to \infty$.) Temos, para todo x,

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x . \tag{3.13}$$

Logaritmos

Utilizaremos as seguintes notações:

$$\begin{aligned}
\lg n &= \log_2 n && \text{(logaritmo binário)}, \\
\ln n &= \log_e n && \text{(logaritmo natural)}, \\
\lg^k n &= (\lg n)^k && \text{(exponenciação)}, \\
\lg \lg n &= \lg(\lg n) && \text{(composição)}.
\end{aligned}$$

Uma importante convenção notacional que adotaremos é que as *funções logarítmicas se aplicarão apenas ao próximo termo na fórmula*; assim, $\lg n + k$ significará $(\lg n) + k$ e não $\lg(n + k)$. Se mantivermos $b > 1$ constante, então para $n > 0$, a função $\log_b n$ será estritamente crescente.

Para todo $a > 0$, $b > 0$, $c > 0$ e n real,

$$a = b^{\log_b a} , \tag{3.14}$$

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

$$\log_b a^n = n \log_b a ,$$

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

$$\log_b(1/a) = -\log_b a , \tag{3.15}$$

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

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

onde, em cada equação anterior, as bases de logaritmos não são iguais a 1.

Pela equação (3.14), a mudança da base de um logaritmo de uma constante para outra só altera o valor do logaritmo por um fator constante, e assim usaremos com freqüência a notação "$\lg n$" quando não nos importarmos com fatores constantes, como na notação O. Os cientistas da computação consideram 2 a base mais natural para logaritmos, porque muitos algoritmos e estruturas de dados envolvem a divisão de um problema em duas partes.

Existe uma expansão de série simples para $\ln(1 + x)$ quando $|x| < 1$:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots .$$

Também temos as seguintes desigualdades para $x > -1$:

$$\frac{x}{1+x} \leq \ln(1+x) \leq x , \tag{3.16}$$

onde a igualdade é válida somente para $x = 0$.

Dizemos que uma função $f(n)$ é ***polilogaritmicamente limitada*** se $f(n) = O(\lg^k n)$ para alguma constante k. Podemos relacionar o crescimento de polinômios e polilogaritmos substituindo n por $\lg n$ e a por 2^a na equação (3.9), resultando em:

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0 .$$

A partir desse limite, podemos concluir que

$$\lg^b n = o(n^a)$$

para qualquer constante $a > 0$. Desse modo, qualquer função polinomial positiva cresce mais rapidamente que qualquer função polilogarítmica.

Fatoriais

A notação $n!$ (lê-se "n fatorial") é definida para inteiros $n \geq 0$ como

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

Então, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

Um limite superior fraco na função fatorial é $n! \leq n^n$, pois cada um dos n termos no produto do fatorial é no máximo n. A ***aproximação de Stirling***,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right), \tag{3.17}$$

onde e é a base do logaritmo natural, nos dá um limite superior mais restrito, e um limite inferior também. Pode-se demonstrar que (consulte o Exercício 3.2-3)

$$\begin{aligned} n! &= o(n^n), \\ n! &= \omega(2^n), \\ \lg(n!) &= \Theta(n \lg n), \end{aligned} \tag{3.18}$$

onde a aproximação de Stirling é útil na demonstração da equação (3.18). A equação a seguir também é válida para todo $n \geq 1$:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha n} \tag{3.19}$$

onde

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}. \tag{3.20}$$

Iteração funcional

Usamos a notação $f^{(i)}(n)$ para denotar a função $f(n)$ aplicada iterativamente i vezes a um valor inicial n. Formalmente, seja $f(n)$ uma função sobre os reais. Para inteiros não negativos i, definimos recursivamente:

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

Por exemplo, se $f(n) = 2n$, então $f^{(i)}(n) = 2^i n$.

A função logaritmo repetido

Usamos a notação $\lg^* n$ (lê-se "log asterisco de n") para denotar o logaritmo repetido, que é definido como a seguir. Seja $\lg^{(i)} n$ definida da maneira anterior, com $f(n)$ $\lg n$. Como o logaritmo de um número não positivo é indefinido, $\lg^{(i)} n$ só é definido se $\lg^{(i-1)} n > 0$. Certifique-se de distinguir $\lg^{(i)} n$ (a função logaritmo aplicada i vezes em sucessão, começando com o argumento n) de $\lg^i n$ (o logaritmo de n elevado à i-ésima potência). A função logaritmo repetido é definida como

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}.$$

O logaritmo repetido é uma função que cresce *muito* lentamente:

$$\begin{aligned}
\lg^* 2 &= 1 , \\
\lg^* 4 &= 2 , \\
\lg^* 16 &= 3 , \\
\lg^* 65536 &= 4 , \\
\lg^* (2^{65536}) &= 5 .
\end{aligned}$$

Tendo em vista que o número de átomos no universo visível é estimado em cerca 10^{80}, que é muito menor que 2^{65536}, raramente encontraremos uma entrada de tamanho n tal que $\lg^* n > 5$.

Números de Fibonacci

Os ***números de Fibonacci*** são definidos pela seguinte recorrência:

$$\begin{aligned}
F_0 &= 0 , \\
F_1 &= 1 , \\
F_i &= F_{i-1} + F_{i-2} \text{ para } i \geq 2.
\end{aligned} \tag{3.21}$$

Portanto, cada número de Fibonacci é a soma dos dois números anteriores, produzindo a seqüência

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots .$$

Os números de Fibonacci estão relacionados com a ***razão áurea*** ϕ e a seu conjugado $\hat{\phi}$, que são dados pelas seguintes fórmulas:

$$\begin{aligned}
\phi &= \frac{1+\sqrt{5}}{2} \\
&= 1{,}61803\ldots , \\
\hat{\phi} &= \frac{1-\sqrt{5}}{2} \\
&= -0{,}61803\ldots .
\end{aligned} \tag{3.22}$$

Especificamente, temos

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} \tag{3.23}$$

que pode ser demonstrada por indução (Exercício 3.2-6). Como $|\hat{\phi}| < 1$, temos $|\hat{\phi}|/\sqrt{5} < 1/\sqrt{5} < 1/2$, de modo que o i-ésimo número de Fibonacci F_i é igual a $\phi^i/\sqrt{5}$ arredondado para o inteiro mais próximo. Desse modo, os números de Fibonacci crescem exponencialmente.

Exercícios

3.2-1

Mostre que, se $f(n)$ e $g(n)$ são funções monotonicamente crescentes, então também o são as funções $f(n) + g(n)$ e $f(g(n))$, e se $f(n)$ e $g(n)$ são além disso não negativas, então $f(n) \cdot g(n)$ é monotonicamente crescente.

3.2-2

Prove a equação (3.15).

3.2-3

Prove a equação (3.18). Prove também que $n! = \omega(2^n)$ e que $n! = o(n^n)$.

3.2-4 ★

A função $\lceil \lg n \rceil!$ é polinomialmente limitada? A função $\lceil \lg \lg n \rceil!$ é polinomialmente limitada?

3.2-5 ★

Qual é assintoticamente maior: $\lg(\lg^* n)$ ou $\lg^*(\lg n)$?

3.2-6

Prove por indução que o i-ésimo número de Fibonacci satisfaz à igualdade

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}},$$

onde ϕ é a razão áurea e $\hat{\phi}$ é seu conjugado.

3.2-7

Prove que, para $i \geq 0$, o $(i + 2)$-ésimo número de Fibonacci satisfaz a $F_{i+2} \geq \phi^i$.

Problemas

3-1 Comportamento assintótico de polinômios

Seja

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

onde $a_d > 0$, um polinômio de grau d em n, e seja k uma constante. Use as definições das notações assintóticas para provar as propriedades a seguir.

a. Se $k \geq d$, então $p(n) = O(n^k)$.

b. Se $k \leq d$, então $p(n) = \Omega(n^k)$.

c. Se $k = d$, então $p(n) = \Theta(n^k)$.

d. Se $k > d$, então $p(n) = o(n^k)$.

e. Se $k < d$, então $p(n) = \omega(n^k)$.

3-2 Crescimentos assintóticos relativos

Indique, para cada par de expressões (A, B) na tabela a seguir, se A é O, o, Ω, ω ou Θ de B. Suponha que $k \geq 1$, $\epsilon > 0$ e $c > 1$ são constantes. Sua resposta deve estar na forma da tabela, com "sim" ou "não" escrito em cada retângulo.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg c}$	$c^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

3-3 Ordenação por taxas de crescimento assintóticas

a. Ordene as funções a seguir por ordem de crescimento; ou seja, encontre um arranjo $g_1, g_2, \ldots, g_{30}$ das funções que satisfazem a $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \ldots, g_{29} = \Omega(g_{30})$. Particione sua lista em classes de equivalência tais que $f(n)$ e $g(n)$ estejam na mesma classe se e somente se $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \,.\, 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	$2n$	$n \lg n$	2^{2n+1}

b. Dê um exemplo de uma única função não negativa $f(n)$ tal que, para todas as funções $g_i(n)$ da parte (a), $f(n)$ não seja nem $O(g_i(n))$, nem $\Omega(g_i(n))$.

3-4 Propriedades da notação assintótica

Sejam $f(n)$ e $g(n)$ funções assintoticamente positivas. Prove ou conteste cada uma das seguintes conjecturas.

a. $f(n) = O(g(n))$ implica $g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

c. $f(n) = O(g(n))$ implica $\lg(f(n)) = O(\lg(g(n)))$, onde $\lg(g(n)) \geq 1$ e $f(n) \geq 1$ para todo n suficientemente grande.

d. $f(n) = O(g(n))$ implica $2^{f(n)} = O(2^{g(n)})$.

e. $f(n) = O((f(n))^2)$.

f. $f(n) = O(g(n))$ implica $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

3-5 Variações sobre O e Ω

Alguns autores definem Ω de um modo ligeiramente diferente do modo como nós definimos; vamos usar $\overset{\infty}{\Omega}$ (lê-se "ômega infinito") para essa definição alternativa. Dizemos que $f(n) = \overset{\infty}{\Omega}(g(n))$ se existe uma constante positiva c tal que $f(n) \geq cg(n) \geq 0$ para infinitamente muitos inteiros n.

a. Mostre que, para duas funções quaisquer $f(n)$ e $g(n)$ que são assintoticamente não negativas, $f(n) = O(g(n))$ ou $f(n) = \overset{\infty}{\Omega}(g(n))$ ou ambas, enquanto isso não é verdade se usamos Ω em lugar de $\overset{\infty}{\Omega}$.

b. Descreva as vantagens e as desvantagens potenciais de se usar $\overset{\infty}{\Omega}$ em vez de Ω para caracterizar os tempos de execução de programas.

Alguns autores também definem O de um modo ligeiramente diferente; vamos usar O' para a definição alternativa. Dizemos que $f(n) = O'(g(n))$ se e somente se $|f(n)| = O(g(n))$.

c. O que acontece para cada sentido de "se e somente se" no Teorema 3.1 se substituirmos O por O', mas ainda usarmos Ω?

Alguns autores definem $\widetilde{O}$ (lê-se "'o' suave") para indicar O com fatores logarítmicos ignorados:

$$\widetilde{O}(g(n)) = \{f(n) : \text{existem constantes positivas } c, k \text{ e } n_0 \text{ tais que } 0 \leq f(n) \leq cg(n) \lg^k(n) \text{ para todo } n \geq n_0\}.$$

d. Defina $\widetilde{\Omega}$ e $\widetilde{\Theta}$ de maneira semelhante. Prove a analogia correspondente ao Teorema 3.1.

3-6 Funções repetidas

O operador de iteração * usado na função lg* pode ser aplicado a qualquer função monotonicamente crescente $f(n)$ sobre os reais. Para uma dada constante $c \in \mathbf{R}$, definimos a função repetida (ou iterada) f_c^* por

$$f_c^*(n) = \min\{i \geq 0 : f^{(i)}(n) \leq c\},$$

que não necessita ser bem definida em todos os casos. Em outras palavras, a quantidade $f_c^*(n)$ é o número de aplicações repetidas da função f necessárias para reduzir seu argumento a c ou menos.

Para cada uma das funções $f(n)$ e constantes c a seguir, forneça um limite tão restrito quanto possível sobre $f_c^*(n)$.

	$f(n)$	c	$f_c^*(n)$
a.	$n-1$	0	
b.	$\lg n$	1	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	
h.	$n/\lg n$	2	

Notas do capítulo

Knuth [182] traça a origem da notação O até um texto de teoria dos números escrito por P. Bachmann em 1892. A notação o foi criada por E. Landau em 1909, para sua descrição da distribuição de números primos. As notações Ω e Θ foram defendidas por Knuth [186] para corrigir a prática popular, mas tecnicamente ruim, de se usar na literatura a notação O para limites superiores e inferiores. Muitas pessoas continuam a usar a notação O onde a notação Θ é mais precisa tecnicamente. Uma discussão adicional da história e do desenvolvimento de notações assintóticas pode ser encontrada em Knuth [182, 186] e em Brassard e Bratley [46].

Nem todos os autores definem as notações assintóticas do mesmo modo, embora as várias definições concordem na maioria das situações comuns. Algumas das definições alternativas envolvem funções que não são assintoticamente não negativas, desde que seus valores absolutos sejam adequadamente limitados.

A equação (3.19) se deve a Robbins [260]. Outras propriedades de funções matemáticas elementares podem ser encontradas em qualquer bom manual de referência de matemática, como Abramowitz e Stegun [1] ou Zwillinger [320], ou em um livro de cálculo, como Apostol [18] ou Thomas e Finney [296]. Knuth [182] e também Graham, Knuth e Patashnik [132] contêm uma grande quantidade de material sobre matemática discreta, como a que se utiliza em ciência da computação.

Capítulo 4

Recorrências

Como observamos na Seção 2.3.2, quando um algoritmo contém uma chamada recursiva a ele próprio, seu tempo de execução pode freqüentemente ser descrito por uma recorrência. Uma ***recorrência*** é uma equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores. Por exemplo, vimos na Seção 2.3.2 que o tempo de execução do pior caso $T(n)$ do procedimento MERGE-SORT poderia ser descrito pela recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1, \\ 2T(n/2) + \Theta(n) & \text{se } n > 1. \end{cases} \tag{4.1}$$

cuja solução se afirmava ser $T(n) = \Theta(n \lg n)$.

Este capítulo oferece três métodos para resolver recorrências – ou seja, para obter limites assintóticos "Θ" ou "O" sobre a solução. No ***método de substituição***, supomos um limite hipotético, e depois usamos a indução matemática para provar que nossa suposição era correta. O ***método de árvore de recursão*** converte a recorrência em uma árvore cujos nós representam os custos envolvidos em diversos níveis da recursão; usamos técnicas para limitar somatórios com a finalidade de solucionar a recorrência. O ***método mestre*** fornece limites para recorrências da forma

$$T(n) = aT(n/b) + f(n),$$

onde $a \geq 1$, $b > 1$ e $f(n)$ é uma função dada; o método requer memorização de três casos mas, depois que você o fizer, será fácil descobrir limites assintóticos para muitas recorrências simples.

Detalhes técnicos

Na prática, negligenciamos certos detalhes técnicos quando enunciamos e resolvemos recorrências. Um bom exemplo de um detalhe que freqüentemente é ignorado é a suposição de argumentos inteiros para funções. Normalmente, o tempo de execução $T(n)$ de um algoritmo só é definido quando n é um inteiro, pois, para a maior parte dos algoritmos, o tamanho da entrada é sempre um inteiro. Por exemplo, a recorrência que descreve o tempo de execução do pior caso de MERGE-SORT é na realidade

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{se } n > 1. \end{cases} \tag{4.2}$$

As condições limite representam outra classe de detalhes que em geral ignoramos. Tendo em vista que o tempo de execução de um algoritmo sobre uma entrada de dimensões constantes é uma constante, as recorrências que surgem dos tempos de execução de algoritmos geralmente têm $T(n) = \Theta(1)$ para n suficientemente pequeno. Em conseqüência disso, por conveniência, em geral omitiremos declarações sobre as condições limite de recorrências e iremos supor que $T(n)$ é constante para n pequeno. Por exemplo, normalmente enunciamos a recorrência (4.1) como

$$T(n) = 2T(n/2) + \Theta(n) , \tag{4.3}$$

sem fornecer explicitamente valores para n pequeno. A razão é que, embora a mudança no valor de $T(1)$ altere a solução para a recorrência, normalmente a solução não muda por mais de um fator constante, e assim a ordem de crescimento não é alterada.

Quando enunciamos e resolvemos recorrências, com freqüência omitimos pisos, tetos e condições limite. Seguimos em frente sem esses detalhes, e mais tarde definimos se eles têm importância ou não. Em geral, eles não têm importância, mas é importante saber quando têm. A experiência ajuda, e também alguns teoremas declarando que esses detalhes não afetam os limites assintóticos de muitas recorrências encontradas na análise de algoritmos (ver Teorema 4.1). Porém, neste capítulo, examinaremos alguns desses detalhes para mostrar os ótimos métodos de solução de pontos de recorrência.

4.1 O método de substituição

O método de substituição para resolver recorrências envolve duas etapas:

1. Pressupor a forma da solução.
2. Usar a indução matemática para encontrar as constantes e mostrar que a solução funciona.

O nome vem da substituição da resposta pressuposta para a função quando a hipótese indutiva é aplicada a valores menores. Esse método é eficiente, mas obviamente só pode ser aplicado em casos nos quais é fácil pressupor a forma da resposta.

O método de substituição pode ser usado para estabelecer limites superiores ou inferiores sobre uma recorrência. Como exemplo, vamos determinar um limite superior sobre a recorrência

$$T(n) = 2T(\lfloor n/2 \rfloor) + n , \tag{4.4}$$

que é semelhante às recorrências (4.2) e (4.3). Supomos que a solução é $T(n) = O(n \lg n)$. Nosso método é provar que $T(n) \le cn \lg n$ para uma escolha apropriada da constante $c > 0$. Começamos supondo que esse limite se mantém válido para $\lfloor n/2 \rfloor$, ou seja, que $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. A substituição na recorrência produz

$$\begin{aligned} T(n) &\le 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\le cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n , \end{aligned}$$

onde o último passo é válido desde que $c \ge 1$.

Agora, a indução matemática exige que mostremos que nossa solução se mantém válida para as condições limite. Normalmente, fazemos isso mostrando que as condições limite são adequadas para casos básicos da prova indutiva. No caso da recorrência (4.4), devemos mostrar que podemos escolher uma constante c grande o suficiente para que o limite $T(n) \leq cn \lg n$ também funcione para as condições limite. Às vezes, essa exigência pode levar a problemas. Vamos supor, como argumento, que $T(1) = 1$ seja a única condição limite da recorrência. Então, para $n = 1$, o limite $T(n) \leq cn \lg n$ produz $T(1) \leq c\ 1 \lg 1 = 0$, o que está em desacordo com $T(1) = 1$. Conseqüentemente, o caso básico de nossa prova indutiva deixa de ser válido.

Essa dificuldade para provar uma hipótese indutiva para uma condição limite específica pode ser facilmente contornada. Por exemplo, na recorrência (4.4), tiramos proveito do fato de que a notação assintótica só exige que demonstremos que $T(n) \leq cn \lg n$ para $n \geq n_0$, onde n_0 é uma constante de nossa escolha. A idéia é remover a difícil condição limite $T(1) = 1$ da consideração na prova indutiva. Observe que, para $n > 3$, a recorrência não depende diretamente de $T(1)$. Desse modo, podemos substituir $T(1)$ por $T(2)$ e $T(3)$ como os casos básicos na prova indutiva, deixando $n_0 = 2$. Observe que fazemos uma distinção entre o caso básico da recorrência ($n = 1$) e os casos básicos da prova indutiva ($n = 2$ e $n = 3$). Da recorrência, derivamos $T(2) = 4$ e $T(3) = 5$. A prova indutiva de que $T(n) \leq cn \lg n$ para alguma constante $c \geq 1$ pode agora ser completada escolhendo-se um valor de c grande o bastante para que $T(2) \leq c\ 2 \lg 2$ e $T(3) \leq c\ 3 \lg 3$. Como observamos, qualquer valor de $c \geq 2$ é suficiente para que os casos básicos de $n = 2$ e $n = 3$ sejam válidos. Para a maioria das recorrências que examinaremos, é simples estender as condições limite para fazer a hipótese indutiva funcionar para n pequeno.

Como fazer um bom palpite

Infelizmente, não há nenhum modo geral de adivinhar as soluções corretas para recorrências. Pressupor uma solução exige experiência e, ocasionalmente, criatividade. Entretanto, por sorte, existem algumas heurísticas que podem ajudá-lo a se tornar um bom criador de suposições. Você também poderá usar árvores de recursão, que veremos na Seção 4.2, para gerar boas hipóteses.

Se uma recorrência for semelhante a uma que você já tenha visto antes, então será razoável supor uma solução semelhante. Como exemplo, considere a recorrência

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n ,$$

que parece difícil devido ao "17" acrescentado ao argumento de T no lado direito. Intuitivamente, porém, esse termo adicional não pode afetar de maneira substancial a solução para a recorrência. Quando n é grande, a diferença entre $T(\lfloor n/2 \rfloor)$ e $T(\lfloor n/2 \rfloor + 17)$ não é tão grande: ambos os termos cortam n quase uniformemente pela metade. Em conseqüência disso, fazemos a suposição de que $T(n) = O(n \lg n)$, que você pode verificar como correto usando o método de substituição (ver Exercício 4.1-5).

Outro caminho para fazer uma boa suposição é provar informalmente limites superiores e inferiores sobre a recorrência, e então reduzir o intervalo de incerteza. Por exemplo, podemos começar com o limite inferior $T(n) = \Omega(n)$ para a recorrência (4.4), pois temos o termo n na recorrência, e podemos demonstrar um limite superior inicial $T(n) = O(n^2)$. Então, podemos diminuir gradualmente o limite superior e elevar o limite inferior, até convergirmos sobre a solução correta, assintoticamente restrita, $T(n) = \Theta(n \lg n)$.

Sutilezas

Há momentos em que é possível fazer uma suposição correta em um limite assintótico sobre a solução de uma recorrência, mas, de algum modo, a matemática não parece funcionar na indu-

ção. Em geral, o problema é que a hipótese indutiva não é forte o bastante para provar o limite detalhado. Quando você chegar a um nó como esse, revisar a suposição subtraindo um termo de mais baixa ordem freqüentemente permitirá que a matemática funcione.

Considere a recorrência

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 .$$

Supomos por hipótese que a solução seja $O(n)$ e tentamos mostrar que $T(n) \le cn$ para uma escolha apropriada da constante c. Substituindo nossa suposição na recorrência, obtemos

$$\begin{aligned} T(n) &\le c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 , \end{aligned}$$

o que não implica $T(n) \le cn$ para qualquer escolha de c. É uma tentação experimentar um palpite maior, digamos $T(n) = O(n^2)$, o que poderia funcionar; porém, de fato, nossa suposição de que a solução é $T(n) = O(n)$ é correta. No entanto, para mostrar isso, devemos criar uma hipótese indutiva mais forte.

Intuitivamente, nossa suposição é quase correta: a única diferença é a constante 1, um termo de mais baixa ordem. Apesar disso, a indução matemática não funciona, a menos que provemos a forma exata da hipótese indutiva. Superamos nossa dificuldade *subtraindo* um termo de mais baixa ordem da nossa suposição anterior. Nossa nova suposição é $T(n) \le cn - b$, onde $b \ge 0$ é constante. Agora temos

$$\begin{aligned} T(n) &\le (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\le cn - b , \end{aligned}$$

desde que $b \ge 1$. Como antes, a constante c deve ser escolhida com um valor grande o suficiente para tratar as condições limite.

A maioria das pessoas acha antiintuitiva a idéia de subtrair um termo de mais baixa ordem. Afinal, se a matemática não funciona, não deveríamos estar aumentando nossa suposição? A chave para entender esse passo é lembrar que estamos usando a indução matemática: podemos provar algo mais forte para um determinado valor, supondo algo mais forte para valores menores.

Como evitar armadilhas

É fácil errar na utilização da notação assintótica. Por exemplo, na recorrência (4.4), podemos "provar" falsamente que $T(n) = O(n)$, supondo $T(n) \le cn$, e então argumentando que

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor + n \\ &\le cn + n \\ &= O(n) , \qquad \Leftarrow \textit{errado!!} \end{aligned}$$

pois c é uma constante. O erro é que não provamos a *forma exata* da hipótese indutiva, ou seja, que $T(n) \le cn$.

Como trocar variáveis

Às vezes, um pouco de manipulação algébrica pode tornar uma recorrência desconhecida semelhante a uma que você já viu antes. Como exemplo, considere a recorrência

$$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n \ ,$$

que parece difícil. Entretanto, podemos simplificar essa recorrência com uma troca de variáveis. Por conveniência, não nos preocuparemos em arredondar valores, como $\sqrt{n}$, para inteiros. Renomear $m = \lg n$ produz

$$T(2^m) = 2T(2^{m/2}) + m \ .$$

Agora podemos renomear $S(m) = T(2^m)$ para produzir a nova recorrência

$$S(m) = 2S(m/2) + m \ ,$$

que é muito semelhante à recorrência (4.4). De fato, essa nova recorrência tem a mesma solução: $S(m) = O(m \lg m)$. Trocando de volta de $S(m)$ para $T(n)$, obtemos $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

Exercícios

4.1-1

Mostre que a solução de $T(n) = T(\lceil n/2 \rceil) + 1$ é $O(\lg n)$.

4.1-2

Vimos que a solução de $T(n) = 2T(\lfloor n/2 \rfloor) + n$ é $O(n \lg n)$. Mostre que a solução dessa recorrência também é $\Omega(n \lg n)$. Conclua que a solução é $\Theta(n \lg n)$.

4.1-3

Mostre que, supondo uma hipótese indutiva diferente, podemos superar a dificuldade com a condição limite $T(1) = 1$ para a recorrência (4.4), sem ajustar as condições limite para a prova indutiva.

4.1-4

Mostre que $\Theta(n \lg n)$ é a solução para a recorrência "exata" (4.2) para a ordenação por intercalação.

4.1-5

Mostre que a solução para $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ é $O(n \lg n)$.

4.1-6

Resolva a recorrência $T(n) = 2T(\sqrt{n})$, fazendo uma troca de variáveis. Sua solução deve ser assintoticamente restrita. Não se preocupe em saber se os valores são integrais.

4.2 O método de árvore de recursão

Embora o método de substituição possa fornecer uma prova sucinta de que uma solução para uma recorrência é correta, às vezes é difícil apresentar uma boa suposição. Traçar uma árvore de recursão, como fizemos em nossa análise da recorrência de ordenação por intercalação na Seção 2.3.2, é um caminho direto para se criar uma boa suposição. Em uma ***árvore de recursão***, cada nó representa o custo de um único subproblema em algum lugar no conjunto de invocações de funções recursivas. Somamos os custos dentro de cada nível da árvore para obter um conjunto de custos por nível, e então somamos todos os custos por nível para determinar o custo total de todos os níveis da recursão. As árvores de recursão são particularmente úteis quando a recorrência descreve o tempo de execução de um algoritmo de dividir e conquistar.

Uma árvore de recursão é mais bem usada para gerar uma boa suposição, que é então verificada pelo método de substituição. Quando utiliza uma árvore de recursão para gerar uma boa suposição, você freqüentemente pode tolerar uma pequena quantidade de "sujeira", pois irá verificar sua suposição mais tarde. Porém, se for muito cuidadoso ao criar uma árvore de recursão e somar os custos, você poderá usar a árvore de recursão como uma prova direta de uma solução para uma recorrência. Nesta seção, usaremos árvores de recursão para gerar boas suposições e, na Seção 4.4, utilizaremos árvores de recursão diretamente para provar o teorema que forma a base do método mestre.

Por exemplo, vamos ver como uma árvore de recursão forneceria uma boa suposição para a recorrência $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Começamos concentrando nossa atenção em encontrar um limite superior para a solução. Considerando que sabemos que pisos e tetos são normalmente insatisfatórios para resolver recorrências (aqui está um exemplo de sujeira que podemos tolerar), criamos uma árvore de recursão para a recorrência $T(n) = 3T(n/4) + cn^2$, tendo estabelecido o coeficiente constante implícito $c > 0$.

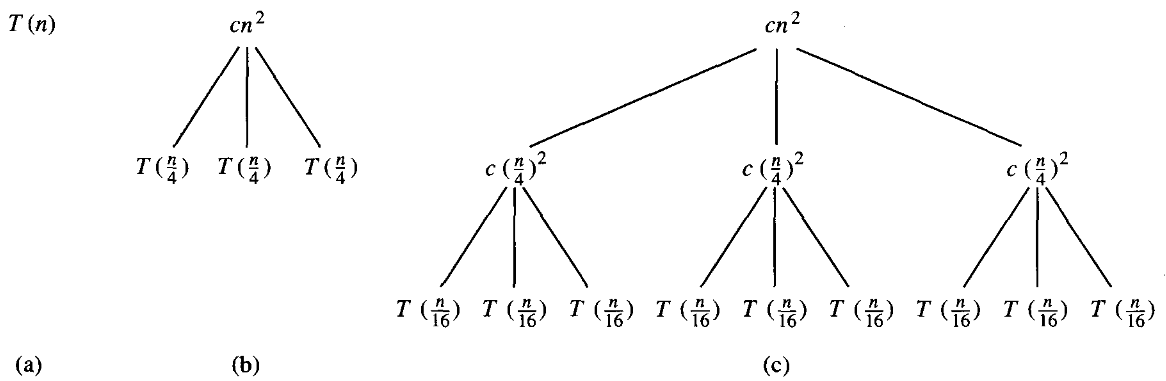
A Figura 4.1 mostra a derivação da árvore de recursão para $T(n) = 3T(n/4) + cn^2$. Por conveniência, supomos que n é uma potência exata de 4 (outro exemplo de sujeira tolerável). A parte (a) da figura mostra $T(n)$ que, na parte (b), foi expandida até uma árvore equivalente representando a recorrência. O termo cn^2 na raiz representa o custo no nível de recursão superior, e as três subárvores da raiz representam os custos resultantes dos subproblemas de tamanho $n/4$. A parte (c) mostra esse processo levado um passo adiante pela expansão de cada nó com custo $T(n/4)$ da parte (b). O custo para cada um dos três filhos da raiz é $c(n/4)^2$. Continuamos a expandir cada nó na árvore, desmembrando-o em suas partes constituintes conforme determinado pela recorrência.

Tendo em vista que os tamanhos de subproblemas diminuem à medida que nos afastamos da raiz, eventualmente temos de alcançar uma condição limite. A que distância da raiz nós a encontramos? O tamanho do subproblema para um nó na profundidade i é $n/4^i$. Desse modo, o tamanho do subproblema chega a $n = 1$ quando $n/4^i = 1$ ou, de modo equivalente, quando $i = \log_4 n$. Assim, a árvore tem $\log_4 n + 1$ níveis $(0, 1, 2, ..., \log_4 n)$.

Em seguida, determinamos o custo em cada nível da árvore. Cada nível tem três vezes mais nós que o nível acima dele, e assim o número de nós na profundidade i é 3^i. Como os tamanhos de subproblemas se reduzem por um fator de 4 para cada nível que descemos a partir da raiz, cada nó na profundidade i, para $i = 0, 1, 2, ..., \log_4 n - 1$, tem o custo de $c(n/4^i)^2$. Multiplicando, vemos que o custo total sobre todos os nós na profundidade i, para $i = 0, 1, 2, ..., \log_4 n - 1$, é $3^i c(n/4^i)^2 = (3/16)^i cn^2$. O último nível, na profundidade $\log_4 n$, tem $3^{\log_4 n} = n^{\log_4 3}$ nós, cada qual contribuindo com o custo $T(1)$, para um custo total de $n^{\log_4 3}T(1)$, que é $\Theta(n^{\log_4 3})$.

Agora somamos os custos sobre todos os níveis para determinar o custo correspondente à árvore inteira:

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})
\end{aligned}
$$



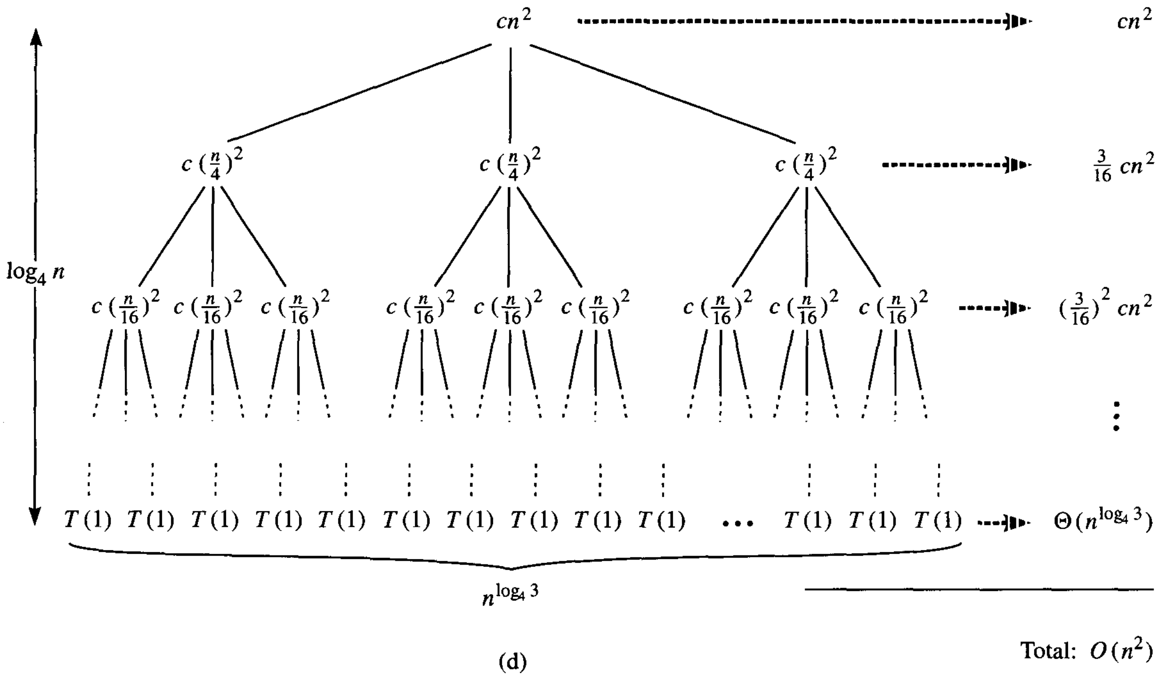


FIGURA 4.1 A construção de uma árvore de recursão para a recorrência $T(n) = 3T(n/4) + cn^2$. A parte **(a)** mostra $T(n)$, que é progressivamente expandido nas partes b a d para formar a árvore de recursão. A árvore completamente expandida da parte (d) tem altura $\log_4 n$ (ela tem $\log_4 n + 1$ níveis)

Essa última fórmula parece um pouco confusa até percebermos que é possível mais uma vez tirar proveito de pequenas porções de sujeira e usar uma série geométrica decrescente infinita como limite superior. Voltando um passo atrás e aplicando a equação (A.6), temos

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) .
\end{aligned}
$$

Desse modo, derivamos uma suposição de $T(n) = O(n^2)$ para nossa recorrência original $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Nesse exemplo, os coeficientes de cn^2 formam uma série geométrica decrescente e, pela equação (A.6), a soma desses coeficientes é limitada na parte superior pela constante 16/13. Tendo em vista que a contribuição da raiz para o custo total é cn^2, a raiz contribui com uma fração constante do custo total. Em outras palavras, o custo total da árvore é dominado pelo custo da raiz.

De fato, se $O(n^2)$ é realmente um limite superior para a recorrência (como verificaremos em breve), então ele deve ser um limite restrito. Por quê? A primeira chamada recursiva contribui com o custo $\Theta(n^2)$, e então $\Omega(n^2)$ deve ser um limite inferior para a recorrência.

Agora podemos usar o método de substituição para verificar que nossa suposição era correta, isto é, $T(n) = O(n^2)$ é um limite superior para a recorrência $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. Queremos mostrar que $T(n) \leq dn^2$ para alguma constante $d > 0$. Usando a mesma constante $c > 0$ de antes, temos

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d\,(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2 , \end{aligned}$$

onde a última etapa é válida desde que $d \geq (16/13)c$.

Como outro exemplo mais complicado, a Figura 4.2 mostra a árvore de recursão para

$$T(n) = T(n/3) + T(2n/3) + O(n) .$$

(Novamente, omitimos as funções piso e teto por simplicidade.) Como antes, c representa o fator constante no termo $O(n)$. Quando somamos os valores em todos os níveis da árvore de recursão, obtemos um valor cn para cada nível. O caminho mais longo da raiz até uma folha é $n \to (2/3)n \to (2/3)^2 n \to \cdots \to 1$. Tendo em vista que $(2/3)^k n = 1$ quando $k = \log_{3/2} n$, a altura da árvore é $\log_{3/2} n$.

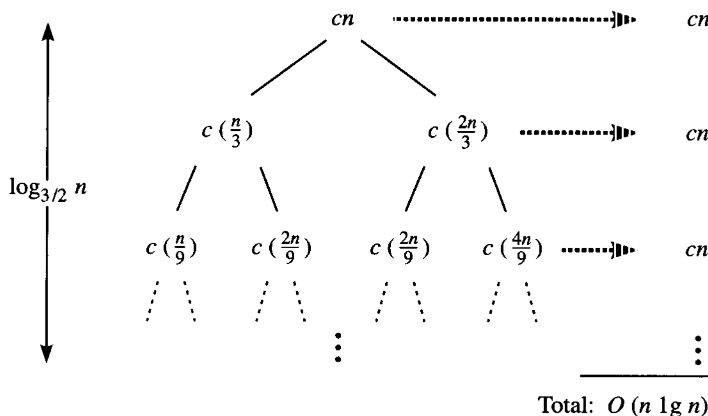


FIGURA 4.2 Uma árvore de recursão para a recorrência $T(n) = T(n/3) + T(2n/3) + cn$

Intuitivamente, esperamos que a solução para a recorrência seja no máximo o número de níveis vezes o custo de cada nível, ou $O(cn \log_{3/2} n) = O(n \lg n)$. O custo total é distribuído de

modo uniforme ao longo dos níveis da árvore de recursão. Existe uma complicação aqui: ainda temos de considerar o custo das folhas. Se essa árvore de recursão fosse uma árvore binária completa de altura $\log_{3/2} n$, haveria $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ folhas. Como o custo de cada folha é uma constante, o custo total de todas as folhas seria então $\Theta(n^{\log_{3/2} 2})$, que é $\omega(n \lg n)$. Contudo, essa árvore de recursão não é uma árvore binária completa, e assim ela tem menos de $n^{\log_{3/2} 2}$ folhas. Além disso, à medida que descemos a partir da raiz, mais e mais nós internos estão ausentes. Conseqüentemente, nem todos os níveis contribuem com um custo exatamente igual a cn; os níveis em direção à parte inferior contribuem menos. Poderíamos desenvolver uma contabilidade precisa de todos os custos, mas lembre-se de que estamos apenas tentando apresentar uma suposição para usar no método de substituição. Vamos tolerar a sujeira e tentar mostrar que uma suposição $O(n \lg n)$ para o limite superior é correta.

Na verdade, podemos usar o método de substituição para verificar aquele $O(n \lg n)$ é um limite superior para a solução da recorrência. Mostramos que $T(n) \le dn \lg n$, onde d é uma constante positiva apropriada. Temos

$$\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3) \lg(n/3) + d(2n/3) \lg(2n/3) + cn \\
&= (d(n/3) \lg n - d(n/3) \lg 3) \\
&\qquad + (d(2n/3) \lg n = d(2n/3) \lg(3/2) + cn \\
&= dn \lg n - d((n/3) \lg 3 + (2n/3) \lg(3/2)) + cn \\
&= dn \lg n - d((n/3) \lg 3 + (2n/3) \lg 3 - (2n/3) \lg 2) + cn \\
&= dn \lg n - dn(\lg 3 - 2/3) + cn \\
&\le dn \lg n \,,
\end{aligned}$$

desde que $d \ge c/(\lg 3 - (2/3))$. Desse modo, não tivemos de executar uma contabilidade de custos mais precisa na árvore de recursão.

Exercícios

4.2-1
Use uma árvore de recursão para determinar um bom limite superior assintótico na recorrência $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Use o método de substituição para verificar sua resposta.

4.2-2
Demonstre que a solução para a recorrência $T(n) = T(n/3) + T(2n/3) + cn$, onde c é uma constante, é $\Omega(n \lg n)$, apelando para uma árvore de recursão.

4.2-3
Trace a árvore de recursão para $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, onde c é uma constante, e forneça um limite assintótico restrito sobre sua solução. Verifique o limite pelo método de substituição.

4.2-4
Use uma árvore de recursão com o objetivo de fornecer uma solução assintoticamente restrita para a recorrência $T(n) = T(n - a) + T(a) + cn$, onde $a \ge 1$ e $c > 0$ são constantes.

4.2-5
Use uma árvore de recursão para fornecer uma solução assintoticamente restrita para a recorrência $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$, onde α é uma constante no intervalo $0 < \alpha < 1$ e $c > 0$ também é uma constante.

4.3 O método mestre

O método mestre fornece um processo de "livro de receitas" para resolver recorrências da forma

$$T(n) = aT(n/b) + f(n) , \tag{4.5}$$

onde $a \geq 1$ e $b > 1$ são constantes e $f(n)$ é uma função assintoticamente positiva. O método mestre exige a memorização de três casos, mas, daí em diante, a solução de muitas recorrências pode ser descoberta com grande facilidade, freqüentemente sem lápis e papel.

A recorrência (4.5) descreve o tempo de execução de um algoritmo que divide um problema de tamanho n em a subproblemas, cada um do tamanho n/b, onde a e b são constantes positivas. Os a subproblemas são resolvidos recursivamente, cada um no tempo $T(n/b)$. O custo de dividir o problema e combinar os resultados dos subproblemas é descrito pela função $f(n)$. (Ou seja, usando a notação da Seção 2.3.2, $f(n) = D(n) + C(n)$.) Por exemplo, a recorrência que surge do procedimento MERGE-SORT tem $a = 2$, $b = 2$ e $f(n) = \Theta(n)$.

Como uma questão de correção técnica, na realidade a recorrência não está bem definida, porque n/b não poderia ser um inteiro. Porém, a substituição de cada um dos a termos $T(n/b)$ por $T(\lfloor n/b \rfloor)$ ou $T(\lceil n/b \rceil)$ não afeta o comportamento assintótico da recorrência. (Provaremos isso na próxima seção.) Por essa razão, em geral, consideramos conveniente omitir as funções piso e teto ao se escreverem recorrências de dividir e conquistar dessa forma.

O teorema mestre

O método mestre depende do teorema a seguir.

Teorema 4.1 (Teorema mestre)
Sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida sobre os inteiros não negativos pela recorrência

$$T(n) = aT(n/b) + f(n) ,$$

onde interpretamos n/b com o significado de $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. Então, $T(n)$ pode ser limitado assintoticamente como a seguir.

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

Antes de aplicar o teorema mestre a alguns exemplos, vamos passar algum tempo tentando entender o que ele significa. Em cada um dos três casos, estamos comparando a função $f(n)$ com a função $n^{\log_b a}$. Intuitivamente, a solução para a recorrência é determinada pela maior das duas funções. Se, como no caso 1, a função $n^{\log_b a}$ for a maior, então a solução será $T(n) = \Theta(n^{\log_b a})$. Se, como no caso 3, a função $f(n)$ for a maior, então a solução será $T(n) = \Theta(f(n))$. Se, como no caso 2, as duas funções tiverem o mesmo tamanho, faremos a multiplicação por um fator logarítmico e a solução será $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Além dessa intuição, existem alguns detalhes técnicos que devem ser entendidos. No primeiro caso, $f(n)$ não só deve ser menor que $n^{\log_b a}$, mas tem de ser *polinomialmente* menor. Ou seja, $f(n)$ deve ser assintoticamente menor que $n^{\log_b a}$ por um fator n^{ϵ} para alguma constante $\epsilon > 0$. No terceiro caso, $f(n)$ não apenas deve ser maior que $n^{\log_b a}$; ela tem de ser polinomialmente maior e, além disso, satisfazer à condição de "regularidade" de que $af(n/b) \leq cf(n)$. Essa condição é satisfeita pela maioria das funções polinomialmente limitadas que encontraremos.

É importante perceber que os três casos não abrangem todas as possibilidades para $f(n)$. Existe uma lacuna entre os casos 1 e 2 quando $f(n)$ é menor que $n^{\log_b a}$, mas não polinomialmente menor. De modo semelhante, há uma lacuna entre os casos 2 e 3 quando $f(n)$ é maior que $n^{\log_b a}$, mas não polinomialmente maior. Se a função $f(n)$ recair em uma dessas lacunas, ou se a condição de regularidade no caso 3 deixar de ser válida, o método mestre não poderá ser usado para resolver a recorrência.

Como usar o método mestre

Para usar o método mestre, simplesmente determinamos qual caso (se houver algum) do teorema mestre se aplica e anotamos a resposta.

Como primeiro exemplo, considere

$$T(n) = 9T(n/3) + n$$

Para essa recorrência, temos $a = 9$, $b = 3$, $f(n) = n$ e, portanto, temos $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Como $f(n) = O(n^{\log_3 9-\epsilon})$, onde $\epsilon = 1$, podemos aplicar o caso 1 do teorema mestre e concluir que a solução é $T(n) = \Theta(n^2)$.

Agora, considere

$$T(n) = T(2n/3) + 1,$$

na qual $a = 1$, $b = 3/2$, $f(n) = 1$ e $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Aplica-se o caso 2, pois $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, e portanto a solução para a recorrência é $T(n) = \Theta(\lg n)$.

Para a recorrência

$$T(n) = 3T(n/4) + n \lg n\ ,$$

temos $a = 3$, $b = 4$, $f(n) = n \lg n$ e $n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$. Como $f(n) = \Omega(n^{\log_4 3+\epsilon})$, onde $\epsilon \approx 0,2$, aplica-se o caso 3 se podemos mostrar que a condição de regularidade é válida para $f(n)$. Para n suficientemente grande, $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ para $c = 3/4$. Conseqüentemente, de acordo com o caso 3, a solução para a recorrência é $T(n) = \Theta(n \lg n)$.

O método mestre não se aplica à recorrência

$$T(n) = 2T(n/2) + n \lg n,$$

mesmo que ela tenha a forma apropriada: $a = 2$, $b = 2$, $f(n) = n \lg n$ e $n^{\log_b a} = n$. Parece que o caso 3 deve se aplicar, pois $f(n) = n \lg n$ é assintoticamente maior que $n^{\log_b a} = n$. O problema é que ela não é *polinomialmente* maior. A razão $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ é assintoticamente menor que n^ϵ para qualquer constante positiva ϵ. Conseqüentemente, a recorrência recai na lacuna entre o caso 2 e o caso 3. (Veja no Exercício 4.4-2 uma solução.)

Exercícios

4.3-1

Use o método mestre para fornecer limites assintóticos restritos para as recorrências a seguir.

a. $T(n) = 4T(n/2) + n$.

b. $T(n) = 4T(n/2) + n^2$.

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

4.3-2
O tempo de execução de um algoritmo A é descrito pela recorrência $T(n) = 7T(n/2) + n^2$. Um algoritmo concorrente A' tem um tempo de execução $T'(n) = aT'(n/4) + n^2$. Qual é o maior valor inteiro para a tal que A' seja assintoticamente mais rápido que A?

4.3-3
Use o método mestre para mostrar que a solução para a recorrência de pesquisa binária $T(n) = T(n/2) + \Theta(1)$ é $T(n) = \Theta(\lg n)$. (Veja no Exercício 2.3-5 uma descrição da pesquisa binária.)

4.3-4
O método mestre pode ser aplicado à recorrência $T(n) = 4T(n/2) + n^2 \lg n$? Por que ou por que não? Forneça um limite superior assintótico para essa recorrência.

4.3-5 ★
Considere a condição de regularidade $af(n/b) \le cf(n)$ para alguma constante $c < 1$, que faz parte do caso 3 do teorema mestre. Dê um exemplo de uma função simples $f(n)$ que satisfaça a todas as condições no caso 3 do teorema mestre, exceto à condição de regularidade.

★ 4.4 Prova do teorema mestre

Esta seção contém uma prova do teorema mestre (Teorema 4.1). A prova não precisa ser entendida para se aplicar o teorema.

A prova tem duas partes. A primeira parte analisa a recorrência "mestre" (4.5), sob a hipótese simplificadora de que $T(n)$ é definida apenas sobre potências exatas de $b > 1$; ou seja, para $n = 1, b, b^2, \ldots$ Essa parte fornece toda a intuição necessária para se entender por que o teorema mestre é verdadeiro. A segunda parte mostra como a análise pode ser estendida a todos os inteiros positivos n e é apenas a técnica matemática aplicada ao problema do tratamento de pisos e tetos.

Nesta seção, algumas vezes abusaremos ligeiramente de nossa notação assintótica, usando-a para descrever o comportamento de funções que só são definidas sobre potências exatas de b. Lembre-se de que as definições de notações assintóticas exigem que os limites sejam provados para todos os números grandes o suficiente, não apenas para aqueles que são potências de b. Tendo em vista que poderíamos produzir novas notações assintóticas que se aplicassem ao conjunto $\{b^i : i = 0, 1, \ldots\}$ em lugar dos inteiros negativos, esse abuso é secundário.

Apesar disso, sempre deveremos nos resguardar quando estivermos usando a notação assintótica sobre um domínio limitado, a fim de não chegarmos a conclusões inadequadas. Por exemplo, provar que $T(n) = O(n)$ quando n é uma potência exata de 2 não garante que $T(n) = O(n)$. A função $T(n)$ poderia ser definida como

$$T(n) = \begin{cases} n & \text{se } n = 1, 2, 4, 8, \ldots, \\ n^2 & \text{em caso contrário,} \end{cases}$$

e, nesse caso, o melhor limite superior que pode ser provado é $T(n) = O(n^2)$. Devido a esse tipo de conseqüência drástica, nunca empregaremos a notação assintótica sobre um domínio limitado sem tornar absolutamente claro a partir do contexto que estamos fazendo isso.

4.4.1 A prova para potências exatas

A primeira parte da prova do teorema mestre analisa a recorrência (4.5),

$$T(n) = aT(n/b) + f(n),$$

para o método mestre, sob a hipótese de que n é uma potência exata de $b > 1$, onde b não precisa ser um inteiro. A análise está dividida em três lemas. O primeiro reduz o problema de resolver a recorrência mestre ao problema de avaliar uma expressão que contém um somatório. O segundo determina limites sobre esse somatório. O terceiro lema reúne os dois primeiros para provar uma versão do teorema mestre correspondente ao caso em que n é uma potência exata de b.

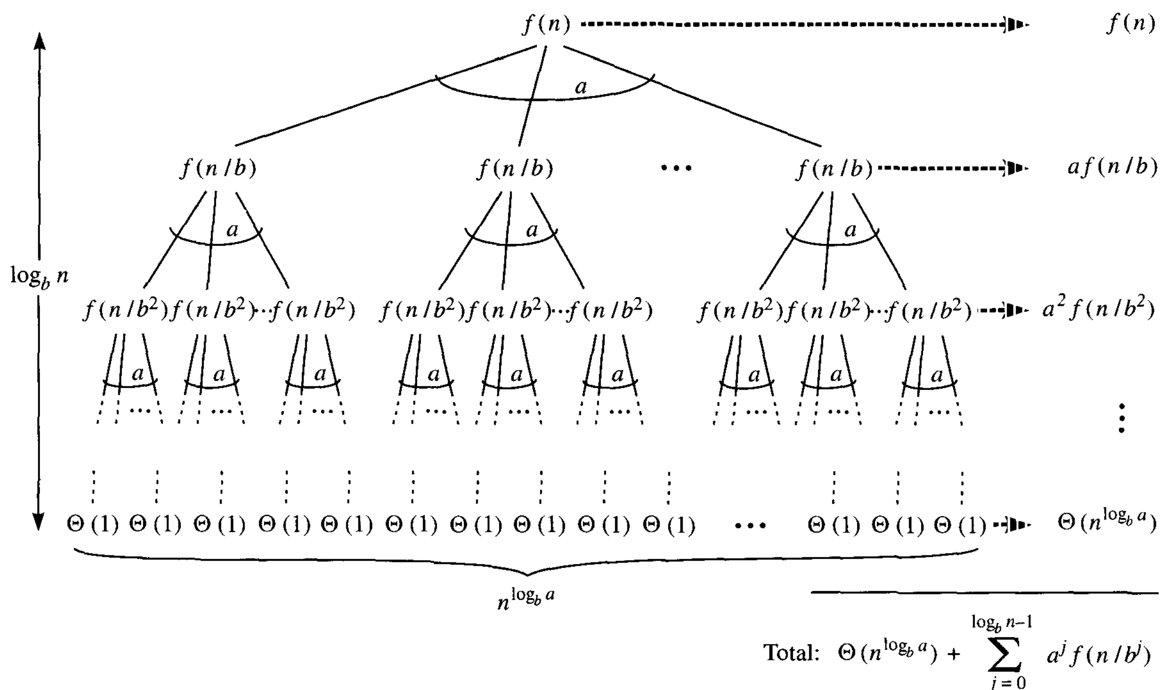


FIGURA 4.3 A árvore de recursão gerada por $T(n) = aT(n/b) + f(n)$. A árvore é uma árvore a-ária completa com $n^{\log_b a}$ folhas e altura $\log_b n$. O custo de cada nível é mostrado à direita, e sua soma é dada na equação (4.6)

Lema 4.2

Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida sobre potências exatas de b. Defina $T(n)$ sobre potências exatas de b pela recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1\,, \\ aT(n/b) + f(n) & \text{se } n = b^i\,, \end{cases}$$

onde i é um inteiro positivo. Então

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)\,. \qquad (4.6)$$

Prova Usamos a árvore de recursão da Figura 4.3. A raiz da árvore tem custo $f(n)$, e ela tem a filhas, cada uma com o custo $f(n/b)$. (É conveniente imaginar a como sendo um inteiro, especialmente ao se visualizar a árvore de recursão, mas a matemática não o exige.) Cada uma dessas filhas tem a filhas com o custo $f(n/b^2)$ e, desse modo, existem a^2 nós que estão à distância 2 da raiz. Em geral, há a^j nós à distância j da raiz, e cada um tem o custo $f(n/b^j)$. O custo de cada folha é $T(1) = \Theta(1)$, e cada folha está a uma distância $\log_b n$ da raiz, pois $n/b^{\log_b n} = 1$. Existem $a^{\log_b n} = n^{\log_b a}$ folhas na árvore.

Podemos obter a equação (4.6) fazendo o somatório dos custos de cada nível da árvore, como mostra a figura. O custo para um nível j de nós internos é $a^j f(n/b^j)$, e assim o total de todos os níveis de nós internos é

$$\sum_{j=0}^{\log_b n-1} a^j (n/b^j) \ .$$

No algoritmo básico de dividir e conquistar, essa soma representa os custos de dividir problemas em subproblemas, e depois combinar novamente os subproblemas. O custo de todas as folhas, que é o custo de efetuar todos os $n^{\log_b a}$ subproblemas de tamanho 1, é $\Theta(n^{\log_b a})$. ■

Em termos da árvore de recursão, os três casos do teorema mestre correspondem a casos nos quais o custo total da árvore é (1) dominado pelos custos nas folhas, (2) distribuído uniformemente entre os níveis da árvore ou (3) dominado pelo custo da raiz.

O somatório na equação (4.6) descreve o custo dos passos de divisão e combinação no algoritmo básico de dividir e conquistar. O lema seguinte fornece limites assintóticos sobre o crescimento do somatório.

Lema 4.3

Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida sobre potências exatas de b. Uma função $g(n)$ definida sobre potências exatas de b por

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \tag{4.7}$$

pode então ser limitada assintoticamente para potências exatas de b como a seguir.

1. Se $f(n) = O\ O(n^{\log_b a-\epsilon})$ para alguma constante $\epsilon > 0$, então $g(n) = O(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a})$, então $g(n) = \Theta(n^{\log_b a} \lg n)$.
3. Se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e para todo $n \geq b$, então $g(n) = \Theta(f(n))$.

Prova Para o caso 1, temos $f(n) = O(n^{\log_b a-\epsilon})$, implicando que $f(n/b^j) = O((n/b^j)^{\log_b a-\epsilon})$. A substituição na equação (4.7) resulta em

$$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right). \tag{4.8}$$

Limitamos o somatório dentro da notação O pela fatoração dos termos e simplificação, o que resulta em uma série geométrica crescente:

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j \\
&= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon \log_b n}-1}{b^\epsilon-1}\right) \\
&= n^{\log_b a-\epsilon} \left(\frac{n^\epsilon-1}{n^\epsilon-1}\right).
\end{aligned}$$

Tendo em vista que b e ϵ são constantes, podemos reescrever a última expressão como $n^{\log_b a-\epsilon} O(n^{\epsilon}) = O(n^{\log_b a})$. Substituindo por essa expressão o somatório na equação (4.8), temos

$$g(n) = O(n^{\log_b a}),$$

e o caso 1 fica provado.

Sob a hipótese de que $f(n) = \Theta(n^{\log_b a})$ para o caso 2, temos que $f(n/b^j) = \Theta((n/b^j)^{\log_b a-\epsilon})$. A substituição na equação (4.7) produz

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right). \tag{4.9}$$

Limitamos o somatório dentro de Θ como no caso 1 mas, dessa vez, não obtemos uma série geométrica. Em vez disso, descobrimos que todo termo do somatório é o mesmo:

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 \\
&= n^{\log_b a} \log_b n .
\end{aligned}$$

Substituindo por essa expressão o somatório da equação (4.9), obtemos

$$\begin{aligned}
g(n) &= \Theta(n^{\log_b a} \log_b n) \\
&= \Theta(n^{\log_b a} \lg n),
\end{aligned}$$

e caso 2 fica provado.

O caso 3 é provado de forma semelhante. Como $f(n)$ aparece na definição (4.7) de $g(n)$ e todas os termos de $g(n)$ são não negativos, podemos concluir que $g(n) = \Omega(f(n))$ para potências exatas de b. Sob a hipótese de que $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todo $n \geq b$, temos $f(n/b) \leq (c/a) f(n)$. Iteragindo j vezes, temos $f(n/b^j) \leq (c/a)^j f(n)$ ou, de modo equivalente, $a^j f(n/b^j) \leq c^j f(n)$. A substituição na equação (4.7) e a simplificação produzem uma série geométrica mas, diferente da série no caso 1, esta tem termos decrescentes:

$$\begin{aligned}
g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\
&\leq \sum_{j=0}^{\log_b n-1} c^j f(n) \\
&\leq f(n) \sum_{j=0}^{\infty} c^j \\
&= f(n)\left(\frac{1}{1-c}\right) \\
&= O(f(n)),
\end{aligned}$$

pois c é constante. Desse modo, podemos concluir que $g(n) = \Theta(f(n))$ para potências exatas de b. O caso 3 fica provado, o que completa a prova do lema. ∎

Agora podemos provar uma versão do teorema mestre para o caso em que n é uma potência exata de b.

Lema 4.4

Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida sobre potências exatas de b. Defina $T(n)$ sobre potências exatas de b pela recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \,, \\ aT(n/b) + f(n) & \text{se } n = b^i \,, \end{cases}$$

onde i é um inteiro positivo. Então, $T(n)$ pode ser limitado assintoticamente para potências exatas de b como a seguir.

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

Prova Empregamos os limites do Lema 4.3 para avaliar o somatório (4.6) a partir do Lema 4.2. Para o caso 1, temos

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + O(n^{\log_b a}) \\ &= \Theta(n^{\log_b a}) \,, \end{aligned}$$

e, para o caso 2,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) \\ &= \Theta(n^{\log_b a} \lg n) \,. \end{aligned}$$

Para o caso 3,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(f(n)) \\ &= \Theta(f(n)) \,, \end{aligned}$$

porque $f(n) = \Omega(n^{\log_b a + \epsilon})$. ∎

4.4.2 Pisos e tetos

Para completar a prova do teorema mestre, devemos agora estender nossa análise à situação em que pisos e tetos são usados na recorrência mestre, de forma que a recorrência seja definida para todos os inteiros, não apenas para potências exatas de b. Obter um limite inferior sobre

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.10}$$

e um limite superior sobre

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.11}$$

é rotina, pois o limite $\lceil n/b \rceil \geq n/b$ pode ser forçado no primeiro caso para produzir o resultado desejado, e o limite $\lfloor n/b \rfloor \leq n/b$ pode ser forçado no segundo caso. A imposição de um limite inferior sobre a recorrência (4.11) exige quase a mesma técnica que a imposição de um limite superior sobre a recorrência (4.10); assim, apresentaremos somente esse último limite.

Modificamos a árvore de recursão da Figura 4.3 para produzir a árvore de recursão da Figura 4.4. À medida que nos aprofundamos na árvore de recursão, obtemos uma seqüência de invocações recursivas sobre os argumentos

$$n\,,$$
$$\lceil n/b \rceil\,,$$
$$\lceil \lceil n/b \rceil /b \rceil\,,$$
$$\lceil \lceil \lceil n/b \rceil /b \rceil /b \rceil\,,$$
$$\vdots$$

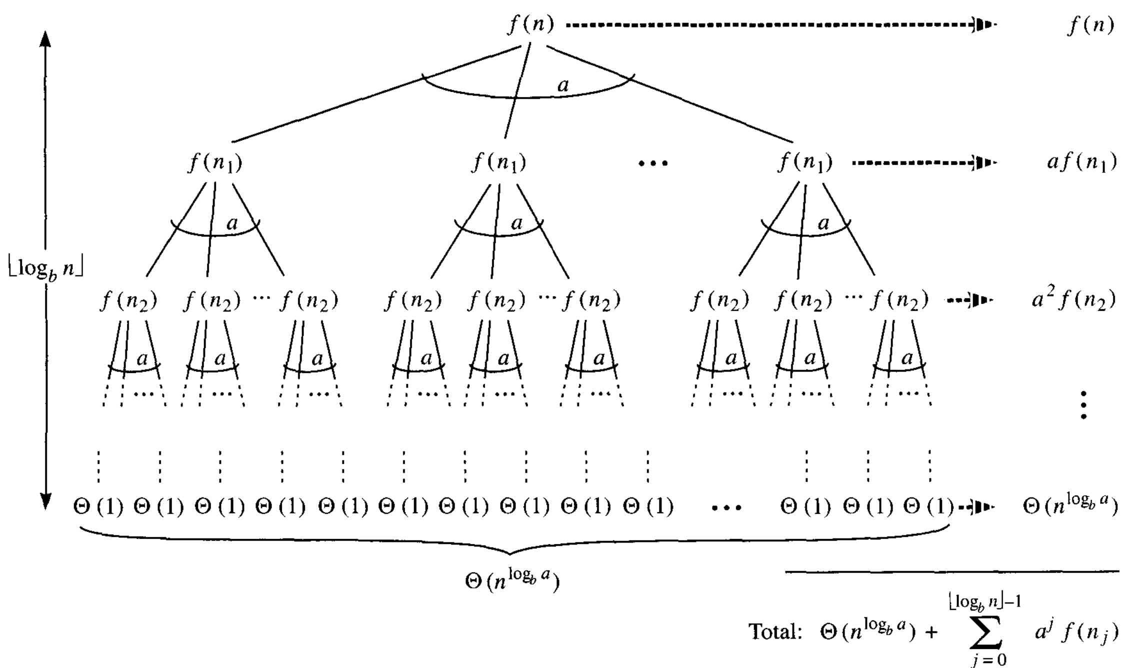


FIGURA 4.4 A árvore de recursão gerada por $T(n) = aT(\lceil n/b \rceil) + f(n)$. O argumento recursivo n_j é dado pela equação (4.12)

Vamos denotar o j-ésimo elemento na seqüência por n_j, onde

$$n_j = \begin{cases} n & \text{se } j = 0\,, \\ \lceil n_{j-1}/b \rceil & \text{se } j > 0\,. \end{cases} \tag{4.12}$$

Nossa primeira meta é determinar a profundidade k tal que n_k é uma constante. Usando a desigualdade $\lceil x \rceil \leq x + 1$, obtemos

$$n_0 \leq n\,,$$
$$n_1 \leq \frac{n}{b} + 1\,,$$
$$n_2 \leq \frac{n}{b^2} + \frac{1}{b} + 1\,,$$
$$n_3 \leq \frac{n}{b^3} + \frac{n}{b^2}, \frac{1}{b} + 1\,,$$
$$\vdots$$

Em geral,

$$
\begin{aligned}
n_j &\le \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \\
&< \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} \\
&= \frac{n}{b^j} + \frac{b}{b-1} .
\end{aligned}
$$

Fazendo $j = \lfloor \log_b n \rfloor$, obtemos

$$
\begin{aligned}
n_{\lfloor \log_b n \rfloor} &< \frac{n}{b^{\lceil \log_b n \rceil}} + \frac{b}{b-1} \\
&\le \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} \\
&= \frac{n}{n/b} + \frac{b}{b-1} \\
&= b + \frac{b}{b-1} \\
&= O(1) ,
\end{aligned}
$$

e desse modo vemos que, à profundidade $\lfloor \log_b n \rfloor$, o tamanho do problema é no máximo uma constante.

Da Figura 4.4, observamos que

$$
T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j j(n_j) , \tag{4.13}
$$

que é quase igual à equação (4.6), a não ser pelo fato de n ser um inteiro arbitrário e não se restringir a ser uma potência exata de b.

Agora podemos avaliar o somatório

$$
g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j j(n_j) \tag{4.14}
$$

a partir de (4.13) de maneira análoga à prova do Lema 4.3. Começando com o caso 3, se $af(\lceil n/b \rceil) \le cf(n)$ para $n > b + b/(b-1)$, onde $c < 1$ é uma constante, segue-se que $a^j f(n_j) \le c^j f(n)$. Portanto, a soma na equação (4.14) pode ser avaliada da mesma maneira que no Lema 4.3. Para o caso 2, temos $f(n) = \Theta(n^{\log_b a})$. Se pudermos mostrar que $f(n_j) = O(n^{\log_b a} / a^j) = O((n/b^j)^{\log_b a})$, então a prova para caso 2 do Lema 4.3 passará. Observe que $j \le \lfloor \log_b n \rfloor$ implica $b^j/n \le 1$. O limite $f(n) = O(n^{\log_b a})$ implica que existe uma constante $c > 0$ tal que, para n_j suficientemente grande,

$$
\begin{aligned}
f(n_j) &\le c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} \\
&= c\left(\frac{n}{b^j}\left(1 + \frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a}
\end{aligned}
$$

$$
\begin{aligned}
&= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\left(\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a} \\
&\leq c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\frac{b}{b-1}\right)^{\log_b a} \\
&= O\left(\frac{n^{\log_b a}}{a^j}\right).
\end{aligned}
$$

pois $c(1+b/(b-1))^{\log_b a}$ é uma constante. Portanto, o caso 2 fica provado. A prova do caso 1 é quase idêntica. A chave é provar o limite $f(n_j) = O(n^{\log_b a-\epsilon})$, semelhante à prova correspondente do caso 2, embora a álgebra seja mais complicada.

Agora provamos os limites superiores no teorema mestre para todos os inteiros n. A prova dos limites inferiores é semelhante.

Exercícios

4.4-1 ★

Forneça uma expressão simples e exata para n_i na equação (4.12) para o caso em que b é um inteiro positivo, em vez de um número real arbitrário.

4.4-2 ★

Mostre que, se $f(n) = \Theta(n^{\log_b a} \lg^k n)$, onde $k \geq 0$, a recorrência mestre tem a solução $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. Por simplicidade, restrinja sua análise a potências exatas de b.

4.4-3 ★

Mostre que o caso 3 do teorema mestre é exagerado, no sentido de que a condição de regularidade $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ implica que existe uma constante $\epsilon > 0$ tal que $f(n) = \Omega(n^{\log_b a+\epsilon})$.

Problemas

4-1 Exemplos de recorrência

Forneça limites assintóticos superiores e inferiores para $T(n)$ em cada uma das recorrências a seguir. Suponha que $T(n)$ seja constante para $n \leq 2$. Torne seus limites tão restritos quanto possível e justifique suas respostas.

a. $T(n) = 2T(n/2) + n^3$.

b. $T(n) = T(9n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n-1) + n$.

h. $T(n) = T(\sqrt{n}) + 1$.

4-2 Como localizar o inteiro perdido

Um arranjo $A[1..n]$ contém todos os inteiros de 0 a n, exceto um. Seria fácil descobrir o inteiro que falta no tempo $O(n)$, usando um arranjo auxiliar $B[0..n]$ para registrar os números que aparecem em A. Porém, neste problema, não podemos ter acesso a todo um inteiro em A com uma única operação. Os elementos de A são representados em binário, e a única operação que pode-

mos usar para obter acesso a eles é "buscar o j-ésimo bit de $A[i]$", que demora um tempo constante.

Mostre que, se usarmos apenas essa operação, ainda poderemos descobrir o inteiro que falta no tempo $O(n)$.

4-3 Custos de passagem de parâmetros

Em todo este livro, partimos da hipótese de que a passagem de parâmetros durante as chamadas de procedimentos demora um tempo constante, mesmo que um arranjo de N elementos esteja sendo passado. Essa hipótese é válida na maioria dos sistemas, porque é passado um ponteiro para o arranjo, e não o próprio arranjo. Este problema examina as implicações de três estratégias de passagem de parâmetros:

1. Um arranjo é passado por ponteiro. Tempo = $\Theta(1)$.

2. Um arranjo é passado por cópia. Tempo = $\Theta(N)$, onde N é o tamanho do arranjo.

3. Um arranjo é passado por cópia somente do subintervalo ao qual o procedimento chamado poderia ter acesso. Tempo = $\Theta(p - q + 1)$ se o subarranjo $A[p\,..\,q]$ for passado.

a. Considere o algoritmo de pesquisa binária recursiva para localizar um número em um arranjo ordenado (ver Exercício 2.3-5). Forneça recorrências para os tempos de execução do pior caso de pesquisa binária quando os arranjos são passados com o uso de cada um dos três métodos anteriores, e forneça bons limites superiores nas soluções das recorrências. Seja N o tamanho do problema original e n o tamanho de um subproblema.

b. Repita a parte (a) para o algoritmo MERGE-SORT da Seção 2.3.1.

4-4 *Outros exemplos de recorrência*

Forneça limites assintóticos superiores e inferiores para $T(n)$ em cada uma das recorrências a seguir. Suponha que $T(n)$ seja constante para n suficientemente pequeno. Torne seus limites tão restritos quanto possível e justifique suas respostas.

a. $T(n) = 3T(n/2) + n \lg n$.

b. $T(n) = 5T(n/5) + n/\lg n$.

c. $T(n) = 4T(n/2) + n^2\sqrt{n}$.

d. $T(n) = 3T(n/3 + 5) + n/2$.

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

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

g. $T(n) = T(n - 1) + 1/n$.

h. $T(n) = T(n - 1) + \lg n$.

i. $T(n) = T(n - 2) + 2 \lg n$.

j. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4-5 Números de Fibonacci

Este problema desenvolve propriedades dos números de Fibonacci, que são definidos pela recorrência (3.21). Usaremos a técnica de gerar funções para resolver a recorrência de Fibonacci. Defina a ***função geradora*** (ou ***série de potências formais***) $\mathcal{F}$ como]

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} F_i z^i$$

$$= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \ldots,$$

onde F_i é o i-ésimo número de Fibonacci.

a. Mostre que $\mathscr{F}(z) = z + z\mathscr{F}(z) + z^2\mathscr{F}(z)$..

b. Mostre que

$$\begin{aligned}\mathscr{F}(z) &= \frac{z}{1-z-z^2} \\ &= \frac{z}{(1-\phi z)(1-\hat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z}\right),\end{aligned}$$

onde

$$\phi = \frac{1+\sqrt{5}}{2} = 1{,}61803\ \ldots$$

e

$$\hat{\phi} = \frac{1-\sqrt{5}}{2} = -0{,}61803\ldots\ .$$

c. Mostre que

$$\mathscr{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i\ .$$

d. Prove que $F_i = \phi^i/\sqrt{5}$ para $i > 0$, arredondado para o inteiro mais próximo. (*Sugestão:* Observe que $|\hat{\phi}| < 1$.)

e. Prove que $F_{i+2} \geq \phi^i$ para $i \geq 0$.

4-6 Testes de chips VLSI

O professor Diógenes tem n chips VLSI[1] supostamente idênticos que, em princípio, são capazes de testar uns aos outros. O aparelho de teste do professor acomoda dois chips de cada vez. Quando o aparelho está carregado, cada chip testa o outro e informa se ele está bom ou ruim. Um chip bom sempre informa com precisão se o outro chip está bom ou ruim, mas a resposta de um chip ruim não é confiável. Portanto, os quatro resultados possíveis de um teste são:

Chip *A* informa	Chip *B* informa	Conclusão
B está bom	*A* está bom	Ambos estão bons ou ambos ruins
B está bom	*A* está ruim	Pelo menos um está ruim
B está ruim	*A* está bom	Pelo menos um está ruim
B está ruim	*A* está ruim	Pelo menos um está ruim

[1]VLSI significa *very large scale integration*, ou integração em escala muito grande, que é a tecnologia de chips de circuitos integrados usada para fabricar a maioria dos microprocessadores de hoje.

a. Mostre que, se mais de $n/2$ chips estão ruins, o professor não pode necessariamente descobrir quais chips estão bons usando qualquer estratégia baseada nessa espécie de teste aos pares. Suponha que os chips ruins possam conspirar para enganar o professor.

b. Considere o problema de descobrir um único chip bom entre n chips, supondo que mais de $n/2$ dos chips estejam bons. Mostre que $\lfloor n/2 \rfloor$ testes de pares sejam suficientes para reduzir o problema a outro com praticamente metade do tamanho.

c. Mostre que os chips bons podem ser identificados com $\Theta(n)$ testes de pares, supondo-se que mais de $n/2$ dos chips estejam bons. Forneça e resolva a recorrência que descreve o número de testes.

4-7 Arranjos de Monge

Um arranjo $m \times n$ A de números reais é um ***arranjo de Monge*** se, para todo i, j, k e l tais que $1 \le i < k \le m$ e $1 \le j < l \le n$, temos

$$A[i,j] + A[k,l] \le A[i,l] + A[k,j].$$

Em outras palavras, sempre que escolhemos duas linhas e duas colunas de um arranjo de Monge e consideramos os quatro elementos nas interseções das linhas e das colunas, a soma dos elementos superior esquerdo e inferior direito é menor ou igual à soma dos elementos inferior esquerdo e superior direito. Por exemplo, o arranjo a seguir é um arranjo de Monge:

```
10 17 13 28 23
17 22 16 29 23
24 28 22 34 24
11 13  6 17  7
45 44 32 37 23
36 33 19 21  6
75 66 51 53 34
```

a. Prove que um arranjo é de Monge se e somente se para todo $i = 1, 2, \ldots, m-1$ e $j = 1, 2, \ldots, n-1$, temos

$$A[i,j] + A[i+1, j+1] \le A[i, j+1] + A[i+1, j].$$

(*Sugestão:* Para a parte "somente se", use a indução separadamente sobre linhas e colunas.)

b. O arranjo a seguir não é de Monge. Troque um elemento para transformá-lo em um arranjo de Monge. (*Sugestão:* Use a parte (a).)

```
37 23 22 32
21  6  7 10
53 34 30 31
32 13  9  6
43 21 15  8
```

c. Seja $f(i)$ o índice da coluna que contém o elemento mínimo mais à esquerda da linha i. Prove que $f(1) \le f(2) \le \ldots \le f(m)$ para qualquer arranjo de Monge $m \times n$.

d. Aqui está uma descrição de um algoritmo de dividir e conquistar que calcula o elemento mínimo mais à esquerda em cada linha de um arranjo de Monge $m \times n$ A:

> Construa uma submatriz A' de A consistindo nas linhas de numeração par de A. Determine recursivamente o mínimo mais à esquerda para cada linha de A'. Em seguida, calcule o mínimo mais à esquerda nas linhas de numeração ímpar de A.

Explique como calcular o mínimo mais à esquerda nas linhas de numeração ímpar de A (dado que o mínimo mais à esquerda das linhas de numeração seja conhecido) no tempo $O(m + n)$.

e. Escreva a recorrência que descreve o tempo de execução do algoritmo descrito na parte (d). Mostre que sua solução é $O(m + n \log m)$.

Notas do capítulo

As recorrências foram estudadas desde 1202 aproximadamente, por L. Fibonacci, e em sua homenagem os números de Fibonacci receberam essa denominação. A. De Moivre introduziu o método de funções geradoras (ver Problema 4-5) para resolver recorrências. O método mestre foi adaptado de Bentley, Haken e Saxe [41], que fornecem o método estendido justificado pelo Exercício 4.4-2. Knuth [182] e Liu [205] mostram como resolver recorrências lineares usando o método de funções geradoras. Purdom e Brown [252] e Graham, Knuth e Patashnik [132] contêm discussões extensas da resolução de recorrências.

Vários pesquisadores, inclusive Akra e Bazzi [13], Roura [262] e Verma [306], forneceram métodos para resolução de recorrências de dividir e conquistar mais gerais do que as que são resolvidas pelo método mestre. Descrevemos aqui o resultado de Akra e Bazzi, que funciona para recorrências da forma

$$T(n) = \sum_{i=1}^{k} a_i T(\lfloor n/b_i \rfloor) + f(n) , \tag{4.15}$$

onde $k \geq 1$; todos os coeficientes a_i são positivos e somam pelo menos 1; todos os valores b_i são maiores que 1; $f(n)$ é limitada, positiva e não decrescente; e, para todas as constantes $c > 1$, existem constantes $n_0, d > 0$ tais que $f(n/c) \geq df(n)$ para todo $n \geq n_0$. Esse método funcionaria sobre uma recorrência como $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$, para a qual o método mestre não se aplica. Para resolver a recorrência (4.15), primeiro encontramos o valor de p tal que $\sum_{i=1}^{k} a_i b_i^{-p} = 1$. (Tal p sempre existe, ele é único e positivo.) A solução para a recorrência é então

$$T(n) = \Theta(n^p) + \Theta\left(n^p \int_{n'}^{n} \frac{f(x)}{x^{p+1}} dx\right),$$

para uma constante n' suficientemente grande. O método de Akra-Bazzi pode ser um tanto difícil de usar, mas ele serve para resolver recorrências que modelam a divisão do problema em subproblemas de tamanhos substancialmente desiguais. O método mestre é mais simples para usar, mas só se aplica quando os tamanhos dos subproblemas são iguais.

Capítulo 5

Análise probabilística e algoritmos aleatórios

Este capítulo introduz a análise probabilística e os algoritmos aleatórios. Se estiver pouco familiarizado com os fundamentos da teoria das probabilidades, leia o Apêndice C, que apresenta uma revisão desse assunto. A análise probabilística e os algoritmos aleatórios serão revistos várias vezes ao longo deste livro.

5.1 O problema da contratação

Suponha que você precise contratar um novo auxiliar de escritório. Suas tentativas anteriores de contratação foram malsucedidas, e você decide usar uma agência de empregos. A agência de empregos lhe enviará um candidato por dia. Você entrevistará a pessoa e então decidirá se deve contratá-la ou não. Você terá de pagar à agência de empregos uma pequena taxa para entrevistar um candidato. Porém, a contratação real de um candidato é mais onerosa, pois você deve despedir seu auxiliar de escritório atual e pagar uma grande taxa de contratação à agência de empregos. Seu compromisso é ter, a cada momento, a melhor pessoa possível para realizar o serviço. Assim, você decide que, depois de entrevistar cada candidato, se esse candidato for mais bem qualificado que o auxiliar de escritório atual, o auxiliar de escritório atual será despedido e o novo candidato será contratado. Você está disposto a pagar o preço resultante dessa estratégia, mas deseja avaliar qual será esse preço.

O procedimento HIRE-ASSISTANT, dado a seguir, expressa em pseudocódigo essa estratégia para contratação. Ele pressupõe que os candidatos ao emprego de auxiliar de escritório são numerados de 1 a n. O procedimento presume que você pode, depois de entrevistar o candidato i, determinar se o candidato i é o melhor candidato que viu até agora. Para inicializar, o procedimento cria um candidato fictício, com o número 0, menos qualificado que cada um dos outros candidatos.

```
HIRE-ASSISTANT (n)
1 melhor ← 0        ▷ candidato 0 é um candidato fictício menos qualificado
2 for i ← 1 to n
3     do entrevistar candidato i
4        if candidato i é melhor que candidato melhor
5           then melhor ← i
6                contratar candidato i
```

O modelo de custo para esse problema difere do modelo descrito no Capítulo 2. Não estamos preocupados com o tempo de execução de HIRE-ASSISTANT, mas sim com o custo referente a entrevistar e contratar. Na superfície, a análise do custo desse algoritmo pode parecer muito diferente de analisar o tempo de execução, digamos, da ordenação por intercalação. Porém, as técnicas analíticas usadas são idênticas, quer estejamos analisando o custo ou o tempo de execução. Em um ou outro caso, estamos contando o número de vezes que certas operações básicas são executadas.

Entrevistar tem um custo baixo, digamos c_i, enquanto contratar é dispendioso, custando c_h. Seja m o número de pessoas contratadas. Então, o custo total associado a esse algoritmo é $O(nc_i + mc_h)$. Independente de quantas pessoas contratamos, sempre entrevistamos n candidatos e, portanto, sempre incorremos no custo nc_i associado a entrevistar. Assim, vamos nos concentrar na análise de mc_h, o custo de contratação. Essa quantidade varia com cada execução do algoritmo.

Esse cenário serve como um modelo para um paradigma computacional comum. Com freqüência, precisamos encontrar o valor máximo ou mínimo em uma seqüência, examinando cada elemento da seqüência e mantendo um "vencedor" atual. O problema da contratação modela a forma como atualizamos freqüentemente nossa noção de qual elemento está vencendo no momento.

Análise do pior caso

No pior caso, realmente contratamos cada candidato que entrevistamos. Essa situação ocorre se os candidatos chegam em ordem crescente de qualidade, e nesse caso contratamos n vezes, com o custo total de contratação $O(nc_h)$.

Contudo, talvez seja razoável esperar que os candidatos nem sempre cheguem em ordem crescente de qualidade. De fato, não temos nenhuma idéia sobre a ordem em que eles chegam, nem temos qualquer controle sobre essa ordem. Portanto, é natural perguntar o que esperamos que aconteça em um caso típico ou médio.

Análise probabilística

A ***análise probabilística*** é o uso da probabilidade na análise de problemas. De modo mais comum, usamos a análise probabilística para analisar o tempo de execução de um algoritmo. Às vezes, nós a usamos para analisar outras quantidades, como o custo da contratação no procedimento HIRE-ASSISTANT. Para executar uma análise probabilística, devemos usar o conhecimento da distribuição das entradas ou fazer suposições sobre ela. Em seguida, analisamos nosso algoritmo, calculando um tempo de execução esperado. A expectativa é tomada sobre a distribuição das entradas possíveis. Desse modo estamos, na verdade, calculando a média do tempo de execução sobre todas as entradas possíveis.

Devemos ser muito cuidadosos ao decidir sobre a distribuição das entradas. Para alguns problemas, é razoável supor algo sobre o conjunto de todas as entradas possíveis, e podemos usar a análise probabilística como uma técnica para projetar um algoritmo eficiente e como um meio de obter informações para resolver um problema. Em outros problemas, não podemos descrever uma distribuição de entradas razoável e, nesses casos, não podemos utilizar a análise probabilística.

No caso do problema da contratação, podemos pressupor que os candidatos chegam em uma ordem aleatória. O que isso significa para esse problema? Supomos que é possível separar dois candidatos quaisquer e decidir qual é o candidato mais bem qualificado; isto é, existe uma ordem total sobre os candidatos. (Consulte o Apêndice B para ver a definição de uma ordem total.) Portanto, podemos ordenar cada candidato com um número exclusivo de 1 a n, usando *ordenação*(i) para denotar a ordenação do candidato i, e adotar a convenção de que uma ordenação mais alta corresponde a um candidato mais bem qualificado. A lista ordenada $\langle$*ordenação*(1), *ordenação*(2), ..., *ordenação*$(n)\rangle$ é uma permutação da lista $\langle 1, 2, ..., n\rangle$. Dizer que os candidatos chegam em uma ordem aleatória é equivalente a dizer que essa lista de ordenações

tem igual probabilidade de ser qualquer uma das $n!$ permutações dos números 1 a n. Como alternativa, dizemos que as ordenações formam uma ***permutação aleatória uniforme***; ou seja, cada uma das $n!$ permutações possíveis aparece com igual probabilidade.

A Seção 5.2 contém uma análise probabilística do problema da contratação.

Algoritmos aleatórios

Para utilizar a análise probabilística, precisamos saber algo a respeito da distribuição sobre as entradas. Em muitos casos, sabemos bem pouco sobre a distribuição das entradas. Mesmo se soubermos algo sobre a distribuição, talvez não possamos modelar esse conhecimento em termos computacionais. Ainda assim, freqüentemente podemos usar a probabilidade e o caráter aleatório como uma ferramenta para projeto e análise de algoritmos, tornando aleatório o comportamento de parte do algoritmo.

No problema da contratação, talvez pareça que os candidatos estão sendo apresentados em ordem aleatória, mas não temos nenhum meio de saber se isso está ou não ocorrendo. Desse modo, para desenvolver um algoritmo aleatório para o problema da contratação, devemos ter maior controle sobre a ordem em que entrevistamos os candidatos. Assim, vamos alterar um pouco o modelo. Diremos que a agência de empregos tem n candidatos, e ela nos envia uma lista dos candidatos com antecedência. A cada dia, escolhemos ao acaso qual candidato entrevistar. Embora não conheçamos nada sobre os candidatos (além de seus nomes), fizemos uma mudança significativa. Em vez de contar com a suposição de que os candidatos virão em ordem aleatória, obtivemos o controle do processo e impusemos uma ordem aleatória.

De modo mais geral, dizemos que um algoritmo é ***aleatório*** se o seu comportamento é determinado não apenas por sua entrada, mas também por valores produzidos por um ***gerador de números aleatórios***. Vamos supor que temos à nossa disposição um gerador de números aleatórios RANDOM. Uma chamada a RANDOM(a, b) retorna um inteiro entre a e b inclusive, sendo cada inteiro igualmente provável. Por exemplo, RANDOM(0, 1) produz 0 com probabilidade 1/2 e produz 1 com probabilidade 1/2. Uma chamada a RANDOM(3, 7) retorna 3, 4, 5, 6 ou 7, cada um com probabilidade 1/5. Cada inteiro retornado por RANDOM é independente dos inteiros retornados em chamadas anteriores. Imagine RANDOM como se fosse o lançamento de um dado de $(b - a + 1)$ lados para obter sua saída. (Na prática, a maioria dos ambientes de programação oferece um ***gerador de números pseudo-aleatórios***: um algoritmo determinístico retornando números que "parecem" estatisticamente aleatórios.)

Exercícios

5.1-1

Mostre que a hipótese de que sempre somos capazes de descobrir qual candidato é o melhor na linha 4 do procedimento HIRE-ASSISTANT implica que conhecemos uma ordem total sobre as ordenações dos candidatos.

5.1-2 ★

Descreva uma implementação do procedimento RANDOM(a, b) que só faça chamadas a RANDOM(0, 1). Qual é o tempo de execução esperado de seu procedimento, como uma função de a e b?

5.1-3 ★

Suponha que você deseja dar saída a 0 com probabilidade 1/2 e a 1 com probabilidade 1/2. Há um procedimento BIASED-RANDOM à sua disposição que dá saída a 0 ou 1. A saída é 1 com alguma probabilidade p e 0 com probabilidade $1 - p$, onde $0 < p < 1$, mas você não sabe qual é o valor de p. Forneça um algoritmo que utilize BIASED-RANDOM como uma sub-rotina e forneça uma resposta imparcial, retornando 0 com probabilidade 1/2 e 1 com probabilidade 1/2. Qual é o tempo de execução esperado de seu algoritmo como uma função de p?

5.2 Indicadores de variáveis aleatórias

Para analisar muitos algoritmos, inclusive o problema da contratação, usaremos indicadores de variáveis aleatórias. Os indicadores de variáveis aleatórias fornecem um método conveniente para conversão entre probabilidades e expectativas. Vamos supor que temos um espaço amostral S e um evento A. Então, o ***indicador de variável aleatória*** $\mathrm{I}\{A\}$ associado ao evento A é definido como

$$I\{A\} = \begin{cases} 1 & \text{se } A \text{ ocorre }, \\ 0 & \text{se } A \text{ não ocorre }. \end{cases} \tag{5.1}$$

Como um exemplo simples, vamos determinar o número esperado de caras que obtemos quando lançamos uma moeda comum. Nosso espaço amostral é $S = \{H, T\}$, e definimos uma variável aleatória Y que assume os valores H e T, cada um com probabilidade 1/2. Podemos então definir um indicador de variável aleatória X_H, associado ao resultado cara no lançamento da moeda, o que podemos expressar como o evento $Y = H$. Essa variável conta o número de caras obtidas nesse lançamento, e tem o valor 1 se a moeda mostra cara e 0 em caso contrário. Escrevemos

$$X_H = \mathrm{I}\{Y = H\} = \begin{cases} 1 & \text{se } Y = H, \\ 0 & \text{se } Y = T. \end{cases}$$

O número esperado de caras obtidas em um lançamento da moeda é simplesmente o valor esperado de nosso indicador de variável X_H:

$$\begin{aligned} \mathrm{E}[X_H] &= \mathrm{E}[\mathrm{I}\{Y = H\}] \\ &= 1 \cdot \Pr\{Y = H\} + 0 \cdot \Pr\{Y = T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 . \end{aligned}$$

Desse modo, o número esperado de caras obtidas por um lançamento de uma moeda comum é 1/2. Como mostra o lema a seguir, o valor esperado de um indicador de variável aleatória associado a um evento A é igual à probabilidade de A ocorrer.

Lema 5.1
Dado um espaço amostral S e um evento A no espaço amostral S, seja $X_A = \mathrm{I}\{A\}$. Então, $\mathrm{E}\{X_A\} = \Pr\{A\}$.

Prova Pela definição de indicador de variável aleatória da equação (5.1) e pela definição de valor esperado, temos

$$\begin{aligned} \mathrm{E}[X_A] &= \mathrm{E}[\mathrm{I}\{A\}] \\ &= 1 \cdot \Pr\{A\} + 0 \cdot \Pr\{\bar{A}\} \\ &= \Pr\{A\} , \end{aligned}$$

onde $\bar{A}$ denota $S - A$, o complemento de A. ∎

Embora os indicadores de variáveis aleatórias possam parecer incômodos para uma aplicação como a contagem do número esperado de caras no lançamento de uma única moeda, eles

são úteis para analisar situações em que realizamos testes aleatórios repetidos. Por exemplo, indicadores de variáveis aleatórias nos dão um caminho simples para chegar ao resultado da equação (C.36). Nessa equação, calculamos o número de caras em n lançamentos de moedas, considerando separadamente a probabilidade de obter 0 cara, 1 cara, 2 caras etc. Contudo, o método mais simples proposto na equação (C.37) na realidade utiliza implicitamente indicadores de variáveis aleatórias. Tornando esse argumento mais explícito, podemos fazer de X_i o indicador de variável aleatória associado ao evento no qual o i-ésimo lançamento mostra cara. Sendo Y_i a variável aleatória que denota o resultado do i-ésimo lançamento, temos que $X_i = \mathrm{I}\{Y_i = H\}$. Seja X a variável aleatória que denota o número total de caras em n lançamentos de moedas; assim,

$$X = \sum_{i=1}^{n} X_i \,.$$

Desejamos calcular o número esperado de caras; para isso, tomamos a expectativa de ambos os lados da equação anterior para obter

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right].$$

O lado esquerdo da equação anterior é a expectativa da soma de n variáveis aleatórias. Pelo Lema 5.1, podemos calcular facilmente a expectativa de cada uma das variáveis aleatórias. Pela equação (C.20) – linearidade de expectativa – é fácil calcular a expectativa da soma: ela é igual à soma das expectativas das n variáveis aleatórias.

A linearidade de expectativa torna o uso de indicadores de variáveis aleatórias uma técnica analítica poderosa; ela se aplica até mesmo quando existe dependência entre as variáveis aleatórias. Agora podemos calcular com facilidade o número esperado de caras:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}\,[X_i] \\
&= \sum_{i=1}^{n} 1/2 \\
&= n/2 \,.
\end{aligned}$$

Desse modo, em comparação com o método empregado na equação (C.36), os indicadores de variáveis aleatórias simplificam muito o cálculo. Utilizaremos indicadores de variáveis aleatórias em todo este livro.

Análise do problema da contratação com o uso de indicadores de variáveis aleatórias

Voltando ao problema da contratação, agora desejamos calcular o número esperado de vezes que contratamos um novo auxiliar de escritório. Com a finalidade de usar uma análise probabilística, supomos que os candidatos chegam em uma ordem aleatória, como discutimos na seção anterior. (Veremos na Seção 5.3 como remover essa suposição.) Seja X a variável aleatória cujo valor é igual ao número de vezes que contratamos um novo auxiliar de escritório. Poderíamos então aplicar a definição de valor esperado da equação (C.19) para obter

$$E[X] = \sum_{i=1}^{n} x \Pr\{X = x\} ,$$

mas esse cálculo seria incômodo. Em vez disso, utilizaremos indicadores de variáveis aleatórias para simplificar bastante o cálculo.

Para usar indicadores de variáveis aleatórias, em lugar de calcular $E[X]$ definindo uma única variável associada ao número de vezes que contratamos um novo auxiliar de escritório, definimos n variáveis relacionadas ao fato de cada candidato específico ser ou não contratado. Em particular, fazemos de X_i o indicador de variável aleatória associado ao evento em que o i-ésimo candidato é contratado. Desse modo,

$$X_i = \text{I}\{\text{candidato } i \text{ é contratado}\} = \begin{cases} 1 & \text{se o candidato } i \text{ é contratado}, \\ 0 & \text{se o candidato } i \text{ não é contratado}. \end{cases} \quad (5.2)$$

e

$$X = X_1 + X_2 + \ldots + X_n . \quad (5.3)$$

Pelo Lema 5.1, temos que

$$E[X_i] = \Pr\{\text{candidato } i \text{ é contratado}\} ,$$

e devemos então calcular a probabilidade de que as linhas 5 e 6 de HIRE-ASSISTANT sejam executadas.

O candidato i é contratado, na linha 5, exatamente quando o candidato i é melhor que cada um dos candidatos 1 a $i - 1$. Como presumimos que os candidatos chegam em uma ordem aleatória, os primeiros i candidatos apareceram em uma ordem aleatória. Qualquer um desses i primeiros candidatos tem igual probabilidade de ser o mais bem qualificado até o momento. O candidato i tem uma probabilidade $1/i$ de ser mais bem qualificado que os candidatos 1 a $i - 1$ e, desse modo, uma probabilidade $1/i$ de ser contratado. Pela Lema 5.1, concluímos que

$$E[X_i] = 1/i . \quad (5.4)$$

Agora podemos calcular $E[X]$:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \quad \text{(pela equação (5.3))} && (5.5) \\ &= \sum_{i=1}^{n} E[X_i] \quad \text{(pela linearidade de expectativa)} \\ &= \sum_{i=1}^{n} 1/i \quad \text{(pela equação (5.4))} \\ &= \ln n + O(1) \quad \text{(pela equação (A.7))}. && (5.6) \end{aligned}$$

Apesar de entrevistarmos n pessoas, na realidade só contratamos aproximadamente $\ln n$ delas, em média. Resumimos esse resultado no lema a seguir.

Lema 5.2
Supondo que os candidatos sejam apresentados em uma ordem aleatória, o algoritmo HIRE-ASSISTANT tem um custo total de contratação $O(c_h \ln n)$.

Prova O limite decorre imediatamente de nossa definição do custo de contratação e da equação (5.6).

O custo esperado de contratação é uma melhoria significativa sobre o custo de contratação do pior caso, $O(n\ c_h)$.

Exercícios

5.2-1
Em HIRE-ASSISTANT, supondo que os candidatos sejam apresentados em uma ordem aleatória, qual é a probabilidade de você contratar exatamente uma vez? Qual é a probabilidade de você contratar exatamente n vezes?

5.2-2
Em HIRE-ASSISTANT, supondo que os candidatos sejam apresentados em uma ordem aleatória, qual é a probabilidade de você contratar exatamente duas vezes?

5.2-3
Use indicadores de variáveis aleatórias para calcular o valor esperado da soma de n dados.

5.2-4
Use indicadores de variáveis aleatórias para resolver o problema a seguir, conhecido como o ***problema da chapelaria***. Cada um entre n clientes entrega um chapéu ao funcionário da chapelaria em um restaurante. O funcionário devolve os chapéus aos clientes em ordem aleatória. Qual é o número esperado de clientes que recebem de volta seus próprios chapéus?

5.2-5
Seja $A[1..n]$ um arranjo de n números distintos. Se $i < j$ e $A[i] > A[j]$, então o par (i,j) é chamado uma ***inversão*** de A. (Veja no Problema 2-4 mais informações sobre inversões.) Suponha que cada elemento de A seja escolhido ao acaso, independentemente e de maneira uniforme no intervalo de 1 a n. Use indicadores de variáveis aleatórias para calcular o número esperado de inversões.

5.3 Algoritmos aleatórios

Na seção anterior, mostramos como o conhecimento de uma distribuição sobre as entradas pode nos ajudar a analisar o comportamento de um algoritmo no caso médio. Muitas vezes, não temos tal conhecimento, e nenhuma análise de caso médio é possível. Como mencionamos na Seção 5.1, talvez possamos usar um algoritmo aleatório.

No caso de um problema como o problema da contratação, no qual é útil supor que todas as permutações da entrada são igualmente prováveis, uma análise probabilística orientará o desenvolvimento de um algoritmo aleatório. Em vez de pressupor uma distribuição de entradas, impomos uma distribuição. Em particular, antes de executar o algoritmo, permutamos ao acaso os candidatos, a fim de impor a propriedade de que cada permutação é igualmente provável. Essa modificação não altera nossa expectativa de contratar um novo auxiliar de escritório aproximadamente $\ln n$ vezes. Contudo, ela significa que, para *qualquer* entrada, esperamos que seja esse o caso, em vez de entradas obtidas a partir de uma distribuição particular.

Agora, exploramos um pouco mais a distinção entre a análise probabilística e os algoritmos aleatórios. Na Seção 5.2 afirmamos que, supondo-se que os candidatos se apresentem em uma ordem aleatória, o número esperado de vezes que contratamos um novo auxiliar de escritório é cerca de $\ln n$. Observe que o algoritmo é determinístico nesse caso; para qualquer entrada particular, o número de vezes que um novo auxiliar de escritório é contratado sempre será o mesmo. Além disso, o número de vezes que contratamos um novo auxiliar de escritório difere para entradas distintas e depende das ordenações dos diversos candidatos. Tendo em vista que esse número depende apenas das ordenações dos candidatos, podemos representar uma entrada particular listando, em ordem, as ordenações dos candidatos, ou seja, $\langle ordenação(1), ordenação(2), \ldots, ordenação(n)\rangle$. Dada lista de ordenação $A_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle$, um novo auxiliar de escritório sempre será contratado 10 vezes, pois cada candidato sucessivo é melhor que o anterior, e as linhas 5 e 6 serão executadas em cada iteração do algoritmo. Dada a lista de ordenações $A_2 = \langle 10, 9, 8, 7, 6, 5, 4, 3, 2, 1\rangle$, um novo auxiliar de escritório será contratado apenas uma vez, na primeira iteração. Dada uma lista de ordenações $A_3 = \langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6\rangle$, um novo auxiliar de escritório será contratado três vezes, após a entrevista com os candidatos de ordenações 5, 8 e 10. Lembrando que o custo de nosso algoritmo depende de quantas vezes contratamos um novo auxiliar de escritório, vemos que existem entradas dispendiosas, como A_1, entradas econômicas, como A_2 e entradas moderadamente dispendiosas, como A_3.

Por outro lado, considere o algoritmo aleatório que primeiro permuta os candidatos, e depois determina o melhor candidato. Nesse caso, a aleatoriedade está no algoritmo, e não na distribuição de entradas. Dada uma entrada específica, digamos a entrada A_3 anterior, não podemos dizer quantas vezes o máximo será atualizado, porque essa quantidade é diferente a cada execução do algoritmo. A primeira vez em que executamos o algoritmo sobre A_3, ele pode produzir a permutação A_1 e executar 10 atualizações enquanto, na segunda vez em que executamos o algoritmo, podemos produzir a permutação A_2 e executar apenas uma atualização. Na terceira vez em que o executamos, podemos produzir algum outro número de atualizações. A cada vez que executamos o algoritmo, a execução depende das escolhas aleatórias feitas e ela provavelmente irá diferir da execução anterior do algoritmo. Para esse algoritmo e muitos outros algoritmos aleatórios, *nenhuma entrada específica induz seu comportamento no pior caso*. Nem mesmo pior inimigo poderá produzir um arranjo de entrada ruim, pois a permutação aleatória torna irrelevante a ordem de entrada. O algoritmo aleatório só funcional mal se o gerador de números aleatórios produzir uma permutação "sem sorte".

No caso do problema da contratação, a única alteração necessária no código tem a finalidade de permutar o arranjo ao acaso.

```
RANDOMIZED-HIRE-ASSISTANT(n)
1  permutar aleatoriamente a lista de candidatos
2  melhor ← 0        ▷ candidato 0 é um candidato fictício menos qualificado
3  for i ← 1 to n
4       do entrevistar candidato i
5          if candidato i é melhor que candidato melhor
6             then melhor ← i
7                  contratar candidato i
```

Com essa mudança simples, criamos um algoritmo aleatório cujo desempenho corresponde ao que obtivemos supondo que os candidatos se apresentavam em uma ordem aleatória.

Lema 5.3
O custo esperado de contratação do procedimento RANDOMIZED-HIRE-ASSISTANT é $O(c_h \ln n)$.

Prova Depois de permutar o arranjo de entrada, chegamos a uma situação idêntica à da análise probabilística de HIRE-ASSISTANT.

A comparação entre os Lemas 5.2 e 5.3 evidencia a diferença entre a análise probabilística e os algoritmos aleatórios. No Lema 5.2, fizemos uma suposição sobre a entrada. No Lema 5.3, não fizemos tal suposição, embora a aleatoriedade da entrada demore algum tempo extra. No restante desta seção, discutiremos algumas questões relacionadas com a permutação aleatória das entradas.

Permutação aleatória de arranjos

Muitos algoritmos aleatórios fazem a aleatoriedade da entrada permutando o arranjo de entrada dado. (Existem outras maneiras de usar a aleatoriedade.) Aqui, descreveremos dois métodos para esse fim. Supomos que temos um arranjo A que, sem perda de generalidade, contém os elementos 1 a n. Nossa meta é produzir uma permutação aleatória do arranjo.

Um método comum é atribuir a cada elemento $A[i]$ do arranjo uma prioridade aleatória $P[i]$, e depois ordenar os elementos de A de acordo com essas prioridades. Por exemplo, se nosso arranjo inicial fosse $A = \langle 1, 2, 3, 4 \rangle$ e escolhêssemos prioridades aleatórias $P = \langle 36, 3, 97, 19 \rangle$, produziríamos um arranjo $B = \langle 2, 4, 1, 3 \rangle$, pois a segunda prioridade é a menor, seguida pela quarta, depois pela primeira e finalmente pela terceira. Denominamos esse procedimento PERMUTE-BY-SORTING:

PERMUTE-BY-SORTING(A)
1 $n \leftarrow$ *comprimento*$[A]$
2 **for** $i \leftarrow 1$ **to** n
3   **do** $P[i] = \text{RANDOM}(1, n^3)$
4 ordenar A, usando P como chaves de ordenação
5 **return** A

A linha 3 escolhe um número aleatório entre 1 e n^3. Usamos um intervalo de 1 a n^3 para tornar provável que todas as prioridades em P sejam exclusivas.

(O Exercício 5.3-5 lhe pede para provar que a probabilidade de todas as entradas serem exclusivas é pelo menos $1 - 1/n$, e o Exercício 5.3-6 pergunta como implementar o algoritmo ainda que duas ou mais prioridades sejam idênticas.) Vamos supor que todas as prioridades sejam exclusivas.

A etapa demorada nesse procedimento é a ordenação na linha 4. Como veremos no Capítulo 8, se usarmos uma ordenação por comparação, a ordenação demorará o tempo $\Omega(n \lg n)$. Podemos alcançar esse limite inferior, pois vimos que a ordenação por intercalação demora o tempo $\Theta(n \lg n)$. (Veremos na Parte II outras ordenações por comparação que tomam o tempo $\Theta(n \lg n)$.) Depois da ordenação, se $P[i]$ for a j-ésima menor prioridade, então $A[i]$ estará na posição j da saída. Dessa maneira, obteremos uma permutação. Resta provar que o procedimento produz uma ***permutação aleatória uniforme***, isto é, que toda permutação dos números de 1 a n tem igual probabilidade de ser produzida.

Lema 5.4
O procedimento PERMUTE-BY-SORTING produz uma permutação aleatória uniforme da entrada, supondo que todas as prioridades são distintas.

Prova Começamos considerando a permutação particular em que cada elemento $A[i]$ recebe a i-ésima menor prioridade. Mostraremos que essa permutação ocorre com probabilidade exatamente igual a $1/n!$. Para $i = 1, 2, \ldots, n$, seja X_i o evento em que o elemento $A[i]$ recebe a i-ésimo menor prioridade. Então, desejamos calcular a probabilidade de que, para todo i, o evento X_i ocorra, que é

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \ldots \cap X_{n-1} \cap X_n\} .$$

Usando o Exercício C.2-6, essa probabilidade é igual a

$$\Pr\{X_1\} \cdot \Pr\{X_2 \mid X_1\} \cdot \Pr\{X_3 \mid X_2 \cap X_1\} \cdot \Pr\{X_4 \mid X_3 \cap X_2 \cap X_1\}$$
$$\cdots \Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} \cdots \Pr\{X_n \mid X_{n-1} \cap \cdots \cap X_1\} .$$

Temos que $\Pr\{X_1\} = 1/n$ porque essa é a probabilidade de que uma prioridade escolhida ao acaso em um conjunto de n seja a menor. Em seguida, observamos que $\Pr\{X_2 \mid X_1\} = 1/(n-1)$ porque, dado que o elemento $A[1]$ tem a menor prioridade, cada um dos $n - 1$ elementos restantes apresenta uma chance igual de ter a segunda menor prioridade. Em geral, para $i = 2, 3, \ldots, n$, temos que $\Pr\{X_i \mid X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\} = 1/(n-i+1)$ pois, dado que os elementos $A[1]$ até $A[i-1]$ têm as $i - 1$ menores prioridades (em ordem), cada um dos $n - (i - 1)$ elementos restantes apresenta uma chance igual de ter a i-ésima menor prioridade. Desse modo, temos

$$\Pr\{X_1 \cap X_2 \cap X_3 \cap \cdots \cap X_{n-1} \cap X_n\} = \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right)\cdots\left(\frac{1}{2}\right)\left(\frac{1}{1}\right)$$
$$= \frac{1}{n!},$$

e mostramos que a probabilidade de obtermos a permutação de identidade é $1/n!$.

Podemos estender essa prova a qualquer permutação de prioridades. Considere qualquer permutação fixa $\sigma = \langle \sigma(1), \sigma(2), \ldots, \sigma(n) \rangle$ do conjunto $\{1, 2, \ldots, n\}$. Vamos denotar por r_i a ordenação da prioridade atribuída ao elemento $A[i]$, onde o elemento com a j-ésima menor prioridade tem a ordenação j. Se definirmos X_i como o evento em que o elemento $A[i]$ recebe a $\sigma(i)$-ésima menor prioridade, ou $r_i = \sigma(i)$, a mesma prova ainda se aplicará. Então, se calcularmos a probabilidade de obter qualquer permutação específica, o cálculo será idêntico ao anterior, de forma que a probabilidade de se obter essa permutação também será $1/n!$.

Poderíamos imaginar que, para provar que uma permutação é uma permutação aleatória uniforme é suficiente mostrar que, para cada elemento $A[i]$, a probabilidade de que ele termine na posição j é $1/n$. O Exercício 5.3-4 mostra que essa condição mais fraca é, de fato, insuficiente.

Um método melhor para gerar uma permutação aleatória é permutar o arranjo dado no local. O procedimento RANDOMIZE-IN-PLACE faz isso no tempo $O(n)$. Na iteração i, o elemento $A[i]$ é escolhido ao acaso entre os elementos $A[i]$ a $A[n]$. Depois da iteração i, $A[i]$ nunca é alterado.

```
RANDOMIZE-IN-PLACE(A)
1  n ← comprimento[A]
2  for i ← 1 to n
3       do trocar A[i] ↔ A[RANDOM(i, n)]
```

Usaremos um loop invariante para mostrar que o procedimento RANDOMIZE-IN-PLACE produz uma permutação aleatória uniforme. Dado um conjunto de n elementos, uma permutação de k é uma seqüência contendo k dos n elementos. (Consulte o Apêndice B.) Existem $n!/(n-k)!$ dessas permutações de k possíveis.

Lema 5.5
O procedimento RANDOMIZE-IN-PLACE calcula uma permutação aleatória uniforme.

Prova Usamos o seguinte loop invariante:

Imediatamente antes da i-ésima iteração do loop **for** das linhas 2 e 3, para cada permutação de $(i-1)$ possível, o subarranjo $A[1 .. i-1]$ contém essa permutação de $(i-1)$ com probabilidade $(n-i+1)!/n!$.

Precisamos mostrar que esse invariante é verdadeiro antes da primeira iteração do loop, que cada iteração do loop mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

Inicialização: Considere a situação imediatamente antes da primeira iteração do loop, de forma que $i = 1$. O loop invariante nos diz que, para cada permutação de 0 possível, o subarranjo $A[1 \,..\, 0]$ contém essa permutação de 0 com probabilidade $(n - i + 1)!/n! = n!/n! = 1$. O subarranjo $A[1 \,..\, 0]$ é um subarranjo vazio e uma permutação de 0 não tem nenhum elemento. Desse modo, $A[1 \,..\, 0]$ contém qualquer permutação de 0 com probabilidade 1, e o loop invariante é válido antes da primeira iteração.

Manutenção: Supomos que, imediatamente antes da $(i - 1)$-ésima iteração, cada permutação de $(i - 1)$ possível aparece no subarranjo $A[1 \,..\, i - 1]$ com probabilidade $(n - i + 1)!/n!$, e mostraremos que, após a i-ésima iteração, cada permutação de i possível aparece no subarranjo $A[1 \,..\, i]$ com probabilidade $(n - i)!/n!$. Incrementar i para a próxima iteração manterá então o loop invariante.

Vamos examinar a i-ésima iteração. Considere uma permutação de i específica e denote os elementos que ela contém por $\langle x_1, x_2, ..., x_i \rangle$. Essa permutação consiste em uma permutação de $(i - 1)$ $\langle x_1, ..., x_{i-1} \rangle$ seguida pelo valor x_i que o algoritmo insere em $A[i]$. Seja E_1 o evento em que as primeiras $i - 1$ iterações criaram a permutação de $(i - 1)$ $\langle x_1, ..., x_{i-1} \rangle$ específica em $A[1 \,..\, i - 1]$. Pelo loop invariante, $\Pr\{E_1\} = (n - i + 1)!/n!$. Seja E_2 o evento em que a i-ésima iteração insere x_i na posição $A[i]$. A permutação de i $\langle x_1, ..., x_i \rangle$ é formada em $A[1 \,..\, i]$ precisamente quando tanto E_1 quanto E_2 ocorrem, e assim desejamos calcular $\Pr\{E_2 \cap E_1\}$. Usando a equação (C.14), temos

$$\Pr\{E_2 \cap E_1\} = \Pr\{E_2 \mid E_1\} \Pr\{E_1\} .$$

A probabilidade $\Pr\{E_2 \mid E_1\}$ é igual a $1/(n - i + 1)$ porque, na linha 3, o algoritmo escolhe x_i ao acaso entre os $n - i + 1$ valores nas posições $A[i \,..\, n]$. Desse modo, temos

$$\begin{aligned} \Pr\{E_2 \cap E_1\} &= \Pr\{E_2 | E_1\} \Pr\{E_1\} \\ &= \frac{1}{n - i + 1} \cdot \frac{(n - i + 1)!}{n!} \\ &= \frac{(n - i)!}{n!} . \end{aligned}$$

Término: No término, $i = n + 1$, e temos que o subarranjo $A[1 \,..\, n]$ é uma permutação de n dada com probabilidade $(n - n)!/n! = 1/n!$.

Portanto, RANDOMIZE-IN-PLACE produz uma permutação aleatória uniforme. ■

Com freqüência, um algoritmo aleatório é a maneira mais simples e eficiente de resolver um problema. Usaremos os algoritmos aleatórios ocasionalmente em todo o livro.

Exercícios

5.3-1

O professor Marceau faz objeções ao loop invariante usado na prova do Lema 5.5. Ele questiona se o loop invariante é verdadeiro antes da primeira iteração. Seu raciocínio é que seria possível

quase com a mesma facilidade declarar que um subarranjo vazio não contém nenhuma permutação de 0. Assim, a probabilidade de um subarranjo vazio conter uma permutação de 0 deve ser 0, invalidando assim o loop invariante antes da primeira iteração. Reescreva o procedimento RANDOMIZE-IN-PLACE, de forma que seu loop invariante associado se aplique a um subarranjo não vazio antes da primeira iteração, e modifique a prova do Lema 5.5 de acordo com seu procedimento.

5.3-2

O professor Kelp decide escrever um procedimento que produzirá ao acaso qualquer permutação além da permutação de identidade. Ele propõe o seguinte procedimento:

```
PERMUTE-WITHOUT-IDENTITY(A)
1 n ← comprimento[A]
2 for i ← 1 to n
3     do trocar A[i] ↔ A[RANDOM(i + 1, n)]
```

Esse código faz o que professor Kelp deseja?

5.3-3

Suponha que, em vez de trocar o elemento $A[i]$ por um elemento aleatório do subarranjo $A[i \,..\, n]$, nós o trocamos por um elemento aleatório de qualquer lugar no arranjo:

```
PERMUTE-WITH-ALL(A)}
1 n ← comprimento[A]
2 for i ← 1 to n
3     do trocar A[i] ← A[RANDOM(1, n)]
```

Esse código produz uma permutação aleatória uniforme? Por que ou por que não?

5.3-4

O professor Armstrong sugere o procedimento a seguir para gerar uma permutação aleatória uniforme:

```
PERMUTE-BY-CYCLIC(A)
1 n ← comprimento[A]
2 deslocamento ← RANDOM(1, n)
3 for i ← 1 to n
4     do dest ← i + deslocamento
5         if dest > n
6            then dest ← dest – n
7         B[dest] ← A[i]
8 return B
```

Mostre que cada elemento $A[i]$ tem uma probabilidade $1/n$ de terminar em qualquer posição particular em B. Em seguida, mostre que o professor Armstrong está equivocado, demonstrando que a permutação resultante não é uniformemente aleatória.

5.3-5 ★

Prove que, no arranjo P do procedimento PERMUTE-BY-SORTING, a probabilidade de todos os elementos serem exclusivos é pelo menos $1 - 1/n$.

5.3-6

Explique como implementar o algoritmo PERMUTE-BY-SORTING para tratar o caso em que duas ou mais prioridades são idênticas. Isto é, seu algoritmo deve produzir uma permutação aleatória uniforme, ainda que duas ou mais prioridades sejam idênticas.

★ 5.4 Análise probabilística e usos adicionais de indicadores de variáveis aleatórias

Esta seção avançada ilustra um pouco mais a análise probabilística por meio de quatro exemplos. O primeiro determina a probabilidade de, em uma sala com k pessoas, algum par compartilhar a mesma data de aniversário. O segundo exemplo examina o lançamento aleatório de bolas em caixas. O terceiro investiga "seqüências" de caras consecutivas no lançamento de moedas. O exemplo final analisa uma variante do problema da contratação, na qual você tem de tomar decisões sem entrevistar realmente todos os candidatos.

5.4.1 O paradoxo do aniversário

Nosso primeiro exemplo é o ***paradoxo do aniversário***. Quantas pessoas devem estar em uma sala antes de existir uma chance de 50% de duas delas terem nascido no mesmo dia do ano? A resposta é um número de pessoas surpreendentemente pequeno. O paradoxo é que esse número é de fato muito menor que o número de dias do ano, ou até menor que metade do número de dias de um ano, como veremos.

Para responder à pergunta, indexamos as pessoas na sala com os inteiros $1, 2, \ldots, k$, onde k é o número de pessoas na sala. Ignoramos a questão dos anos bissextos e supomos que todos os anos têm $n = 365$ dias. Para $i = 1, 2, \ldots, k$, seja b_i o dia do ano em que recai o aniversário da pessoa i, onde $1 \le b_i \le n$. Supomos também que os aniversários estão uniformemente distribuídos ao longo dos n dias do ano, de tal forma que $\Pr\{b_i = r\} = 1/n$ para $i = 1, 2, \ldots, k$ e $r = 1, 2, \ldots, n$.

A probabilidade de que duas pessoas, digamos i e j, tenham datas de aniversário coincidentes depende do fato de ser independente a seleção aleatória dos aniversários. Supomos de agora em diante que os aniversários são independentes, e então a probabilidade de que o aniversário de i e o aniversário de j recaiam ambos no dia r é

$$\begin{aligned} \Pr\{b_i = r \text{ e } b_j = r\} &= \Pr\{b_i = r\} \Pr\{b_j = r\} \\ &= 1/n^2 . \end{aligned}$$

Desse modo, a probabilidade de que ambos recaiam no mesmo dia é

$$\begin{aligned} \Pr\{b_i = b_j\} &= \sum_{r=1}^{n} \Pr\{b_i = r \text{ e } b_j = r\} \\ &= \sum_{r=1}^{n} (1/n^2) \\ &= 1/n . \end{aligned} \tag{5.7}$$

Mais intuitivamente, uma vez que b_i é escolhido, a probabilidade de b_j ser escolhido com o mesmo valor é $1/n$. Desse modo, a probabilidade de i e j terem o mesmo dia de aniversário é igual à probabilidade de o aniversário de um deles recair em um determinado dia. Porém, observe que essa coincidência depende da suposição de que os dias de aniversário são independentes.

Podemos analisar a probabilidade de pelo menos 2 entre k pessoas terem aniversários coincidentes examinando o evento complementar. A probabilidade de pelo menos dois aniversários coincidirem é 1 menos a probabilidade de todos os aniversários serem diferentes. O evento em que k pessoas têm aniversários distintos é

$$B_k = \bigcap_{i=1}^{k} A_i ,$$

onde A_i é o evento em que o dia do aniversário da pessoa i é diferente do aniversário da pessoa j para todo $j < i$. Tendo em vista que podemos escrever $B_k = A_k \cap B_{k-1}$, obtemos da equação (C.16) a recorrência

$$\Pr\{B_k\} = \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} , \tag{5.8}$$

onde consideramos $\Pr\{B_1\} = \Pr\{A_1\} = 1$ uma condição inicial. Em outras palavras, a probabilidade de que $b_1, b_2, \ldots, b_k$ sejam aniversários distintos é a probabilidade de $b_1, b_2, \ldots, b_{k-1}$ serem aniversários distintos vezes a probabilidade de que $b_k \ldots b_i$ para $i = 1, 2, \ldots, k-1$, dado que $b_1, b_2, \ldots, b_{k-1}$ são distintos.

Se $b_1, b_2, \ldots, b_{k-1}$ são distintos, a probabilidade condicional de que $b_k \ldots b_i$ para $i = 1, 2, \ldots, k-1$ é $\Pr\{A_k \mid B_{k-1}\} = (n-k+1)/n$, pois, dos n dias, existem $n-(k-1)$ que não são tomados. Aplicamos de forma iterativa a recorrência (5.8) para obter

$$\begin{aligned}
\Pr\{B_k\} &= \Pr\{B_{k-1}\} \Pr\{A_k \mid B_{k-1}\} \\
&= \Pr\{B_{k-1}\} \Pr\{A_{k-1} \mid B_{k-2}\} \Pr\{A_k \mid B_{k-1}\} \\
&\;\;\vdots \\
&= \Pr\{B_1\} \Pr\{A_2\} \mid B_1\} \Pr\{A_3 \mid B_2\} \ldots \Pr\{A_k \mid B_{k-1}\} \\
&= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right) \cdots \left(\frac{n-k+1}{n}\right) \\
&= 1 \cdot \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right) \cdots \left(1-\frac{k-1}{n}\right).
\end{aligned}$$

A desigualdade (3.11), $1 + x \le e^x$, nos fornece

$$\begin{aligned}
\Pr\{B_k\} &\le e^{-1/n} e^{-2/n} \ldots e^{-(k-1)/n} \\
&= e^{-\sum_{i=1}^{k-1} i/n} \\
&= e^{-k(k-1)/2n} \\
&\le 1/2
\end{aligned}$$

quando $-k(k-1)/2n \le \ln(1/2)$. A probabilidade de que todos os k aniversários sejam distintos é no máximo 1/2 quando $k(k-1) \ge 2n \ln 2$ ou, resolvendo a equação quadrática, quando $k \ge (1+\sqrt{1+(8 \ln 2)n})/2$. Para $n = 365$, devemos ter $k \ge 23$. Portanto, se pelo menos 23 pessoas estão em uma sala, a probabilidade é de pelo menos 1/2 de que no mínimo duas pessoas tenham a mesma data de aniversário. Em Marte, um ano tem a duração de 669 dias marcianos; então, seriam necessários 31 marcianos para conseguirmos o mesmo efeito.

Uma análise usando indicadores de variáveis aleatórias

Podemos usar indicadores de variáveis aleatórias para fornecer uma análise mais simples, embora aproximada, do paradoxo do aniversário. Para cada par (i, j) das k pessoas na sala, vamos definir o indicador de variável aleatória X_{ij}, para $1 \leq i < j \leq k$, por

$$X_{ij} = \text{I}\,\{\text{a pessoa } i \text{ e a pessoa } j \text{ têm o mesmo dia de aniversário}\}$$

$$= \begin{cases} 1 & \text{se a pessoa } i \text{ e a pessoa } j \text{ têm o mesmo dia de aniversário},\\ 0 & \text{em caso contrário}. \end{cases}$$

Pela equação (5.7), a probabilidade de que duas pessoas tenham aniversários coincidentes é $1/n$ e, desse modo, pelo Lema 5.1, temos

$$\begin{aligned} \text{E}[X_{ij}] &= \Pr\{\text{a pessoa } i \text{ e a pessoa } j \text{ têm o mesmo dia aniversário}\}\\ &= 1/n\,. \end{aligned}$$

Sendo X a variável aleatória que conta o número de pares de indivíduos que têm a mesma data de aniversário, temos

$$X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\,.$$

Tomando as expectativas de ambos os lados e aplicando a linearidade de expectativa, obtemos

$$\begin{aligned} \text{E}[X] &= \text{E}\left[\sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\right]\\ &= \sum_{i=1}^{k} \sum_{j=i+1}^{k} \text{E}[X_{ij}]\\ &= \binom{k}{2}\frac{1}{n}\\ &= \frac{k(k-1)}{2n}\,. \end{aligned}$$

Quando $k(k-1) \geq 2n$, o número esperado de pares de pessoas com a mesma data de aniversário é pelo menos 1. Desse modo, se tivermos pelo menos $\sqrt{2n} + 1$ indivíduos em uma sala, poderemos esperar que no mínimo dois deles façam aniversário no mesmo dia. Para $n = 365$, se $k = 28$, o número esperado de pares com o mesmo dia de aniversário é $(28 \cdot 27)/(2 \cdot 365) \approx 1{,}0356$. Assim, com pelo menos 28 pessoas, esperamos encontrar no mínimo um par de aniversários coincidentes. Em Marte, onde um ano corresponde a 669 dias marcianos, precisaríamos de pelo menos 38 marcianos.

A primeira análise, que usou somente probabilidades, determinou o número de pessoas necessárias para que a probabilidade de existir um par de datas de aniversário coincidentes exceda 1/2, e a segunda análise, que empregou indicadores de variáveis aleatórias, determinou o número tal que a quantidade esperada de aniversários coincidentes é 1. Embora os números exatos de pessoas sejam diferentes nas duas situações, eles são assintoticamente iguais: $\Theta(\sqrt{n})$.

5.4.2 Bolas e caixas

Considere o processo de lançar aleatoriamente bolas idênticas em b caixas, com a numeração 1, 2, ..., b. Os lançamentos são independentes, e em cada lançamento a bola tem igual probabilidade de terminar em qualquer caixa. A probabilidade de uma bola lançada cair em qualquer caixa dada é $1/b$. Desse modo, o processo de lançamento de bolas é uma seqüência de experiências de Bernoulli (consulte o Apêndice C, Seção C.4) com uma probabilidade de sucesso $1/b$, onde sucesso significa que a bola cai na caixa dada. Esse modelo é particularmente útil para analisar o hash (consulte o Capítulo 11), e podemos responder a uma variedade de perguntas interessantes sobre o processo de lançamento de bolas. (O problema C-1 formula perguntas adicionais sobre bolas e caixas.)

Quantas bolas caem em uma determinada caixa? O número de bolas que caem em uma caixa dada segue a distribuição binomial $b(k; n, 1/b)$. Se n bolas são lançadas, a equação (C.36) nos informa que o número esperado de bolas que caem na caixa dada é n/b.

Quantas bolas devem ser lançadas, em média, até uma caixa dada conter uma bola? O número de lances até a caixa dada receber uma bola segue a distribuição geométrica com probabilidade $1/b$ e, pela equação (C.31), o número esperado de lances até o sucesso é $1/(1/b) = b$.

Quantas bolas devem ser lançadas até toda caixa conter pelo menos uma bola? Vamos chamar um lançamento em que uma bola cai em uma caixa vazia de "acerto". Queremos saber o número esperado n de lançamentos necessários para se conseguir b acertos.

Os acertos podem ser usados para particionar os n lançamentos em fases. A i-ésima fase consiste nos lançamentos depois do $(i-1)$-ésimo acerto até o i-ésimo acerto. A primeira fase consiste no primeiro lançamento, pois temos a garantia de um acerto quando todas as caixas estão vazias. Para cada lançamento durante a i-ésima fase, existem $i-1$ caixas que contêm bolas e $b-i+1$ caixas vazias. Desse modo, para cada lançamento na i-ésima fase, a probabilidade de se obter um acerto é $(b-i+1)/b$.

Seja n_i o número de lançamentos na i-ésima fase. Portanto, o número de lançamentos exigidos para se conseguir b acertos é $n = \sum_{i=1}^{b} n_i$. Cada variável aleatória n_i tem uma distribuição geométrica com probabilidade de sucesso $(b-i+1)/b$ e, pela equação (C.31),

$$E[n_i] = \frac{b}{b-i+1}.$$

Por linearidade de expectativa,

$$\begin{aligned}
E[n] &= E\left[\sum_{i=1}^{b} n_i\right] \\
&= \sum_{i=1}^{b} E[n_i] \\
&= \sum_{i=1}^{b} \frac{b}{b-i+1} \\
&= b\sum_{i=1}^{b} \frac{1}{i} \\
&= b(\ln b + O(1)).
\end{aligned}$$

A última linha decorre do limite (A.7) sobre a série harmônica. Então, ocorrem aproximadamente $b \ln b$ lançamentos antes de podermos esperar que toda caixa tenha uma bola. Esse pro-

blema também é conhecido como o ***problema do colecionador de cupons***, e nos diz que uma pessoa que tenta colecionar cada um de b cupons diferentes deve adquirir aproximadamente $b \ln b$ cupons obtidos ao acaso para ter sucesso.

5.4.3 Seqüências

Suponha que você lance uma moeda comum n vezes. Qual é a seqüência mais longa de caras consecutivas que você espera ver? A resposta é $\Theta(\lg n)$, como mostra a análise a seguir.

Primeiro, provamos que o comprimento esperado da mais longa seqüência de caras é $O(\lg n)$. A probabilidade de que cada lançamento de moeda seja uma cara é 1/2. Seja A_{ik} o evento em que uma seqüência de caras de comprimento no mínimo k começa com o i-ésimo lançamento de moeda ou, mais precisamente, o evento em que os k lançamentos consecutivos de moedas $i, i + 1, \ldots, i + k - 1$ produzem somente caras, onde $1 \le k \le n$ e $1 \le i \le n - k + 1$. Como os lançamentos de moedas são mutuamente independentes, para qualquer evento dado A_{ik}, a probabilidade de que todos os k lançamentos sejam caras é

$$\Pr\{A_{ik}\} = 1/2^k . \tag{5.9}$$

Para $k = 2\lceil \lg n \rceil$,

$$\begin{aligned}\Pr\{A_{i,2\lceil \lg n \rceil}\} &= 1/2^{2\lceil \lg n \rceil} \\ &\le 1/2^{2 \lg n} \\ &= 1/n^2 ,\end{aligned}$$

e, portanto, a probabilidade de uma seqüência de caras de comprimento pelo menos igual a $2\lceil \lg n \rceil$ começar na posição i é bastante pequena. Há no máximo $n - 2\lceil \lg n \rceil + 1$ posições onde tal seqüência pode começar. A probabilidade de uma seqüência de caras de comprimento pelo menos $2\lceil \lg n \rceil$ começar em qualquer lugar é então

$$\begin{aligned}\Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil}\right\} &\le \sum_{i=1}^{n-2\lceil \lg n \rceil+1} 1/n^2 \\ &< \sum_{i=1}^{n} 1/n^2 \\ &= 1/n ,\end{aligned} \tag{5.10}$$

pois, pela desigualdade de Boole (C.18), a probabilidade de uma união de eventos é no máximo a soma das probabilidades dos eventos individuais. (Observe que a desigualdade de Boole é válida até mesmo para eventos como esses, que não são independentes.)

Agora usamos a desigualdade (5.10) para limitar o comprimento da seqüência mais longa. Para $j = 0, 1, 2, \ldots, n$, seja L_j o evento em que a seqüência mais longa de caras tem comprimento exatamente j, e seja L o comprimento da seqüência mais longa. Pela definição de valor esperado,

$$E[L] = \sum_{j=0}^{n} j \Pr\{L_j\} . \tag{5.11}$$

Poderíamos tentar avaliar essa soma usando limites superiores sobre cada Pr $\{L_j\}$ semelhantes aos que foram calculados na desigualdade (5.10). Infelizmente, esse método produziria limites fracos. Porém, podemos usar alguma intuição obtida pela análise anterior para obter um bom limite. Informalmente, observamos que para nenhum termo individual no somatório da equação (5.11) ambos os fatores, j e Pr $\{L_j\}$, são grandes. Por quê? Quando $j \geq 2\lceil \lg n \rceil$, então Pr $\{L_j\}$ é muito pequeno e, quando $j < 2\lceil \lg n \rceil$, então j é bastante pequeno. De maneira mais formal, notamos que os eventos L_j para $j = 0, 1, \ldots, n$ são disjuntos, e assim a probabilidade de uma seqüência de caras de comprimento no mínimo $2\lceil \lg n \rceil$ começar em qualquer lugar é $\sum_{j=2\lceil \lg n \rceil}^{n}$ $\Pr\{L_j\}$. Pela desigualdade (5.10), temos $\sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} < 1/n$. Além disso, notando que $\sum_{j=0}^{n} \Pr\{L_j\} = 1$, temos $\sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} \leq 1$. Desse modo, obtemos

$$\begin{aligned}
\mathrm{E}[L] &= \sum_{j=0}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{2\lceil \lg n \rceil - 1} j \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} j \Pr\{L_j\} \\
&= \sum_{j=0}^{2\lceil \lg n \rceil - 1} (2\lceil \lg n \rceil) \Pr\{L_j\} + \sum_{j=2\lceil \lg n \rceil}^{n} n \Pr\{L_j\} \\
&= 2\lceil \lg n \rceil \sum_{j=0}^{2\lceil \lg n \rceil - 1} \Pr\{L_j\} + n \sum_{j=2\lceil \lg n \rceil}^{n} \Pr\{L_j\} \\
&< 2\lceil \lg n \rceil \cdot 1 + n \cdot (1/n) \\
&= O(\lg n) .
\end{aligned}$$

As chances de que uma seqüência de caras exceda $r\lceil \lg n \rceil$ lançamentos diminui rapidamente com r. Para $r \geq 1$, a probabilidade de uma seqüência de $r\lceil \lg n \rceil$ caras começar na posição i é

$$\begin{aligned}
\Pr\{A_{i, r\lceil \lg n \rceil}\} &= 1/2^{r\lceil \lg n \rceil} \\
&\leq 1/n^r .
\end{aligned}$$

Desse modo, a probabilidade de que a seqüência mais longa seja pelo menos $r\lceil \lg n \rceil$ é no máximo igual a $n/n^r = 1/n^{r-1}$ ou, de modo equivalente, a probabilidade de que a seqüência mais longa tenha comprimento menor que $r\lceil \lg n \rceil$ é no mínimo $1 - 1/n^{r-1}$.

Como um exemplo, para $n = 1000$ lançamentos de moedas, a probabilidade de haver uma seqüência de pelo menos $2\lceil \lg n \rceil = 20$ caras é no máximo $1/n = 1/1000$. As chances de haver uma seqüência mais longa que $3\lceil \lg n \rceil = 30$ caras é no máximo $1/n^2 = 1/1.000.000$.

Agora, vamos provar um limite complementar inferior: o comprimento esperado da seqüência mais longa de caras em n lançamentos de moedas é $\Omega(\lg n)$. Para provar esse limite, procuramos por seqüências de comprimento s particionando os n lançamentos em aproximadamente n/s grupos de s lançamentos cada. Se escolhermos $s = \lfloor (\lg n)/2 \rfloor$, poderemos mostrar que é provável que pelo menos um desses grupos mostre apenas caras e, conseqüentemente, é provável que a seqüência mais longa tenha comprimento pelo menos igual a $s = \Omega(\lg n)$. Mostraremos então que a seqüência mais longa tem comprimento esperado $\Omega(\lg n)$.

Particionamos os n lançamentos de moedas em pelo menos $\lfloor n/\lfloor (\lg n)/2 \rfloor \rfloor$ grupos de $\lfloor (\lg n)/2 \rfloor$ lançamentos sucessivos, e limitamos a probabilidade de não surgir nenhum grupo formado apenas por caras. Pela equação (5.9), a probabilidade do grupo que começa na posição i mostrar apenas caras é

$$\Pr\{A_{i,\lfloor(\lg n/2\rfloor}\} = 1/2^{\lfloor(\lg n/2\rfloor}$$
$$\geq 1/\sqrt{n}\ .$$

A probabilidade de uma seqüência de caras de comprimento pelo menos igual a $\lfloor(\lg n)/2\rfloor$ não começar na posição i é então no máximo $1 - 1/\sqrt{n}$. Tendo em vista que os $\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor$ grupos são formados a partir de lançamentos de moedas mutuamente exclusivos e independentes, a probabilidade de que cada um desses grupos *deixe* de ser uma seqüência de comprimento $\lfloor(\lg n)/2\rfloor$ é no máximo

$$(1-1/\sqrt{n})^{\lfloor n/\lfloor(\lg n)/2\rfloor\rfloor} \leq (1-1/\sqrt{n})^{n/\lfloor(\lg n)/2\rfloor-1}$$

$$= (1-1/\sqrt{n})^{2n/\lg n-1}$$

$$= e^{-(2n/\lg n-1)/\sqrt{n}}$$

$$= O(e^{-\lg n})$$

$$= O(1/n)\ .$$

Para esse argumento, usamos a desigualdade (3.11), $1 + x \leq e^x$ e o fato, que talvez você deseje verificar, de que $(2n/\lg n - 1)/\sqrt{n} \geq \lg n$ para n suficientemente grande.

Desse modo, a probabilidade de que a seqüência mais longa exceda $\lfloor(\lg n)/2\rfloor$ é

$$\sum_{j=\lfloor(\lg n)/2\rfloor+1}^{n} \Pr\{L_j\} \geq 1-O(1/n)\ . \tag{5.12}$$

Agora podemos calcular um limite inferior sobre o comprimento esperado da seqüência mais longa, começando com a equação (5.11) e procedendo de forma semelhante à nossa análise do limite superior:

$$\mathrm{E}[L] = \sum_{j=0}^{n} j\Pr\{L_j\}$$

$$= \sum_{j=0}^{\lceil(\lg n)/2\rceil} j\Pr\{L_j\} + \sum_{j=\lceil(\lg n)/2\rceil+1}^{n} j\Pr\{L_j\}$$

$$= \sum_{j=0}^{\lceil(\lg n)/2\rceil} 0\cdot\Pr\{L_j\} + \sum_{j=\lceil(\lg n)/2\rceil+1}^{n} \lfloor(\lg n)/2\rfloor\Pr\{L_j\}$$

$$= 0\cdot\sum_{j=0}^{\lceil(\lg n)/2\rceil} \Pr\{L_j\} + \lfloor(\lg n)/2\rfloor \sum_{j=\lceil(\lg n)/2\rceil+1}^{n} \Pr\{L_j\}$$

$$\geq 0 + \lfloor(\lg n)/2\rfloor\,(1 - O(1/n)) \qquad \text{(pela desigualdade (5.12))}$$

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

Como ocorre no caso do paradoxo do aniversário, podemos obter uma análise mais simples, embora aproximada, usando indicadores de variáveis aleatórias. Seja $X_{ik} = \mathrm{I}\{A_{ik}\}$ o indicador de variável aleatória associado a uma seqüência de caras de comprimento no mínimo k que começam com o i-ésimo lançamento de moeda. Para contar o número total de tais seqüências, definimos

$$X = \sum_{i=1}^{n-k+1} X_{ik} \; .$$

Tomando as expectativas e usando a linearidade de expectativa, temos

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-k+1} X_{ik}\right] \\
&= \sum_{i=1}^{n-k+1} \mathrm{E}[X_{ik}] \\
&= \sum_{i=1}^{n-k+1} \Pr\{A_{ik}\} \\
&= \sum_{i=1}^{n-k+1} 1/2^k \\
&\frac{n-k+1}{2^k} \; .
\end{aligned}$$

Conectando diversos valores para k, podemos calcular o número esperado de seqüências de comprimento k. Se esse número é grande (muito maior que 1), então espera-se que ocorram muitas seqüências de comprimento k, e a probabilidade de ocorrer uma é alta. Se esse número é pequeno (muito menor que 1), então espera-se que ocorram muito poucas seqüências de comprimento k, e a probabilidade de ocorrer uma é baixa. Se $k = c \lg n$, para alguma constante positiva c, obtemos

$$\begin{aligned}
\mathrm{E}[X] &= \frac{n - c\lg n + 1}{2^{c \lg n}}. \\
&= \frac{n - c\lg n + 1}{n^c} \\
&= \frac{1}{n^{c-1}} - \frac{(c \lg n - 1)/n}{n^{c-1}} \\
&= \Theta(1/n^{c-1}) \; .
\end{aligned}$$

Se c é grande, o número esperado de seqüências de comprimento $c \lg n$ é muito pequeno, e concluímos que é improvável que elas ocorram. Por outro lado, se $c < 1/2$, então obtemos $\mathrm{E}[X] = \Theta(1/n^{1/2-1}) = \Theta(n^{1/2})$, e esperamos que exista um número grande de seqüências de comprimento $(1/2) \lg n$. Portanto, é muito provável que ocorra uma seqüência de tal comprimento. Somente a partir dessas estimativas grosseiras, podemos concluir que o comprimento esperado da seqüência mais longa é $\Theta(\lg n)$.

5.4.4 O problema da contratação on-line

Como um exemplo final, examinaremos uma variante do problema da contratação. Suponha agora que não desejamos entrevistar todos os candidatos a fim de encontrar o melhor. Também não desejamos contratar e despedir à medida que encontrarmos candidatos cada vez melhores. Em vez disso, estamos dispostos a aceitar um candidato próximo do melhor, em troca de contratar exatamente uma vez. Devemos obedecer a um requisito da empresa: depois de cada entrevista, devemos oferecer imediatamente o cargo ao candidato ou dizer a ele que não será possível contratá-lo. Qual é o compromisso entre minimizar a quantidade de entrevistas e maximizar a qualidade do candidato contratado?

Podemos modelar esse problema da maneira ilustrada a seguir. Após encontrar um candidato, somos capazes de dar a cada um deles uma pontuação; seja *pontuação*(i) a pontuação dada ao i-ésimo candidato, e suponha que não existam dois candidatos que recebam a mesma pontuação. Depois de ver j candidatos, sabemos qual dos j candidatos tem a pontuação mais alta, mas não sabemos se qualquer dos $n - j$ candidatos restantes terá uma pontuação mais alta. Decidimos adotar a estratégia de selecionar um inteiro positivo $k < n$, entrevistando e depois rejeitando os primeiros k candidatos, e contratando daí em diante o primeiro candidato que tem uma pontuação mais alta que todos os candidatos anteriores. Se notarmos que o candidato mais bem qualificado se encontrava entre os k primeiros entrevistados, então contrataremos o n-ésimo candidato. Essa estratégia é formalizada no procedimento ON-LINE-MAXIMUM(k, n), que aparece a seguir. O procedimento ON-LINE-MAXIMUM retorna o índice do candidato que desejamos contratar.

```
ON-LINE-MAXIMUM(k, n)
1 melhorpontuação ← -∞
2 for i ← 1 to k
3     do if pontuação(i) > melhorpontuação
4           then melhorpontuação ← pontuação(i)
5 for i ← k + 1 to n
6     do if pontuação(i) > melhorpontuação
7           then return i
8 return n
```

Desejamos determinar, para cada valor possível de k, a probabilidade de contratarmos o candidato mais bem qualificado. Em seguida, escolheremos o melhor possível k, e implementaremos a estratégia com esse valor. Por enquanto, suponha que k seja fixo. Seja $M(j) = \max_{1 \le i \le j} \{pontuação(i)\}$ a pontuação máxima entre os candidatos 1 a j. Seja S o evento em que temos sucesso na escolha do candidato mais bem qualificado, e seja S_i o evento em que temos sucesso quando o candidato mais bem qualificado for o i-ésimo entrevistado. Tendo em vista que diversos S_i são disjuntos, temos que $\Pr\{S\} = \sum_{i=1}^{n} \Pr\{S_i\}$. Observando que nunca temos sucesso quando o candidato mais bem qualificado é um dos k primeiros, temos que $\Pr\{S_i\} = 0$ para $i = 1, 2, ..., k$. Desse modo, obtemos

$$\Pr\{S\} = \sum_{i=k+1}^{n} \Pr\{S_i\} . \tag{5.13}$$

Agora calculamos $\Pr\{S_i\}$. Para se ter sucesso quando o candidato mais bem qualificado é o i-ésimo, duas coisas devem acontecer. Primeiro, o candidato mais bem qualificado deve estar na posição i, um evento que denotamos por B_i. Segundo, o algoritmo não deve selecionar quaisquer dos candidatos nas posições $k + 1$ a $i - 1$, o que acontece somente se, para cada j tal que $k + 1 \le j \le i - 1$, encontramos *pontuação*(j) < *melhorpontuação* na linha 6. (Como as pontuações são exclusivas, podemos ignorar a possibilidade de *pontuação*(j) = *melhorpontuação*.) Em ou-

tras palavras, deve ser o caso de que todos os valores *pontuação*$(k + 1)$ até *pontuação*$(i - 1)$ são menores que $M(k)$; se quaisquer deles forem maiores que $M(k)$, retornaremos em vez disso o índice do primeiro que for maior. Usamos O_i para denotar o evento em que nenhum dos candidatos nas posições $k + 1$ a $i - 1$ é escolhido. Felizmente, os dois eventos B_i e O_i são independentes. O evento O_i depende apenas da ordenação relativa dos valores nas posições 1 a $i - 1$, enquanto B_i depende apenas do fato de o valor na posição i SER maior que todos os valores de 1 até $i - 1$. A ordenação das posições 1 a $i - 1$ não afeta o fato de i ser maior que todas elas, e o valor de i não afeta a ordenação das posições 1 a $i - 1$. Desse modo, podemos aplicar a equação (C.15) para obter

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\} \Pr\{O_i\} .$$

A probabilidade $\Pr\{B_i\}$ é claramente $1/n$, pois o máximo tem igual probabilidade de estar em qualquer uma das n posições. Para o evento O_i ocorrer, o valor máximo nas posições 1 a $i - 1$ deve estar em uma das k primeiras posições, e é igualmente provável que esteja em qualquer dessas $i - 1$ posições. Conseqüentemente, $\Pr\{O_i\} = k/(i - 1)$ e $\Pr\{S_i\} = k/(n(i - 1))$. Usando a equação (5.13), temos

$$\begin{aligned} \Pr\{S\} &= \sum_{i=k+1}^{n} \Pr\{S_i\} \\ &= \sum_{i=k+1}^{n} \frac{k}{n(i-1)} \\ &= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1} \\ &= \frac{k}{n} \sum_{i=k}^{n-1} \frac{1}{i} . \end{aligned}$$

Fazemos a aproximação por integrais para limitar esse somatório acima e abaixo. Pelas desigualdades (A.12), temos

$$\int_{k}^{n} \frac{1}{x} dx \le \sum_{i=k}^{n-1} \le \int_{k-1}^{n-1} \frac{1}{x} dx .$$

A avaliação dessas integrais definidas nos dá os limites

$$\frac{k}{n}(\ln n - \ln k) \le \Pr\{S\} \le \frac{k}{n}(\ln(n-1) - \ln(k-1)) ,$$

que fornecem um limite bastante restrito para $\Pr\{S\}$. Como desejamos maximizar nossa probabilidade de sucesso, vamos nos concentrar na escolha do valor de k que maximiza o limite inferior sobre $\Pr\{S\}$. (Além disso, a expressão do limite inferior é mais fácil de maximizar que a expressão do limite superior.) Fazendo a diferenciação da expressão $(k/n)(\ln n - \ln k)$ com relação a k, obtemos

$$\frac{1}{n}(\ln n - \ln k - 1) .$$

Definindo essa derivada como igual a 0, vemos que o limite inferior sobre a probabilidade é maximizado quando $\ln k = \ln n - 1 = \ln(n/e)$ ou, de modo equivalente, quando $k = n/e$. Desse modo, se implementarmos nossa estratégia com $k = n/e$, teremos sucesso na contratação do nosso candidato mais bem qualificado com probabilidade pelo menos $1/e$.

Exercícios

5.4-1

Quantas pessoas deve haver em uma sala para que a probabilidade de que alguém tenha a mesma data de aniversário que você seja pelo menos 1/2? Quantas pessoas deve haver para que a probabilidade de que pelo menos duas pessoas façam aniversário em 7 de setembro seja maior que 1/2?

5.4-2

Suponha que sejam lançadas bolas em b caixas. Cada lançamento é independente, e cada bola tem a mesma probabilidade de cair em qualquer caixa. Qual é o número esperado de lançamentos de bolas antes que pelo menos uma das caixas contenha duas bolas?

5.4-3 ★

Para a análise do paradoxo do aniversário, é importante que os aniversários sejam mutuamente independentes, ou é suficiente a independência aos pares? Justifique sua resposta.

5.4-4 ★

Quantas pessoas devem ser convidadas para uma festa, a fim de tornar provável que existam *três* pessoas com a mesma data de aniversário?

5.4-5 ★

Qual é a probabilidade de uma cadeia de k elementos sobre um conjunto de tamanho n ser na realidade uma permutação de k elementos? De que modo essa pergunta se relaciona ao paradoxo do aniversário?

5.4-6 ★

Suponha que n bolas sejam lançadas em n caixas, onde cada lançamento é independente e a bola tem igual probabilidade de cair em qualquer caixa. Qual é o número esperado de caixas vazias? Qual é o número esperado de caixas com exatamente uma bola?

5.4-7 ★

Torne mais nítido o limite inferior sobre o comprimento da seqüência mostrando que, em n lançamentos de uma moeda comum, a probabilidade de não ocorrer nenhuma seqüência mais longa que $\lg n - 2 \lg \lg n$ caras consecutivas é menor que $1/n$.

Problemas

5-1 Contagem probabilística

Com um contador de b bits, só podemos contar normalmente até $2^b - 1$. Com a ***contagem probabilística*** de R. Morris, podemos contar até um valor muito maior, a expensas de alguma perda de precisão.

Seja um valor de contador i que representa uma contagem n_i para $i = 0, 1, \ldots, 2^b - 1$, onde os n_i formam uma seqüência crescente de valores não negativos. Supomos que o valor inicial do contador é 0, representando uma contagem de $n_0 = 0$. A operação de INCREMENT atua de maneira probabilística sobre um contador que contém o valor i. Se $i = 2^b - 1$, então é relatado um erro de estouro (overflow). Caso contrário, o contador é incrementado em 1 com probabilidade $1/(n_{i+1} - n_i)$, e permanece inalterado com probabilidade $1 - 1/(n_{i+1} - n_i)$.

Se selecionarmos $n_i = i$ para todo $i \geq 0$, então o contador será um contador comum. Surgirão situações mais interessantes se selecionarmos, digamos, $n_i = 2^{i-1}$ para $i > 0$ ou $n_i = F_i$ (o i-ésimo número de Fibonacci – consulte a Seção 3.2).

Para este problema, suponha que n_{2^b-1} seja é grande o suficiente para que a probabilidade de um erro de estouro seja desprezível.

a. Mostre que o valor esperado representado pelo contador depois de n operações de INCREMENT serem executadas é exatamente n.

b. A análise da variância da contagem representada pelo contador depende da seqüência dos n_i. Vamos considerar um caso simples: $n_i = 100i$ para todo $i \geq 0$. Estime a variância no valor representado pelo registrador depois de n opções de INCREMENT terem sido executadas.

5-2 Pesquisa em um arranjo não ordenado

Este problema examina três algoritmos para pesquisar um valor x em um arranjo não ordenado A que consiste em n elementos.

Considere a estratégia aleatória a seguir: escolha um índice aleatório i em A. Se $A[i] = x$, então terminamos; caso contrário, continuamos a pesquisa escolhendo um novo índice aleatório em A. Continuamos a escolher índices aleatórios em A até encontrarmos um índice j tal que $A[j] = x$ ou até verificarmos todos os elementos de A. Observe que cada escolha é feita a partir do conjunto inteiro de índices, de forma que podemos examinar um dado elemento mais de uma vez.

a. Escreva pseudocódigo para um procedimento RANDOM-SEARCH para implementar a estratégia anterior. Certifique-se de que o algoritmo termina depois que todos os índices em A são escolhidos.

b. Suponha que exista exatamente um índice i tal que $A[i] = x$. Qual é o número esperado de índices em A que devem ser escolhidos antes de x ser encontrado e RANDOM-SEARCH terminar?

c. Generalizando sua solução para a parte (b), suponha que existam $k \geq 1$ índices i tais que $A[i] = x$. Qual é o número esperado de índices em A que devem ser escolhidos antes de x ser encontrado e RANDOM-SEARCH terminar? Sua resposta deve ser uma função de n e k.

d. Suponha que não exista nenhum índice i tal que $A[i] = x$. Qual é o número esperado de índices em A que devem ser escolhidos antes de todos os elementos de A terem sido verificados e RANDOM-SEARCH terminar?

Agora, considere um algoritmo de pesquisa linear determinística, que denominamos DETERMINISTIC-SEARCH. Especificamente, o algoritmo pesquisa A para x em ordem, considerando $A[1], A[2], A[3], \ldots, A[n]$ até $A[i] = x$ ser encontrado ou alcançar o fim do arranjo. Suponha que todas as permutações possíveis do arranjo de entrada sejam igualmente prováveis.

e. Suponha que exista exatamente um índice i tal que $A[i] = x$. Qual é o tempo de execução esperado de DETERMINISTIC-SEARCH? Qual é o tempo de execução no pior caso de DETERMINISTIC-SEARCH?

f. Generalizando sua solução para parte (e), suponha que existam $k \geq 1$ índices i tais que $A[i] = x$. Qual é o tempo de execução esperado de DETERMINISTIC-SEARCH? Qual é tempo de execução no pior caso de DETERMINISTIC-SEARCH? Sua resposta deve ser uma função de n e k.

g. Suponha que não exista nenhum índice i tal que $A[i] = x$. Qual é o tempo de execução esperado de DETERMINISTIC-SEARCH? Qual é o tempo de execução no pior caso de DETERMINISTIC-SEARCH?

Finalmente, considere um algoritmo aleatório SCRAMBLE-SEARCH que funciona primeiro permutando aleatoriamente o arranjo de entrada e depois executando a pesquisa linear determinística anterior sobre o arranjo permutado resultante.

h. Sendo k o número de índices i tais que $A[i] = x$, forneça os tempos de execução no pior caso e esperado de SCRAMBLE-SEARCH para os casos em que $k = 0$ e $k = 1$. Generalize sua solução para tratar o caso em que $k \geq 1$.

i. Qual dos três algoritmos de pesquisa você usaria? Explique sua resposta.

Notas do capítulo

Bollobás [44], Hofri [151] e Spencer [283] contêm um grande número de técnicas probabilísticas avançadas. As vantagens dos algoritmos aleatórios são discutidas e pesquisadas por Karp [174] e Rabin [253]. O livro-texto de Motwani e Raghavan [228] apresenta um tratamento extensivo de algoritmos aleatórios.

Têm sido amplamente estudadas diversas variantes do problema da contratação. Esses problemas são mais comumente referidos como "problemas da secretária". Um exemplo de trabalho nessa área é o artigo de Ajtai, Meggido e Waarts [12].

Parte II

Ordenação e estatísticas de ordem

Introdução

Esta parte apresenta vários algoritmos que resolvem o ***problema de ordenação*** a seguir:

Entrada: Uma seqüência de n números $\langle a_1, a_2, ..., a_n \rangle$.

Saída: Uma permutação (reordenação) $\langle a'_1, a'_2, ..., a'_n \rangle$ da seqüência de entrada, tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

A seqüência de entrada normalmente é um arranjo de n elementos, embora possa ser representada de algum outro modo, como uma lista ligada.

A estrutura dos dados

Na prática, os números a serem ordenados raramente são valores isolados. Em geral, cada um deles faz parte de uma coleção de dados chamada ***registro***. Cada registro contém uma ***chave***, que é o valor a ser ordenado, e o restante do registro consiste em ***dados satélite***, que quase sempre são transportados junto com a chave. Na prática, quando um algoritmo de ordenação permuta as chaves, ele também deve permutar os dados satélite. Se cada registro inclui uma grande quantidade de dados satélite, muitas vezes permutamos um arranjo de ponteiros para os registros em lugar dos próprios registros, a fim de minimizar a movimentação de dados.

De certo modo, são esses detalhes de implementação que distinguem um algoritmo de um programa completamente implementado. O fato de ordenarmos números individuais ou grandes registros que contêm números é irrelevante para o *método* pelo qual um procedimento de ordenação determina a seqüência ordenada. Desse modo, quando nos concentramos no problema de ordenação, em geral supomos que a entrada consiste apenas em números. A tradução de um algoritmo para ordenação de números em um programa para ordenação de registros é conceitualmente direta, embora em uma situação específica de engenharia possam surgir outras sutilezas que fazem da tarefa real de programação um desafio.

Por que ordenar?

Muitos cientistas de computação consideram a ordenação o problema mais fundamental no estudo de algoritmos. Há várias razões:

- Às vezes, a necessidade de ordenar informações é inerente a uma aplicação. Por exemplo, para preparar os extratos de clientes, os bancos precisam ordenar os cheques pelo número do cheque.
- Os algoritmos freqüentemente usam a ordenação como uma sub-rotina chave. Por exemplo, um programa que apresenta objetos gráficos dispostos em camadas uns sobre os outros talvez tenha de ordenar os objetos de acordo com uma relação "acima", de forma a poder desenhar esses objetos de baixo para cima. Neste texto, veremos numerosos algoritmos que utilizam a ordenação como uma sub-rotina.
- Existe uma ampla variedade de algoritmos de ordenação, e eles empregam um rico conjunto de técnicas. De fato, muitas técnicas importantes usadas ao longo do projeto de algoritmos são representadas no corpo de algoritmos de ordenação que foram desenvolvidos ao longo dos anos. Desse modo, a ordenação também é um problema de interesse histórico.
- A ordenação é um problema para o qual podemos demonstrar um limite inferior não trivial (como faremos no Capítulo 8). Nossos melhores limites superiores correspondem ao limite inferior assintoticamente, e assim sabemos que nossos algoritmos de ordenação são assintoticamente ótimos. Além disso, podemos usar o limite inferior da ordenação com a finalidade de demonstrar limites inferiores para alguns outros problemas.
- Muitas questões de engenharia surgem quando se implementam algoritmos de ordenação. O programa de ordenação mais rápido para uma determinada situação pode depender de muitos fatores, como o conhecimento anterior a respeito das chaves e dos dados satélite, da hierarquia de memória (caches e memória virtual) do computador host e do ambiente de software. Muitas dessas questões são mais bem tratadas no nível algorítmico, em vez de ser necessário "mexer" no código.

Algoritmos de ordenação

Introduzimos no Capítulo 2 dois algoritmos para ordenação de n números reais. A ordenação de inserção leva o tempo $\Theta(n^2)$ no pior caso. Porém, pelo fato de seus loops internos serem compactos, ela é um rápido algoritmo de ordenação local para pequenos tamanhos de entrada. (Lembre-se de que um algoritmo de ordenação efetua a ordenação ***local*** se somente um número constante de elementos do arranjo de entrada sempre são armazenados fora do arranjo.) A ordenação por intercalação tem um tempo assintótico de execução melhor, $\Theta(n \lg n)$, mas o procedimento MERGE que ela utiliza não opera no local.

Nesta parte, apresentaremos mais dois algoritmos que ordenam números reais arbitrários. O heapsort, apresentado no Capítulo 6, efetua a ordenação de n números localmente, no tempo $O(n \lg n)$. Ele usa uma importante estrutura de dados, chamada heap (monte), com a qual também podemos implementar uma fila de prioridades.

O quicksort, no Capítulo 7, também ordena n números localmente, mas seu tempo de execução no pior caso é $\Theta(n^2)$. Porém, seu tempo de execução no caso médio é $\Theta(n \lg n)$, e ele em geral supera o heapsort na prática. Como a ordenação por inserção, o quicksort tem um código compacto, e assim o fator constante oculto em seu tempo de execução é pequeno. Ele é um algoritmo popular para ordenação de grandes arranjos de entrada.

A ordenação por inserção, a ordenação por intercalação, o heapsort e o quicksort são todos ordenações por comparação: eles determinam a seqüência ordenada de um arranjo de entrada pela comparação dos elementos. O Capítulo 8 começa introduzindo o modelo de árvore de de-

cisão, a fim de estudar as limitações de desempenho de ordenações por comparação. Usando esse modelo, provamos um limite inferior igual a $\Omega(n \lg n)$ no tempo de execução do pior caso de qualquer ordenação por comparação sobre n entradas, mostrando assim que o heapsort e a ordenação por intercalação são ordenações por comparação assintoticamente ótimas.

Em seguida, o Capítulo 8 mostra que poderemos superar esse limite inferior $\Omega(n \lg n)$ se for possível reunir informações sobre a seqüência ordenada da entrada por outros meios além da comparação dos elementos. Por exemplo, o algoritmo de ordenação por contagem pressupõe que os números da entrada estão no conjunto $\{1, 2, ..., k\}$. Usando a indexação de arranjos como uma ferramenta para determinar a ordem relativa, a ordenação por contagem pode ordenar n números no tempo $(k + n)$. Desse modo, quando $k = O(n)$, a ordenação por contagem é executada em um tempo linear no tamanho do arranjo de entrada. Um algoritmo relacionado, a radix sort (ordenação da raiz), pode ser usado para estender o intervalo da ordenação por contagem. Se houver n inteiros para ordenar, cada inteiro tiver d dígitos e cada dígito estiver no conjunto $\{1, 2, ..., k\}$, a radix sort poderá ordenar os números em um tempo $O(d(n + k))$. Quando d é uma constante e k é $O(n)$, a radix sort é executada em tempo linear. Um terceiro algoritmo, bucket sort (ordenação por balde), requer o conhecimento da distribuição probabilística dos números no arranjo de entrada. Ele pode ordenar n números reais distribuídos uniformemente no intervalo meio aberto $[0, 1)$ no tempo do caso médio $O(n)$.

Estatísticas de ordem

A i-ésima estatística de ordem de um conjunto de n números é o i-ésimo menor número no conjunto. É claro que uma pessoa pode selecionar a i-ésima estatística de ordem, ordenando a entrada e indexando o i-ésimo elemento da saída. Sem quaisquer suposições a respeito da distribuição da entrada, esse método é executado no tempo $\Omega(n \lg n)$, como mostra o limite inferior demonstrado no Capítulo 8.

No Capítulo 9, mostramos que é possível encontrar o i-ésimo menor elemento no tempo $O(n)$, mesmo quando os elementos são números reais arbitrários. Apresentamos um algoritmo com pseudocódigo compacto que é executado no tempo $\Theta(n^2)$ no pior caso, mas em tempo linear no caso médio. Também fornecemos um algoritmo mais complicado que é executado em um tempo $O(n)$ no pior caso.

Experiência necessária

Embora a maioria das seções desta parte não dependa de conceitos matemáticos difíceis, algumas seções exigem uma certa sofisticação matemática. Em particular, as análises do caso médio do quicksort, do bucket sort e do algoritmo de estatística de ordem utilizam a probabilidade, o que revisamos no Apêndice C; o material sobre a análise probabilística e os algoritmos aleatórios é estudado no Capítulo 5. A análise do algoritmo de tempo linear do pior caso para estatísticas de ordem envolve matemática um pouco mais sofisticada que as análises do pior caso desta parte.

Capítulo 6

Heapsort

Neste capítulo, introduzimos outro algoritmo de ordenação. Como a ordenação por intercalação, mas diferente da ordenação por inserção, o tempo de execução do heapsort é $O(n \lg n)$. Como a ordenação por inserção, mas diferente da ordenação por intercalação, o heapsort ordena localmente: apenas um número constante de elementos do arranjo é armazenado fora do arranjo de entrada em qualquer instante. Desse modo, o heapsort combina os melhores atributos dos dois algoritmos de ordenação que já discutimos.

O heapsort também introduz outra técnica de projeto de algoritmos: o uso de uma estrutura de dados, nesse caso uma estrutura que chamamos "heap" (ou "monte") para gerenciar informações durante a execução do algoritmo. A estrutura de dados heap não é útil apenas para o heapsort (ou ordenação por heap); ela também cria uma eficiente fila de prioridades. A estrutura de dados heap reaparecerá em algoritmos de capítulos posteriores.

Observamos que o termo "heap" foi cunhado originalmente no contexto do heapsort, mas, desde então, ele passou a se referir ao "espaço para armazenamento do lixo coletado" como o espaço proporcionado pelas linguagens de programação Lisp e Java. Nossa estrutura de dados heap *não* é um espaço para armazenamento do lixo coletado e, sempre que mencionarmos heaps neste livro, estaremos fazendo referência à estrutura de dados definida neste capítulo.

6.1 Heaps

A estrutura de dados ***heap (binário)*** é um objeto arranjo que pode ser visto como uma árvore binária praticamente completa (ver Seção B.5.3), como mostra a Figura 6.1. Cada nó da árvore corresponde a um elemento do arranjo que armazena o valor no nó. A árvore está completamente preenchida em todos os níveis, exceto talvez no nível mais baixo, que é preenchido a partir da esquerda até certo ponto. Um arranjo A que representa um heap é um objeto com dois atributos: *comprimento*$[A]$, que é o número de elementos no arranjo, e *tamanho-do-heap*$[A]$, o número de elementos no heap armazenado dentro do arranjo A. Ou seja, embora $A[1 \,..\, \textit{comprimento}[A]]$ possa conter números válidos, nenhum elemento além de $A[\textit{tamanho-do-heap}[A]]$, onde $\textit{tamanho-do-heap}[A] \leq \textit{comprimento}[A]$, é um elemento do heap. A raiz da árvore é $A[1]$ e, dado o índice i de um nó, os índices de seu pai PARENT(i), do filho da esquerda LEFT(i) e do filho da direita RIGHT(i) podem ser calculados de modo simples:

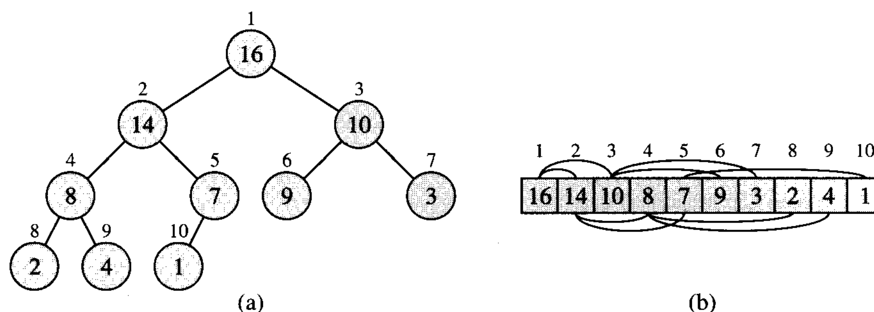


FIGURA 6.1 Um heap máximo visto como **(a)** uma árvore binária e **(b)** um arranjo. O número dentro do círculo em cada nó na árvore é o valor armazenado nesse nó. O número acima de um nó é o índice correspondente no arranjo. Acima e abaixo do arranjo encontramos linhas mostrando relacionamentos pai-filho; os pais estão sempre à esquerda de seus filhos. A árvore tem altura três; o nó no índice 4 (com o valor 8) tem altura um

PARENT(i)
  return $\lfloor i/2 \rfloor$

LEFT(i)
  return $2i$

RIGHT(i)
  return $2i + 1$

Na maioria dos computadores, o procedimento LEFT pode calcular $2i$ em uma única instrução, simplesmente deslocando a representação binária de i uma posição de bit para a esquerda. De modo semelhante, o procedimento RIGHT pode calcular rapidamente $2i + 1$ deslocando a representação binária de i uma posição de bit para a esquerda e inserindo 1 como valor do bit de baixa ordem. O procedimento PARENT pode calcular $\lfloor i/2 \rfloor$ deslocando i uma posição de bit para a direita. Em uma boa implementação de heapsort, esses três procedimentos são executados freqüentemente como "macros" ou como procedimentos "em linha".

Existem dois tipos de heaps binários: heaps máximos e heaps mínimos. Em ambos os tipos, os valores nos nós satisfazem a uma ***propriedade de heap***, cujos detalhes específicos dependem do tipo de heap. Em um ***heap máximo***, a ***propriedade de heap máximo*** é que, para todo nó i diferente da raiz,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

isto é, o valor de um nó é no máximo o valor de seu pai. Desse modo, o maior elemento em um heap máximo é armazenado na raiz, e a subárvore que tem raiz em um nó contém valores menores que o próprio nó. Um ***heap mínimo*** é organizado de modo oposto; a ***propriedade de heap mínimo*** é que, para todo nó i diferente da raiz,

$$A[\text{PARENT}(i)] \leq A[i] .$$

O menor elemento em um heap mínimo está na raiz.

Para o algoritmo de heapsort, usamos heaps máximos. Heaps mínimos são comumente empregados em filas de prioridades, que discutimos na Seção 6.5. Seremos precisos ao especificar se necessitamos de um heap máximo ou de um heap mínimo para qualquer aplicação específica e, quando as propriedades se aplicarem tanto a heaps máximos quanto a heaps mínimos, simplesmente usaremos o termo "heap".

Visualizando um heap como uma árvore, definimos a ***altura*** de um nó em um heap como o número de arestas no caminho descendente simples mais longo desde o nó até uma folha, e definimos a altura do heap como a altura de sua raiz. Tendo em vista que um heap de n elementos é baseado em uma árvore binária completa, sua altura é $\Theta(\lg n)$ (ver Exercício 6.1-2). Veremos que as operações básicas sobre heaps são executadas em um tempo máximo proporcional à altura da árvore, e assim demoram um tempo $O(\lg n)$. O restante deste capítulo apresenta alguns procedimentos básicos e mostra como eles são usados em um algoritmo de ordenação e em uma estrutura de dados de fila de prioridades.

- O procedimento MAX-HEAPIFY, executado no tempo $O(\lg n)$, é a chave para manter a propriedade de heap máximo (6.1).
- O procedimento BUILD-MAX-HEAP, executado em tempo linear, produz um heap a partir de um arranjo de entrada não ordenado.
- O procedimento HEAPSORT, executado no tempo $O(n \lg n)$, ordena um arranjo localmente.
- Os procedimentos MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY e HEAP-MAXIMUM, executados no tempo $O(\lg n)$, permitem que a estrutura de dados heap seja utilizada como uma fila de prioridades.

Exercícios

6.1-1
Quais são os números mínimo e máximo de elementos em um heap de altura h?

6.1-2
Mostre que um heap de n elementos tem altura $\lfloor \lg n \rfloor$.

6.1-3
Mostre que, em qualquer subárvore de um heap máximo, a raiz da subárvore contém o maior valor que ocorre em qualquer lugar nessa subárvore.

6.1-4
Onde em um heap máximo o menor elemento poderia residir, supondo-se que todos os elementos sejam distintos?

6.1-5
Um arranjo que está em seqüência ordenada é um heap mínimo?

6.1-6
A seqüência $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ é um heap máximo?

6.1-7
Mostre que, com a representação de arranjo para armazenar um heap de n elementos, as folhas são os nós indexados por $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Manutenção da propriedade de heap

MAX-HEAPIFY é uma sub-rotina importante para manipulação de heaps máximos. Suas entradas são um arranjo A e um índice i para o arranjo. Quando MAX-HEAPIFY é chamado, supomos que as árvores binárias com raízes em LEFT(i) e RIGHT(i) são heaps máximos, mas que $A[i]$ pode ser menor que seus filhos, violando assim a propriedade de heap máximo. A função de MAX-HEAPIFY é deixar que o valor em $A[i]$ "flutue para baixo" no heap máximo, de tal forma que a subárvore com raiz no índice i se torne um heap máximo.

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ tamanho-do-heap[A] e A[l] > A[i]
 4     then maior ← l
 5     else maior ← i
 6  if r ≤ tamanho-do-heap[A] e A[r] > A[maior]
 7     then maior ← r
 8  if maior ≠ i
 9     then trocar A[i] ↔ A[maior]
10          MAX-HEAPIFY(A, maior)
```

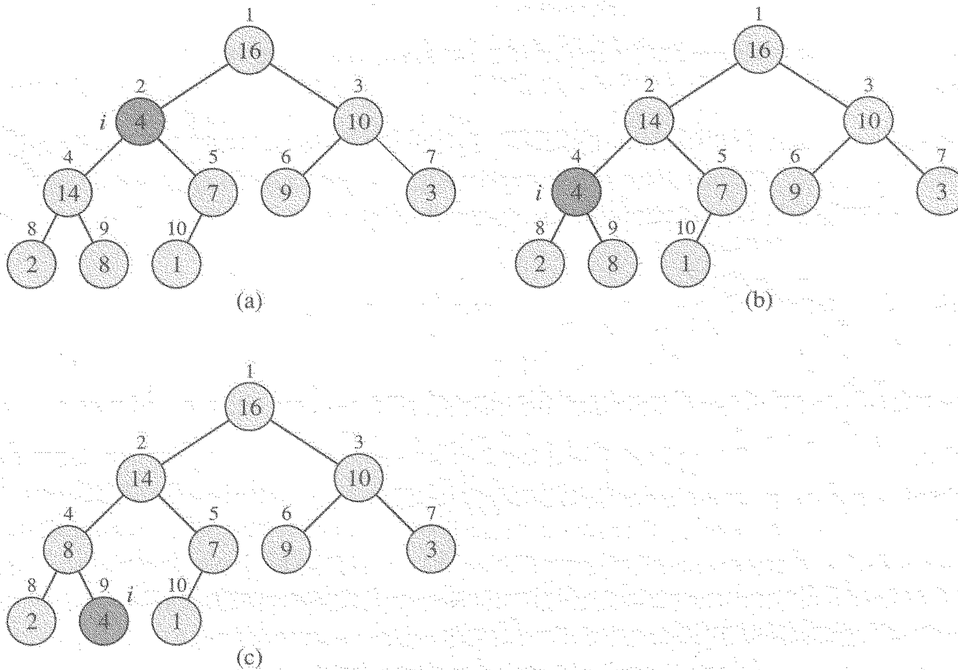


FIGURA 6.2 A ação de MAX-HEAPIFY(A, 2), onde *tamanho-do-heap*[A] = 10. **(a)** A configuração inicial, com $A[2]$ no nó $i = 2$, violando a propriedade de heap máximo, pois ele não é maior que ambos os filhos. A propriedade de heap máximo é restabelecida para o nó 2 em **(b)** pela troca de $A[2]$ por $A[4]$, o que destrói a propriedade de heap máximo para o nó 4. A chamada recursiva MAX-HEAPIFY(A, 4) agora define $i = 4$. Após a troca de $A[4]$ por $A[9]$, como mostramos em **(c)**, o nó 4 é corrigido, e a chamada recursiva a MAX-HEAPIFY(A, 9) não produz nenhuma mudança adicional na estrutura de dados

A Figura 6.2 ilustra a ação de MAX-HEAPIFY. Em cada passo, o maior entre os elementos $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$ é determinado, e seu índice é armazenado em *maior*. Se $A[i]$ é maior, então a subárvore com raiz no nó i é um heap máximo e o procedimento termina. Caso contrário, um dos dois filhos tem o maior elemento, e $A[i]$ é trocado por $A[maior]$, o que faz o nó i e seus filhos satisfazerem à propriedade de heap máximo. Porém, agora o nó indexado por *maior* tem o valor original $A[i]$ e, desse modo, a subárvore com raiz em *maior* pode violar a propriedade de heap máximo. Em conseqüência disso, MAX-HEAPIFY deve ser chamado recursivamente nessa subárvore.

O tempo de execução de MAX-HEAPIFY em uma subárvore de tamanho n com raiz em um dado nó i é o tempo $\Theta(1)$ para corrigir os relacionamentos entre os elementos $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$, mais o tempo para executar MAX-HEAPIFY em uma subárvore com raiz em um dos filhos do nó i. As subárvores de cada filho têm tamanho máximo igual a $2n/3$ – o pior caso ocorre quando a última linha da árvore está exatamente metade cheia – e o tempo de execução de MAX-HEAPIFY pode então ser descrito pela recorrência

$T(n) \le T(2n/3) + \Theta(1)$.

A solução para essa recorrência, de acordo com o caso 2 do teorema mestre (Teorema 4.1), é $T(n) = O(\lg n)$. Como alternativa, podemos caracterizar o tempo de execução de MAX-HEAPIFY em um nó de altura h como $O(h)$.

Exercícios

6.2-1

Usando a Figura 6.2 como modelo, ilustre a operação de MAX-HEAPIFY(A, 3) sobre o arranjo $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2-2

Começando com o procedimento MAX-HEAPIFY, escreva pseudocódigo para o procedimento MIN-HEAPIFY(A, i), que executa a manipulação correspondente sobre um heap mínimo. Como o tempo de execução de MIN-HEAPIFY se compara ao de MAX-HEAPIFY?

6.2-3

Qual é o efeito de chamar MAX-HEAPIFY(A, i) quando o elemento $A[i]$ é maior que seus filhos?

6.2-4

Qual é o efeito de chamar MAX-HEAPIFY(A, i) para $i >$ *tamanho-do-heap*$[A]/2$?

6.2-5

O código de MAX-HEAPIFY é bastante eficiente em termos de fatores constantes, exceto possivelmente para a chamada recursiva da linha 10, que poderia fazer alguns compiladores produzirem um código ineficiente. Escreva um MAX-HEAPIFY eficiente que use uma construção de controle iterativa (um loop) em lugar da recursão.

6.2-6

Mostre que o tempo de execução do pior caso de MAX-HEAPIFY sobre um heap de tamanho n é $\Omega(\lg n)$. (*Sugestão*: Para um heap com n nós, forneça valores de nós que façam MAX-HEAPIFY ser chamado recursivamente em todo nó sobre um caminho desde a raiz até uma folha.)

6.3 A construção de um heap

Podemos usar o procedimento MAX-HEAPIFY de baixo para cima, a fim de converter um arranjo $A[1..n]$, onde $n =$ *comprimento*$[A]$, em um heap máximo. Pelo Exercício 6.1-7, os elementos no subarranjo $A[(\lfloor n/2 \rfloor + 1)..n]$ são todos folhas da árvore, e então cada um deles é um heap de 1 elemento com o qual podemos começar. O procedimento BUILD-MAX-HEAP percorre os nós restantes da árvore e executa MAX-HEAPIFY sobre cada um.

BUILD-MAX-HEAP(A)
1 *tamanho-do-heap*$[A] \leftarrow$ *comprimento*$[A]$
2 **for** $i \leftarrow \lfloor$*comprimento*$[A]/2\rfloor$ **downto** 1
3   **do** MAX-HEAPIFY(A, i)

A Figura 6.3 ilustra um exemplo da ação de BUILD-MAX-HEAP.

Para mostrar por que BUILD-MAX-HEAP funciona corretamente, usamos o seguinte loop invariante:

> No começo de cada iteração do loop **for** das linhas 2 e 3, cada nó $i + 1, i + 2, ..., n$ é a raiz de um heap máximo.

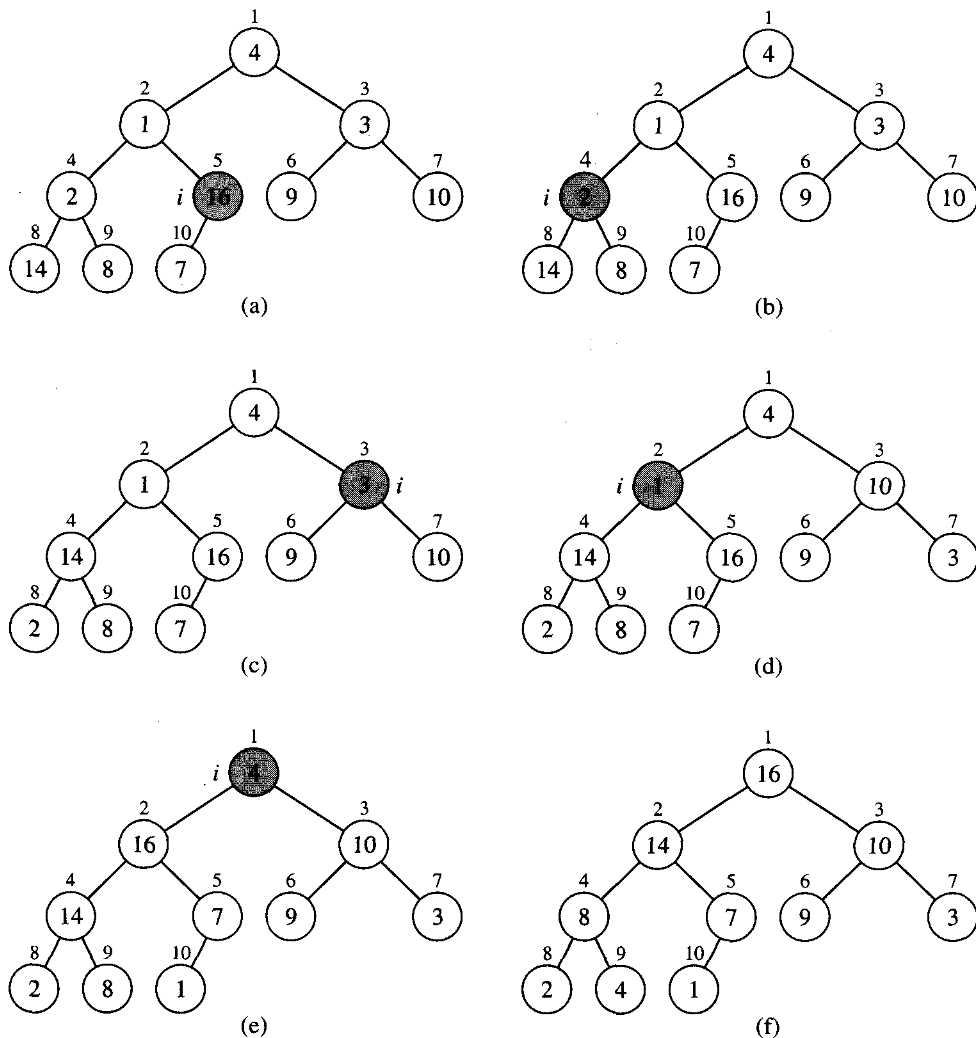


FIGURA 6.3 A operação de BUILD-MAX-HEAP, mostrando a estrutura de dados antes da chamada a MAX-HEAPIFY na linha 3 de BUILD-MAX-HEAP. **(a)** Um arranjo de entrada de 10 elementos A e a árvore binária que ele representa. A figura mostra que o índice de loop i se refere ao nó 5 antes da chamada MAX-HEAPIFY(A, i). **(b)** A estrutura de dados que resulta. O índice de loop i para a próxima iteração aponta para o nó 4. **(c)–(e)** Iterações subseqüentes do loop **for** em BUILD-MAX-HEAP. Observe que, sempre que MAX-HEAPIFY é chamado em um nó, as duas subárvores desse nó são ambas heaps máximos. **(f)** O heap máximo após o término de BUILD-MAX-HEAP

Precisamos mostrar que esse invariante é verdade antes da primeira iteração do loop, que cada iteração do loop mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

Inicialização: Antes da primeira iteração do loop, $i = \lfloor n/2 \rfloor$. Cada nó $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ é uma folha, e é portanto a raiz de um heap máximo trivial.

Manutenção: Para ver que cada iteração mantém o loop invariante, observe que os filhos do nó i são numerados com valores mais altos que i. Assim, pelo loop invariante, ambos são raízes de heaps máximos. Essa é precisamente a condição exigida para a chamada MAX-HEAPIFY(A, i) para tornar o nó i a raiz de um heap máximo. Além disso, a chamada a MAX-HEAPIFY preserva a propriedade de que os nós $i + 1, i + 2, \ldots, n$ são todos raízes de heaps máximos. Decrementar i na atualização do loop **for** restabelece o loop invariante para a próxima iteração.

Término: No término, $i = 0$. Pelo loop invariante, cada nó $1, 2, ..., n$ é a raiz de um heap máximo. Em particular, o nó 1 é uma raiz.

Podemos calcular um limite superior simples sobre o tempo de execução de BUILD-MAX-HEAP como a seguir. Cada chamada a MAX-HEAPIFY custa o tempo $O(\lg n)$, e existem $O(n)$ dessas chamadas. Desse modo, o tempo de execução é $O(n \lg n)$. Esse limite superior, embora correto, não é assintoticamente restrito.

Podemos derivar um limite mais restrito observando que o tempo de execução de MAX-HEAPIFY sobre um nó varia com a altura do nó na árvore, e as alturas na maioria dos nós são pequenas. Nossa análise mais restrita se baseia nas propriedades de que um heap de n elementos tem altura $\lfloor \lg n \rfloor$ (ver Exercício 6.1-2) e no máximo $\lceil n/2^{h+1} \rceil$ nós de qualquer altura h (ver Exercício 6.3-3).

O tempo exigido por MAX-HEAPIFY quando é chamado em um nó de altura h é $O(h)$; assim, podemos expressar o custo total de BUILD-MAX-HEAP por

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right).$$

O último somatório pode ser calculado substituindo-se $x = 1/2$ na fórmula (A.8), o que produz

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

Desse modo, o tempo de execução de BUILD-MAX-HEAP pode ser limitado como

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n) .$$

Conseqüentemente, podemos construir um heap máximo a partir de um arranjo não ordenado em tempo linear.

Podemos construir um heap mínimo pelo procedimento BUILD-MIN-HEAP, que é igual a BUILD-MAX-HEAP, mas tem a chamada a MAX-HEAPIFY na linha 3 substituída por uma chamada a MIN-HEAPFY (ver Exercício 6.2-2). BUILD-MIN-HEAP produz um heap mínimo a partir de um arranjo linear não ordenado em tempo linear.

Exercícios

6.3-1

Usando a Figura 6.3 como modelo, ilustre a operação de BUILD-MAX-HEAP sobre o arranjo $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2

Por que queremos que o índice de loop i na linha 2 de BUILD-MAX-HEAP diminua de $\lfloor comprimento[A]/2 \rfloor$ até 1, em vez de aumentar de 1 até $\lfloor comprimento[A]/2 \rfloor$?

6.3-3

Mostre que existem no máximo $\lceil n/2^{h+1} \rceil$ nós de altura h em qualquer heap de n elementos.

6.4 O algoritmo heapsort

O algoritmo heapsort (ordenação por monte) começa usando BUILD-MAX-HEAP para construir um heap no arranjo de entrada $A[1..n]$, onde n = *comprimento*$[A]$. Tendo em vista que o elemento máximo do arranjo está armazenado na raiz $A[1]$, ele pode ser colocado em sua posição final correta, trocando-se esse elemento por $A[n]$. Se agora "descartarmos" o nó n do heap (diminuindo *tamanho-do-heap*$[A]$), observaremos que $A[1 .. (n-1)]$ pode ser facilmente transformado em um heap máximo. Os filhos da raiz continuam sendo heaps máximos, mas o novo elemento raiz pode violar a propriedade de heap máximo. Porém, tudo o que é necessário para restabelecer a propriedade de heap máximo é uma chamada a MAX-HEAPIFY(A, 1), que deixa um heap máximo em $A[1 .. (n-1)]$. Então, o algoritmo heapsort repete esse processo para o heap de tamanho $n-1$, descendo até um heap de tamanho 2. (Veja no Exercício 6.4-2 um loop invariante preciso.)

HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 **for** $i \leftarrow$ *comprimento*$[A]$ **downto** 2
3   **do** trocar $A[1] \leftrightarrow A[i]$
4   *tamanho-do-heap*$[A] \leftarrow$ *tamanho-do-heap*$[A] - 1$
5   MAX-HEAPIFY(A, 1)

A Figura 6.4 mostra um exemplo da operação de heapsort depois do heap máximo ser inicialmente construído. Cada heap máximo é mostrado no princípio de uma iteração do loop **for** das linhas 2 a 5.

O procedimento HEAPSORT demora o tempo $O(n \lg n)$, pois a chamada a BUILD-MAX-HEAP demora o tempo $O(n)$, e cada uma das $n-1$ chamadas a MAX-HEAPIFY demora o tempo $O(\lg n)$.

Exercícios

6.4-1
Usando a Figura 6.4 como modelo, ilustre a operação de HEAPSORT sobre o arranjo $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2
Discuta a correção de HEAPSORT usando o loop invariante a seguir:

> No início de cada iteração do loop **for** das linhas 2 a 5, o subarranjo $A[1 .. i]$ é um heap máximo contendo os i menores elementos de $A[1 .. n]$, e o subarranjo $A[i+1 .. n]$ contém os $n-i$ maiores elementos de $A[1 .. n]$, ordenados.

6.4-3
Qual é o tempo de execução de heapsort sobre um arranjo A de comprimento n que já está ordenado em ordem crescente? E em ordem decrescente?

6.4-4
Mostre que o tempo de execução do pior caso de heapsort é $\Omega(n \lg n)$.

6.4-5
Mostre que, quando todos os elementos são distintos, o tempo de execução do melhor caso de heapsort é $\Omega(n \lg n)$.

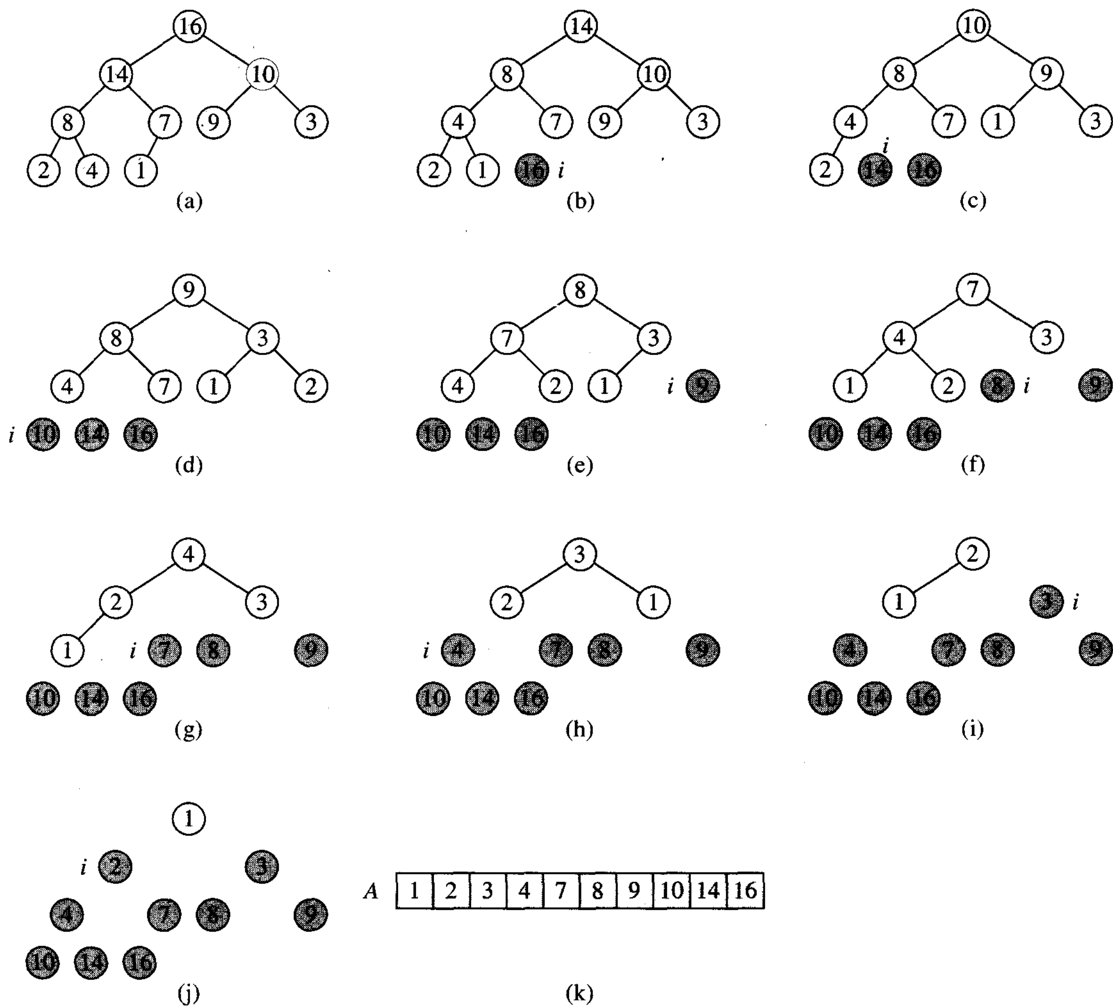


FIGURA 6.4 A operação de HEAPSORT. **(a)** A estrutura de dados heap máximo, logo após ter sido construída por BUILD-MAX-HEAP. **(b)–(j)** O heap máximo logo após cada chamada de MAX-HEAPIFY na linha 5. O valor de i nesse instante é mostrado. Apenas os nós levemente sombreados permanecem no heap. **(k)** O arranjo ordenado resultante A

6.5 Filas de prioridades

O heapsort é um algoritmo excelente, mas uma boa implementação de quicksort, apresentado no Capítulo 7, normalmente o supera na prática. Não obstante, a estrutura de dados de heap propriamente dita tem uma utilidade enorme. Nesta seção, apresentaremos uma das aplicações mais populares de um heap: seu uso como uma fila de prioridades eficiente. Como ocorre no caso dos heaps, existem dois tipos de filas de prioridades: as filas de prioridade máxima e as filas de prioridade mínima. Focalizaremos aqui a implementação das filas de prioridade máxima, que por sua vez se baseiam em heaps máximos; o Exercício 6.5-3 lhe pede para escrever os procedimentos correspondentes a filas de prioridade mínima.

Uma ***fila de prioridades*** é uma estrutura de dados para manutenção de um conjunto S de elementos, cada qual com um valor associado chamado ***chave***. Uma fila de prioridade máxima admite as operações a seguir.

INSERT(S, x) insere o elemento x no conjunto S. Essa operação poderia ser escrita como $S \leftarrow S \cup \{x\}$.

MAXIMUM(S) retorna o elemento de S com a maior chave.

EXTRACT-MAX(S) remove e retorna o elemento de S com a maior chave.

INCREASE-KEY(S, x, k) aumenta o valor da chave do elemento x para o novo valor k, que se presume ser pelo menos tão grande quanto o valor da chave atual de x.

Uma aplicação de filas de prioridade máxima é programar trabalhos em um computador compartilhado. A fila de prioridade máxima mantém o controle dos trabalhos a serem executados e de suas prioridades relativas. Quando um trabalho termina ou é interrompido, o trabalho de prioridade mais alta é selecionado dentre os trabalhos pendentes, com o uso de EXTRACT-MAX. Um novo trabalho pode ser adicionado à fila em qualquer instante, com a utilização de INSERT.

Como alternativa, uma ***fila de prioridade mínima*** admite as operações INSERT, MINIMUM, EXTRACT-MIN e DECREASE-KEY. Uma fila de prioridade mínima pode ser usada em um simulador orientado a eventos. Os itens na fila são eventos a serem simulados, cada qual com um tempo de ocorrência associado que serve como sua chave. Os eventos devem ser simulados em ordem de seu momento de ocorrência, porque a simulação de um evento pode provocar outros eventos a serem simulados no futuro. O programa de simulação utiliza EXTRACT-MIN em cada etapa para escolher o próximo evento a simular. À medida que novos eventos são produzidos, eles são inseridos na fila de prioridade mínima com o uso de INSERT. Veremos outros usos de filas de prioridade mínima destacando a operação DECREASE-KEY, nos Capítulos 23 e 24.

Não surpreende que possamos usar um heap para implementar uma fila de prioridades. Em uma dada aplicação, como a programação de trabalhos ou a simulação orientada a eventos, os elementos de uma fila de prioridades correspondem a objetos na aplicação. Freqüentemente é necessário determinar que objeto da aplicação corresponde a um dado elemento de fila de prioridades e vice-versa. Então, quando um heap é usado para implementar uma fila de prioridades, com freqüência precisamos armazenar um ***descritor*** para o objeto da aplicação correspondente em cada elemento do heap. A constituição exata do descritor (isto é, um ponteiro, um inteiro etc.) depende da aplicação. De modo semelhante, precisamos armazenar um descritor para o elemento do heap correspondente em cada objeto da aplicação. Aqui, o descritor em geral seria um índice de arranjo. Como os elementos do heap mudam de posições dentro do arranjo durante operações de heap, uma implementação real, ao reposicionar um elemento do heap, também teria de atualizar o índice do arranjo no objeto da aplicação correspondente. Tendo em vista que os detalhes de acesso a objetos de aplicações dependem muito da aplicação e de sua implementação, não os examinaremos aqui, exceto por observar que na prática esses descritores precisam ser mantidos corretamente.

Agora descreveremos como implementar as operações de uma fila de prioridade máxima. O procedimento HEAP-MAXIMUM implementa a operação MAXIMUM no tempo $\Theta(1)$.

HEAP-MAXIMUM(A)
1 **return** $A[1]$

O procedimento HEAP-EXTRACT-MAX implementa a operação EXTRACT-MAX. Ele é semelhante ao corpo do loop **for** (linhas 3 a 5) do procedimento HEAPSORT:

HEAP-EXTRACT-MAX(A)
1 **if** *tamanho-do-heap*[A] < 1
2 **then error** "heap underflow"
3 *max* ← $A[1]$
4 $A[1]$ ← A[*tamanho-do-heap*[A]]
5 *tamanho-do-heap*[A] ← *tamanho-do-heap*[A] – 1
6 MAX-HEAPIFY(A, 1)
7 **return** *max*

O tempo de execução de HEAP-EXTRACT-MAX é $O(\lg n)$, pois ele executa apenas uma porção constante do trabalho sobre o tempo $O(\lg n)$ para MAX-HEAPIFY.

O procedimento HEAP-INCREASE-KEY implementa a operação INCREASE-KEY. O elemento da fila de prioridades cuja chave deve ser aumentada é identificado por um índice i no arranjo. Primeiro, o procedimento atualiza a chave do elemento $A[i]$ para seu novo valor. Em seguida, como o aumento da chave de $A[i]$ pode violar a propriedade de heap máximo, o procedimento, de um modo que é uma reminiscência do loop de inserção (linhas 5 a 7) de INSERTION-SORT da Seção 2.1, percorre um caminho desde esse nó em direção à raiz, até encontrar um lugar apropriado para o elemento recém-aumentado. Durante essa travessia, ele compara repetidamente um elemento a seu pai, permutando suas chaves e continuando se a chave do elemento é maior, e terminando se a chave do elemento é menor, pois a propriedade de heap máximo agora é válida. (Veja no Exercício 6.5-5 um loop invariante preciso.)

```
HEAP-INCREASE-KEY(A, i, chave)
1 if chave < A[i]
2    then error "nova chave é menor que chave atual"
3 A[i] ← chave
4 while i > 1 e A[PARENT(i)] < A[i]
5      do troca A[i] ↔ A[PARENT(i)]
6         i ← PARENT(i)
```

A Figura 6.5 mostra um exemplo de uma operação de HEAP-INCREASE-KEY. O tempo de execução de HEAP-INCREASE-KEY sobre um heap de n elementos é $O(\lg n)$, pois o caminho traçado desde o nó atualizado na linha 3 até a raiz tem o comprimento $O(\lg n)$.

O procedimento MAX-HEAP-INSERT implementa a operação INSERT. Ele toma como uma entrada a chave do novo elemento a ser inserido no heap máximo A. Primeiro, o procedimento expande o heap máximo, adicionando à árvore uma nova folha cuja chave é $-\infty$. Em seguida, ele chama HEAP-INCREASE-KEY para definir a chave desse novo nó com seu valor correto e manter a propriedade de heap máximo.

```
MAX-HEAP-INSERT(A, chave)
1 tamanho-do-heap[A] ← tamanho-do-heap[A] + 1
2 A[tamanho-do-heap[A]] ← - ∞
3 HEAP-INCREASE-KEY(A, tamanho-do-heap[A], chave)
```

O tempo de execução de MAX-HEAP-INSERT sobre um heap de n elementos é $O(\lg n)$.

Em resumo, um heap pode admitir qualquer operação de fila de prioridades em um conjunto de tamanho n no tempo $O(\lg n)$.

Exercícios

6.5-1

Ilustre a operação de HEAP-EXTRACT-MAX sobre o heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-2

Ilustre a operação de MAX-HEAP-INSERT(A, 10) sobre o heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$. Use o heap da Figura 6.5 como modelo para a chamada de HEAP-INCREASE-KEY.

6.5-3

Escreva pseudocódigo para os procedimentos HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY e MIN-HEAP-INSERT que implementam uma fila de prioridade mínima com um heap mínimo.

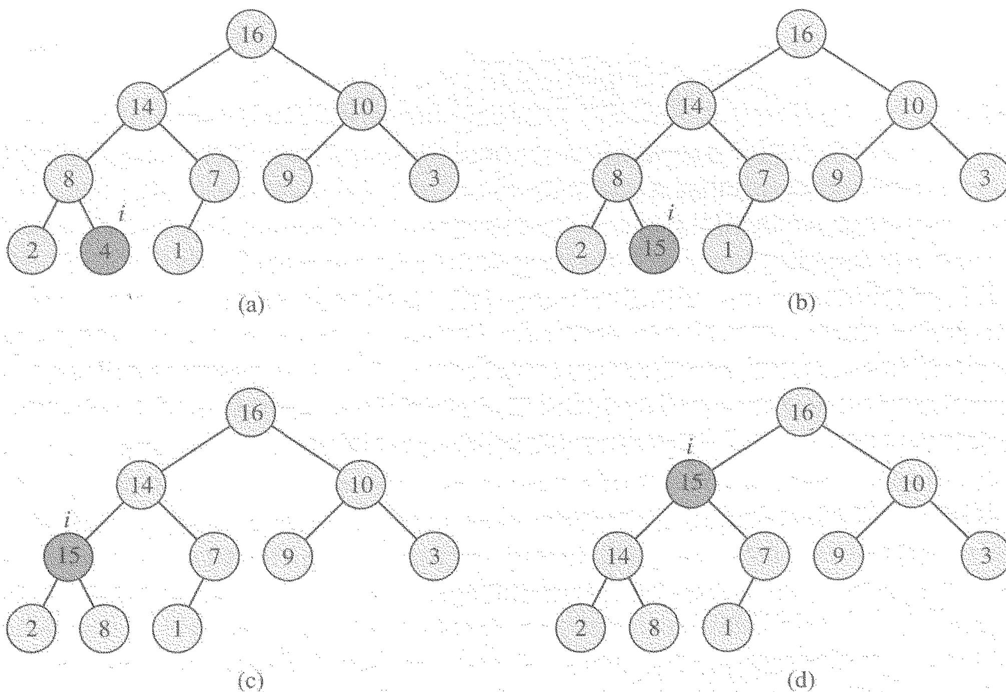


FIGURA 6.5 A operação de HEAP-INCREASE-KEY. **(a)** O heap máximo da Figura 6.4(a) com um nó cujo índice é i, fortemente sombreado. **(b)** Esse nó tem sua chave aumentada para 15. **(c)** Depois de uma iteração do loop **while** das linhas 4 a 6, o nó e seu pai trocaram chaves, e o índice i sobe para o pai. **(d)** O heap máximo depois de mais uma iteração do loop **while**. Nesse momento, $A[\text{PARENT}(i)] \geq A[i]$. Agora, a propriedade de heap máximo é válida, e o procedimento termina

6.5-4

Por que nos preocupamos em definir a chave do nó inserido como $-\infty$ na linha 2 de MAX-HEAP-INSERT quando a nossa próxima ação é aumentar sua chave para o valor desejado?

6.5-5

Demonstre a correção de HEAP-INCREASE-KEY usando este loop invariante:

> No começo de cada iteração do loop **while** das linhas 4 a 6, o arranjo $A[1 \,..\, \textit{tamanho-do-heap}[A]]$ satisfaz à propriedade de heap máximo, a não ser pelo fato de que é possível haver uma violação: $A[i]$ pode ser maior que $A[\text{PARENT}(i)]$.

6.5-6

Mostre como implementar uma fila de primeiro a entrar, primeiro a sair com uma fila de prioridades. Mostre como implementar uma pilha com uma fila de prioridades. (Filas e pilhas são definidas na Seção 10.1.)

6.5-7

A operação HEAP-DELETE(A, i) elimina o item do nó i do heap A. Forneça uma implementação de HEAP-DELETE que seja executada no tempo $O(\lg n)$ para um heap máximo de n elementos.

6.5-8

Forneça um algoritmo de tempo $O(n \lg k)$ para intercalar k listas ordenadas em uma única lista ordenada, onde n é o número total de elementos em todas as listas de entrada. (*Sugestão*: Use um heap mínimo para fazer a intercalação de k modos.)

Problemas

6-1 Construção de um heap com o uso de inserção

O procedimento BUILD-MAX-HEAP na Seção 6.3 pode ser implementado pelo uso repetido de MAX-HEAP-INSERT para inserir os elementos no heap. Considere a implementação a seguir:

BUILD-MAX-HEAP′(A)
1 *tamanho-do-heap*[A] $\leftarrow$ 1
2 **for** $i \leftarrow 2$ **to** *comprimento*[A]
3 **do** MAX-HEAP-INSERT(A, $A[i]$)

a. Os procedimentos BUILD-MAX-HEAP e BUILD-MAX-HEAP′ sempre criam o mesmo heap quando são executados sobre o mesmo arranjo de entrada? Prove que eles o fazem, ou então forneça um contra-exemplo.

b. Mostre que no pior caso, BUILD-MAX-HEAP′ exige o tempo $\Theta(n \lg n)$ para construir um heap de n elementos.

6-2 Análise de heaps d-ários

Um ***heap d-ário*** é semelhante a um heap binário, mas (com uma única exceção possível) nós que não são de folhas têm d filhos em vez de dois filhos.

a. Como você representaria um heap d-ário em um arranjo?

b. Qual é a altura de um heap d-ário de n elementos em termos de n e d?

c. Dê uma implementação eficiente de EXTRACT-MAX em um heap máximo d-ário. Analise seu tempo de execução em termos de d e n.

d. Forneça uma implementação eficiente de INSERT em um heap máximo d-ário. Analise seu tempo de execução em termos de d e n.

e. Dê uma implementação eficiente de HEAP-INCREASE-KEY(A, i, k), que primeiro configura $A[i] \leftarrow \max(A[i], k)$ e depois atualiza adequadamente a estrutura de heap máximo d-ário. Analise seu tempo de execução em termos de d e n.

6-3 Quadros de Young

Um ***quadro de Young*** $m \times n$ é uma matriz $m \times n$ *tal que* as entradas de cada linha estão em seqüência ordenada da esquerda para a direita, e as entradas de cada coluna estão em seqüência ordenada de cima para baixo. Algumas das entradas de um quadro de Young podem ser ∞, o que tratamos como elementos inexistentes. Desse modo, um quadro de Young pode ser usado para conter $r \leq mn$ números finitos.

a. Trace um quadro de Young 4×4 contendo os elementos $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Demonstre que um quadro de Young Y $m \times n$ é vazio se $Y[1, 1] = \infty$. Demonstre que Y é completo (contém mn elementos) se $Y[m, n] < \infty$.

c. Forneça um algoritmo para implementar EXTRACT-MIN em um quadro de Young $m \times n$ não vazio que funcione no tempo $O(m+n)$. Seu algoritmo deve usar uma sub-rotina recursiva que solucione um problema $m \times n$ resolvendo recursivamente um subproblema $(m-1) \times n$ ou $m \times (n-1)$. (*Sugestão*: Pense em MAX-HEAPIFY.) Defina $T(p)$, onde $p = m + n$, como o tempo de execução máximo de EXTRACT-MIN em qualquer quadro de Young $m \times n$. Forneça e resolva uma recorrência para $T(p)$ que produza o limite de tempo $O(m + n)$.

d. Mostre como inserir um novo elemento em um quadro de Young $m \times n$ não completo no tempo $O(m + n)$.

e. Sem usar nenhum outro método de ordenação como uma sub-rotina, mostre como utilizar um quadro de Young $n \times n$ para ordenar n^2 números no tempo $O(n^3)$.

f. Forneça um algoritmo de tempo $O(m + n)$ para determinar se um dado número está armazenado em um determinado quadro de Young $m \times n$.

Notas do capítulo

O algoritmo heapsort foi criado por Williams [316], que também descreveu como implementar uma fila de prioridades com um heap. O procedimento BUILD-MAX-HEAP foi sugerido por Floyd [90].

Usamos heaps mínimos para implementar filas de prioridade mínima nos Capítulos 16, 23 e 24. Também damos uma implementação com limites de tempo melhorados para certas operações nos Capítulos 19 e 20.

Implementações mais rápidas de filas de prioridades são possíveis para dados inteiros. Uma estrutura de dados criada por van Emde Boas [301] admite as operações MINIMUM, MAXIMUM, INSERT, DELETE, SEARCH, EXTRACT-MIN, EXTRACT-MAX, PREDECESSOR e SUCCESSOR, no tempo de pior caso $O(\lg \lg C)$, sujeito à restrição de que o universo de chaves é o conjunto $\{1, 2, ..., C\}$. Se os dados são inteiros de b bits e a memória do computador consiste em palavras de b bits endereçáveis, Fredman e Willard [99] mostraram como implementar MINIMUM no tempo $O(1)$, e INSERT e EXTRACT-MIN no tempo $O(\sqrt{\lg n})$. Thorup [299] melhorou o limite $O(\sqrt{\lg n})$ para o tempo $O((\lg \lg n)^2)$. Esse limite usa uma quantidade de espaço ilimitada em n, mas pode ser implementado em espaço linear com o uso de hash aleatório.

Um caso especial importante de filas de prioridades ocorre quando a seqüência de operações de EXTRACT-MIN é ***monotônica*** ou ***monótona***, ou seja, os valores retornados por operações sucessivas de EXTRACT-MIN são monotonicamente crescentes com o tempo. Esse caso surge em várias aplicações importantes, como o algoritmo de caminhos mais curtos de origem única de Dijkstra, discutido no Capítulo 24, e na simulação de eventos discretos. Para o algoritmo de Dijkstra é particularmente importante que a operação DECREASE-KEY seja implementada de modo eficiente.

No caso monotônico, se os dados são inteiros no intervalo 1, 2, ..., C, Ahuja, Melhorn, Orlin e Tarjan [8] descrevem como implementar EXTRACT-MIN e INSERT no tempo amortizado $O(\lg C)$ (consulte o Capítulo 17 para obter mais informações sobre análise amortizada) e DECREASE-KEY no tempo $O(1)$, usando uma estrutura de dados chamada heap de raiz. O limite $O(\lg C)$ pode ser melhorado para $O(\sqrt{\lg C})$ com o uso de heaps de Fibonacci (consulte o Capítulo 20) em conjunto com heaps de raiz. O limite foi melhorado ainda mais para o tempo esperado $O(\lg^{1/3 + \varepsilon} C)$ por Cherkassky, Goldberg e Silverstein [58], que combinam a estrutura de baldes em vários níveis de Denardo e Fox [72] com o heap de Thorup mencionado antes. Raman [256] melhorou mais ainda esses resultados para obter um limite de $O(\min(\lg^{1/4 + \varepsilon} C, \lg^{1/3 + \varepsilon} n))$, para qualquer $\varepsilon > 0$. Discussões mais detalhadas desses resultados podem ser encontradas em trabalhos de Raman [256] e Thorup [299].

Capítulo 7

Quicksort

O quicksort (ordenação rápida) é um algoritmo de ordenação cujo tempo de execução do pior caso é $\Theta(n^2)$ sobre um arranjo de entrada de n números. Apesar desse tempo de execução lento no pior caso, o quicksort com freqüência é a melhor opção prática para ordenação, devido a sua notável eficiência na média: seu tempo de execução esperado é $\Theta(n \lg n)$, e os fatores constantes ocultos na notação $\Theta(n \lg n)$ são bastante pequenos. Ele também apresenta a vantagem da ordenação local (ver Capítulo 2) e funciona bem até mesmo em ambientes de memória virtual.

A Seção 7.1 descreve o algoritmo e uma sub-rotina importante usada pelo quicksort para particionamento. Pelo fato do comportamento de quicksort ser complexo, começaremos com uma discussão intuitiva de seu desempenho na Seção 7.2 e adiaremos sua análise precisa até o final do capítulo. A Seção 7.3 apresenta uma versão de quicksort que utiliza a amostragem aleatória. Esse algoritmo tem um bom tempo de execução no caso médio, e nenhuma entrada específica induz seu comportamento do pior caso. O algoritmo aleatório é analisado na Seção 7.4, onde mostraremos que ele é executado no tempo $\Theta(n^2)$ no pior caso e no tempo $O(n \lg n)$ em média.

7.1 Descrição do quicksort

O quicksort, como a ordenação por intercalação, se baseia no paradigma de dividir e conquistar introduzido na Seção 2.3.1. Aqui está o processo de dividir e conquistar em três passos para ordenar um subarranjo típico $A[p \mathinner{..} r]$.

Dividir: O arranjo $A[p \mathinner{..} r]$ é particionado (reorganizado) em dois subarranjos (possivelmente vazios) $A[p \mathinner{..} q-1]$ e $A[q+1 \mathinner{..} r]$ tais que cada elemento de $A[p \mathinner{..} q-1]$ é menor que ou igual a $A[q]$ que, por sua vez, é igual ou menor a cada elemento de $A[q+1 \mathinner{..} r]$. O índice q é calculado como parte desse procedimento de particionamento.

Conquistar: Os dois subarranjos $A[p \mathinner{..} q-1]$ e $A[q+1 \mathinner{..} r]$ são ordenados por chamadas recursivas a quicksort.

Combinar: Como os subarranjos são ordenados localmente, não é necessário nenhum trabalho para combiná-los: o arranjo $A[p \mathinner{..} r]$ inteiro agora está ordenado.

O procedimento a seguir implementa o quicksort.

```
QUICKSORT(A, p, r)
1 if p < r
2    then q ← PARTITION(A, p, r)
3       QUICKSORT(A, p, q – 1)
4       QUICKSORT(A, q + 1, r)
```

Para ordenar um arranjo A inteiro, a chamada inicial é QUICKSORT(A, 1, *comprimento*[A]).

Particionamento do arranjo

A chave para o algoritmo é o procedimento PARTITION, que reorganiza o subarranjo $A[p \,..\, r]$ localmente.

```
PARTITION(A, p, r)
1 x ← A[r]
2 i ← p – 1
3 for j ← p to r – 1
4     do if A[j] ≤ x
5           then i ← i + 1
6                trocar A[i] ← A[j]
7 trocar A[i + 1] ↔ A[r]
8 return i + 1
```

A Figura 7.1 mostra a operação de PARTITION sobre um arranjo de 8 elementos. PARTITION sempre seleciona um elemento $x = A[r]$ como um elemento ***pivô*** ao redor do qual será feito o particionamento do subarranjo $A[p \,..\, r]$. À medida que o procedimento é executado, o arranjo é particionado em quatro regiões (possivelmente vazias). No início de cada iteração do loop **for** nas linhas 3 a 6, cada região satisfaz a certas propriedades, que podemos enunciar como um loop invariante:

No início de cada iteração do loop das linhas 3 a 6, para qualquer índice de arranjo k,

1. Se $p \leq k \leq i$, então $A[k] \leq x$.
2. Se $i + 1 \leq k \leq j - 1$, então $A[k] > x$.
3. Se $k = r$, então $A[k] = x$.

A Figura 7.2 resume essa estrutura. Os índices entre j e $r - 1$ não são cobertos por quaisquer dos três casos, e os valores nessas entradas não têm nenhum relacionamento particular para o pivô x.

Precisamos mostrar que esse loop invariante é verdadeiro antes da primeira iteração, que cada iteração do loop mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando o loop termina.

Inicialização: Antes da primeira iteração do loop, $i = p - 1$ e $j = p$. Não há nenhum valor entre p e i, e nenhum valor entre $i + 1$ e $j - 1$; assim, as duas primeiras condições do loop invariante são satisfeitas de forma trivial. A atribuição na linha 1 satisfaz à terceira condição.

Manutenção: Como mostra a Figura 7.3, existem dois casos a considerar, dependendo do resultado do teste na linha 4. A Figura 7.3(a) mostra o que acontece quando $A[j] > x$; a única ação no loop é incrementar j. Depois que j é incrementado, a condição 2 é válida para $A[j - 1]$ e todas as outras entradas permanecem inalteradas.

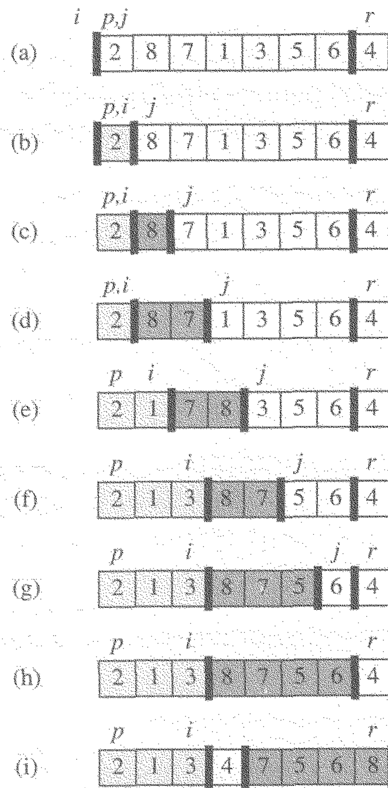


FIGURA 7.1 A operação de PARTITION sobre um exemplo de arranjo. Os elementos do arranjo ligeiramente sombreados estão todos na primeira partição com valores não maiores que x. Elementos fortemente sombreados estão na segunda partição com valores maiores que x. Os elementos não sombreados ainda não foram inseridos em uma das duas primeiras partições, e o elemento final branco é o pivô. **(a)** O arranjo inicial e as configurações de variáveis. Nenhum dos elementos foi inserido em qualquer das duas primeiras partições. **(b)** O valor 2 é "trocado com ele mesmo" e inserido na partição de valores menores. **(c)**–**(d)** Os valores 8 e 7 foram acrescentados à partição de valores maiores. **(e)** Os valores 1 e 8 são permutados, e a partição menor cresce. **(f)** Os valores 3 e 8 são permutados, e a partição menor cresce. **(g)**–**(h)** A partição maior cresce até incluir 5 e 6 e o loop termina. **(i)** Nas linhas 7 e 8, o elemento pivô é permutado de forma a residir entre as duas partições

A Figura 7.3(b) mostra o que acontece quando $A[j] \leq x$; i é incrementado, $A[i]$ e $A[j]$ são permutados, e então j é incrementado. Por causa da troca, agora temos que $A[i] \leq x$, e a condição 1 é satisfeita. De modo semelhante, também temos que $A[j-1] > x$, pois o item que foi trocado em $A[j-1]$ é, pelo loop invariante, maior que x.

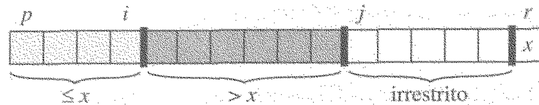


FIGURA 7.2 As quatro regiões mantidas pelo procedimento PARTITION em um subarranjo $A[p \,..\, r]$. Os valores em $A[p \,..\, i]$ são todos menores que ou iguais a x, os valores em $A[i+1 \,..\, j-1]$ são todos maiores que x, e $A[r] = x$. Os valores em $A[j \,..\, r-1]$ podem ser quaisquer valores

Término: No término, $j = r$. Então, toda entrada no arranjo está em um dos três conjuntos descritos pelo invariante, e particionamos os valores no arranjo em três conjuntos: os que são menores que ou iguais a x, os maiores que x, e um conjunto unitário contendo x.

As duas linhas finais de PARTITION movem o elemento pivô para seu lugar no meio do arranjo, permutando-o com o elemento mais à esquerda que é maior que x. A saída de PARTITION agora satisfaz às especificações dadas para a etapa de dividir.

O tempo de execução de PARTITION sobre o subarranjo $A[p \,..\, r]$ é $\Theta(n)$, onde $n = r - p + 1$ (ver Exercício 7.1-3).

Exercícios

7.1-1
Usando a Figura 7. 1 como modelo, ilustre a operação de PARTITION sobre o arranjo $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

7.1-2
Que valor de q PARTITION retorna quando todos os elementos no arranjo $A[p \,..\, r]$ têm o mesmo valor? Modifique PARTITION de forma que $q = (p + r)/2$ quando todos os elementos no arranjo $A[p \,..\, r]$ têm o mesmo valor.

7.1-3
Forneça um breve argumento mostrando que o tempo de execução de PARTITION sobre um subarranjo de tamanho n é $\Theta(n)$.

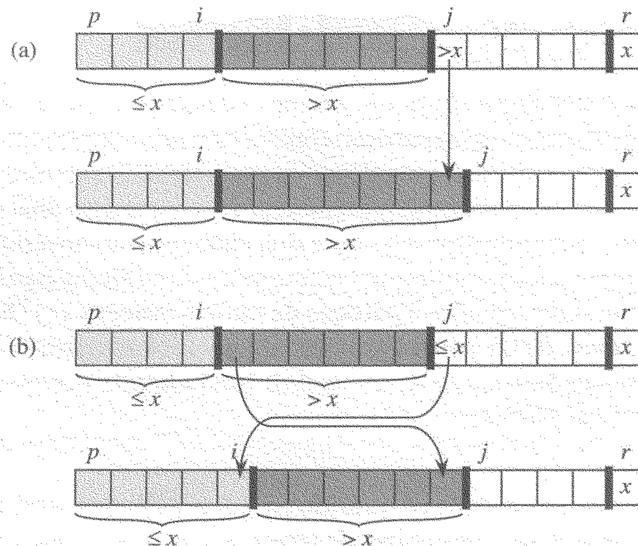


FIGURA 7.3 Os dois casos para uma iteração do procedimento PARTITION. **(a)** Se $A[j] > x$, a única ação é incrementar j, o que mantém o loop invariante. **(b)** Se $A[j] \leq x$, o índice i é incrementado, $A[i]$ e $A[j]$ são permutados, e então j é incrementado. Novamente, o loop invariante é mantido

7.1-4
De que maneira você modificaria QUICKSORT para fazer a ordenação em ordem não crescente?

7.2 O desempenho de quicksort

O tempo de execução de quicksort depende do fato de o particionamento ser balanceado ou não balanceado, e isso por sua vez depende de quais elementos são usados para particionar. Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto a ordenação por intercalação. Contudo, se o particionamento é não balanceado, ele pode ser executado assintoticamente de forma tão lenta quanto a ordenação por inserção. Nesta seção, investigaremos informalmente como o quicksort é executado sob as hipóteses de particionamento balanceado e particionamento não balanceado.

Particionamento no pior caso

O comportamento do pior caso para o quicksort ocorre quando a rotina de particionamento produz um subproblema com $n - 1$ elementos e um com 0 elementos. (Essa afirmativa é demonstrada na Seção 7.4.1.) Vamos supor que esse particionamento não balanceado surge em cada chamada recursiva. O particionamento custa o tempo $\Theta(n)$. Tendo em vista que a chamada recursiva sobre um arranjo de tamanho 0 simplesmente retorna, $T(0) = \Theta(1)$, e a recorrência para o tempo de execução é

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

Intuitivamente, se somarmos os custos incorridos a cada nível da recursão, conseguimos uma série aritmética (equação (A.2)), que tem o valor $\Theta(n^2)$. Na realidade, é simples usar o método de substituição para provar que a recorrência $T(n) = T(n-1) + \Theta(n)$ tem a solução $T(n) = \Theta(n^2)$. (Veja o Exercício 7.2-1.)

Portanto, se o particionamento é não balanceado de modo máximo em cada nível recursivo do algoritmo, o tempo de execução é $\Theta(n^2)$. Por conseguinte, o tempo de execução do pior caso do quicksort não é melhor que o da ordenação por inserção. Além disso, o tempo de execução $\Theta(n^2)$ ocorre quando o arranjo de entrada já está completamente ordenado – uma situação comum na qual a ordenação por inserção é executada no tempo $O(n)$.

Particionamento do melhor caso

Na divisão mais uniforme possível, PARTITION produz dois subproblemas, cada um de tamanho não maior que $n/2$, pois um tem tamanho $\lfloor n/2 \rfloor$ e o outro tem tamanho $\lceil n/2 \rceil - 1$. Nesse caso, o quicksort é executado com muito maior rapidez. A recorrência pelo tempo de execução é então

$$T(n) \leq 2T(n/2) + \Theta(n)$$

que, pelo caso 2 do teorema mestre (Teorema 4.1) tem a solução $T(n) = O(n \lg n)$. Desse modo, o balanceamento equilibrado dos dois lados da partição em cada nível da recursão produz um algoritmo assintoticamente mais rápido.

Particionamento balanceado

O tempo de execução do caso médio de quicksort é muito mais próximo do melhor caso que do pior caso, como mostrarão as análises da Seção 7.4. A chave para compreender por que isso poderia ser verdade é entender como o equilíbrio do particionamento se reflete na recorrência que descreve o tempo de execução.

Por exemplo, suponha que o algoritmo de particionamento sempre produza uma divisão proporcional de 9 para 1, que a princípio parece bastante desequilibrada. Então, obtemos a recorrência

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

no tempo de execução de quicksort, onde incluímos explicitamente a constante c oculta no termo $\Theta(n)$. A Figura 7.4 mostra a árvore de recursão para essa recorrência. Note que todo nível da árvore tem custo cn, até ser alcançada uma condição limite à profundidade $\log_{10} n = \Theta(\lg n)$, e então os níveis têm o custo máximo cn. A recursão termina na profundidade $\log_{10/9} n = \Theta(\lg n)$. O custo total do quicksort é portanto $O(n \lg n)$. Desse modo, com uma divisão na proporção de

9 para 1 em todo nível de recursão, o que intuitivamente parece bastante desequilibrado, o quicksort é executado no tempo $O(n \lg n)$ – assintoticamente o mesmo tempo que teríamos se a divisão fosse feita exatamente ao meio. De fato, até mesmo uma divisão de 99 para 1 produz um tempo de execução igual a $O(n \lg n)$. A razão é que qualquer divisão de proporcionalidade *constante* produz uma árvore de recursão de profundidade $\Theta(\lg n)$, onde o custo em cada nível é $O(n)$. Portanto, o tempo de execução será então $O(n \lg n)$ sempre que a divisão tiver proporcionalidade constante.

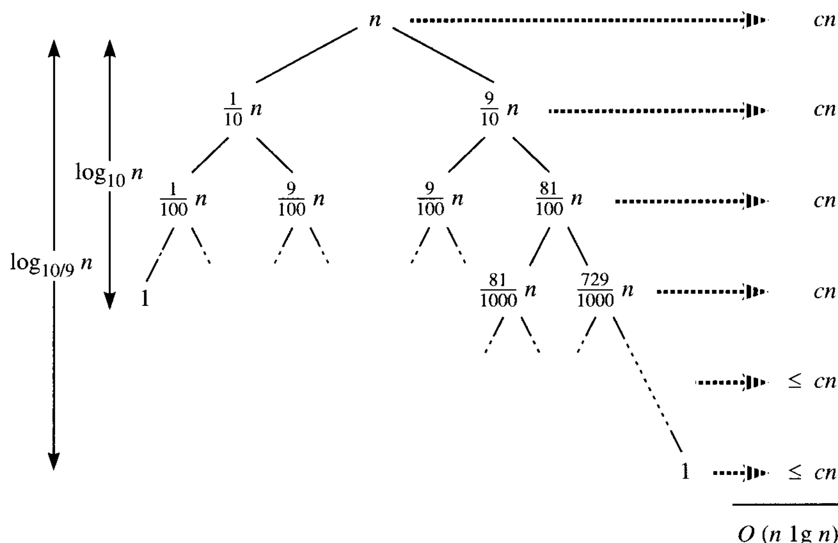


FIGURA 7.4 Uma árvore de recursão para QUICKSORT, na qual PARTITION sempre produz uma divisão de 9 para 1, resultando no tempo de execução $O(n \lg n)$. Os nós mostram tamanhos de subproblemas, com custos por nível à direita. Os custos por nível incluem a constante c implícita no termo $\Theta(n)$

Intuição para o caso médio

Para desenvolver uma noção clara do caso médio de quicksort, devemos fazer uma suposição sobre a freqüência com que esperamos encontrar as várias entradas. O comportamento de quicksort é determinado pela ordenação relativa dos valores nos elementos do arranjo dados como entrada, e não pelos valores específicos no arranjo. Como em nossa análise probabilística do problema da contratação na Seção 5.2, iremos supor por enquanto que todas as permutações dos números de entrada são igualmente prováveis.

Quando executamos o quicksort sobre um arranjo de entrada aleatório, é improvável que o particionamento sempre ocorra do mesmo modo em todo nível, como nossa análise informal pressupôs. Esperamos que algumas divisões sejam razoavelmente bem equilibradas e que algumas sejam bastante desequilibradas. Por exemplo, o Exercício 7.2-6 lhe pede para mostrar que, em cerca de 80% do tempo, PARTITION produz uma divisão mais equilibrada que 9 para 1, e mais ou menos em 20% do tempo ele produz uma divisão menos equilibrada que 9 para 1.

No caso médio, PARTITION produz uma mistura de divisões "boas" e "ruins". Em uma árvore de recursão para uma execução do caso médio de PARTITION, as divisões boas e ruins estão distribuídas aleatoriamente ao longo da árvore. Porém, suponha para fins de intuição, que as divisões boas e ruins alternem seus níveis na árvore, e que as divisões boas sejam divisões do melhor caso e as divisões ruins sejam divisões do pior caso. A Figura 7.5(a) mostra as divisões em dois níveis consecutivos na árvore de recursão. Na raiz da árvore, o custo é n para particionamento e os subarranjos produzidos têm tamanhos $n - 1$ e 1: o pior caso. No nível seguinte, o subarranjo de tamanho $n - 1$ é particionado no melhor caso em dois subarranjos de tamanho $(n - 1)/2 - 1$ e $(n - 1)/2$. Vamos supor que o custo da condição limite seja 1 para o subarranjo de tamanho 0.

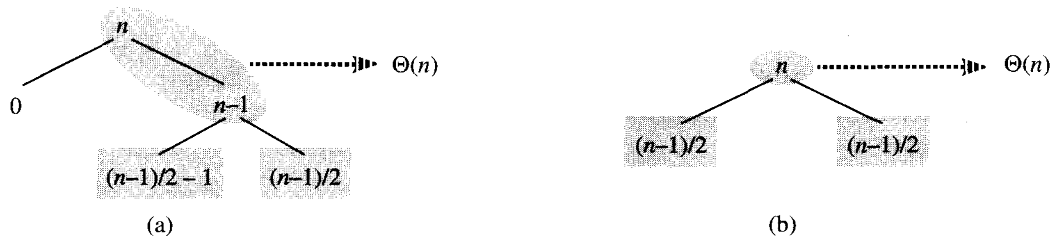


FIGURA 7.5 **(a)** Dois níveis de uma árvore de recursão para quicksort. O particionamento na raiz custa n e produz uma divisão "ruim": dois subarranjos de tamanhos 0 e $n - 1$. O particionamento do subarranjo de tamanho $n - 1$ custa $n - 1$ e produz uma divisão "boa": subarranjos de tamanho $(n - 1)/2 - 1$ e $(n - 1)/2$. **(b)** Um único nível de uma árvore de recursão que está muito bem equilibrada. Em ambas as partes, o custo de particionamento para os subproblemas mostrados com sombreamento elíptico é $\Theta(n)$. Ainda assim, os subproblemas que restam para serem resolvidos em (a), mostrados com sombreamento retangular, não são maiores que os subproblemas correspondentes que restam para serem resolvidos em (b)

A combinação da divisão ruim seguida pela divisão boa produz três subarranjos de tamanhos 0, $(n - 1)/2 - 1$ e $(n - 1)/2$, a um custo de particionamento combinado de $\Theta(n) + \Theta(n - 1) = \Theta(n)$. Certamente, essa situação não é pior que a da Figura 7.5(b), ou seja, um único nível de particionamento que produz dois subarranjos de tamanho $(n - 1)/2$, ao custo $\Theta(n)$. Ainda assim, essa última situação é equilibrada! Intuitivamente, o custo $\Theta(n - 1)$ da divisão ruim pode ser absorvido no custo $\Theta(n)$ da divisão boa, e a divisão resultante é boa. Desse modo, o tempo de execução do quicksort, quando os níveis se alternam entre divisões boas e ruins, é semelhante ao tempo de execução para divisões boas sozinhas: ainda $O(n \lg n)$, mas com uma constante ligeiramente maior oculta pela notação de O. Faremos uma análise rigorosa do caso médio na Seção 7.4.2.

Exercícios

7.2-1

Use o método de substituição para provar que a recorrência $T(n) = T(n - 1) + \Theta(n)$ tem a solução $T(n) = \Theta(n^2)$, como afirmamos no início da Seção 7.2.

7.2-2

Qual é o tempo de execução de QUICKSORT quando todos os elementos do arranjo A têm o mesmo valor?

7.2-3

Mostre que o tempo de execução de QUICKSORT é $\Theta(n^2)$ quando o arranjo A contém elementos distintos e está ordenado em ordem decrescente.

7.2-4

Os bancos freqüentemente registram transações em uma conta na ordem dos horários das transações, mas muitas pessoas gostam de receber seus extratos bancários com os cheques relacionados na ordem de número do cheque. Em geral, as pessoas preenchem seus cheques na ordem do número do cheque, e os comerciantes normalmente os descontam com uma presteza razoável. Portanto, o problema de converter a ordenação pela hora da transação na ordenação pelo número do cheque é o problema de ordenar uma entrada quase ordenada. Demonstre que o procedimento INSERTION-SORT tenderia a superar o procedimento QUICKSORT nesse problema.

7.2-5

Suponha que as divisões em todo nível de quicksort estejam na proporção $1 - \alpha$ para α, onde $0 < \alpha \leq 1/2$ é uma constante. Mostre que a profundidade mínima de uma folha na árvore de recursão é aproximadamente $-\lg n/\lg \alpha$ e a profundidade máxima é aproximadamente $-\lg n/\lg(1 - \alpha)$. (Não se preocupe com o arredondamento até inteiro.)

7.2-6 ★

Demonstre que, para qualquer constante $0 < \alpha \leq 1/2$, a probabilidade de que, em um arranjo de entradas aleatórias, PARTITION produza uma divisão mais equilibrada que $1 - \alpha$ para α é aproximadamente $1 - 2\alpha$.

7.3 Uma versão aleatória de quicksort

Na exploração do comportamento do caso médio de quicksort, fizemos uma suposição de que todas as permutações dos números de entrada são igualmente prováveis. Porém, em uma situação de engenharia nem sempre podemos esperar que ela se mantenha válida. (Ver Exercício 7.2-4.) Como vimos na Seção 5.3, às vezes adicionamos um caráter aleatório a um algoritmo para obter bom desempenho no caso médio sobre todas as entradas. Muitas pessoas consideram a versão aleatória resultante de quicksort o algoritmo de ordenação preferido para entrada grandes o suficiente.

Na Seção 5.3, tornamos nosso algoritmo aleatório permutando explicitamente a entrada. Também poderíamos fazer isso para quicksort, mas uma técnica de aleatoriedade diferente, chamada amostragem aleatória, produz uma análise mais simples. Em vez de sempre usar $A[r]$ como pivô, usaremos um elemento escolhido ao acaso a partir do subarranjo $A[p \,..\, r]$. Faremos isso permutando o elemento $A[r]$ com um elemento escolhido ao acaso de $A[p \,..\, r]$. Essa modificação, em que fazemos a amostragem aleatória do intervalo $p, ..., r$, assegura que o elemento pivô $x = A[r]$ tem a mesma probabilidade de ser qualquer um dos $r - p + 1$ elementos do subarranjo. Como o elemento pivô é escolhido ao acaso, esperamos que a divisão do arranjo de entrada seja razoavelmente bem equilibrada na média.

As mudanças em PARTITION e QUICKSORT são pequenas. No novo procedimento de partição, simplesmente implementamos a troca antes do particionamento real:

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  trocar A[p] ↔ A[i]
3  return PARTITION(A, p, r)
```

O novo quicksort chama RANDOMIZED-PARTITION em lugar de PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2     then q ← RANDOMIZED-PARTITION(A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q - 1)
4          RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Analisaremos esse algoritmo na próxima seção.

Exercícios

7.3-1

Por que analisamos o desempenho do caso médio de um algoritmo aleatório e não seu desempenho no pior caso?

7.3-2

Durante a execução do procedimento RANDOMIZED-QUICKSORT, quantas chamadas são feitas ao gerador de números aleatórios RANDOM no pior caso? E no melhor caso? Dê a resposta em termos de notação Θ.

7.4 Análise de quicksort

A Seção 7.2 forneceu alguma intuição sobre o comportamento do pior caso do quicksort e sobre o motivo pelo qual esperamos que ele funcione rapidamente. Nesta seção, analisaremos o comportamento do quicksort de forma mais rigorosa. Começaremos com uma análise do pior caso, que se aplica a QUICKSORT ou a RANDOMIZED-QUICKSORT, e concluímos com uma análise do caso médio de RANDOMIZED-QUICKSORT.

7.4.1 Análise do pior caso

Vimos na Seção 7.2 que uma divisão do pior caso em todo nível de recursão do quicksort produz um tempo de execução igual a $\Theta(n^2)$ que, intuitivamente, é o tempo de execução do pior caso do algoritmo. Agora, vamos provar essa afirmação.

Usando o método de substituição (ver Seção 4.1), podemos mostrar que o tempo de execução de quicksort é $O(n^2)$. Seja $T(n)$ o tempo no pior caso para o procedimento QUICKSORT sobre uma entrada de tamanho n. Então, temos a recorrência

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)\,, \qquad (7.1)$$

onde o parâmetro q varia de 0 a $n-1$, porque o procedimento PARTITION produz dois subproblemas com tamanho total $n-1$. Supomos que $T(n) \le cn^2$ para alguma constante c. Pela substituição dessa suposição na recorrência (7.1), obtemos

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)\,. \end{aligned}$$

A expressão $q^2 + (n-q-1)^2$ alcança um máximo sobre o intervalo $0 \le q \le n-1$ do parâmetro em um dos pontos extremos, como pode ser visto pelo fato da segunda derivada da expressão com relação a q ser positiva (ver Exercício 7.4-3). Essa observação nos dá o limite $\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$. Continuando com nossa definição do limite de $T(n)$, obtemos

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2\,, \end{aligned}$$

pois podemos escolher a constante c grande o suficiente para que o termo $c(2n-1)$ domine o termo $\Theta(n)$. Portanto, $T(n) = O(n^2)$. Vimos na Seção 7.2 um caso específico em que quicksort demora o tempo $\Omega(n^2)$: quando o particionamento é desequilibrado. Como alternativa, o Exercício 7.4-1 lhe pede para mostrar que a recorrência (7.1) tem uma solução $T(n) = \Omega(n^2)$. Desse modo, o tempo de execução (no pior caso) de quicksort é $\Theta(n^2)$.

7.4.2 Tempo de execução esperado

Já fornecemos um argumento intuitivo sobre o motivo pelo qual o tempo de execução do caso médio de RANDOMIZED-QUICKSORT é $O(n \lg n)$: se, em cada nível de recursão, a divisão induzida por RANDOMIZED-PARTITION colocar qualquer fração constante dos elementos em um lado da partição, então a árvore de recursão terá a profundidade $\Theta(\lg n)$, e o trabalho $O(n)$ será

executado em cada nível. Ainda que sejam adicionados novos níveis com a divisão mais desequilibrada possível entre esses níveis, o tempo total continuará a ser $O(n \lg n)$. Podemos analisar o tempo de execução esperado de RANDOMIZED-QUICKSORT com precisão, compreendendo primeiro como o procedimento de particionamento opera, e depois usando essa compreensão para derivar um limite $O(n \lg n)$ sobre o tempo de execução esperado. Esse limite superior no tempo de execução esperado, combinado com o limite no melhor caso $\Theta(n \lg n)$ que vimos na Seção 7.2, resulta em um tempo de execução esperado $\Theta(n \lg n)$.

Tempo de execução e comparações

O tempo de execução de QUICKSORT é dominado pelo tempo gasto no procedimento PARTITION. Toda vez que o procedimento PARTITION é chamado, um elemento pivô é selecionado, e esse elemento nunca é incluído em quaisquer chamadas recursivas futuras a QUICKSORT e PARTITION. Desse modo, pode haver no máximo n chamadas a PARTITION durante a execução inteira do algoritmo de quicksort. Uma chamada a PARTITION demora o tempo $O(1)$ mais um período de tempo proporcional ao número de iterações do loop **for** das linhas 3 a 6. Cada iteração desse loop **for** executa uma comparação na linha 4, comparando o elemento pivô a outro elemento do arranjo A. Assim, se pudermos contar o número total de vezes que a linha 4 é executada, poderemos limitar o tempo total gasto no loop **for** durante toda a execução de QUICKSORT.

***Lema** 7.1*
Seja X o número de comparações executadas na linha 4 de PARTITION por toda a execução de QUICKSORT sobre um arranjo de n elementos. Então, o tempo de execução de QUICKSORT é $O(n + X)$.

Prova Pela discussão anterior, existem n chamadas a PARTITION, cada uma das quais faz uma proporção constante do trabalho e depois executa o loop **for** um certo número de vezes. Cada iteração do loop **for** executa a linha 4. ■

Portanto, nossa meta é calcular X, o número total de comparações executadas em todas as chamadas a PARTITION. Não tentaremos analisar quantas comparações são feitas em *cada* chamada a PARTITION. Em vez disso, derivaremos um limite global sobre o número total de comparações. Para fazê-lo, devemos reconhecer quando o algoritmo compara dois elementos do arranjo e quando ele não o faz. Para facilitar a análise, renomeamos os elementos do arranjo A como $z_1, z_2, \ldots, z_n$, com z_i sendo o i-ésimo menor elemento. Também definimos o conjunto $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ como o conjunto de elementos entre z_i e z_j, inclusive.

Quando o algoritmo compara z_i e z_j? Para responder a essa pergunta, primeiro observamos que cada par de elementos é comparado no máximo uma vez. Por quê? Os elementos são comparados apenas ao elemento pivô e, depois que uma chamada específica de PARTITION termina, o elemento pivô usado nessa chamada nunca é comparado novamente a quaisquer outros elementos.

Nossa análise utiliza variáveis indicadoras aleatórias (consulte a Seção 5.2). Definimos

$$X_{ij} = \mathrm{I}\,\{z_i \text{ é comparado a } z_j\}\ ,$$

onde estamos considerando se a comparação tem lugar em qualquer instante durante a execução do algoritmo, não apenas durante uma iteração ou uma chamada de PARTITION. Tendo em vista que cada par é comparado no máximo uma vez, podemos caracterizar facilmente o número total de comparações executadas pelo algoritmo:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\ .$$

Tomando as expectativas em ambos os lados, e depois usando a linearidade de expectativa e o Lema 5.1, obtemos

$$\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ é comparado a } z_j\} . \qquad (7.2)
\end{aligned}$$

Resta calcular Pr $\{z_i$ é comparado a $z_j\}$. Nossa análise parte do princípio de que cada pivô é escolhido ao acaso e de forma independente.

É útil imaginar quando dois itens *não* são comparados. Considere uma entrada para quicksort dos números 1 a 10 (em qualquer ordem) e suponha que o primeiro elemento pivô seja 7. Então, a primeira chamada a PARTITION separa os números em dois conjuntos: $\{1, 2, 3, 4, 5, 6\}$ e $\{8, 9, 10\}$. Fazendo-se isso, o elemento pivô 7 é comparado a todos os outros elementos, mas nenhum número do primeiro conjunto (por exemplo, 2) é ou jamais será comparado a qualquer número do segundo conjunto (por exemplo, 9).

Em geral, uma vez que um pivô x é escolhido com $z_i < x < z_j$, sabemos que z_i e z_j não podem ser comparados em qualquer momento subseqüente. Se, por outro lado, z_i for escolhido como um pivô antes de qualquer outro item em Z_{ij}, então z_i será comparado a cada item em Z_{ij}, exceto ele mesmo. De modo semelhante, se z_j for escolhido como pivô antes de qualquer outro item em Z_{ij}, então z_j será comparado a cada item em Z_{ij}, exceto ele próprio. Em nosso exemplo, os valores 7 e 9 são comparados porque 7 é o primeiro item de $Z_{7,9}$ a ser escolhido como pivô. Em contraste, 2 e 9 nunca serão comparados, porque o primeiro elemento pivô escolhido de $Z_{2,9}$ é 7. Desse modo, z_i e z_j são comparados se e somente se o primeiro elemento a ser escolhido como pivô de Z_{ij} é z_i ou z_j.

Agora, calculamos a probabilidade de que esse evento ocorra. Antes do ponto em que um elemento de Z_{ij} é escolhido como pivô, todo o conjunto Z_{ij} está reunido na mesma partição. Por conseguinte, qualquer elemento de Z_{ij} tem igual probabilidade de ser o primeiro escolhido como pivô. Pelo fato do conjunto Z_{ij} ter $j - i + 1$ elementos e tendo em vista que os pivôs são escolhidos ao acaso e de forma independente, a probabilidade de qualquer elemento dado ser o primeiro escolhido como pivô é $1/(j - i + 1)$. Desse modo, temos

$$\begin{aligned}
\Pr\{z_i \text{ é comparado a } z_j\} &= \Pr\{z_i \text{ ou } z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \qquad (7.3) \\
&= \Pr\{z_i \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ é o primeiro pivô escolhido de } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} .
\end{aligned}$$

A segunda linha se segue porque os dois eventos são mutuamente exclusivos. Combinando as equações (7.2) e (7.3), obtemos

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} .$$

Podemos avaliar essa soma usando uma troca de variáveis ($k = j - i$) e o limite sobre a série harmônica na equação (A.7):

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) . \qquad (7.4)
\end{aligned}
$$

Desse modo concluímos que, usando-se RANDOMIZED-PARTITION, o tempo de execução esperado de quicksort é $O(n \lg n)$.

Exercícios

7.4-1

Mostre que, na recorrência

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) ,$$

$T(n) = \Omega(n^2)$.

7.4-2

Mostre que o tempo de execução do quicksort no melhor caso é $\Omega(n \lg n)$.

7.4-3

Mostre que $q^2 + (n-q-1)^2$ alcança um máximo sobre $q = 0, 1, ..., n-1$ quando $q = 0$ ou $q = n-1$.

7.4-4

Mostre que o tempo de execução esperado do procedimento RANDOMIZED-QUICKSORT é $\Omega(n \lg n)$.

7.4-5

O tempo de execução do quicksort pode ser melhorado na prática, aproveitando-se o tempo de execução muito pequeno da ordenação por inserção quando sua entrada se encontra "quase" ordenada. Quando o quicksort for chamado em um subarranjo com menos de k elementos, deixe-o simplesmente retornar sem ordenar o subarranjo. Após o retorno da chamada de alto nível a quicksort, execute a ordenação por inserção sobre o arranjo inteiro, a fim de concluir o processo de ordenação. Mostre que esse algoritmo de ordenação é executado no tempo esperado $O(nk + n \lg(n/k))$. Como k deve ser escolhido, tanto na teoria quanto na prática?

7.4-6 ★

Considere a modificação do procedimento PARTITION pela escolha aleatória de três elementos do arranjo A e pelo particionamento sobre sua mediana (o valor médio dos três elementos). Faça a aproximação da probabilidade de se obter na pior das hipóteses uma divisão de α para $(1 - \alpha)$, como uma função de α no intervalo $0 < \alpha < 1$.

Problemas

7-1 Correção da partição de Hoare

A versão de PARTITION dada neste capítulo não é o algoritmo de particionamento original. Aqui está o algoritmo de partição original, devido a T. Hoare:

```
HOARE-PARTITION(A, p, r)
 1 x ← A[p]
 2 i ← p - 1
 3 j ← r + 1
 4 while TRUE
 5     do repeat j ← j - 1
 6          until A[j] ≤ x
 7        repeat i ← i + 1
 8          until A[i] ≥ x
 9        if i < j
10          then trocar A[i] ↔ A[j]
11          else return j
```

a. Demonstre a operação de HOARE-PARTITION sobre o arranjo $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, mostrando os valores do arranjo e os valores auxiliares depois de cada iteração do loop **for** das linhas 4 a 11.

As três perguntas seguintes lhe pedem para apresentar um argumento cuidadoso de que o procedimento HOARE-PARTITION é correto. Prove que:

b. Os índices i e j são tais que nunca acessamos um elemento de A fora do subarranjo $A[p \,..\, r]$.

c. Quando HOARE-PARTITION termina, ele retorna um valor j tal que $p \leq j < r$.

d. Todo elemento de $A[p \,..\, j]$ é menor que ou igual a todo elemento de $A[j + 1 \,..\, r]$ quando HOARE-PARTITION termina.

O procedimento PARTITION da Seção 7.1 separa o valor do pivô (originalmente em $A[r]$) das duas partições que ele forma. Por outro lado, o procedimento HOARE-PARTITION sempre insere o valor do pivô (originalmente em $A[p]$) em uma das duas partições $A[p \,..\, j]$ e $A[j + 1 \,..\, r]$. Como $p \leq j < r$, essa divisão é sempre não trivial.

e. Reescreva o procedimento QUICKSORT para usar HOARE-PARTITION.

7-2 Análise alternativa de quicksort

Uma análise alternativa do tempo de execução de quicksort aleatório se concentra no tempo de execução esperado de cada chamada recursiva individual a QUICKSORT, em vez de se ocupar do número de comparações executadas.

a. Demonstre que, dado um arranjo de tamanho n, a probabilidade de que qualquer elemento específico seja escolhido como pivô é $1/n$. Use isso para definir variáveis indicadoras aleatórias $X_i = \mathrm{I}\{i\text{-ésimo menor elemento é escolhido como pivô}\}$. Qual é $\mathrm{E}[X_i]$?

b. Seja $T(n)$ uma variável aleatória denotando o tempo de execução de quicksort sobre um arranjo de tamanho n. Demonstre que

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q) + \Theta(n))\right]. \tag{7.5}$$

c. Mostre que a equação (7.5) é simplificada para

$$E[T(n)] = \frac{2}{n}\sum_{q=0}^{n-1} E[T(q)] + \Theta(n) . \quad (7.6)$$

d. Mostre que

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 . \quad (7.7)$$

(*Sugestão*: Divida o somatório em duas partes, uma para $k = 1, 2, ..., \lceil n/2 \rceil - 1$ e uma para $k = \lceil n/2 \rceil, ..., n - 1$.)

e. Usando o limite da equação (7.7), mostre que a recorrência na equação (7.6) tem a solução $E[T(n)] = \Theta(n \lg n)$. (*Sugestão*: Mostre, por substituição, que $E[T(n)] \leq an \log n - bn$ para algumas constantes positivas a e b.)

7-3 Ordenação do pateta

Os professores Howard, Fine e Howard propuseram o seguinte algoritmo de ordenação "elegante":

```
STOOGE-SORT(A, i, j)
1 if A[i] > A[j]
2    then trocar A[i] ↔ A[j]
3 if i + 1 ≥ j
4    then return
5 k ← ⌊(j – i + 1)/3⌋            ▷ Arredonda para menos.
6 STOOGE-SORT(A, i, j – k)       ▷ Primeiros dois terços.
7 STOOGE-SORT(A, i + k, j)       ▷ Últimos dois terços.
8 STOOGE-SORT(A, i, j – k)       ▷ Primeiros dois terços novamente.
```

a. Mostre que, se n = *comprimento*[A], então STOOGE-SORT(A, 1, *comprimento*[A]) ordena corretamente o arranjo de entrada $A[1 .. n]$.

b. Forneça uma recorrência para o tempo de execução no pior caso de STOOGE-SORT e um limite assintótico restrito (notação Θ) sobre o tempo de execução no pior caso.

c. Compare o tempo de execução no pior caso de STOOGE-SORT com o da ordenação por inserção, da ordenação por intercalação, de heapsort e de quicksort. Os professores merecem a estabilidade no emprego?

7-4 Profundidade de pilha para quicksort

O algoritmo QUICKSORT da Seção 7.1 contém duas chamadas recursivas a ele próprio. Após a chamada a PARTITION, o subarranjo da esquerda é ordenado recursivamente, e depois o subarranjo da direita é ordenado recursivamente. A segunda chamada recursiva em QUICKSORT não é realmente necessária; ela pode ser evitada pelo uso de uma estrutura de controle iterativa. Essa técnica, chamada ***recursão do final***, é automaticamente fornecida por bons compiladores. Considere a versão de quicksort a seguir, que simula a recursão do final.

```
QUICKSORT'(A, p, r)
1 while p < r
2    do ▷ Particiona e ordena o subarranjo esquerdo
3       q ← PARTITION(A, p, r)
4       QUICKSORT'(A, p, q – 1)
5       p ← q + 1
```

a. Mostre que QUICKSORT'(A, 1, *comprimento*[A]) ordena corretamente o arranjo A.

Os compiladores normalmente executam procedimentos recursivos usando uma ***pilha*** que contém informações pertinentes, inclusive os valores de parâmetros, para cada chamada recursiva. As informações para a chamada mais recente estão na parte superior da pilha, e as informações para a chamada inicial encontram-se na parte inferior. Quando um procedimento é invocado, suas informações são ***empurradas*** sobre a pilha; quando ele termina, suas informações são ***extraídas***. Tendo em vista nossa suposição de que os parâmetros de arranjos são na realidade representados por ponteiros, as informações correspondentes a cada chamada de procedimento na pilha exigem o espaço de pilha $O(1)$. A ***profundidade de pilha*** é a quantidade máxima de espaço da pilha usado em qualquer instante durante uma computação.

b. Descreva um cenário no qual a profundidade de pilha de QUICKSORT' é $\Theta(n)$ sobre um arranjo de entrada de n elementos.

c. Modifique o código de QUICKSORT' de tal modo que a profundidade de pilha no pior caso seja $\Theta(\lg n)$. Mantenha o tempo de execução esperado $O(n \lg n)$ do algoritmo.

7-5 Partição de mediana de 3

Um modo de melhorar o procedimento RANDOMIZED-QUICKSORT é criar uma partição em torno de um elemento x escolhido com maior cuidado que a simples escolha de um elemento aleatório do subarranjo. Uma abordagem comum é o método da ***mediana de 3***: escolha x como a mediana (o elemento intermediário) de um conjunto de 3 elementos selecionados aleatoriamente a partir do subarranjo. Para esse problema, vamos supor que os elementos no arranjo de entrada $A[1 \,..\, n]$ sejam distintos e que $n \geq 3$. Denotamos o arranjo de saída ordenado por $A'[1 \,..\, n]$. Usando o método da mediana de 3 para escolher o elemento pivô x, defina $p_i = \Pr\{x = A'[i]\}$.

a. Dê uma fórmula exata para pi como uma função de n e i para $i = 2, 3, ..., n - 1$. (Observe que $p_1 = p_n = 0$.)

b. Por qual valor aumentamos a probabilidade de escolher $x = A'[\lfloor (n + 1)/2 \rfloor]$, a mediana de $A[1 \,..\, n]$, em comparação com a implementação comum? Suponha que $n \to \infty$ e forneça a razão de limitação dessas probabilidades.

c. Se definimos uma "boa" divisão com o significado de escolher o pivô como $x = A'[i]$, onde $n/3 \leq i \leq 2n/3$, por qual quantidade aumentamos a probabilidade de se obter uma boa divisão em comparação com a implementação comum? (*Sugestão*: Faça uma aproximação da soma por uma integral.)

d. Mostre que, no tempo de execução $\Omega(n \lg n)$ de quicksort, o método da mediana de 3 só afeta o fator constante.

7-6 Ordenação nebulosa de intervalos

Considere um problema de ordenação no qual os números não são conhecidos exatamente. Em vez disso, para cada número, conhecemos um intervalo sobre a linha real a que ele pertence. Ou seja, temos n intervalos fechados da forma $[a_i, b_i]$, onde $a_i \leq b_i$. O objetivo é fazer a ***ordenação nebulosa*** desses intervalos, isto é, produzir uma permutação $\langle i_1, i_2, ..., i_n \rangle$ dos intervalos tal que exista $c_j \in [a_{i_j}, b_{i_j}]$ que satisfaça a $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Projete um algoritmo para ordenação do pateta de n intervalos. Seu algoritmo devia ter a estrutura geral de um algoritmo que faz o quicksort das extremidades esquerdas (os valores a_i), mas deve tirar proveito da sobreposição de intervalos para melhorar o tempo de execução. (À medida que os intervalos se sobrepõem cada vez mais, o problema de fazer a ordenação do pateta dos intervalos fica cada vez mais fácil. Seu algoritmo deve aproveitar tal sobreposição, desde que ela exista.)

b. Demonstre que seu algoritmo é executado no tempo esperado $\Theta(n \lg n)$ em geral, mas funciona no tempo esperado $\Theta(n)$ quando todos os intervalos se sobrepõem (isto é, quando existe um valor x tal que $x \in [a_i, b_i]$ para todo i). O algoritmo não deve verificar esse caso de forma explícita; em vez disso, seu desempenho deve melhorar naturalmente à medida que aumentar a proporção de sobreposição.

Notas do capítulo

O procedimento quicksort foi criado por Hoare [147]; a versão de Hoare aparece no Problema 7-1. O procedimento PARTITION dado na Seção 7.1 se deve a N. Lomuto. A análise da Seção 7.4 se deve a Avrim Blum. Sedgewick [268] e Bentley [40] fornecem uma boa referência sobre os detalhes de implementação e como eles são importantes.

McIlroy [216] mostrou como engenheiro um "adversário matador" que produz um arranjo sobre o qual virtualmente qualquer implementação de quicksort demora o tempo $\Theta(n^2)$. Se a implementação é aleatória, o adversário produz o arranjo depois de ver as escolhas ao acaso do algoritmo de quicksort.

Capítulo 8

Ordenação em tempo linear

Apresentamos até agora diversos algoritmos que podem ordenar n números no tempo $O(n \lg n)$. A ordenação por intercalação e o heapsort alcançam esse limite superior no pior caso; quicksort o alcança na média. Além disso, para cada um desses algoritmos, podemos produzir uma seqüência de n números de entrada que faz o algoritmo ser executado no tempo $\Omega(n \lg n)$.

Esses algoritmos compartilham uma propriedade interessante: *a seqüência ordenada que eles determinam* se *baseia apenas em comparações entre os elementos de entrada*. Chamamos esses algoritmos de ordenação de ***ordenações por comparação***. Todos os algoritmos de ordenação apresentados até agora são portanto ordenações por comparação.

Na Seção 8.1, provaremos que qualquer ordenação por comparação deve efetuar $\Omega(n \lg n)$ comparações no pior caso para ordenar n elementos. Desse modo, a ordenação por intercalação e heapsort são assintoticamente ótimas, e não existe nenhuma ordenação por comparação que seja mais rápida por mais de um fator constante.

As Seções 8.2, 8.3 e 8.4 examinam três algoritmos de ordenação – ordenação por contagem, radix sort (ordenação da raiz) e bucket sort (ordenação por balde) – que são executados em tempo linear. É desnecessário dizer que esses algoritmos utilizam outras operações diferentes de comparações para determinar a seqüência ordenada. Em conseqüência disso, o limite inferior $\Omega(n \lg n)$ não se aplica a eles.

8.1 Limites inferiores para ordenação

Em uma ordenação por comparação, usamos apenas comparações entre elementos para obter informações de ordem sobre uma seqüência de entrada $\langle a_1, a_2, ..., a_n \rangle$. Ou seja, dados dois elementos a_i e a_j, executamos um dos testes $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ ou $a_i > a_j$, para determinar sua ordem relativa. Podemos inspecionar os valores dos elementos ou obter informações de ordem sobre eles de qualquer outro modo.

Nesta seção, vamos supor sem perda de generalidade que todos os elementos de entrada são distintos. Dada essa hipótese, as comparações da forma $a_i = a_j$ são inúteis; assim, podemos supor que não é feita nenhuma comparação dessa forma. Também observamos que as comparações $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ e $a_i < a_j$ são todas equivalentes, em virtude de produzirem informações idênticas sobre a ordem relativa de a_i e a_j. Por essa razão, vamos supor que todas as comparações têm a forma $a_i \leq a_j$.

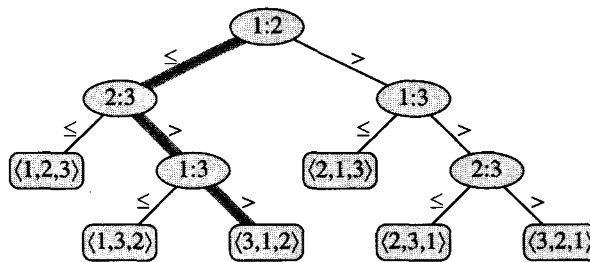


FIGURA 8.1 A árvore de decisão para ordenação por inserção, operando sobre três elementos. Um nó interno anotado por $i{:}j$ indica uma comparação entre a_i e a_j. Uma folha anotada pela permutação $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indica a ordenação $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. O caminho sombreado indica as decisões tomadas durante a ordenação da seqüência de entrada $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; a permutação $\langle 3, 1, 2 \rangle$ na folha indica que a seqüência ordenada é $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. Existem $3! = 6$ permutações possíveis dos elementos de entrada; assim, a árvore de decisão deve ter no mínimo 6 folhas

O modelo de árvore de decisão

As ordenações por comparação podem ser vistas de modo abstrato em termos de ***árvores de decisão***. Uma árvore de decisão é uma árvore binária completa que representa as comparações executadas por um algoritmo de ordenação quando ele opera sobre uma entrada de um tamanho dado. Controle, movimentação de dados e todos os outros aspectos do algoritmo são ignorados. A Figura 8.1 mostra a árvore de decisão correspondente ao algoritmo de ordenação por inserção da Seção 2.1, operando sobre uma seqüência de entrada de três elementos.

Em uma árvore de decisão, cada nó interno é anotado por $i{:}j$ para algum i e j no intervalo $1 \leq i, j \leq n$, onde n é o número de elementos na seqüência de entrada. Cada folha é anotada por uma permutação $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. (Consulte a Seção C.1 para adquirir experiência em permutações.) A execução do algoritmo de ordenação corresponde a traçar um caminho desde a raiz da árvore de decisão até uma folha. Em cada nó interno, é feita uma comparação $a_i \leq a_j$. A subárvore da esquerda determina então comparações subseqüentes para $a_i \leq a_j$, e a subárvore da direita determina comparações subseqüentes para $a_i > a_j$. Quando chegamos a uma folha, o algoritmo de ordenação estabelece a ordem $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Como qualquer algoritmo de ordenação correto deve ser capaz de produzir cada permutação de sua entrada, uma condição necessária para uma ordenação por comparação ser correta é que cada uma das $n!$ permutações sobre n elementos deve aparecer como uma das folhas da árvore de decisão, e que cada uma dessas folhas deve ser acessível a partir da raiz por um caminho correspondente a uma execução real da ordenação por comparação. (Iremos nos referir a tais folhas como "acessíveis".) Desse modo, consideraremos apenas árvores de decisão em que cada permutação aparece como uma folha acessível.

Um limite inferior para o pior caso

O comprimento do caminho mais longo da raiz de uma árvore de decisão até qualquer de suas folhas acessíveis representa o número de comparações do pior caso que o algoritmo de ordenação correspondente executa. Conseqüentemente, o número de comparações do pior caso para um dado algoritmo de ordenação por comparação é igual à altura de sua árvore de decisão. Um limite inferior sobre as alturas de todas as árvores de decisão em que cada permutação aparece como uma folha acessível é portanto um limite inferior sobre o tempo de execução de qualquer algoritmo de ordenação por comparação. O teorema a seguir estabelece esse limite inferior.

Teorema 8.1

Qualquer algoritmo de ordenação por comparação exige $\Omega(n \lg n)$ comparações no pior caso.

Prova Da discussão precedente, basta determinar a altura de uma árvore de decisão em que cada permutação aparece como uma folha acessível. Considere uma árvore de decisão de altura h com l folhas acessíveis correspondente a uma ordenação por comparação sobre n elementos. Como cada uma das $n!$ permutações da entrada aparece como alguma folha, temos $n! \leq l$. Tendo em vista que uma árvore binária de altura h não tem mais de 2^h folhas, temos

$$n! \leq l \leq 2^h ,$$

que, usando-se logaritmos, implica

$$\begin{aligned} h &\geq \lg(n!) && \text{(pois a função lg é monotonicamente crescente)} \\ &= \Omega(n \lg n) && \text{(pela equação (3.18))} . \end{aligned}$$

■

Corolário 8.2
O heapsort e a ordenação por intercalação são ordenações por comparação assintoticamente ótimas.

Prova Os $O(n \lg n)$ limites superiores sobre os tempos de execução para heapsort e ordenação por intercalação correspondem ao limite inferior $\Omega(n \lg n)$ do pior caso do Teorema 8.1. ■

Exercícios

8.1-1
Qual é a menor profundidade possível de uma folha em uma árvore de decisão para uma ordenação por comparação?

8.1-2
Obtenha limites assintoticamente restritos sobre $\lg(n!)$ sem usar a aproximação de Stirling. Em vez disso, avalie o somatório $\sum_{k=1}^{n} \lg k$, empregando técnicas da Seção A.2.

8.1-3
Mostre que não existe nenhuma ordenação por comparação cujo tempo de execução seja linear para pelo menos metade das $n!$ entradas de comprimento n. E no caso de uma fração $1/n$ das entradas de comprimento n? E no caso de uma fração $1/2^n$?

8.1-4
Você recebeu uma seqüência de n elementos para ordenar. A seqüência de entrada consiste em n/k subseqüências, cada uma contendo k elementos. Os elementos em uma dada subseqüência são todos menores que os elementos na subseqüência seguinte e maiores que os elementos na subseqüência precedente. Desse modo, tudo que é necessário para ordenar a seqüência inteira de comprimento n é ordenar os k elementos em cada uma das n/k subseqüências. Mostre um limite inferior $\Omega(n \lg k)$ sobre o número de comparações necessárias para resolver essa variação do problema de ordenação. (*Sugestão*: Não é rigoroso simplesmente combinar os limites inferiores para as subseqüências individuais.)

8.2 Ordenação por contagem

A ***ordenação por contagem*** pressupõe que cada um dos n elementos de entrada é um inteiro no intervalo de 1 a k, para algum inteiro k. Quando $k = O(n)$, a ordenação é executada no tempo $\Theta(n)$.

A idéia básica da ordenação por contagem é determinar, para cada elemento de entrada x, o número de elementos menores que x. Essa informação pode ser usada para inserir o elemento x

diretamente em sua posição no arranjo de saída. Por exemplo, se há 17 elementos menores que x, então x é colocado na posição de saída 18. Esse esquema deve ser ligeiramente modificado para manipular a situação na qual vários elementos têm o mesmo valor, pois não queremos inserir todos eles na mesma posição.

No código para ordenação por contagem, partimos da suposição de que a entrada é um arranjo $A[1 \, .. \, n]$ e, portanto, $comprimento[A] = n$. Exigimos dois outros arranjos: o arranjo $B[1 \, .. \, n]$ contém a saída ordenada, e o arranjo $C[0 \, .. \, k]$ fornece um espaço de armazenamento de trabalho temporário.

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2      do C[i] ← 0
 3  for j ← 1 to comprimento[A]
 4      do C[A[j]] ← C[A[j]] + 1
 5  ▷ Agora C[i] contém o número de elementos iguais a i.
 6  for i ← 1 to k
 7      do C[i] ← C[i] + C[i – 1]
 8  ▷ Agora C[i] contém o número de elementos menores que ou iguais a i.
 9  for j ← comprimento[A] downto 1
10      do B[C[A[j]]] ← A[j]
11         C[A[j]] ← C[A[j]] – 1
```

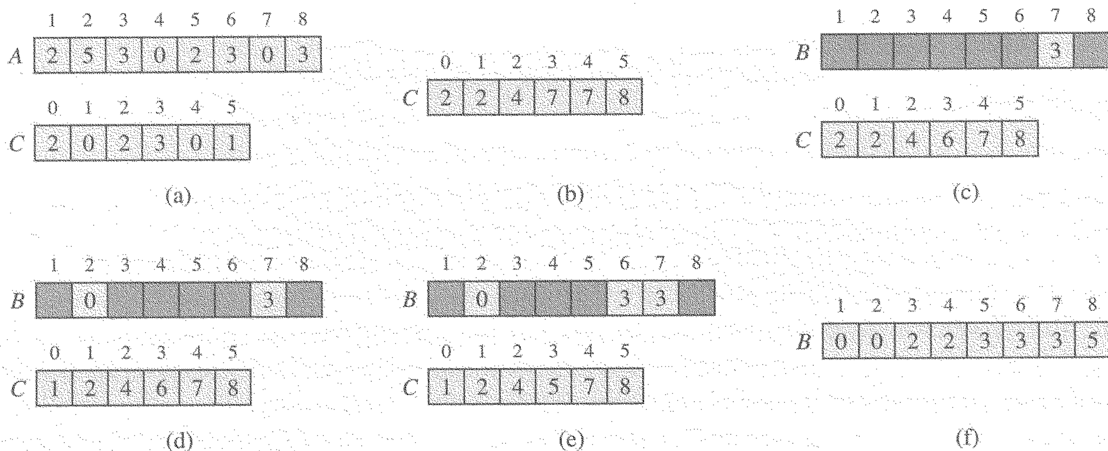


FIGURA 8.2 A operação de COUNTING-SORT sobre um arranjo de entrada $A[1 \, .. \, 8]$, onde cada elemento de A é um inteiro não negativo não maior que $k = 5$. **(a)** O arranjo A e o arranjo auxiliar C após a linha 4. **(b)** O arranjo C após a linha 7. **(c)–(e)** O arranjo de saída B e o arranjo auxiliar C após uma, duas e três iterações do loop nas linhas 9 a 11, respectivamente. Apenas os elementos levemente sombreados do arranjo B foram preenchidos. **(f)** O arranjo de saída final ordenado B

A ordenação por contagem é ilustrada na Figura 8.2. Após a inicialização no loop **for** das linhas 1 e 2, inspecionamos cada elemento de entrada no loop **for** das linhas 3 e 4. Se o valor de um elemento de entrada é i, incrementamos $C[i]$. Desse modo, depois da linha 4, $C[i]$ contém um número de elementos de entrada igual a i para cada inteiro $i = 0, 1, ..., k$. Nas linhas 6 e 7 determinamos, para cada $i = 0, 1, ..., k$, quantos elementos de entrada são menores que ou iguais a i; mantendo uma soma atualizada do arranjo C.

Finalmente, no loop **for** das linhas 9 a 11, colocamos cada elemento $A[j]$ em sua posição ordenada correta no arranjo de saída B. Se todos os n elementos forem distintos, então, quando entrarmos pela primeira vez na linha 9, para cada $A[j]$, o valor $C[A[j]]$ será a posição final correta

de $A[j]$ no arranjo de saída, pois existem $C[A[j]]$ elementos menores que ou iguais a $A[j]$. Como os elementos podem não ser distintos, decrementamos $C[A[j]]$ toda vez que inserimos um valor $A[j]$ no arranjo B; isso faz com que o próximo elemento de entrada com um valor igual a $A[j]$, se existir algum, vá para a posição imediatamente anterior a $A[j]$ no arranjo de saída.

Quanto tempo a ordenação por contagem exige? O loop **for** das linhas 1 e 2 demora o tempo $\Theta(k)$, o loop **for** das linhas 3 e 4 demora o tempo $\Theta(n)$, o loop **for** das linhas 6 e 7 demora o tempo $\Theta(k)$ e o loop **for** das linhas 9 a 11 demora o tempo $\Theta(n)$. Portanto, o tempo total é $\Theta(k + n)$. Na prática, normalmente usamos a ordenação por contagem quando temos $k = O(n)$, em cujo caso o tempo de execução é $\Theta(n)$.

A ordenação por contagem supera o limite inferior de $\Omega(n \lg n)$ demonstrado na Seção 8.1, porque não é uma ordenação por comparação. De fato, nenhuma comparação entre elementos de entrada ocorre em qualquer lugar no código. Em vez disso, a ordenação por contagem utiliza os valores reais dos elementos para efetuar a indexação em um arranjo. O limite inferior $\Omega(n \lg n)$ para ordenação não se aplica quando nos afastamos do modelo de ordenação por comparação.

Uma propriedade importante da ordenação por contagem é o fato de ela ser ***estável***: números com o mesmo valor aparecem no arranjo de saída na mesma ordem em que se encontram no arranjo de entrada. Ou seja, os vínculos entre dois números são rompidos pela regra de que qualquer número que aparecer primeiro no arranjo de entrada aparecerá primeiro no arranjo de saída. Normalmente, a propriedade de estabilidade só é importante quando dados satélite são transportados juntamente com o elemento que está sendo ordenado. A estabilidade da ordenação por contagem é importante por outra razão: a ordenação por contagem é usada freqüentemente como uma sub-rotina em radix sort. Como veremos na próxima seção, a estabilidade da ordenação por contagem é crucial para a correção da radix sort.

Exercícios

8.2-1

Usando a Figura 8.2 como modelo, ilustre a operação de COUNTING-SORT sobre o arranjo $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

8.2-2

Prove que COUNTING-SORT é estável.

8.2-3

Suponha que o cabeçalho do loop **for** na linha 9 do procedimento COUNTING-SORT seja reescrito:

9 **for** $j \leftarrow 1$ **to** *comprimento*$[A]$

Mostre que o algoritmo ainda funciona corretamente. O algoritmo modificado é estável?

8.2-4

Descreva um algoritmo que, dados n inteiros no intervalo de 0 a k, realiza o pré-processamento de sua entrada e depois responde a qualquer consulta sobre quantos dos n inteiros recaem em um intervalo $[a \,..\, b]$ no tempo $O(1)$. Seu algoritmo deve utilizar o tempo de pré-processamento $\Theta(n + k)$.

8.3 Radix sort

A ***radix sort*** (ou ***ordenação da raiz***) é o algoritmo usado pelas máquinas de ordenação de cartões que agora são encontradas apenas nos museus de informática. Os cartões estão organizados em 80 colunas, e em cada coluna pode ser feita uma perfuração em uma de 12 posições. O

ordenador pode ser "programado" mecanicamente para examinar uma determinada coluna de cada cartão em uma pilha e distribuir o cartão em uma de 12 caixas, dependendo de qual foi o local perfurado. Um operador pode então juntar os cartões caixa por caixa, de modo que os cartões com a primeira posição perfurada fiquem sobre os cartões com a segunda posição perfurada e assim por diante.

329		720		720		329
457		355		329		355
657		436		436		436
839	⇒	457	⇒	839	⇒	457
436		657		355		657
720		329		457		720
355		839		657		839

FIGURA 8.3 A operação de radix sort sobre uma lista de sete números de 3 dígitos. A primeira coluna é a entrada. As colunas restantes mostram a lista após ordenações sucessivas sobre posições de dígitos cada vez mais significativas. As setas verticais indicam a posição do dígito sobre o qual é feita a ordenação para produzir cada lista a partir da anterior

No caso de dígitos decimais, apenas 10 posições são utilizadas em cada coluna. (As outras duas posições são usadas para codificação de caracteres não numéricos.) Assim, um número de d dígitos ocuparia um campo de d colunas. Tendo em vista que o ordenador de cartões pode examinar apenas uma coluna de cada vez, o problema de ordenar n cartões em um número de d dígitos requer um algoritmo de ordenação.

Intuitivamente, poderíamos ordenar números sobre seu dígito *mais significativo*, ordenar cada uma das caixas resultantes recursivamente, e então combinar as pilhas em ordem. Infelizmente, como os cartões em 9 das 10 caixas devem ser postos de lado para se ordenar cada uma das caixas, esse procedimento gera muitas pilhas intermediárias de cartões que devem ser controladas. (Ver Exercício 8.3-5.)

A radix sort resolve o problema da ordenação de cartões de modo contra-intuitivo ordenando primeiro sobre o dígito *menos significativo*. Os cartões são então combinados em uma única pilha, com os cartões na caixa 0 precedendo os cartões na caixa 1, que precedem os cartões na caixa 2 e assim por diante. Então, a pilha inteira é ordenada novamente sobre o segundo dígito menos significativo e recombinada de maneira semelhante. O processo continua até os cartões terem sido ordenados sobre todos os d dígitos. É interessante observar que, nesse ponto, os cartões estão completamente ordenados sobre o número de d dígitos. Desse modo, apenas d passagens pela pilha são necessárias para se fazer a ordenação. A Figura 8.3 mostra como a radix sort opera sobre uma "pilha" (ou um "deck") de sete números de 3 dígitos.

É essencial que as ordenações de dígitos nesse algoritmo sejam estáveis. A ordenação executada por um ordenador de cartões é estável, mas o operador tem de ser cauteloso para não alterar a ordem dos cartões à medida que eles são retirados de uma caixa, ainda que todos os cartões em uma caixa tenham o mesmo dígito na coluna escolhida.

Em um computador típico, que é uma máquina seqüencial de acesso aleatório, a radix sort é usada às vezes para ordenar registros de informações chaveados por vários campos. Por exemplo, talvez fosse desejável ordenar datas por três chaves: ano, mês e dia. Poderíamos executar um algoritmo de ordenação com uma função de comparação que, dadas duas datas, comparasse anos e, se houvesse uma ligação, comparasse meses e, se ocorresse outra ligação, comparasse dias. Como outra alternativa, poderíamos ordenar as informações três vezes com uma ordenação estável: primeiro sobre o dia, em seguida sobre o mês, e finalmente sobre o ano.

O código para radix sort é direto. O procedimento a seguir supõe que cada elemento no arranjo de n elementos A tem d dígitos, onde o dígito 1 é o dígito de mais baixa ordem e o dígito d é o dígito de mais alta ordem.

RADIX-SORT(A, d)
1 **for** $i \leftarrow 1$ **to** d
2 **do** usar uma ordenação estável para ordenar o arranjo A sobre o dígito i

Lema 8.3
Dados n números de d dígitos em que cada dígito pode assumir até k valores possíveis, RADIX-SORT ordena corretamente esses números no tempo $\Theta(d(n + k))$.

Prova A correção de radix sort se segue por indução sobre a coluna que está sendo ordenada (ver Exercício 8.3-3). A análise do tempo de execução depende da ordenação estável usada como algoritmo de ordenação intermediária. Quando cada dígito está no intervalo de 0 a $k - 1$ (de modo que possa assumir até k valores possíveis) e k não é muito grande, a ordenação por contagem é a escolha óbvia. Cada passagem sobre n números de d dígitos leva então o tempo $\Theta(n + k)$. Há d passagens; assim, o tempo total para radix sort é $\Theta(d(n + k))$. ■

Quando d é constante e $k = O(n)$, radix sort é executada em tempo linear. Mais geralmente, temos alguma flexibilidade em como quebrar cada chave em dígitos.

Lema 8.4
Dados n números de b bits e qualquer inteiro positivo $r \leq b$, RADIX-SORT ordena corretamente esses números no tempo $\Theta((b/r)(n + 2^r))$.

Prova Para um valor $r \leq b$, visualizamos cada chave como tendo $d = \lceil b/r \rceil$ dígitos de r bits cada. Cada dígito é um inteiro no intervalo 0 a $2^r - 1$, de forma que podemos usar a ordenação por contagem com $k = 2^r - 1$. (Por exemplo, podemos visualizar uma palavra de 32 bits como tendo 4 dígitos de 8 bits, de forma que $b = 32$, $r = 8$, $k = 2^r - 1 = 255$ e $d = b/r = 4$.) Cada passagem da ordenação por contagem leva o tempo $\Theta(n + k) = \Theta(n + 2^r)$ e há d passagens, dando um tempo de execução total igual a $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. ■

Para valores de n e b dados, desejamos escolher o valor de r, com $r \leq b$, que minimiza a expressão $(b/r)(n + 2^r)$. Se $b < \lfloor \lg n \rfloor$, para qualquer valor de $r \leq b$, temos $(n + 2^r) = \Theta(n)$. Desse modo, a escolha de $r = b$ produz um tempo de execução $(b/b)(n + 2^b) = \Theta(n)$, que é assintoticamente ótimo. Se $b \geq \lfloor \lg n \rfloor$, então a escolha de $r = \lfloor \lg n \rfloor$ fornece o melhor tempo dentro de um fator constante, que podemos ver como a seguir. A escolha de $r = \lfloor \lg n \rfloor$ produz um tempo de execução $\Theta(bn/\lg n)$. À medida que aumentamos r acima de $\lfloor \lg n \rfloor$, o termo 2^r no numerador aumenta mais rápido que o termo r no denominador, e assim o aumento de r acima de $\lfloor \lg n \rfloor$ resulta em um tempo de execução $\Omega(bn/\lg n)$. Se, em vez disso, diminuirmos r abaixo de $\lfloor \lg n \rfloor$, então o termo b/r aumentará, e o termo $n + 2^r$ permanecerá em $\Theta(n)$.

É preferível radix sort a um algoritmo de ordenação baseado em comparação, como quicksort? Se $b = O(\lg n)$, como é freqüentemente o caso, e escolhemos $r \approx \lg n$, então o tempo de execução de radix sort é $\Theta(n)$, que parece ser melhor que o tempo do caso médio de quicksort, $\Theta(n \lg n)$. Porém, os fatores constantes ocultos na notação Θ são diferentes. Embora radix sort possa fazer menos passagens que quicksort sobre as n chaves, cada passagem de radix sort pode tomar um tempo significativamente maior. Determinar o algoritmo de ordenação preferível depende das características das implementações, da máquina subjacente (por exemplo, quicksort utiliza com freqüência caches de hardware de modo mais eficaz que radix sort) e dos dados de entrada. Além disso, a versão de radix sort que utiliza a ordenação por contagem como ordenação estável intermediária não efetua a ordenação local, o que é feito por muitas das ordenações por comparação no tempo $\Theta(n \lg n)$. Desse modo, quando o espaço de armazenamento da memória primária é importante, um algoritmo local como quicksort pode ser preferível.

Exercícios

8.3-1

Usando a Figura 8.3 como modelo, ilustre a operação de RADIX-SORT sobre a seguinte lista de palavras em inglês: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

8.3-2

Quais dos seguintes algoritmos de ordenação são estáveis: ordenação por inserção, ordenação por intercalação, heapsort e quicksort? Forneça um esquema simples que torne estável qualquer algoritmo de ordenação. Quanto tempo e espaço adicional seu esquema requer?

8.3-3

Use a indução para provar que radix sort funciona. Onde sua prova necessita da hipótese de que a ordenação intermediária é estável?

8.3-4

Mostre como ordenar n inteiros no intervalo de 0 a $n^2 - 1$ no tempo $O(n)$.

8.3-5 ★

No primeiro algoritmo de ordenação de cartões desta seção, exatamente quantas passagens de ordenação são necessárias para ordenar números decimais de d dígitos no pior caso? Quantas pilhas de cartões um operador precisaria controlar no pior caso?

8.4 Bucket sort

A ***bucket sort*** (ou ***ordenação por balde***) funciona em tempo linear quando a entrada é gerada a partir de uma distribuição uniforme. Como a ordenação por contagem, a bucket sort é rápida porque pressupõe algo sobre a entrada. Enquanto a ordenação por contagem presume que a entrada consiste em inteiros em um intervalo pequeno, bucket sort presume que a entrada é gerada por um processo aleatório que distribui elementos uniformemente sobre o intervalo $[0, 1)$. (Consulte a Seção C.2 para ver uma definição de distribuição uniforme.)

A idéia de bucket sort é dividir o intervalo $[0, 1)$ em n subintervalos de igual tamanho, ou ***baldes***, e depois distribuir os n números de entrada entre os baldes. Tendo em vista que as entradas são uniformemente distribuídas sobre $[0, 1)$, não esperamos que muitos números caiam em cada balde. Para produzir a saída, simplesmente ordenamos os números em cada balde, e depois percorremos os baldes em ordem, listando os elementos contidos em cada um.

Nosso código para bucket sort pressupõe que a entrada é um arranjo de n elementos A, e que cada elemento $A[i]$ no arranjo satisfaz a $0 \leq A[i] < 1$. O código exige um arranjo auxiliar $B[0 \,..\, n-1]$ de listas ligadas (baldes) e pressupõe que existe um mecanismo para manter tais listas. (A Seção 10.2 descreve como implementar operações básicas sobre listas ligadas.)

BUCKET-SORT(A)
1 $n \leftarrow$ *comprimento*$[A]$
2 **for** $i \leftarrow 1$ **to** n
3 **do** inserir $A[i]$ na lista $B[\lfloor nA[i] \rfloor]$
4 **for** $i \leftarrow 0$ **to** $n - 1$
5 **do** ordenar lista $B[i]$ com ordenação por inserção
6 concatenar as listas $B[0]$, $B[1]$, ..., $B[n-1]$ juntas em ordem

A Figura 8.4 mostra a operação de bucket sort sobre um arranjo de entrada de 10 números.

Para ver que esse algoritmo funciona, considere dois elementos $A[i]$ e $A[j]$. Suponha sem perda de generalidade que $A[i] \leq A[j]$. Tendo em vista que $\lfloor nA[i] \rfloor \leq \lfloor nA[j] \rfloor$, o elemento $A[i]$ é inserido no mesmo balde que $A[j]$ ou em um balde com índice mais baixo. Se $A[i]$ e $A[j]$ são inseri-

dos no mesmo balde, então o loop **for** das linhas 4 e 5 os coloca na ordem adequada. Se $A[i]$ e $A[j]$ são inseridos em baldes diferentes, a linha 6 os coloca na ordem adequada. Portanto, a bucket sort funciona corretamente.

Para analisar o tempo de execução, observe que todas as linhas exceto a linha 5 demoram o tempo $O(n)$ no pior caso. Resta equilibrar o tempo total ocupado pelas n chamadas à ordenação por inserção na linha 5.

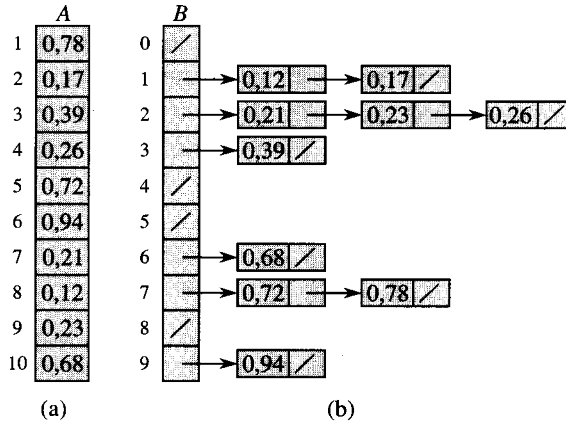


FIGURA 8.4 A operação de BUCKET-SORT. (**a**) O arranjo de entrada $A[1 \,..\, 10]$. (**b**) O arranjo $B[0 \,..\, 9]$ de listas ordenadas (baldes) depois da linha 5 do algoritmo. O balde i contém valores no intervalo $[i/10, (i + 1)/10)$. A saída ordenada consiste em uma concatenação em ordem das listas $B[0], B[1], ..., B[9]$

Para analisar o custo das chamadas para ordenação por inserção, seja ni a variável aleatória que denota o número de elementos inseridos no balde $B[i]$. Tendo em vista que a ordenação por inserção funciona em tempo quadrático (consulte a Seção 2.2), o tempo de execução de bucket sort é

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

Tomando as expectativas de ambos os lados e usando a linearidade de expectativa, temos

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \qquad \text{(por linearidade de expectativa)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) \qquad \text{(pela equação (C.21))} . \qquad (8.1) \end{aligned}$$

Afirmamos que

$$\mathrm{E}[n_i^2] = 2 - 1/n \qquad (8.2)$$

para $i = 0, 1, ..., n - 1$. Não surpreende que cada balde i tenha o mesmo valor de $\mathrm{E}[n_i^2]$, pois cada valor no arranjo de entrada A tem igual probabilidade de cair em qualquer balde. Para provar a equação (8.2), definimos variáveis indicadoras aleatórias

$X_{ij} = \mathrm{I}\ \{A[j] \text{ recai no balde } i\}$

para $i = 0, 1, ..., n - 1$ e $j = 1, 2, ..., n$. Desse modo,

$$n_i = \sum_{j=1}^{n} X_{ij}\ .$$

Para calcular $\mathrm{E}[n_i^2]$, expandimos o quadrado e reagrupamos os termos:

$$\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le j\le n \\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \mathrm{E}[X_{ij}^2] + \sum_{1\le j\le n}\ \sum_{\substack{1\le j\le n \\ k\ne j}} \mathrm{E}[X_{ij}X_{ik}]\ , \qquad (8.3)
\end{aligned}$$

onde a última linha se segue por linearidade de expectativa. Avaliamos os dois somatórios separadamente. A variável indicadora aleatória X_{ij} é 1 com probabilidade $1/n$ e 0 em caso contrário e, portanto,

$$\begin{aligned}
\mathrm{E}[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\
&= \frac{1}{n}\ .
\end{aligned}$$

Quando $k \ne j$, as variáveis X_{ij} e X_{ik} são independentes e, por conseguinte,

$$\begin{aligned}
\mathrm{E}[X_{ij}X_{ik}] &= \mathrm{E}[X_{ij}]\ \mathrm{E}[X_{ik}] \\
&= \frac{1}{n} \cdot \frac{1}{n} \\
&= \frac{1}{n^2}\ .
\end{aligned}$$

Substituindo esses dois valores esperados na equação (8.3), obtemos

$$\begin{aligned}
\mathrm{E}[X_{ij}^2] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1\le j\le n}\ \sum_{\substack{1\le j\le n \\ k\ne j}} \frac{1}{n^2} \\
&= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2}
\end{aligned}$$

$$= 1 + \frac{n-1}{n}$$

$$= 2 - \frac{1}{n},$$

o que prova a equação (8.2).

Usando esse valor esperado na equação (8.1), concluímos que o tempo esperado para bucket sort é $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$. Desse modo, o algoritmo de bucket sort inteiro funciona no tempo esperado linear.

Mesmo que a entrada não seja obtida a partir de uma distribuição uniforme, bucket sort ainda pode ser executada em tempo linear. Como a entrada tem a propriedade de que a soma dos quadrados dos tamanhos de baldes é linear no número total de elementos, a equação (8.1) nos diz que bucket sort funcionará em tempo linear.

Exercícios

8.4-1

Usando a Figura 8.4 como modelo, ilustre a operação de BUCKET-SORT no arranjo $A = \langle 0{,}79, 0{,}13, 0{,}16, 0{,}64, 0{,}39, 0{,}20, 0{,}89, 0{,}53, 0{,}71, 0{,}42 \rangle$.

8.4-2

Qual é o tempo de execução do pior caso para o algoritmo de bucket sort? Que alteração simples no algoritmo preserva seu tempo de execução esperado linear e torna seu tempo de execução no pior caso igual a $O(n \lg n)$?

8.4-3

Seja X uma variável aleatória que é igual ao número de caras em dois lançamentos de uma moeda comum. Qual é $E[X^2]$? Qual é $E^2[X]$?

8.4-4 ★

Temos n pontos no círculo unitário, $p_i = (x_i, y_i)$, tais que $0 < x_i^2 + y_i^2 \le 1$ para $i = 1, 2, ..., n$. Suponha que os pontos estejam distribuídos uniformemente; ou seja, a probabilidade de se encontrar um ponto em qualquer região do círculo é proporcional à área dessa região. Projete um algoritmo de tempo esperado $\Theta(n)$ para ordenar os n pontos por suas distâncias $d_i = \sqrt{x_i^2 + y_i^2}$ a partir da origem. (*Sugestão*: Projete os tamanhos de baldes em BUCKET-SORT para refletir a distribuição uniforme dos pontos no círculo unitário.)

8.4-5 ★

Uma ***função de distribuição de probabilidades*** $P(x)$ para uma variável aleatória X é definida por $P(x) = \Pr\{X \le x\}$. Suponha que uma lista de n variáveis aleatórias $X_1, X_2, ..., X_n$ seja obtida a partir de uma função de distribuição de probabilidades contínuas P que possa ser calculada no tempo $O(1)$. Mostre como ordenar esses números em tempo esperado linear.

Problemas

8-1 Limites inferiores do caso médio na ordenação por comparação

Neste problema, provamos um limite inferior $\Omega(n \lg n)$ sobre o tempo de execução esperado de qualquer ordenação por comparação determinística ou aleatória sobre n elementos de entrada distintos. Começamos examinando uma ordenação por comparação determinística A com árvore de decisão T_A. Supomos que toda permutação de entradas de A é igualmente provável.

a. Suponha que cada folha de T_A seja identificada com a probabilidade de ser alcançada dada uma entrada aleatória. Prove que exatamente $n!$ folhas são identificadas com $1/n!$ e que as restantes são identificadas com 0.

b. Seja $D(T)$ um valor que denota o comprimento do caminho externo de uma árvore de decisão T; isto é, $D(T)$ é a soma das profundidades de todas as folhas de T. Seja T uma árvore de decisão com $k > 1$ folhas, e sejam RT e LT as subárvores direita e esquerda de T. Mostre que $D(T) = D(LT) + D(RT) + k$.

c. Seja $d(k)$ o valor mínimo de $D(T)$ sobre todas as árvores de decisão T com $k > 1$ folhas. Mostre que $d(k) = \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$. (*Sugestão*: Considere uma árvore de decisão T com k folhas que alcance o mínimo. Seja i_0 o número de folhas em LT e $k - i_0$ o número de folhas em RT.)

d. Prove que, para um dado valor de $k > 1$ e i no intervalo $1 \le i \le k - 1$, a função $i \lg i + (k - i) \lg(k - i)$ é minimizada em $i = k/2$. Conclua que $d(k) = \Omega(k \lg k)$.

e. Prove que $D(T_A) = \Omega(n! \lg(n!))$ e conclua que o tempo esperado para ordenar n elementos é $\Omega(n \lg n)$.

Agora, considere uma ordenação por comparação *aleatória* B. Podemos estender o modelo de árvore de decisão para tratar a aleatoriedade, incorporando dois tipos de nós: os nós de comparação comum e os nós "aleatórios". Um nó aleatório modela uma opção aleatória da forma RANDOM $(1, r)$ feita pelo algoritmo B; o nó tem r filhos, cada um dos quais tem igual probabilidade de ser escolhido durante uma execução do algoritmo.

f. Mostre que, para qualquer ordenação por comparação aleatória B, existe uma ordenação por comparação determinística A que não faz mais comparações sobre a média que B.

8-2 Ordenação local em tempo linear

Vamos supor que temos um arranjo de n registros de dados para ordenar e que a chave de cada registro tem o valor 0 ou 1. Um algoritmo para ordenar tal conjunto de registros poderia ter algum subconjunto das três características desejáveis a seguir:

1. O algoritmo é executado no tempo $O(n)$.
2. O algoritmo é estável.
3. O algoritmo ordena localmente, sem utilizar mais que uma quantidade constante de espaço de armazenamento além do arranjo original.

a. Dê um algoritmo que satisfaça aos critérios 1 e 2 anteriores.

b. Dê um algoritmo que satisfaça aos critérios 1 e 3 anteriores.

c. Dê um algoritmo que satisfaça aos critérios 2 e 3 anteriores.

d. Algum dos seus algoritmos de ordenação das partes (a)-(c) pode ser usado para ordenar n registros com chaves de b bits usando radix sort no tempo $O(bn)$? Explique como ou por que não.

e. Suponha que os n registros tenham chaves no intervalo de 1 a k. Mostre como modificar a ordenação por contagem de tal forma que os registros possam ser ordenados localmente no tempo $O(n + k)$. Você pode usar o espaço de armazenamento $O(k)$ fora do arranjo de entrada. Seu algoritmo é estável? (*Sugestão*: Como você faria isso para $k = 3$?)

8-3 Ordenação de itens de comprimento variável

a. Você tem um arranjo de inteiros, no qual diferentes inteiros podem ter números de dígitos distintos, mas o número total de dígitos sobre *todos* os inteiros no arranjo é n. Mostre como ordenar o arranjo no tempo $O(n)$.

b. Você tem um arranjo de cadeias, no qual diferentes cadeias podem ter números de caracteres distintos, mas o número total de caracteres em todas as cadeias é n. Mostre como ordenar as cadeias no tempo $O(n)$.

(Observe que a ordem desejada aqui é a ordem alfabética padrão; por exemplo, `a < ab < b`.)

8-4 *Jarros de água*
Vamos supor que você tem n jarros de água vermelhos e n jarros azuis, todos de diferentes formas e tamanhos. Todos os jarros vermelhos contêm quantidades diferentes de água, como também os jarros azuis. Além disso, para todo jarro vermelho, existe um jarro azul que contém a mesma quantidade de água e vice-versa.

Sua tarefa é encontrar um agrupamento dos jarros em pares de jarros vermelhos e azuis que contêm a mesma quantidade de água. Para isso, você pode executar a seguinte operação: escolher um par de jarros no qual um é vermelho e um é azul, encher o jarro vermelho com água, e depois despejar a água no jarro azul. Essa operação lhe informará se o jarro vermelho ou o jarro azul pode conter mais água, ou se eles têm o mesmo volume. Suponha que tal comparação demore uma unidade de tempo. Seu objetivo é encontrar um algoritmo que faça um número mínimo de comparações para determinar o agrupamento. Lembre-se de que você não pode comparar diretamente dois jarros vermelhos ou dois jarros azuis.

a. Descreva um algoritmo determinístico que use $\Theta(n^2)$ comparações para agrupar os jarros em pares.

b. Prove um limite inferior $\Omega(n \lg n)$ para o número de comparações que um algoritmo que resolve esse problema deve efetuar.

c. Dê um algoritmo aleatório cujo número esperado de comparações seja $O(n \lg n)$ e prove que esse limite é correto. Qual é o número de comparações no pior caso do seu algoritmo?

8-5 *Ordenação por média*
Suponha que, em vez de ordenar um arranjo, simplesmente exigimos que os elementos aumentem na média. De modo mais preciso, chamamos um arranjo de n elementos A de ***k-ordenado*** se, para todo $i = 1, 2, \ldots, n - k$, é válida a desigualdade a seguir:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

a. O que significa um arranjo ser 1-ordenado?

b. Forneça uma permutação dos números 1, 2, ..., 10 que seja 2-ordenada, mas não ordenada.

c. Prove que um arranjo de n elementos é k-ordenado se e somente se $A[i] \leq A[i + k]$ para todo $i = 1, 2, \ldots, n - k$.

d. Forneça um algoritmo que faça a k-ordenação de um arranjo de n elementos no tempo $O(n \lg(n/k))$.

Também podemos mostrar um limite inferior sobre o tempo para produzir um arranjo k-ordenado, quando k é uma constante.

e. Mostre que um arranjo k-ordenado de comprimento n pode ser ordenado no tempo $O(n \lg k)$. (*Sugestão*: Use a solução do Exercício 6.5-8.)

f. Mostre que, quando k é uma constante, é necessário o tempo $\Omega(n \lg n)$ para fazer a k-ordenação de um arranjo de n elementos. (*Sugestão*: Use a solução para a parte anterior, juntamente com o limite inferior sobre ordenações por comparação.)

8-6 *Limite inferior sobre a intercalação de listas ordenadas*
O problema de intercalar duas listas ordenadas surge com freqüência. Ele é usado como uma sub-rotina de MERGE-SORT, e o procedimento para intercalar duas listas ordenadas é dado como MERGE na Seção 2.3.1. Neste problema, mostraremos que existe um limite inferior $2n - 1$ sobre o número de comparações no pior caso exigidas para intercalar duas listas ordenadas, cada uma contendo n itens.

Primeiro, mostraremos um limite inferior de $2n - o(n)$ comparações, usando uma árvore de decisão.

a. Mostre que, dados $2n$ números, existem $\binom{2n}{n}$ maneiras possíveis de dividi-los em duas listas ordenadas, cada uma com n números.

b. Usando uma árvore de decisão, mostre que qualquer algoritmo que intercala corretamente duas listas ordenadas utiliza pelo menos $2n - o(n)$ comparações.

Agora, mostraremos um limite $2n - 1$, ligeiramente mais restrito.

c. Mostre que, se dois elementos são consecutivos na seqüência ordenada e vêm de listas opostas, então eles devem ser comparados.

d. Use sua resposta à parte anterior para mostrar um limite inferior de $2n - 1$ comparações para intercalar duas listas ordenadas.

Notas do capítulo

O modelo de árvore de decisão para o estudo de ordenações por comparação foi introduzido por Ford e Johnson [94]. O tratamento abrangente de Knuth sobre a ordenação [185] cobre muitas variações sobre o problema da ordenação, inclusive o limite inferior teórico de informações sobre a complexidade da ordenação fornecida aqui. Os limites inferiores para ordenação com o uso de generalizações do modelo de árvore de decisão foram estudados amplamente por Ben-Or [36].

Knuth credita a H. H. Seward a criação da ordenação por contagem em 1954, e também a idéia de combinar a ordenação por contagem com a radix sort. A radix sort começando pelo dígito menos significativo parece ser um algoritmo popular amplamente utilizado por operadores de máquinas mecânicas de ordenação de cartões. De acordo com Knuth, a primeira referência ao método publicada é um documento de 1929 escrito por L. J. Comrie que descreve o equipamento de perfuração de cartões. A bucket sort esteve em uso desde 1956, quando a idéia básica foi proposta por E. J. Isaac e R. C. Singleton.

Munro e Raman [229] fornecem um algoritmo de ordenação estável que executa $O(n^{1+\varepsilon})$ comparações no pior caso, onde $0 < \varepsilon \leq 1$ é qualquer constante fixa. Embora qualquer dos algoritmos de tempo $O(n \lg n)$ efetue um número menor de comparações, o algoritmo de Munro e Raman move os dados apenas $O(n)$ vezes e opera localmente.

O caso de ordenar n inteiros de b bits no tempo $o(n \lg n)$ foi considerado por muitos pesquisadores. Vários resultados positivos foram obtidos, cada um sob hipóteses um pouco diferentes a respeito do modelo de computação e das restrições impostas sobre o algoritmo. Todos os resultados pressupõem que a memória do computador está dividida em palavras endereçáveis de b bits. Fredman e Willard [99] introduziram a estrutura de dados de árvores de fusão e a empregaram para ordenar n inteiros no tempo $O(n \lg n / \lg \lg n)$. Esse limite foi aperfeiçoado mais tarde para o tempo $O(n\sqrt{\lg n})$ por Andersson [16]. Esses algoritmos exigem o uso de multiplicação e de várias constantes pré-calculadas. Andersson, Hagerup, Nilsson e Raman [17] mostraram como ordenar n inteiros no tempo $O(n \lg \lg n)$ sem usar multiplicação, mas seu método exige espaço de armazenamento que pode ser ilimitado em termos de n. Usando-se o hash multiplicativo, é possível reduzir o espaço de armazenamento necessário para $O(n)$, mas o limite $O(n \lg \lg n)$ do pior caso sobre o tempo de execução se torna um limite de tempo esperado. Generalizando as árvores de pesquisa exponencial de Andersson [16], Thorup [297] forneceu um algoritmo de ordenação de tempo $O(n(\lg \lg n)^2)$ que não usa multiplicação ou aleatoriedade, e que utiliza espaço linear. Combinando essas técnicas com algumas idéias novas, Han [137] melhorou o limite para ordenação até o tempo $O(n \lg \lg n \lg \lg \lg n)$. Embora esses algoritmos sejam inovações teóricas importantes, todos eles são bastante complicados e neste momento parece improvável que venham a competir na prática com algoritmos de ordenação existentes.

Capítulo 9

Medianas e estatísticas de ordem

A i-ésima ***estatística de ordem*** de um conjunto de n elementos é o i-ésimo menor elemento. Por exemplo, o ***mínimo*** de um conjunto de elementos é a primeira estatística de ordem ($i = 1$), e o ***máximo*** é a n-ésima estatística de ordem ($i = n$). Informalmente, uma ***mediana*** é o "ponto médio" do conjunto. Quando n é ímpar, a mediana é única, ocorrendo em $i = (n + 1)/2$. Quando n é par, existem duas medianas, ocorrendo em $i = n/2$ e $i = n/2 + 1$. Desse modo, independentemente da paridade de n, as medianas ocorrem em $i = \lfloor (n + 1)/2 \rfloor$ (a ***mediana inferior***) e $i = \lceil (n + 1)/2 \rceil$ (a ***mediana superior***). Porém, por simplicidade neste texto, usaremos de forma coerente a expressão "a mediana" para nos referirmos à mediana inferior.

Este capítulo focaliza o problema de selecionar a i-ésima estatística de ordem de um conjunto de n números distintos. Supomos por conveniência que o conjunto contém números distintos, embora virtualmente tudo que fizermos se estenda à situação na qual um conjunto contém valores repetidos. O ***problema de seleção*** pode ser especificado formalmente do seguinte modo:

Entrada: Um conjunto A de n números (distintos) e um número i, com $1 \leq i \leq n$.

Saída: O elemento $x \in A$ que é maior que exatamente $i - 1$ outros elementos de A.

O problema de seleção pode ser resolvido no tempo $O(n \lg n)$, pois podemos ordenar os números usando heapsort ou ordenação por intercalação, e depois simplesmente indexar o i-ésimo elemento no arranjo de saída. Contudo, existem algoritmos mais rápidos.

Na Seção 9.1, examinamos o problema de selecionar o mínimo e o máximo de um conjunto de elementos. Mais interessante é o problema de seleção geral, que é investigado nas duas seções subseqüentes. A Seção 9.2 analisa um algoritmo prático que alcança um limite $O(n)$ sobre o tempo de execução no caso médio. A Seção 9.3 contém um algoritmo de maior interesse teórico, que alcança o tempo de execução $O(n)$ no pior caso.

9.1 Mínimo e máximo

Quantas comparações são necessárias para determinar o mínimo de um conjunto de n elementos? Podemos obter facilmente um limite superior de $n - 1$ comparações: examine cada elemento do conjunto isoladamente e mantenha o controle do menor elemento visto até então. No procedimento a seguir, vamos supor que o conjunto reside no arranjo A, onde *comprimento*$[A] = n$.

```
MINIMUM(A)
1 min ← A[1]
2 for i ← 2 to comprimento[A]
3    do if min > A[i]
4       then min ← A[i]
5 return min
```

A localização do máximo também pode, é claro, ser realizada com $n - 1$ comparações.

Isso é o melhor que podemos fazer? Sim, desde que possamos obter um limite inferior de $n - 1$ comparações para o problema de determinar o mínimo. Imagine qualquer algoritmo que determina o mínimo como um torneio entre os elementos. Cada comparação é uma partida no torneio, na qual o menor dos dois elementos vence. A observação chave é que todo elemento, exceto o vencedor, deve perder pelo menos uma partida. Conseqüentemente, $n - 1$ comparações são necessárias para determinar o mínimo, e o algoritmo MINIMUM é ótimo com relação ao número de comparações executadas.

Mínimo e máximo simultâneos

Em algumas aplicações, devemos localizar tanto o mínimo quanto o máximo de um conjunto de n elementos. Por exemplo, um programa gráfico talvez precise ajustar a escala de um conjunto de (x, y) dados para se encaixar em uma tela de exibição retangular ou em outro dispositivo de saída gráfica. Para fazer isso, o programa deve primeiro determinar o mínimo e o máximo de cada coordenada.

Não é difícil criar um algoritmo que possa encontrar tanto o mínimo quanto o máximo de n elementos usando $\Theta(n)$ comparações, que é o número assintoticamente ótimo de comparações. Simplesmente localize o mínimo e o máximo independentemente, usando $n - 1$ comparações para cada um deles, o que fornece um total de $2n - 2$ comparações.

De fato, no máximo $3\lfloor n/2 \rfloor$ comparações são suficientes para se encontrar tanto o mínimo quanto o máximo. A estratégia é manter os elementos mínimo e máximo vistos até agora. Porém, em lugar de processar cada elemento da entrada comparando-o com o mínimo e o máximo atuais, a um custo de duas comparações por elemento, processamos os elementos aos pares. Primeiro, comparamos pares de elementos da entrada *uns com os outros*, depois comparamos o menor com o mínimo atual e o maior com o máximo atual, a um custo de três comparações para cada dois elementos.

A definição de valores iniciais para o mínimo e o máximo atuais depende do fato de n ser ímpar ou par. Se n é ímpar, definimos tanto o mínimo quanto o máximo com o valor do primeiro elemento e, em seguida, processamos os elementos restantes aos pares. Se n é par, executamos uma comparação sobre os dois primeiros elementos para determinar os valores iniciais do mínimo e do máximo, e depois processamos os elementos restantes aos pares, como no caso de n ímpar.

Vamos analisar o número total de comparações. Se n é ímpar, executamos $3\lfloor n/2 \rfloor$ comparações. Se n é par, executamos uma comparação inicial seguida por $3(n - 2)/2$ comparações, dando um total de $3n/2 - 2$. Desse modo, em qualquer caso, o número total de comparações é no máximo $3\lfloor n/2 \rfloor$.

Exercícios

9.1-1

Mostre que o segundo menor entre n elementos pode ser encontrado com $n + \lceil \lg n \rceil - 2$ comparações no pior caso. (*Sugestão*: Encontre também o menor elemento.)

9.1-2 ★
Mostre que $\lceil 3n/2 \rceil - 2$ comparações são necessárias no pior caso para localizar tanto o máximo quanto o mínimo entre n números. (*Sugestão*: Considere quantos números são potencialmente o máximo ou o mínimo, e investigue como uma comparação afeta essas contagens.)

9.2 Seleção em tempo esperado linear

O problema de seleção geral parece mais difícil que o problema simples de se achar um mínimo, ainda que surpreendentemente o tempo de execução assintótico para ambos os problemas seja o mesmo: $\Theta(n)$. Nesta seção, apresentamos um algoritmo de dividir e conquistar para o problema de seleção. O algoritmo RANDOMIZED-SELECT é modelado sobre o algoritmo quicksort do Capítulo 7. Como no quicksort, a idéia é particionar o arranjo de entrada recursivamente. Porém, diferente de quicksort, que processa recursivamente ambos os lados da partição, RANDOMIZED-SELECT só funciona sobre um lado da partição. Essa diferença fica evidente na análise: enquanto quicksort tem um tempo de execução esperado $\Theta(n \lg n)$, o tempo esperado de RANDOMIZED-SELECT é $\Theta(n)$.

RANDOMIZED-SELECT utiliza o procedimento RANDOMIZED-PARTITION introduzido na Seção 7.3. Desse modo, como RANDOMIZED-QUICKSORT, ele é um algoritmo aleatório, pois seu comportamento é determinado em parte pela saída de um gerador de números aleatórios. O código a seguir para RANDOMIZED-SELECT retorna o i-ésimo menor elemento do arranjo $A[p \mathinner{..} r]$.

```
RANDOMIZED-SELECT(A, p, r, i)
1 if p = r
2    then return A[p]
3 q ← RANDOMIZED-PARTITION(A, p, r)
4 k ← q - p + 1
5 if i = k          ▷ O valor pivô é a resposta
6    then return A[q]
7 elseif i < k
8    then return RANDOMIZED-SELECT(A, p, q - 1, i)
9 else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

Após RANDOMIZED-PARTITION ser executado na linha 3 do algoritmo, o arranjo $A[p \mathinner{..} r]$ é particionado em dois subarranjos (possivelmente vazios) $A[p \mathinner{..} q-1]$ e $A[q+1 \mathinner{..} r]$ tais que cada elemento de $A[p \mathinner{..} q-1]$ é menor que ou igual a $A[q]$ que, por sua vez, é menor que cada elemento de $A[q+1 \mathinner{..} r]$. Como em quicksort, vamos nos referir a $A[q]$ como o elemento ***pivô***. A linha 4 de RANDOMIZED-SELECT calcula o número k de elementos no subarranjo $A[p \mathinner{..} q]$, ou seja, o número de elementos no lado baixo da partição, mais uma unidade para o elemento pivô. A linha 5 verifica então se $A[q]$ é o i-ésimo menor elemento. Se for, então $A[q]$ é retornado. Caso contrário, o algoritmo determina em qual dos dois subarranjos $A[p \mathinner{..} q-1]$ e $A[q+1 \mathinner{..} r]$ o i-ésimo menor elemento se encontra. Se $i < k$, então o elemento desejado está no lado baixo da partição, e é recursivamente selecionado do subarranjo na linha 8. Porém, se $i > k$, o elemento desejado reside no lado alto da partição. Como já conhecemos k valores que são menores que o i-ésimo menor elemento de $A[p \mathinner{..} r]$ – isto é, os elementos de $A[p \mathinner{..} q]$ – o elemento desejado é o $(i-k)$-ésimo menor elemento de $A[q+1 \mathinner{..} r]$, encontrado recursivamente na linha 9. O código parece permitir chamadas recursivas a subarranjos com 0 elementos, mas o Exercício 9.2-1 lhe pede para mostrar que essa situação não pode acontecer.

O tempo de execução do pior caso para RANDOMIZED-SELECT é $\Theta(n^2)$, mesmo para se encontrar o mínimo, porque poderíamos estar extremamente sem sorte e sempre efetuar a partição em torno do maior elemento restante, e o particionamento levará o tempo $\Theta(n)$. Entretanto, o algoritmo funciona bem no caso médio e, porque ele é aleatório, nenhuma entrada específica surge do comportamento no pior caso.

O tempo exigido por RANDOMIZED-SELECT em um arranjo de entrada $A[p \,..\, r]$ de n elementos é uma variável aleatória que denotamos por $T(n)$, e obtemos um limite superior sobre $E[T(n)]$ como a seguir. O procedimento RANDOMIZED-PARTITION tem igual probabilidade de retornar qualquer elemento como pivô. Assim, para cada k tal que $1 \le k \le n$, o subarranjo $A[p \,..\, q]$ tem k elementos (todos menores que ou iguais ao pivô) com probabilidade $1/n$. Para $k = 1, 2, \ldots, n$, definimos variáveis indicadoras aleatórias X_k, nas quais

$$X_k = I\,\{\text{o subarranjo } A[p \,..\, q] \text{ tem exatamente } k \text{ elementos}\}\ ,$$

e assim temos

$$E[X_k] = 1/n\ . \qquad (9.1)$$

Quando chamamos RANDOMIZED-SELECT e escolhemos $A[q]$ como o elemento pivô, não sabemos *a priori* se terminaremos imediatamente com a resposta correta, faremos a recursão no subarranjo $A[p \,..\, q-1]$ ou faremos a recursão no subarranjo $A[q+1 \,..\, r]$. Essa decisão depende de onde o i-ésimo menor elemento ficará em relação a $A[q]$. Supondo que $T(n)$ seja monotonicamente crescente, podemos limitar o tempo necessário para a chamada recursiva pelo tempo necessário para a chamada recursiva sobre a maior entrada possível. Em outras palavras, supomos, para obter um limite superior, que o i-ésimo elemento está sempre no lado da partição com o maior número de elementos. Para uma dada chamada de RANDOMIZED-SELECT, a variável indicadora aleatória X_k tem o valor 1 para exatamente um valor de k, e é 0 para todos os outros k. Quando $X_k = 1$, os dois subarranjos sobre os quais podemos fazer a recursão têm tamanhos $k-1$ e $n-k$. Conseqüentemente, temos a recorrência

$$\begin{aligned} T(n) &\le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1,\ n-k)) + O(n)) \\ &= \sum_{k=1}^{n} (X_k \cdot T(\max(k-1,\ n-k)) + O(n))\ . \end{aligned}$$

Tomando valores esperados, temos

$$\begin{aligned} &E[T(n)] \\ &\quad \le E\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1,\ n-k)) + O(n)\right] \\ &\quad = \sum_{k=1}^{n} E[X_k \cdot T(\max(k-1,\ n-k))] + O(n) && \text{(por linearidade de expectativa)} \\ &\quad = \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k-1,\ n-k))] + O(n) && \text{(pela equação (C.23))} \\ &\quad = \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k-1,\ n-k))] + O(n) && \text{(pela equação (9.1))}\ . \end{aligned}$$

Para aplicar a equação (C.23), dependemos do fato de X_k e $T(\max(k-1, n-k))$ serem variáveis aleatórias independentes. O Exercício 9.2-2 lhe pede para justificar essa afirmação.

Vamos considerar a expressão $\max(k-1, n-k)$. Temos

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{se } k > \lceil n/2 \rceil, \\ n-k & \text{se } k \le \lceil n/2 \rceil. \end{cases}$$

Se n é par, cada termo de $T(\lceil n/2 \rceil)$ até $T(n-1)$ aparece exatamente duas vezes no somatório e, se n é ímpar, todos esses termos aparecem duas vezes e o termo $T(\lfloor n/2 \rfloor)$ aparece uma vez. Desse modo, temos

$$\mathrm{E}[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} \mathrm{E}[T(k)] + O(n).$$

Resolvemos a recorrência por substituição. Suponha que $T(n) \le cn$ para alguma constante c que satisfaça às condições iniciais da recorrência. Supomos que $T(n) = O(1)$ para n menor que alguma constante; escolheremos essa constante mais adiante. Também escolheremos uma constante a tal que a função descrita pelo termo $O(n)$ anterior (que descreve o componente não recursivo do tempo de execução do algoritmo) seja limitado acima por an para todo $n > 0$. Usando essa hipótese indutiva, temos

$$\begin{aligned}
\mathrm{E}[T(n)] &\le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\
&= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \\
&= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\
&= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{(n^2/4 - 3n/2 + 2)}{2} \right) + an \\
&= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\
&= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \\
&\le \frac{3cn}{4} + \frac{c}{2} + an. \\
&= cn\left(\frac{cn}{4} - \frac{c}{2} - an \right).
\end{aligned}$$

Para completar a prova, precisamos mostrar que, para n suficientemente grande, essa última expressão é no máximo cn ou, de modo equivalente, que $cn/4 - c/2 - an \ge 0$. Se adicionarmos $c/2$ a ambos os lados e fatorarmos n, obteremos $n(c/4 - a) \ge c/2$. Desde que a constante c seja escolhida de modo que $c/4 - a > 0$, isto é, $c > 4a$, poderemos dividir ambos os lados por $c/4 - a$, obtendo

$$n \ge \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}.$$

Desse modo, se considerarmos que $T(n) = O(1)$ para $n < 2c/(c - 4a)$, teremos $T(n) = O(n)$. Concluímos que qualquer estatística de ordem, e em particular a mediana, pode ser determinada em média no tempo linear.

Exercícios

9.2-1
Mostre que, em RANDOMIZED-SELECT, não é feita nenhuma chamada recursiva para um arranjo de comprimento 0.

9.2-2
Demonstre que a variável indicadora aleatória X_k e o valor $T(\max(k - 1, n - k))$ são independentes.

9.2-3
Escreva uma versão iterativa de RANDOMIZED-SELECT.

9.2-4
Suponhamos que RANDOMIZED-SELECT seja utilizado para selecionar o elemento mínimo do arranjo $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1\rangle$. Descreva uma seqüência de partições que resulte em um desempenho do pior caso de RANDOMIZED-SELECT.

9.3 Seleção em tempo linear no pior caso

Agora, vamos examinar um algoritmo de seleção cujo tempo de execução é $O(n)$ no pior caso. Como RANDOMIZED-SELECT, o algoritmo SELECT localiza o elemento desejado particionando recursivamente o arranjo de entrada. Porém, a idéia que rege o algoritmo é *garantir* uma boa divisão quando o arranjo é particionado. SELECT utiliza o algoritmo de particionamento determinístico PARTITION de quicksort (ver Seção 7.1), modificado para tomar o elemento a particionar como um parâmetro de entrada.

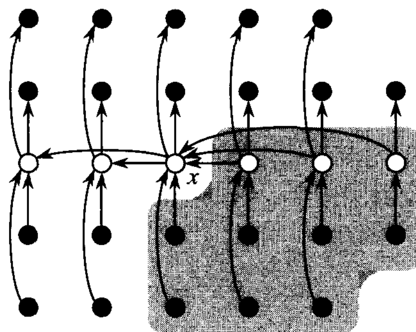


FIGURA 9.1 Análise do algoritmo SELECT. Os n elementos são representados por pequenos círculos, e cada grupo ocupa uma coluna. As medianas dos grupos são brancas, e a mediana de medianas está identificada como x. (Quando encontramos a mediana de um número par de elementos, usamos a mediana inferior.) São traçadas setas de elementos maiores para elementos menores e, a partir disso, podemos ver que 3 elementos em cada grupo de 5 elementos à direita de x são maiores que x, e 3 em cada grupo de 5 elementos à esquerda de x são menores que x. Os elementos maiores que x são mostrados sobre um plano de fundo sombreado

O algoritmo SELECT determina o i-ésimo menor elemento de um arranjo de entrada de $n > 1$ elementos, executando as etapas a seguir. (Se $n = 1$, então SELECT simplesmente retorna seu único valor de entrada como o i-ésimo menor.)

1. Dividir os n elementos do arranjo de entrada em $\lfloor n/5 \rfloor$ grupos de 5 elementos cada e no máximo um grupo formado pelos n mod 5 elementos restantes.
2. Encontrar a mediana de cada um dos $\lceil n/5 \rceil$ grupos, primeiro através da ordenação por inserção dos elementos de cada grupo (dos quais existem 5 no máximo), e depois escolhendo a mediana da lista ordenada de elementos de grupos.
3. Usar SELECT recursivamente para encontrar a mediana x das $\lceil n/5 \rceil$ medianas localizadas na Etapa 2. (Se existe um número par de medianas, então, por nossa convenção, x é a mediana inferior.)
4. Particionar o arranjo de entrada em torno da mediana de medianas x, usando uma versão modificada de PARTITION. Seja k uma unidade maior que o número de elementos no lado baixo da partição, de forma que x seja o k-ésimo menor elemento e existam $n - k$ elementos no lado alto da partição.
5. Se $i = k$, então retornar x. Caso contrário, usar SELECT recursivamente para encontrar o i-ésimo menor elemento no lado baixo se $i \leq k$, ou então o $(i - k)$-ésimo menor elemento no lado alto, se $i > k$.

Para analisar o tempo de execução de SELECT, primeiro determinamos um limite inferior sobre o número de elementos que são maiores que o elemento de particionamento x. A Figura 9.1 é útil na visualização dessa contabilidade. Pelo menos metade das medianas encontradas na Etapa 2 é maior que[1] a mediana de medianas x. Portanto, pelo menos metade dos $\lceil n/5 \rceil$ grupos contribui com 3 elementos maiores que x, exceto pelo único grupo que tem menos de 5 elementos se 5 não dividir n exatamente, e pelo único grupo contendo o próprio x.

Descontando esses dois grupos, segue-se que o número de elementos maiores que x é pelo menos

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6 .$$

De modo semelhante, o número de elementos menores que x é no mínimo $3n/10 - 6$. Desse modo, no pior caso, SELECT é chamado recursivamente sobre no máximo $7n/10 + 6$ elementos na Etapa 5.

Agora, podemos desenvolver uma recorrência para o tempo de execução do pior caso $T(n)$ do algoritmo SELECT. As Etapas 1, 2 e 4 demoram o tempo $O(n)$. (A Etapa 2 consiste em $O(n)$ chamadas de ordenação por inserção sobre conjuntos de tamanho $O(1)$.) A Etapa 3 demora o tempo $T(\lceil n/5 \rceil)$, e a Etapa 5 demora no máximo o tempo $T(7n/10 + 6)$, supondo-se que T seja monotonicamente crescente. Adotamos a hipótese, que a princípio parece sem motivo, de que qualquer entrada de 140 ou menos elementos exige o tempo $O(1)$; a origem da constante mágica 140 ficará clara em breve. Portanto, podemos obter a recorrência

$$T(n) \leq T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 140 , \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{se } n > 140 . \end{cases}$$

Mostramos que o tempo de execução é linear por substituição. Mais especificamente, mostraremos que $T(n) \leq cn$ para alguma constante c grande o bastante e para todo $n > 0$. Começamos supondo que $T(n) \leq cn$ para alguma constante c grande o bastante e para todo $n \leq 140$; essa hipótese se mantém válida se c é suficientemente grande. Também escolhemos uma constante a tal que a função descrita pelo termo $O(n)$ anterior (que descreve o componente não recursivo do tempo de execução do algoritmo) é limitado acima por an para todo $n > 0$. Substituindo essa hipótese indutiva no lado direito da recorrência, obtemos

[1] Em conseqüência de nossa hipótese de que os números são distintos, podemos dizer "maior que" e "menor que" sem nos preocuparmos com a igualdade.

$$\begin{aligned} T(n) &\le c\lceil n/5\rceil + c(7n/10 + 6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an)\ , \end{aligned}$$

que é no máximo cn se

$$-cn/10 + 7c + an \le 0\ . \qquad (9.2)$$

A desigualdade (9.2) é equivalente à desigualdade $c \ge 10a(n/(n-70))$ quando $n > 70$. Considerando que supomos $n \ge 140$, temos $n/(n-70) \le 2$, e assim a escolha de $c \ge 20a$ satisfará à desigualdade (9.2). (Observe que não existe nada de especial sobre a constante 140; poderíamos substituí-la por qualquer inteiro estritamente maior que 70 e depois escolher c de acordo.) Então, o tempo de execução do pior caso de SELECT é linear.

Como em uma ordenação por comparação (ver Seção 8.1), SELECT e RANDOMIZED-SELECT descobrem informações sobre a ordem relativa de elementos apenas pela comparação de elementos. Vimos no Capítulo 8 que a ordenação exige o tempo $\Omega(n \lg n)$ no modelo de comparação, mesmo na média (ver Problema 8-1). Os algoritmos de ordenação de tempo linear do Capítulo 8 fazem suposições sobre a entrada. Em contraste, os algoritmos de seleção de tempo linear deste capítulo não exigem quaisquer hipóteses sobre a entrada. Eles não estão sujeitos ao limite inferior $\Omega(n \lg n)$ porque conseguem resolver o problema de seleção sem ordenação.

Desse modo, o tempo de execução é linear porque esses algoritmos não efetuam a ordenação; o comportamento de tempo linear não é um resultado de hipóteses sobre a entrada, como foi o caso para os algoritmos de ordenação do Capítulo 8. A ordenação requer o tempo $\Omega(n \lg n)$ no modelo de comparação, mesmo na média (ver Problema 8-1), e assim o método de ordenação e indexação apresentado na introdução a este capítulo é assintoticamente ineficiente.

Exercícios

9.3-1

No algoritmo SELECT, os elementos de entrada estão divididos em grupos de 5. O algoritmo funcionará em tempo linear se eles forem divididos em grupos de 7? Demonstre que SELECT não será executado em tempo linear se forem usados grupos de 3 elementos.

9.3-2

Analise SELECT para mostrar que, se $n \ge 140$, pelo menos $\lceil n/4\rceil$ elementos são maiores que a mediana de medianas x e pelo menos $\lceil n/4\rceil$ elementos são menores que x.

9.3-3

Mostre como quicksort pode ser desenvolvido para ser executado no tempo $O(n \lg n)$ no pior caso.

9.3-4 *

Suponha que um algoritmo utilize apenas comparações para encontrar o i-ésimo menor elemento em um conjunto de n elementos. Mostre que ele também pode encontrar os $i - 1$ menores elementos e os $n - i$ maiores elementos sem executar quaisquer comparações adicionais.

9.3-5

Dada uma sub-rotina de "caixa-preta" de mediana de tempo linear no pior caso, forneça um algoritmo simples de tempo linear que resolva o problema de seleção para uma estatística de ordem arbitrária.

9.3-6
Os k-ésimos ***quantis*** de um conjunto de n elementos são as $k - 1$ estatísticas de ordem que dividem o conjunto ordenado em k conjuntos de igual tamanho (até dentro de 1). Forneça um algoritmo de tempo $O(n \lg k)$ para listar os k-ésimos quantis de um conjunto.

9.3-7
Descreva um algoritmo de tempo $O(n)$ que, dados um conjunto S de n números distintos e um inteiro positivo $k \leq n$, determine os k números em S que estão mais próximos da mediana de S.

9.3-8
Sejam $X[1 \,..\, n]$ e $Y[1 \,..\, n]$ dois arranjos, cada um contendo n números já em seqüência ordenada. Forneça um algoritmo de tempo $O(\lg n)$ para localizar a mediana de todos os $2n$ elementos nos arranjos X e Y.

9.3-9
O professor Olavo é consultor de uma empresa petrolífera que está planejando um grande oleoduto de leste para oeste através de um campo petrolífero de n poços. De cada poço, um oleoduto auxiliar deve ser conectado diretamente ao oleoduto principal ao longo de um caminho mais curto (para o norte ou para o sul), como mostra a Figura 9.2. Dadas as coordenadas x e y dos poços, de que modo o professor deve escolher a localização ótima do oleoduto principal (aquela que minimiza o comprimento total dos dutos auxiliares)? Mostre que a localização ótima pode ser determinada em tempo linear.

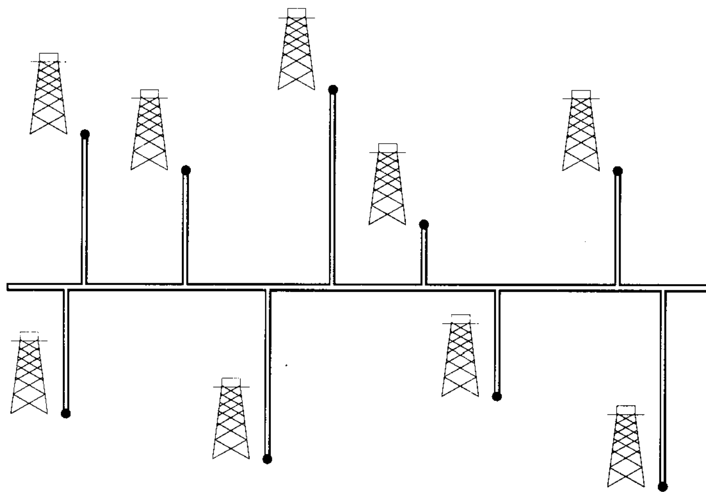


FIGURA 9.2 O professor Olavo precisa determinar a posição do oleoduto no sentido leste-oeste que minimiza o comprimento total dos dutos auxiliares norte-sul

Problemas

9-1 Os i maiores números em seqüência ordenada
Dado um conjunto de n números, queremos encontrar os i maiores em seqüência ordenada, usando um algoritmo baseado em comparação. Descubra o algoritmo que implementa cada um dos métodos a seguir com o melhor tempo de execução assintótico no pior caso e analise os tempos de execução dos algoritmos em termos de n e i.

a. Classifique os números e liste os i maiores.

b. Construa uma fila de prioridade a partir dos números e chame EXTRACT-MAX i vezes.

c. Use um algoritmo de estatística de ordem para localizar o i-ésimo maior número, particionar e ordenar os i maiores números.

9-2 Mediana ponderada

Para n elementos distintos $x_1, x_2, ..., x_n$, com pesos positivos $w_1, w_2, ..., w_n$ tais que $\sum_{i=1}^{n} w_i = 1$, a ***mediana ponderada (inferior)*** é o elemento x_k que satisfaz a

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

e

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2}.$$

a. Mostre que a mediana de $x_1, x_2, ..., x_n$ é a mediana ponderada dos x_i com pesos $w_i = 1/n$ para $i = 1, 2, ..., n$.

b. Mostre como calcular a mediana ponderada de n elementos no tempo $O(n \lg n)$ no pior caso usando ordenação.

c. Mostre como calcular a mediana ponderada no tempo $\Theta(n)$ no pior caso, usando um algoritmo de mediana de tempo linear como SELECT da Seção 9.3.

O ***problema da localização da agência postal*** é definido como a seguir. Temos n pontos $p_1, p_2, ..., p_n$ com pesos associados $w_1, w_2, ..., w_n$. Desejamos encontrar um ponto p (não necessariamente um dos pontos de entrada) que minimize o somatório $\sum_{i=1}^{n} w_i d(p, p_i)$, onde $d(a, b)$ é a distância entre os pontos a e b.

d. Mostre que a mediana ponderada é uma solução melhor para o problema da localização de agência postal unidimensional, no qual os pontos são simplesmente números reais e a distância entre os pontos a e b é $d(a, b) = |a - b|$.

e. Encontre a melhor solução para o problema de localização da agência postal bidimensional, no qual os pontos são pares de coordenadas (x, y) e a distância entre os pontos $a = (x_1, y_1)$ e $b = (x_2, y_2)$ é a ***distância de Manhattan*** dada por $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

9-3 Estatísticas de ordem menores

Mostramos que o número $T(n)$ de comparações no pior caso usadas por SELECT para selecionar a i-ésima estatística de ordem de n números satisfaz a $T(n) = \Theta(n)$, mas a constante oculta pela notação Θ é bastante grande. Quando i é pequena em relação a n, podemos implementar um procedimento diferente que utiliza SELECT como uma sub-rotina, mas que efetua menos comparações no pior caso.

a. Descreva um algoritmo que utilize $U_i(n)$ comparações para encontrar o i-ésimo menor de n elementos, onde

$$U_i(n) = T(n) = \begin{cases} T(n) & \text{se } i \geq n/2 \text{ ,} \\ \lfloor n/2 \rfloor + U_i(\lfloor n/2 \rfloor) + T(2i) & \text{em caso contrário.} \end{cases}$$

(*Sugestão*: Comece com $\lfloor n/2 \rfloor$ comparações de pares disjuntos e efetue a recursão sobre o conjunto que contém o menor elemento de cada par.)

b. Mostre que, se $i < n/2$, então $U_i(n) = n + O(T(2i) \lg(n/i))$.

c. Mostre que, se i é uma constante menor que $n/2$, então $U_i(n) = n + O(\lg n)$.

d. Mostre que, se $i = n/k$ para $k \geq 2$, então $U_i(n) = n + O(T(2n/k) \lg k)$.

Notas do capítulo

O algoritmo de tempo linear no pior caso para localização da mediana foi criado por Blum, Floyd, Pratt, Rivest e Tarjan [43]. A versão de tempo médio mais rápido se deve a Hoare [146]. Floyd e Rivest [92] desenvolveram uma versão de tempo médio melhor que efetua a partição em torno de um elemento selecionado recursivamente a partir de uma amostra pequena dos elementos.

Ainda não se sabe exatamente quantas comparações são necessárias para determinar a mediana. Um limite inferior de $2n$ comparações para a localização de medianas foi dado por Bent e John [38]. Um limite superior de $3n$ foi dado por Schonhage, Paterson e Pippenger [265]. Dor e Zwick [79] fizeram melhorias nesses dois limites; seu limite superior é ligeiramente menor que $2{,}95n$ e o limite inferior é ligeiramente maior que $2n$. Paterson [239] descreve esses resultados juntamente com outro trabalho relacionado.

Parte III

Estruturas de dados

Introdução

Os conjuntos são tão fundamentais para a ciência da computação quanto o são para a matemática. Enquanto os conjuntos matemáticos são invariáveis, os conjuntos manipulados por algoritmos podem crescer, encolher ou sofrer outras mudanças ao longo do tempo. Chamamos tais conjuntos de conjuntos ***dinâmicos***. Os próximos cinco capítulos apresentam algumas técnicas básicas para representar conjuntos dinâmicos finitos e para manipular esses conjuntos em um computador.

Os algoritmos podem exigir vários tipos diferentes de operações a serem executadas sobre conjuntos. Por exemplo, muitos algoritmos precisam apenas da capacidade de inserir elementos em, eliminar elementos de, e testar a pertinência de elementos a um conjunto. Um conjunto dinâmico que admite essas operações é chamado ***dicionário***. Outros algoritmos exigem operações mais complicadas. Por exemplo, filas de prioridade mínima, que foram introduzidas no Capítulo 6 no contexto da estrutura de dados do tipo heap (monte), admitem as operações de inserção de um elemento em e de extração do menor elemento de um conjunto. A melhor maneira de implementar um conjunto dinâmico depende das operações que devem ser admitidas.

Os elementos de um conjunto dinâmico

Em uma implementação típica de um conjunto dinâmico, cada elemento é representado por um objeto cujos campos podem ser examinados e manipulados se tivermos um ponteiro para o objeto. (A Seção 10.3 discute a implementação de objetos e ponteiros em ambientes de programação que não contêm esses objetos e ponteiros como tipos de dados básicos.) Alguns tipos de conjuntos dinâmicos pressupõem que um dos campos do objeto é um campo de ***chave*** de identificação. Se as chaves são todas diferentes, podemos imaginar o conjunto dinâmico como um conjunto de valores de chaves. O objeto pode conter ***dados satélite***, que são transportados em campos de outro objeto, mas não são utilizados de outro modo pela implementação do conjunto. Ele também pode ter campos que são manipulados pelas operações de conjuntos; esses campos podem conter dados ou ponteiros para outros objetos no conjunto.

Alguns conjuntos dinâmicos pressupõem que as chaves são extraídas de um conjunto totalmente ordenado, como o dos números reais, ou ainda o conjunto de todas as palavras que se-

guem a ordenação alfabética usual. (Um conjunto totalmente ordenado satisfaz à propriedade de tricotomia, definida no Capítulo 3.) Uma ordenação total nos permite definir o elemento mínimo do conjunto, por exemplo, ou falar do próximo elemento maior que um dado elemento em um conjunto.

Operações sobre conjuntos dinâmicos

As operações sobre um conjunto dinâmico podem ser agrupadas em duas categorias: ***consultas***, que simplesmente retornam informações sobre o conjunto, e ***operações de modificação***, que alteram o conjunto. Aqui está uma lista de operações típicas. Qualquer aplicação específica normalmente exigirá que apenas algumas dessas operações sejam implementadas.

SEARCH(S, k)

> Uma consulta que, dado um conjunto S e um valor de chave k, retorna um ponteiro x para um elemento em S tal que $chave[x] = k$, ou NIL se nenhum elemento desse tipo pertencer a S.

INSERT(S, x)

> Uma operação de modificação que aumenta o conjunto S com o elemento apontado por x. Normalmente, supomos que quaisquer campos no elemento x necessários para a implementação do conjunto já tenham sido inicializados.

DELETE(S, x)

> Uma operação de modificação que, dado um ponteiro x para um elemento no conjunto S, remove x de S. (Observe que essa operação utiliza um ponteiro para um elemento x, não um valor de chave.)

MINIMUM(S)

> Uma consulta sobre um conjunto totalmente ordenado S que retorna um ponteiro para o elemento de S com a menor chave.

MAXIMUM(S)

> Uma consulta sobre um conjunto totalmente ordenado S que retorna um ponteiro para o elemento de S com a maior chave.

SUCCESSOR(S, x)

> Uma consulta que, dado um elemento x cuja chave é de um conjunto totalmente ordenado S, retorna um ponteiro para o maior elemento seguinte em S, ou NIL se x é o elemento máximo.

PREDECESSOR(S, x)

> Uma consulta que, dado um elemento x cuja chave é de um conjunto totalmente ordenado S, retorna um ponteiro para o menor elemento seguinte em S, ou NIL se x é o elemento mínimo.

As consultas SUCCESSOR e PREDECESSOR freqüentemente são estendidas a conjuntos com chaves não-distintas. Para um conjunto sobre n chaves, a suposição normal é que uma chamada a MINIMUM seguida por $n - 1$ chamadas a SUCCESSOR enumera os elementos no conjunto em seqüência ordenada.

O tempo empregado para executar uma operação de conjunto é medido normalmente em termos do tamanho do conjunto dado como um de seus argumentos. Por exemplo, o Capítulo 13 descreve uma estrutura de dados que pode admitir quaisquer das operações listadas anteriormente sobre um conjunto de tamanho n no tempo $O(\lg n)$.

Visão geral da Parte III

Os Capítulos 10 a 14 descrevem várias estruturas de dados que podem ser usadas para implementar conjuntos dinâmicos; muitas dessas estruturas serão utilizadas mais tarde para construir algoritmos eficientes destinados à solução de uma variedade de problemas. Outra estrutura de dados importante – o heap (ou monte) – já foi apresentada no Capítulo 6.

O Capítulo 10 apresenta os detalhes essenciais do trabalho com estruturas de dados simples como pilhas, filas, listas ligadas e árvores enraizadas. Ele também mostra como objetos e ponteiros podem ser implementados em ambientes de programação que não os admitem como primitivas. Grande parte desse material deve ser familiar para qualquer pessoa que tenha freqüentado um curso introdutório de programação.

O Capítulo 11 introduz as tabelas hash, que admitem as operações de dicionário INSERT, DELETE e SEARCH. No pior caso, o hash exige o tempo $1(n)$ para executar uma operação SEARCH, mas o tempo esperado para operações sobre tabelas hash é $O(1)$. A análise do hash se baseia na probabilidade, mas a maior parte do capítulo não requer nenhuma experiência no assunto.

As árvores de pesquisa binária, que são focalizadas no Capítulo 12, admitem todas as operações sobre conjuntos dinâmicos listadas anteriormente. No pior caso, cada operação demora um tempo $1(n)$ em uma árvore com n elementos, mas, em uma árvore de pesquisa binária construída aleatoriamente, o tempo esperado para cada operação é $O(\lg n)$. As árvores de pesquisa binária servem como base para muitas outras estruturas de dados.

As árvores vermelho-preto, uma variante de árvores de pesquisa binária, são introduzidas no Capítulo 13. Diferentes das árvores de pesquisa binária comuns, as árvores vermelho-preto oferecem a garantia de funcionar bem: as operações demoram o tempo $O(\lg n)$ no pior caso. Uma árvore vermelho-preto é uma árvore de pesquisa balanceada; o Capítulo 18 apresenta outro tipo de árvore de pesquisa balanceada, chamada árvore B. Embora a mecânica das árvores vermelho-preto seja um pouco complicada, você pode descobrir a maior parte de suas propriedades a partir do capítulo, sem estudar a mecânica em detalhes. Apesar disso, o exame do código pode ser bastante instrutivo.

No Capítulo 14, mostramos como aumentar as árvores vermelho-preto para oferecer suporte a operações diferentes das operações básicas listadas antes. Primeiro, aumentamos essas árvores de modo a podermos manter dinamicamente estatísticas de ordem para um conjunto de chaves. Em seguida, nós as aumentamos de modo diferente, a fim de manter intervalos de números reais.

Parte VIII

Apêndice: Fundamentos de matemática

Introdução

A análise de algoritmos freqüentemente exige a utilização de um conjunto de ferramentas matemáticas. Algumas dessas ferramentas são tão simples quanto a álgebra do ensino de segundo grau, mas outras talvez sejam novas para você. Este apêndice é um compêndio de vários outros conceitos e métodos que empregamos para analisar algoritmos. Conforme observamos na introdução à Parte I, é possível que você tenha visto grande parte do material deste apêndice antes de ler este livro (embora as convenções específicas de notação que utilizamos possam diferir ocasionalmente do que você encontrou em outros livros). Conseqüentemente, você deve tratar o conteúdo deste apêndice como material de referência. No entanto, como no restante do livro, incluímos exercícios e problemas para que você possa melhorar seus conhecimentos nessas áreas específicas.

O Apêndice A oferece métodos para avaliação e delimitação de somatórios, os quais surgirão com freqüência na análise de algoritmos. Muitas das fórmulas desse capítulo poderão ser encontradas em qualquer texto de cálculo, mas você achará conveniente ter esses métodos compilados em um único lugar.

O Apêndice B contém definições e notações básicas para conjuntos, relações, funções, grafos e árvores. Esse capítulo também apresenta algumas propriedades básicas desses objetos matemáticos.

O Apêndice C começa com princípios elementares de contagem: permutações, combinações e assuntos semelhantes. O restante do capítulo contém definições e propriedades de probabilidade básica. A maior parte dos algoritmos deste livro não exige nenhum conhecimento de probabilidade para sua análise e, desse modo, você poderá omitir facilmente as últimas seções do capítulo em uma primeira leitura, até mesmo sem folheá-las. Mais tarde, quando encontrar uma análise probabilística e desejar compreendê-la melhor, você encontrará no Apêndice C um guia bem organizado para fins de referência.

Apêndice A

Somatórios

Quando um algoritmo contém uma construção de controle iterativo (ou repetitivo) como um loop **while** ou **for**, seu tempo de execução pode ser expresso como a soma dos tempos gastos em cada execução do corpo do loop. Por exemplo, descobrimos na Seção 2.2 que a j-ésima iteração da ordenação por inserção demorou um tempo proporcional a j no pior caso. Somando o tempo gasto em cada iteração, obtivemos o somatório (ou a série)

$$\sum_{j=2}^{n} j .$$

A avaliação desse somatório produziu um limite de $\Theta(n^2)$ no tempo de execução do pior caso do algoritmo. Esse exemplo indica a importância geral de se entender como manipular e limitar somatórios.

A Seção A.1 lista diversas fórmulas básicas que envolvem somatórios. A Seção A.2 oferece técnicas úteis para limitar somatórios. As fórmulas da Seção A.1 são dadas sem demonstração, embora as provas de algumas delas sejam apresentadas na Seção A.2 para ilustrar os métodos dessa seção. A maioria das outras provas pode ser encontrada em qualquer texto de cálculo.

A.1 Fórmulas e propriedades de somatórios

Dada uma seqüência de números $a_1, a_2, \ldots$, a soma finita $a_1 + a_2 + \ldots + a_n$, onde n é um inteiro não negativo, pode ser escrita como

$$\sum_{k=1}^{n} a_k .$$

Se $n = 0$, o valor do somatório é definido como 0. O valor de uma série finita é sempre bem definido, e seus termos podem ser somados em qualquer ordem.

Dada uma seqüência de números $a_1, a_2, \ldots$, a soma infinita $a_1 + a_2 + \ldots$ pode ser escrita como

$$\sum_{k=1}^{\infty} a_k ,$$

que é interpretada com o significado

$$\lim_{n \to \infty} \sum_{k=1}^{n} a_k .$$

Se o limite não existe, a série ***diverge***; caso contrário, ela ***converge***. Os termos de uma série convergente nem sempre podem ser adicionados em qualquer ordem. Contudo, podemos reorganizar os termos de uma ***série absolutamente convergente***, ou seja, uma série $\sum_{k=1}^{\infty} a_k$ para a qual a série $\sum_{k=1}^{\infty} |a_k|$ também converge.

Linearidade

Para qualquer número real c e quaisquer seqüências finitas $a_1, a_2, \ldots, a_n$ e $b_1, b_2, \ldots, b_n$,

$$\sum_{k=1}^{n}(ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k \ .$$

A propriedade de linearidade também é obedecida por séries convergentes infinitas.

A propriedade de linearidade pode ser explorada para manipular somatórios que incorporem notação assintótica. Por exemplo,

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right).$$

Nessa equação, a notação Θ no lado esquerdo se aplica à variável k mas, no lado direito, ela se aplica a n. Tais manipulações também podem ser aplicadas a séries convergentes infinitas.

Série aritmética

O somatório

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n \ ,$$

é uma ***série aritmética*** e tem o valor

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1) \tag{A.1}$$

$$= \Theta(n^2) \ . \tag{A.2}$$

Somas de quadrados e cubos

Temos os seguintes somatórios de quadrados e cubos:

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \ , \tag{A.3}$$

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4} \ . \tag{A.4}$$

Série geométrica

Para o real $x \neq 1$, o somatório

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n$$

é uma ***série geométrica*** ou ***exponencial*** e tem o valor

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1} . \qquad (A.5)$$

Quando o somatório é infinito e $|x| < 1$, temos a série geométrica infinitamente decrescente

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} . \qquad (A.6)$$

Série harmônica

Para inteiros positivos n, o n-ésimo ***número harmônico*** é

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} .$$

$$= \sum_{k=1}^{n} \frac{1}{k}$$

$$= \ln n + O(1) . \qquad (A.7)$$

(Provaremos esse limite na Seção A.2.)

Integração e diferenciação de séries

Fórmulas adicionais podem ser obtidas por integração ou diferenciação das fórmulas anteriores. Por exemplo, diferenciando-se ambos os lados da série geométrica infinita (A.6) e multiplicando por x, obtemos

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \qquad (A.8)$$

para $|x| < 1$.

Como inserir séries

Para qualquer seqüência $a_1, a_2, \ldots, a_n$,

$$\sum_{k=1}^{\infty} (a_k - a_{k-1}) = a_n - a_0 , \qquad (A.9)$$

desde que cada um dos termos $a_1, a_2, \ldots, a_{n-1}$ seja adicionado exatamente uma vez e subtraído exatamente uma vez. Dizemos que a soma ***se insere***. De modo semelhante,

$$\sum_{k=1}^{n-1} (a_k - a_{k-1}) = a_0 - a_n .$$

Como exemplo de uma soma por inserção, considere a série

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} .$$

Tendo em vista que podemos reescrever cada termo como

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1},$$

obtemos

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1}\right).$$

Produtos

O produto finito $a_1 a_2 \ldots a_n$ pode ser escrito como

$$\prod_{k=1}^{n} a_k .$$

Se $n = 0$, o valor do produto é definido como 1. Podemos converter uma fórmula com um produto em uma fórmula com um somatório, utilizando a identidade

$$\lg\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \lg a_k .$$

Exercícios

A.1-1
Encontre uma fórmula simples para $\sum_{k=1}^{n} (2k-1)$.

A.1-2 ★
Mostre que $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + O(1)$, manipulando a série harmônica.

A.1-3
Mostre que $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$ para $0 < |x| < 1$.

A.1-4 ★
Mostre que $\sum_{k=1}^{\infty} (k-1)/2^k = 0$.

A.1-5 ★
Avalie a soma $\sum_{k=1}^{\infty} (2k+1)x^{2k}$.

A.1-6
Prove que $\sum_{k=1}^{n} O(f_k(n)) = O(\sum_{k=1}^{n} f_k(n))$ usando a propriedade de linearidade de somatórios.

A.1-7
Avalie o produto $\prod_{k=1}^{n} 2 \cdot 4^k$.

A.1-8 ★
Avalie o produto $\prod_{k=2}^{n} (1 - 1/k^2)$.

A.2 Como limitar somatórios

Existem muitas técnicas disponíveis para limitar os somatórios que descrevem os tempos de execução de algoritmos. Aqui estão alguns dos métodos mais freqüentemente empregados.

Indução matemática

O caminho mais comum para se definir o valor de uma série é usar a indução matemática. Como exemplo, vamos demonstrar que a série aritmética $\sum_{k=1}^{n} k$ tem o valor $\frac{1}{2}n(n+1)$. Isso pode ser verificado facilmente para $n = 1$; assim, criamos a hipótese indutiva de que ela é válida para n e demonstramos que ela vale para $n + 1$. Temos

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1).$$

$$= \frac{1}{2}n(n+1) + (n+1)$$

$$= \frac{1}{2}(n+1)(n+2).$$

Não é necessário adivinhar o valor exato de um somatório para usar a indução matemática. A indução também pode ser usada para mostrar um limite. Como exemplo, vamos demonstrar que a série geométrica $\sum_{k=0}^{n} 3^k$ é $O(3^n)$. Mais especificamente, vamos provar que $\sum_{k=0}^{n} 3^k \le c3^n$ para alguma constante c. No caso da condição inicial $n = 0$, temos $\sum_{k=0}^{n} 3^k = 1 \le c$ enquanto $c \ge 1$. Supondo que o limite se mantenha válido para n, vamos provar que ele é válido para $n + 1$. Temos

$$\sum_{k=0}^{n+1} 3^k = \sum_{k=0}^{n} 3^k + 3^{n+1}$$

$$\le c3^n + c3^{n+1}$$

$$= \left(\frac{1}{3} + \frac{1}{c}\right)c3^{n+1}$$

$$\le c3^{n+1}$$

,

desde que $(1/3 + 1/c) \le 1$ ou, de modo equivalente, $c \ge 3/2$. Portanto, $\sum_{k=0}^{n} 3^k = O(3^n)$, como desejávamos demonstrar.

Temos de ser cuidadosos quando usarmos a notação assintótica para provar limites por indução. Considere a seguinte prova falaciosa de que $\sum_{k=1}^{n} k = O(n)$. Certamente, $\sum_{k=1}^{n} k = O(1)$. Partindo da hipótese de que o limite é válido para n, podemos agora demonstrá-lo para $n + 1$:

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1)$$

$$= O(n) + (n+1) \qquad \text{<= } \textit{errado!!}$$

$$= O(n+1).$$

O erro no argumento é que a "constante" oculta pelo "*O* maiúsculo" cresce com n e, portanto, não é constante. Não mostramos que a mesma constante funciona para *todo* n.

Limitando os termos

Às vezes, um bom limite superior em uma série pode ser obtido limitando-se cada termo da série, e com freqüência é suficiente utilizar o maior termo para limitar os outros. Por exemplo, um limite superior rápido sobre a série aritmética (A.1) é

$$\sum_{k=1}^{n+1} k \le \sum_{k=1}^{n} n$$
$$= n^2 .$$

Em geral, para uma série $\sum_{k=1}^{n} a_k$, se considerarmos $a_{max} = \max_{1 \le k \le n} a_k$, então

$$\sum_{k=1}^{n} a_k \le n a_{max} .$$

A técnica de limitar cada termo em uma série pelo maior termo é um método fraco quando a série pode de fato ser limitada por uma série geométrica. Dada a série $\sum_{k=0}^{n} a_k$, suponha que $a_{k+1}/a_k \le r$ para todo $k \ge 0$, onde $0 < r < 1$ é uma constante. A soma pode ser limitada por uma série geométrica decrescente infinita, pois $a_k \le a_0 r^k$ e, desse modo,

$$\sum_{k=0}^{n} a_k \le \sum_{k=0}^{\infty} a_0 r^k$$
$$= a_0 \sum_{k=0}^{\infty} r^k$$
$$= a_0 \frac{1}{1-r} .$$

Podemos aplicar esse método para limitar o somatório $\sum_{k=1}^{\infty} (k/3^k)$. Para iniciar o somatório em $k = 0$, nós o reescrevemos como $\sum_{k=0}^{\infty} ((k+1)/3^{k+1})$. O primeiro termo ($a_0$) é 1/3, e a razão ($r$) entre os termos sucessivos é

$$\frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} = \frac{1}{3} \cdot \frac{k+2}{k+1}$$
$$\le \frac{2}{3}$$

para todo $k \ge 1$. Desse modo, temos

$$\sum_{k=1}^{\infty} \frac{k}{3^k} = \sum_{k=0}^{\infty} \frac{k+1}{3^{k+1}}$$
$$\le \frac{1}{3} \cdot \frac{1}{1-2/3}$$
$$= 1 .$$

Um erro comum na aplicação desse método é mostrar que a razão entre termos sucessivos é menor que 1, e então admitir como hipótese que o somatório é limitado por uma série geométrica. Um exemplo é a série harmônica infinita, que diverge desde

$$\sum_{k=1}^{\infty} \frac{1}{k} = \lim_{n \to \infty} \sum_{k=1}^{n} \frac{1}{k}$$
$$= \lim_{n \to \infty} \Theta(\lg n)$$
$$= \infty .$$

A razão entre o $(k + 1)$-ésimo e o k-ésimo termos nessa série é $k/(k + 1) < 1$, mas a série não é limitada por uma série geométrica decrescente. Para limitar uma série por uma série geométrica, deve-se mostrar que existe um $r < 1$ que é uma *constante*, tal que a razão entre todos os pares de termos sucessivos nunca exceda r. Na série harmônica, não existe tal r porque a razão se torna arbitrariamente próxima de 1.

Divisão de somatórios

Uma das maneiras de obter limites em um somatório difícil é expressar a série como a soma de duas ou mais séries, particionando-se o intervalo do índice e, em seguida, limitando-se cada uma das séries resultantes. Por exemplo, suponha a tentativa de encontrar um limite inferior da série aritmética $\sum_{k=1}^{n} k$, a qual já mostramos que tem um limite superior n^2. Poderíamos tentar limitar cada termo no somatório pelo menor termo mas, como esse termo é 1, obtemos um limite inferior n para o somatório – bem longe do nosso limite superior n^2.

Podemos obter um limite inferior melhor dividindo primeiro o somatório. Por conveniência, suponha que n seja par. Temos

$$
\begin{aligned}
\sum_{k=1}^{n} k &= \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \\
&\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} (n/2) \\
&= (n/2)^2 \\
&= \Omega(n^2) ,
\end{aligned}
$$

que é um limite assintoticamente restrito, considerando-se que $\sum_{k=1}^{n} k = O(n^2)$.

Para um somatório que surge da análise de um algoritmo, podemos freqüentemente dividir o somatório e ignorar um número constante dos termos iniciais. Em geral, essa técnica se aplica quando cada termo a_k em um somatório $\sum_{k=0}^{n} a_k$ é independente de n. Então, para qualquer constante $k_0 > 0$, podemos escrever

$$
\begin{aligned}
\sum_{k=0}^{n} a_k &= \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k \\
&= \Theta(1) + \sum_{k=k_0}^{n} a_k ,
\end{aligned}
$$

pois os termos iniciais do somatório são todos constantes e existe um número constante deles. Podemos então usar outros métodos para limitar $\sum_{k=k_0}^{n} a_k$. Por exemplo, encontrar um limite superior assintótico sobre

$$
\sum_{k=0}^{\infty} \frac{k^2}{2^k} ,
$$

observamos que a razão entre termos sucessivos é

$$
\begin{aligned}
\frac{(k+1)^2/2^{k+1}}{k^2/2^k} &= \frac{(k+2)^2}{2k^2} \\
&\leq \frac{8}{9}
\end{aligned}
$$

se $k \geq 3$. Portanto, o somatório também pode ser dividido em

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k} = \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k}$$

$$\leq \sum_{k=0}^{2} \frac{k^2}{2^k} + \frac{8}{9} \sum_{k=0}^{\infty} \left(\frac{8}{9} \right)^k$$

$$= O(1) ,$$

pois o segundo somatório é uma série geométrica decrescente.

A técnica de dividir somatórios pode ser usada para definir limites assintóticos em situações muito mais difíceis. Por exemplo, podemos obter um limite $O(\lg n)$ na série harmônica (A.7):

$$H_n = \sum_{k=1}^{n} \frac{1}{k} .$$

A idéia é dividir o intervalo de 1 a n em $\lfloor \lg n \rfloor$ fragmentos e estabelecer um limite superior como contribuição de cada fragmento em 1. Cada fragmento consiste nos termos que começam em $1/2^i$ e que vão até $1/2^{i+1}$, sem incluí-lo, fornecendo

$$\sum_{k=1}^{n} \frac{1}{k} \leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i - 1} \frac{1}{2^i + j}$$

$$\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i - 1} \frac{1}{2^i}$$

$$\leq \sum_{i=0}^{\lfloor \lg n \rfloor} 1$$

$$\leq \lg n + 1 . \qquad \text{(A.10)}$$

Aproximação por integrais

Quando um somatório pode ser expresso como $\sum_{k=m}^{n} f(k)$, onde $f(k)$ é uma função monotonicamente crescente, é possível fazer a aproximação do somatório por integrais:

$$\int_{m-1}^{n} f(x)dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)dx . \qquad \text{(A.11)}$$

A justificativa para essa aproximação é mostrada na Figura A.1. O somatório é representado como a área dos retângulos na figura, e a integral é a região sombreada sob a curva. Quando $f(k)$ é uma função monotonicamente decrescente, podemos usar um método semelhante para fornecer os limites

$$\int_{m-1}^{n+1} f(x)dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m-1}^{n} f(x)dx . \qquad \text{(A.12)}$$

A aproximação integral (A.12) fornece uma estimativa restrita para o n-ésimo número harmônico. No caso de um limite inferior, obtemos

$$\sum_{k=1}^{n} \frac{1}{k} \geq \int_{1}^{n+1} \frac{dx}{x}$$

$$= \ln(n + 1) . \qquad \text{(A.13)}$$

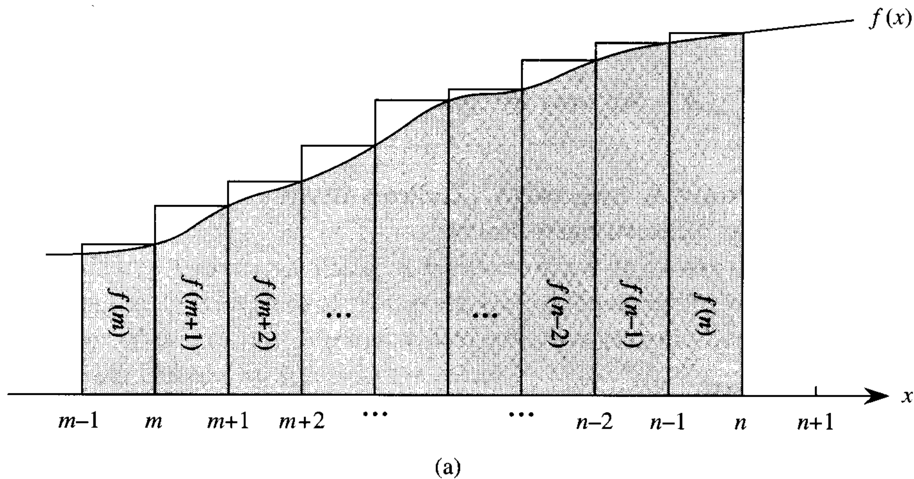
Para o limite superior, derivamos a desigualdade

$$\sum_{k=2}^{n} \frac{1}{k} \leq \int_{1}^{n} \frac{dx}{x}$$

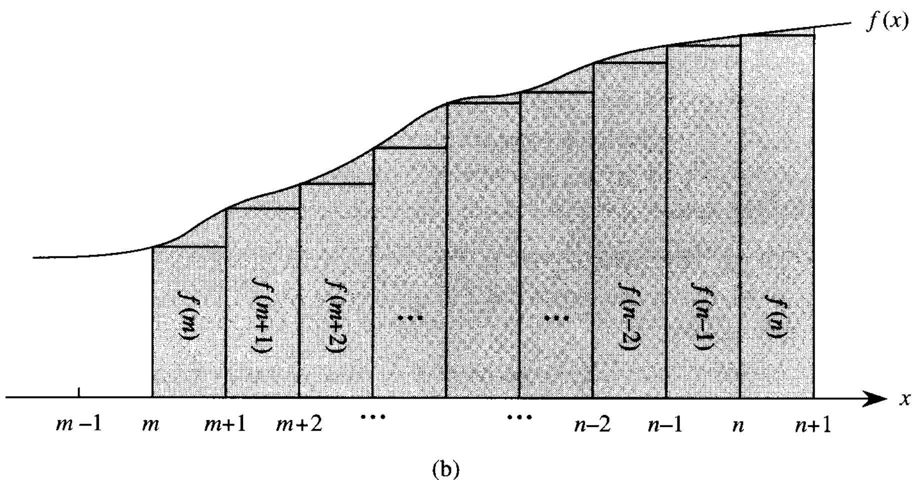
$$= \ln n ,$$

que produz o limite

$$\sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1 . \qquad \text{(A.14)}$$



(a)



(b)

FIGURA A.1 Aproximação de $\sum_{k=m}^{n} f(x)$ por integrais. A área de cada retângulo é mostrada dentro do retângulo, e a área total dos retângulos representa o valor do somatório. A integral é representada pela área sombreada sob a curva. Comparando as áreas em (**a**), obtemos $\int_{m-1}^{n} f(x)dx \leq \sum_{k=m}^{n} f(k)$, e depois, deslocando os retângulos uma unidade para a direita, obtemos $\sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)dx$ em (**b**)

Exercícios

A.2-1
Mostre que $\sum_{k=1}^{n} 1/k^2$ é limitada acima por uma constante.

A.2-2
Encontre um limite superior assintótico sobre o somatório

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil$$

A.2-3
Mostre que o n-ésimo número harmônico é $\Omega(\lg n)$, dividindo o somatório.

A.2-4
Faça a aproximação de $\sum_{k=1}^{n} k^3$ com uma integral.

A.2-5
Por que não usamos a aproximação integral (A.12) diretamente em $\sum_{k=1}^{n} 1/k$ para obter um limite superior sobre o n-ésimo número harmônico?

Problemas

A-1 Limitando somatórios
Forneça limites assintoticamente restritos sobre os somatórios a seguir. Suponha que $r \geq 0$ e $s \geq 0$ sejam constantes.

a. $\sum_{k=1}^{n} k^r$.

b. $\sum_{k=1}^{n} \lg^s k$.

c. $\sum_{k=1}^{n} k^r \lg^s k$.

Notas do capítulo

Knuth [182] é uma excelente referência para o material apresentado neste capítulo. As propriedades básicas de séries podem ser encontradas em qualquer bom livro de cálculo, como Apostol [18] ou Thomas e Finney [296].

Apêndice B

Conjuntos e outros temas

Muitos capítulos deste livro mencionam os fundamentos de matemática discreta. Este capítulo reexamina de forma mais completa as notações, definições e propriedades elementares de conjuntos, relações, funções, grafos e árvores. Os leitores que já estão bem versados nesses assuntos só precisam dar uma olhada rápida neste capítulo.

B.1 Conjuntos

Um conjunto é uma coleção de objetos distintos, que são chamados ***elementos*** ou ***membros*** do conjunto. Se um objeto x é um elemento de (ou pertence a) um conjunto S, escrevemos $x \in S$ (lê-se "x é um membro de S", "x é elemento de S", "x pertence a S" ou, de modo mais abreviado, "x está em S"). Se x não é um elemento de S, escrevemos que $x \notin S$. Podemos descrever um conjunto relacionando explicitamente seus elementos como uma lista entre chaves. Por exemplo, é possível definir um conjunto S contendo exatamente os números 1, 2 e 3, escrevendo-se $S = \{1, 2, 3\}$. Como 2 é um elemento do conjunto S, podemos escrever $2 \in S$; como 4 não é um elemento do conjunto, temos $4 \notin S$. Um conjunto não pode conter o mesmo objeto mais de uma vez,[1] e seus elementos não são ordenados. Dois conjuntos A e B são ***iguais***, sendo representados por $A = B$, se eles contêm os mesmos elementos. Por exemplo, $\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$.

Adotamos notações especiais para conjuntos encontrados com freqüência.

- $\emptyset$ denota o ***conjunto vazio***, isto é, o conjunto que não contém nenhum elemento.
- **Z** denota o conjunto de ***números inteiros***, isto é, o conjunto $\{..., -2, -1, 0, 1, 2, ...\}$.
- **R** denota o conjunto de ***números reais***.
- **N** denota o conjunto de ***números naturais***, isto é, o conjunto $\{0, 1, 2, ...\}$.[2]

Se todos os elementos de um conjunto A estão contidos em um conjunto B, ou seja, se $x \in A$ implica $x \in B$, escrevemos $A \subseteq B$ e dizemos que A é um ***subconjunto*** de B. Um conjunto A é um ***subconjunto próprio*** de B, representado por A B, se $A \subseteq B$ mas $A \neq B$. (Alguns autores usam o símbolo "" para denotar a relação de subconjunto comum, em lugar da relação de subconjunto

[1] Uma variação de um conjunto, que pode conter o mesmo objeto mais de uma vez, é chamada um ***multiconjunto***.

[2] Alguns autores iniciam os números naturais com 1 em vez de 0. A tendência moderna parece ser a de iniciar esse conjunto com 0.

próprio.) Para qualquer conjunto A, temos $A \subseteq A$. No caso de dois conjuntos A e B, temos $A = B$ se e somente se $A \subseteq B$ e $B \subseteq A$. Para três conjuntos A, B e C quaisquer, se $A \subseteq B$ e $B \subseteq C$, então $A \subseteq C$. Para qualquer conjunto A, temos $\emptyset \subseteq A$.

Algumas vezes, definimos conjuntos em termos de outros conjuntos. Dado um conjunto A, podemos definir um conjunto $B \subseteq A$ declarando uma propriedade que distingue os elementos de B. Por exemplo, podemos definir o conjunto de números inteiros pares por $\{x : x \in \mathbf{Z} \text{ e } x/2 \text{ é um inteiro}\}$. Nessa notação, o sinal de dois-pontos significa "tal que". (Alguns autores usam uma barra vertical em lugar do sinal de dois-pontos.)

Dados dois conjuntos A e B, também podemos definir novos conjuntos aplicando ***operações de conjuntos***:

- A ***interseção*** de conjuntos A e B é o conjunto

$$A \cap B = \{x : x \in A \text{ e } x \in B\} .$$

- A ***união*** de conjuntos A e B é o conjunto

$$A \cup B = \{x : x \in A \text{ ou } x \in B\} .$$

- A ***diferença*** entre dois conjuntos A e B é o conjunto

$$A - B = \{x : x \in A \text{ e } x \notin B\} .$$

As operações de conjuntos obedecem às leis enunciadas a seguir.

Leis de conjuntos vazios:

$$A \cap \emptyset = \emptyset,$$

$$A \cup \emptyset = A .$$

Leis de idempotência:

$$A \cap A = A ,$$

$$A \cup A = A .$$

Leis comutativas:

$$A \cap B = B \cap A ,$$

$$A \cup B = B \cup A .$$

Leis associativas:

$$A \cap (B \cap C) = (A \cap B) \cap C ,$$

$$A \cup (B \cup C) = (A \cup B) \cup C .$$

Leis distributivas:

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) , \quad \text{(B.1)}$$

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C) .$$

Leis de absorção:

$$A \cap (A \cup B) = A \,,$$

$$A \cup (A \cap B) = A \,.$$

Leis de DeMorgan:

$$A - (B \cap C) = (A - B) \cup (A - C) \,,$$

$$A - (B \cup C) = (A - B) \cap (A - C) \,. \qquad \text{(B.2)}$$

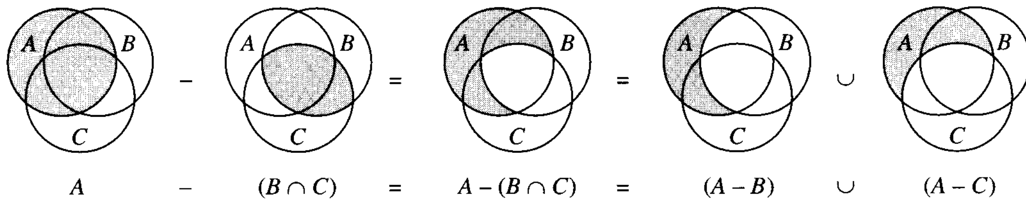


FIGURA B.1 Um diagrama de Venn que ilustra as primeiras leis de DeMorgan (B.2). Cada um dos conjuntos A, B e C é representado como um círculo

A primeira das leis de DeMorgan está ilustrada na Figura B.1 com o uso de um ***diagrama de Venn***, uma imagem gráfica na qual os conjuntos são representados como regiões do plano.

Muitas vezes, todos os conjuntos sob consideração são subconjuntos de algum conjunto maior U chamado ***universo***. Por exemplo, se estivermos considerando vários conjuntos formados apenas por inteiros, o conjunto **Z** de inteiros será um universo apropriado. Dado um universo U, definimos o ***complemento*** de um conjunto A como $\overline{A} = U - A$. Para qualquer conjunto $A \subseteq U$, temos as seguintes leis:

$$\overline{\overline{A}} = A \,,$$

$$A \cap \overline{A} = \emptyset \,,$$

$$A \cup \overline{A} = U \,.$$

As leis de DeMorgan (B.2) podem ser reescritas com complementos. Para dois conjuntos quaisquer $A, B \subseteq U$, temos

$$\overline{B \cap C} = \overline{B} \cup \overline{C} \,,$$

$$\overline{B \cup C} = \overline{B} \cap \overline{C}$$

Dois conjuntos A e B são ***disjuntos*** se não têm nenhum elemento em comum, ou seja, se $A \cap B = \emptyset$. Uma coleção $\mathcal{S} = \{S_i\}$ de conjuntos não vazios forma uma ***partição*** de um conjunto S se:

- os conjuntos são ***disjuntos aos pares***, isto é, $S_i, S_j \in \mathcal{S}$ e $i \neq j$ implicam $S_i \cap S_j = \emptyset$, e
- sua união é S, isto é,

$$S = \bigcup_{S_i \in \mathcal{S}} S_i \,.$$

Em outras palavras, $\mathcal{S}$ forma uma partição de S se cada elemento de S aparece em exatamente um $S_i \in \mathcal{S}$.

O número de elementos em um conjunto é chamado ***cardinalidade*** (ou ***tamanho***) do conjunto, denotada por $|S|$. Dois conjuntos têm a mesma cardinalidade se seus elementos podem ser colocados em uma correspondência de um para um. A cardinalidade do conjunto vazio é $|\phi| = 0$. Se a cardinalidade de um conjunto é um número natural, dizemos que o conjunto é ***finito***; caso contrário, ele é ***infinito***. Um conjunto infinito que pode ser colocado em uma correspondência de um para um com os números naturais **N** é ***infinito contável***; caso contrário, ele é ***não contável***. Os inteiros **Z** são contáveis, mas os reais **R** são não contáveis.

Para dois conjuntos finitos A e B, temos a identidade

$$|A \cup B| = |A| + |B| - |A \cap B| , \tag{B.3}$$

da qual podemos concluir que

$$|A \cup B| \leq |A| + |B| .$$

Se A e B são disjuntos, então $|A \cap B| = 0$ e, portanto, $|A \cup B| = |A| + |B|$. Se $A \subseteq B$, então $|A| \leq |B|$.

Um conjunto finito formado por n elementos às vezes é chamado um ***conjunto de n elementos***. Um conjunto de um elemento é chamado ***unitário***. Um subconjunto de k elementos de um conjunto às vezes é chamado um ***subconjunto de k elementos***.

O conjunto de todos os subconjuntos de um conjunto S, inclusive o conjunto vazio e o próprio conjunto S, é denotado por 2^S e é chamado ***conjunto potência*** de S. Por exemplo, $2^{\{a, b\}} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. O conjunto potência de um conjunto finito S tem cardinalidade $2^{|S|}$.

Algumas vezes, utilizamos estruturas semelhantes a conjuntos, nas quais os elementos estão ordenados. Um ***par ordenado*** de dois elementos a e b é denotado por (a, b) e pode ser definido formalmente como o conjunto $(a, b) = \{a, \{a, b\}\}$. Desse modo, o par ordenado (a, b) não é igual ao par ordenado (b, a).

O ***produto cartesiano*** de dois conjuntos A e B, denotado por $A \times B$, é o conjunto de todos os pares ordenados tais que o primeiro elemento do par é um elemento de A e o segundo é um elemento de B. De modo mais formal,

$$A \times B = \{(a, b) : a \in A \text{ e } b \in B\} .$$

Por exemplo, $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$. Quando A e B são conjuntos finitos, a cardinalidade de seu produto cartesiano é

$$|A \times B| = |A| \cdot |B| . \tag{B.4}$$

O produto cartesiano de n conjuntos $A_1, A_2, ..., A_n$ é o conjunto de ***n-tuplas***

$$A_1 \times A_2 \times \ldots \times A_n = (a_1, a_2, ..., a_n) : a_i \in A_i, i = 1, 2, ..., n) ,$$

cuja cardinalidade é

$$|A_1 \times A_2 \times \ldots \times A_n| = |A_1| \cdot |A_2| \ldots |A_n|$$

se todos conjuntos são finitos. Denotamos um produto cartesiano de n termos sobre um único conjunto A pelo conjunto

$$A^n = A \times A \times \ldots \times A ,$$

cuja cardinalidade é $|A^n| = |A|^n$ se A é finito. Uma n-tupla também pode ser visualizada como uma seqüência finita de comprimento n (ver Seção B.3).

Exercícios

B.1-1

Trace diagramas de Venn que ilustrem a primeira das leis distributivas (B.1).

B.1-2

Prove a generalização das leis de DeMorgan para qualquer coleção finita de conjuntos:

$$\overline{A_1 \cap A_2 \cap \ldots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \ldots \cup \overline{A_n} ,$$

$$\overline{A_1 \cup A_2 \cup \ldots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \ldots \cap \overline{A_n} .$$

B.1-3 ★

Prove a generalização da equação (B.3), que é chamada ***princípio de inclusão e exclusão***:

$$\begin{aligned} |A_1 \cup A_2 \cup \ldots \cup A_n| = {} & \\ & |A_1| + |A_2| + \ldots + |A_n| \\ & -|A_1 \cap A_2| + \ldots + |A_1 \cap A_3| - \ldots && \text{(todos os pares)} \\ & +|A_1 \cap A_2 \cap A_3| + \ldots && \text{(todas as triplas)} \\ & \vdots \\ & +(-1)^{n-1}|A_1 \cap A_2 \cap A_3| . \end{aligned}$$

B.1-4

Mostre que o conjunto de números naturais ímpares é contável.

B.1-5

Mostre que, para qualquer conjunto finito S, o conjunto potência 2^S tem $2^{|S|}$ (ou seja, existem $2^{|S|}$ subconjuntos distintos de S).

B.1-6

Dê uma definição indutiva para uma n-tupla, estendendo a definição da teoria dos conjuntos para um par ordenado.

B.2 Relações

Uma ***relação binária*** R sobre dois conjuntos A e B é um subconjunto do produto cartesiano $A \times B$. Se $(a, b) \in R$, algumas vezes escrevemos $a\,R\,b$. Quando dizemos que R é uma relação binária sobre um conjunto A, queremos dizer que R é um subconjunto de $A \times A$. Por exemplo, a relação "menor que" sobre os números naturais é o conjunto $\{(a, b) : a, b \in \mathbf{N} \text{ e } a < b\}$. Uma relação n-ária sobre conjuntos $A_1, A_2, \ldots, A_n$ é um subconjunto de $A_1 \times A_2 \times \ldots \times A_n$.

Uma relação binária $R \subseteq A \times A$ é ***reflexiva*** se

$$a\,R\,a$$

para todo $a \in A$. Por exemplo, "=" e "≤" são relações reflexivas em **N**, mas "<" não é. A relação R é ***simétrica*** se

$$a\,R\,b \text{ implica } b\,R\,a$$

para todo $a, b \in A$. Por exemplo, "=" é simétrica, mas "<" e "≤" não são. A relação R é ***transitiva*** se

$$a\,R\,b \text{ e } b\,R\,c \text{ implicam } a\,R\,c$$

para todo $a, b, c \in A$. Por exemplo, as relações "<", "≤" e "=" são transitivas, mas a relação $R = \{(a, b) : a, b \in \mathbf{N} \text{ e } a = b - 1\}$ não é, pois $3 R 4$ e $4 R 5$ não implicam $3 R 5$.

Uma relação que é reflexiva, simétrica e transitiva é uma ***relação de equivalência***. Por exemplo, "=" é uma relação de equivalência sobre os números naturais, mas "<" não é. Se R é uma relação de equivalência em um conjunto A, então, para $a \in A$, a ***classe de equivalência*** de a é o conjunto $[a] = \{b \in A : a R b\}$, ou seja, o conjunto de todos os elementos equivalentes a a. Por exemplo, se definimos $R = \{(a, b) : a, b \in \mathbf{N} \text{ e } a + b \text{ é um número par}\}$, então R é uma relação de equivalência, pois $a + a$ é par (reflexiva), $a + b$ é par implica $b + a$ é par (simétrica) e $a + b$ é par e $b + c$ é par implicam $a + c$ é par (transitiva). A classe de equivalência de 4 é $[4] = \{0, 2, 4, 6, ...\}$, e a classe de equivalência de 3 é $[3] = \{1, 3, 5, 7, ...\}$. Um teorema básico de classes de equivalência é dado a seguir.

Teorema B.1 (Uma relação de equivalência é o mesmo que uma partição)
As classes de equivalência de qualquer relação de equivalência R sobre um conjunto A formam uma partição de A, e qualquer partição de A determina uma relação de equivalência sobre A para a qual os conjuntos na partição são as classes de equivalência.

Prova Como primeira parte da prova, devemos mostrar que as classes de equivalência de R são conjuntos não vazios e disjuntos aos pares cuja união é A. Como R é reflexiva, $a \in [a]$, e assim as classes de equivalência são não vazias; além disso, tendo em vista que todo elemento $a \in A$ pertence à classe de equivalência $[a]$, a união das classes de equivalência é A. Resta mostrar que as classes de equivalência são conjuntos disjuntos aos pares, isto é, se duas classes de equivalência $[a]$ e $[b]$ têm um elemento c em comum, então elas são de fato o mesmo conjunto. Agora, $a R c$ e $b R c$ que, por simetria e transitividade, implicam $a R b$. Portanto, para qualquer elemento arbitrário $x \in [a]$, temos que $x R a$ implica $x R b$ e, desse modo, $[a] \subseteq [b]$. De modo semelhante, $[b] \subseteq [a]$ e, assim sendo, $[a] = [b]$.

Como segunda parte da prova, seja $\mathscr{A} = \{A_i\}$ uma partição de A, e defina $R = \{(a, b) : \text{existe } i \text{ tal que } a \in A_i \text{ e } b \in A_i\}$. Afirmamos que R é uma relação de equivalência em A. A reflexividade se mantém, pois $a \in A_i$ implica $a R a$. A simetria se mantém porque, se $a R b$, então a e b estão no mesmo conjunto A_i e, conseqüentemente, $b R a$. Se $a R b$ e $b R c$, então todos os três elementos estão no mesmo conjunto e, desse modo, $a R c$, e a transitividade é mantida. Para verificar que os conjuntos na partição são as classes de equivalência de R, observe que, se $a \in A_i$, então $x \in [a]$ implica $x \in A_i$, e $x \in A_i$ implica $x \in [a]$. ■

Uma relação binária R sobre um conjunto A é ***anti-simétrica*** se

$$a R b \text{ e } b R a \text{ implicam } a = b .$$

Por exemplo, a relação "<" sobre os números naturais é anti-simétrica, pois $a \leq b$ e $b \leq a$ implicam $a = b$. Uma relação que é reflexiva, anti-simétrica e transitiva é uma ***ordem parcial***, e chamamos um conjunto no qual uma ordem parcial é definida de ***conjunto parcialmente ordenado***. Por exemplo, a relação "é descendente de" é uma ordem parcial no conjunto de todas as pessoas (se visualizarmos indivíduos como sendo seus próprios descendentes).

Em um conjunto parcialmente ordenado A, não pode haver nenhum elemento "máximo" a tal que $b R a$ para todo $b \in A$. Em vez disso, pode haver diversos elementos ***máximos*** a tais que, para nenhum $b \in A$, onde $b \neq a$, ocorre que $a R b$. Por exemplo, em uma coleção de caixas de diferentes tamanhos podem existir várias caixas máximas que não se ajustam dentro de qualquer outra caixa, ainda que não exista nenhuma caixa máxima única dentro da qual se ajustará qualquer outra caixa.[3]

---

[3] Para sermos exatos, com a finalidade de fazer a relação "caber em" ser uma ordem parcial, precisamos visualizar uma caixa como cabendo dentro dela própria.

Uma ordem parcial R sobre um conjunto A é uma ***ordem total*** ou ***ordem linear*** se, para todo $a, b \in A$, temos $a\,R\,b$ ou $b\,R\,a$, ou seja, se for possível relacionar entre si todos os pares de elementos de A em R. Por exemplo, a relação "$\leq$" é uma ordem total sobre os números naturais, mas a relação "é descendente de" não é uma ordem total sobre o conjunto de todas as pessoas, pois existem indivíduos que não são descendentes uns dos outros.

Exercícios

B.2-1
Prove que a relação de subconjunto "$\subseteq$" em todos os subconjuntos de **Z** é uma ordem parcial mas não uma ordem total.

B.2-2
Mostre que, para qualquer inteiro positivo n, a relação "equivalente de módulo n" é uma relação de equivalência sobre os inteiros. (Dizemos que $a \equiv b \pmod{n}$ se existe um inteiro q tal que $a - b = qn$.) Em que classes de equivalência essa relação particiona os inteiros?

B.2-3
Dê exemplos de relações que sejam

a. reflexivas e simétricas, mas não transitivas,

b. reflexivas e transitivas, mas não simétricas,

c. simétricas e transitivas, mas não reflexivas.

B.2-4
Seja S um conjunto finito, e seja R uma relação de equivalência sobre $S \times S$. Mostre que, se uma adição R é anti-simétrica, então as classes de equivalência de S com respeito a R são unitárias.

B.2-5
O professor Narciso afirma que, se uma relação R é simétrica e transitiva, então ela também é reflexiva. Ele oferece a seguinte prova. Por simetria, $a\,R\,b$ implica $b\,R\,a$. Por essa razão, a transitividade implica $a\,R\,a$. O professor está certo?

B.3 Funções

Dados dois conjuntos A e B, uma ***função*** f é uma relação binária em $A \times B$ tal que, para $a \in A$, existe exatamente um $b \in B$ tal que $(a, b) \in f$. O conjunto A é chamado ***domínio*** de f e o conjunto B é chamado ***contradomínio*** de f. Algumas vezes, escrevemos $f : A \to B$; e, se $(a, b) \in f$, escrevemos $b = f(a)$, pois b é determinado de forma unívoca pela escolha de a.

Intuitivamente, a função f atribui um elemento de B a cada elemento de A. Nenhum elemento de A recebe a atribuição de dois elementos diferentes de B, mas o mesmo elemento de B pode receber a atribuição de dois elementos diferentes de A. Por exemplo, a relação binária

$$f = \{(a, b) : a, b \in \mathbf{N} \text{ e } b = a \bmod 2\}$$

é uma função $f : \mathbf{N} \to \{0, 1\}$ porque, para cada número natural a, existe exatamente um valor b em $\{0, 1\}$ tal que $b = a \bmod 2$. Para esse exemplo, $0 = f(0)$, $1 = f(1)$, $0 = f(2)$ etc. Em contraste, a relação binária

$$g = \{(a, b) : a, b \in \mathbf{N} \text{ e } a + b \text{ é par}\}$$

não é uma função, pois $(1, 3)$ e $(1, 5)$ estão ambos em g e, desse modo, para a opção $a = 1$, não existe exatamente um b tal que $(a, b) \in g$.

Dada uma função $f : A \to B$, se $b = f(a)$, dizemos que a é o ***argumento*** de f e que b é o ***valor*** de f em a. Podemos definir uma função declarando seu valor para cada elemento de seu domínio. Por exemplo, poderíamos definir $f(n) = 2n$ para $n \in \mathbf{N}$, o que significa $f = \{(n, 2n) : n \in \mathbf{N}\}$. Duas funções f e g são ***iguais*** se elas têm o mesmo domínio e contradomínio e se, para todo a no domínio, $f(a) = g(a)$.

Uma ***seqüência finita*** de comprimento n é uma função f cujo domínio é o conjunto de n inteiros $\{0, 1, ..., n-1\}$. Com freqüência, denotamos uma seqüência finita listando seus valores: $\langle f(0), f(1), ..., f(n-1)\rangle$. Uma ***seqüência infinita*** é uma função cujo domínio é o conjunto $\mathbf{N}$ dos números naturais. Por exemplo, a seqüência de Fibonacci definida pela recorrência (3.21), é a seqüência infinita $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, ...\rangle$.

Quando o domínio de uma função f é um produto cartesiano, freqüentemente omitimos os parênteses extras que envolvem o argumento de f. Por exemplo, se tivéssemos uma função $f : A_1 \times A_2 \times ... \times A_n \to B$, escreveríamos $b = f(a_1, a_2, ..., a_n)$ em vez de $b = f((a_1, a_2, ..., a_n))$. Também chamamos cada a_i, um ***argumento*** para a função f, embora tecnicamente o (único) argumento para f seja a n-tupla $(a_1, a_2, ..., a_n)$.

Se $f : A \to B$ é uma função e $b = f(a)$, então dizemos às vezes que b é a ***imagem*** de a sob f. A imagem de um conjunto $A' \subseteq A$ sob f é definida por

$$f(A') = \{b \in B : b = f(a) \text{ para algum } a \in A'\} .$$

O ***intervalo*** de f é a imagem do seu domínio, ou seja, $f(A)$. Por exemplo, o intervalo da função $f : \mathbf{N} \to \mathbf{N}$ definida por $f(n) = 2n$ é $f(\mathbf{N}) = \{m : m = 2n \text{ para algum } n \in \mathbf{N}\}$.

Uma função é uma ***sobrejeção*** se seu intervalo é seu contradomínio. Por exemplo, a função $f(n) = \lfloor n/2 \rfloor$ é uma função sobrejetora de $\mathbf{N}$ para $\mathbf{N}$, pois todo elemento em $\mathbf{N}$ aparece como o valor de f para algum argumento. Em contraste, a função $f(n) = 2n$ não é uma função sobrejetora de $\mathbf{N}$ para $\mathbf{N}$, porque nenhum argumento de f pode produzir 3 como um valor. Porém, a função $f(n) = 2n$ é uma função sobrejetora dos números naturais para os números pares. Uma sobrejeção $f : A \to B$ é descrita algumas vezes como o mapeamento de A ***sobre*** B. Quando dizemos que f está sobre, queremos dizer que ela é sobrejetora.

Uma função $f : A \to B$ é uma ***injeção*** se argumentos distintos de f produzem valores distintos, isto é, se $a \neq a'$ implica $f(a) \neq f(a')$. Por exemplo, a função $f(n) = 2n$ é um função injetora de $\mathbf{N}$ para $\mathbf{N}$, pois cada número par b é a imagem sob f de no máximo um elemento do domínio, ou seja, $b/2$. A função $f(n) = \lfloor n/2 \rfloor$ não é injetora, pois o valor 1 é produzido por dois argumentos: 2 e 3. Algumas vezes, uma injeção é chamada uma função de ***um para um***.

Uma função $f : A \to B$ é um ***bijeção*** se ela é injetora e sobrejetora. Por exemplo, a função $f(n) = (-1)^n \lceil n/2 \rceil$ é uma bijeção de $\mathbf{N}$ para $\mathbf{Z}$:

$0 \to 0,$

$1 \to -1,$

$2 \to 1,$

$3 \to -2,$

$4 \to 2,$

$\vdots$

A função é injetora, porque nenhum elemento de $\mathbf{Z}$ é a imagem de mais de um elemento de $\mathbf{N}$. Ela é sobrejetora, pois todo elemento de $\mathbf{Z}$ aparece como imagem de algum elemento de $\mathbf{N}$. Então, a função é bijetora. Às vezes, uma bijeção é chamada uma ***correspondência de um para um***, porque emparelha elementos do domínio e do contradomínio. Uma bijeção de um conjunto A para ele mesmo é chamada às vezes ***permutação***.

Quando uma função f é bijetora, sua ***inversa*** f^{-1} é definida como

$$f^{-1}(b) = a \text{ se e somente se } f(a) = b .$$

Por exemplo, a inversa da função $f(n) = (-1)^n \lceil n/2 \rceil$ é

$$f^{-1}(m) = \begin{cases} 2m & \text{se } m \geq 0 , \\ -2m & \text{se } m < 0 . \end{cases}$$

Exercícios

B.3-1
Sejam A e B conjuntos finitos, e seja $f : A \rightarrow B$ uma função. Mostre que

a. se f é injetora, então $|A| \leq |B|$;

b. se f é sobrejetora, então $|A| \geq |B|$.

B.3-2
A função $f(x) = x + 1$ é bijetora quando o domínio e o contradomínio são **N**? Ela é bijetora quando o domínio e o contradomínio são **Z**?

B.3-3
Dê uma definição natural para a inversa de uma relação binária tal que, se uma relação é de fato uma função bijetora, sua inversa relacional é sua inversa funcional.

B.3-4 ★
Dê uma bijeção de **Z** para **Z** × **Z**.

B.4 Grafos

Esta seção apresenta dois tipos de grafos: orientado e não orientado. O leitor deve saber que certas definições na literatura diferem das que são dadas aqui mas, em sua maioria, as diferenças são ligeiras. A Seção 22.1 mostra como os grafos podem ser representados na memória do computador.

Um ***grafo orientado*** G é um par (V, E), onde V é um conjunto finito e E é uma relação binária em V. O conjunto V é chamado ***conjunto de vértices*** de G, e seus elementos são chamados ***vértices***. O conjunto E é chamado ***conjunto de arestas*** de G, e seus elementos são chamados ***arestas***. A Figura B.2(a) é uma representação pictórica de um grafo orientado sobre o conjunto de vértices $\{1, 2, 3, 4, 5, 6\}$. Os vértices são representados por círculos na figura, e as arestas são representadas por setas. Observe que são possíveis ***autoloops*** – arestas de um vértice para ele próprio.

Em um ***grafo não orientado*** $G = (V, E)$, o conjunto de arestas E consiste em pares de vértices ***não ordenados***, em lugar de pares ordenados. Isto é, uma aresta é um conjunto $\{u, v\}$, onde $u, v \in V$ e $u \neq v$. Por convenção, usamos a notação (u, v) para uma aresta, em lugar da notação de conjuntos $\{u, v\}$, e (u, v) e (v, u) são consideradas a mesma aresta. Em um grafo não orientado, autoloops são proibidos, e assim toda aresta consiste em exatamente dois vértices distintos. A Figura B.2(b) é uma representação pictórica de um grafo não orientado no conjunto de vértices $\{1, 2, 3, 4, 5, 6\}$.

Muitas definições para grafos orientados e não orientados são idênticas, embora certos termos tenham significados ligeiramente diferentes nos dois contextos. Se (u, v) é uma aresta em um grafo orientado $G = (V, E)$, dizemos que (u, v) é ***incidente do*** ou ***sai do*** vértice u e é ***incidente no*** ou ***entra no*** vértice v. Por exemplo, as arestas que deixam o vértice 2 na Figura B.2(a) são $(2, 2)$, $(2, 4)$ e $(2, 5)$. As arestas que entram no vértice 2 são $(1, 2)$ e $(2, 2)$. Se (u, v) é uma aresta em um grafo não orientado $G = (V, E)$, dizemos que (u, v) é ***incidente nos*** vértices u e v. Na Figura B.2(b), as arestas incidentes no vértice 2 são $(1, 2)$ e $(2, 5)$.

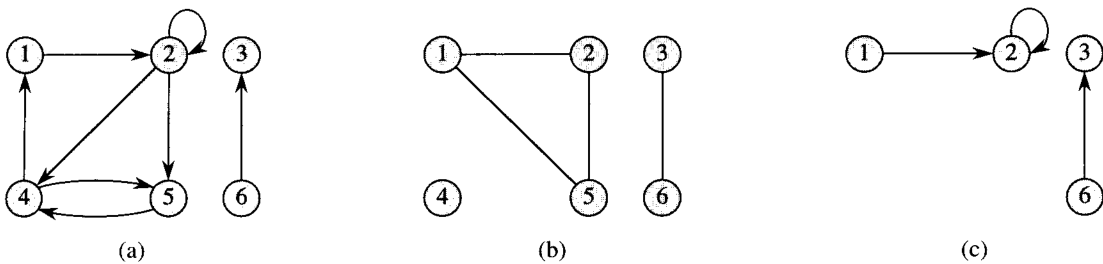


FIGURA B.2 Grafos orientados e não orientados. (**a**) Um grafo orientado $G = (V, E)$, onde $V = \{1, 2, 3, 4, 5, 6\}$ e $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. A aresta (2, 2) é um autoloop. (**b**) Um grafo não orientado $G = (V, E)$, onde $V = \{1, 2, 3, 4, 5, 6\}$ e $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. O vértice 4 é isolado. (**c**) O subgrafo do grafo da parte (a) induzido pelo conjunto de vértices {1, 2, 3, 6}

Se (u, v) é uma aresta em um grafo $G = (V, E)$, dizemos que o vértice v é ***adjacente*** ao vértice u. Quando o grafo é não orientado, a relação de adjacência é simétrica. Quando o grafo é orientado, a relação de adjacência não é necessariamente simétrica. Se v é adjacente a u em um grafo orientado, às vezes escrevemos $u \to v$. Nas partes (a) e (b) da Figura B.2, o vértice 2 é adjacente ao vértice 1, pois a aresta (1, 2) pertence a ambos os grafos. O vértice 1 *não* é adjacente ao vértice 2 na Figura B.2(a), pois a aresta (2, 1) não pertence ao grafo.

O ***grau*** de um vértice em um grafo não orientado é o número de arestas incidentes nele. Por exemplo, o vértice 2 na Figura B.2(b) tem grau 2. Em um grafo orientado, o ***grau de saída*** de um vértice é o número de arestas que saem dele, e o ***grau de entrada*** de um vértice é o número de arestas que entram nele. O ***grau*** de um vértice em um grafo orientado é seu grau de entrada somado a seu grau de saída. O vértice 2 na Figura B.2(a) tem grau de entrada 2, grau de saída 3 e grau 5.

Um ***caminho*** de ***comprimento*** k de um vértice u até um vértice u' em um grafo $G = (V, E)$ é uma seqüência $\langle v_0, v_1, v_2, .., v_k\rangle$ de vértices tais que $u = v_0$, $u' = v_k$ e (v_{i-1}, vi) para $i = 1, 2, ..., k$. O comprimento do caminho é o número de arestas no caminho. O caminho ***contém*** os vértices v_0, $v_1, \ldots v_k$ e as arestas $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. (Sempre existe um caminho de comprimento 0 de u até u.) Se existe um caminho p de u até u', dizemos que u' é ***acessível*** a partir de u via p, o que escrevemos algumas vezes como $u \overset{p}{\leadsto} u'$, se G é orientado. Um caminho é ***simples*** se todos os vértices no caminho são distintos. Na Figura B.2(a), o caminho ⟨1, 2, 5, 4⟩ é um caminho simples de comprimento 3. O caminho ⟨2, 5, 4, 5⟩ não é simples.

Um ***subcaminho*** do caminho $p = \langle v_0, v_1, .., v_k\rangle$ é uma subseqüência contígua de seus vértices. Isto é, para qualquer $0 \leq i \leq j \leq k$, a subseqüência de vértices $\langle v_i, v_{i+1}, .., v_j\rangle$ é um subcaminho de p.

Em um grafo orientado, um caminho $\langle v_0, v_1, .., v_k\rangle$ forma um ***ciclo*** se $v_0 = v_k$ e o caminho contém pelo menos uma aresta. O ciclo é ***simples*** se, além disso, $v_1, v_2, ..., v_k$ são distintos. Um autoloop é um ciclo de comprimento 1. Dois caminhos $\langle v_0, v_1, v_2, .., v_{k-1}, v_0\rangle$ e $\langle v'_0, v'_1, v'_2, \ldots, v'_{k-1}, v'_0, \rangle$ formam o mesmo ciclo se existe um inteiro j tal que $v'_i = v_{(i+j) \bmod k}$ para $i = 0, 1, ..., k-1$. Na Figura B.2(a), o caminho ⟨1, 2, 4, 1⟩ forma o mesmo ciclo que os caminhos ⟨2, 4, 1, 2⟩ e ⟨4, 1, 2, 4⟩. Esse ciclo é simples, mas o ciclo ⟨1, 2, 4, 5, 4, 1⟩ não é. O ciclo ⟨2, 2⟩ formado pela aresta (2, 2) é um autoloop. Um grafo orientado sem autoloops é ***simples***. Em um grafo não orientado, um caminho $\langle v_0, v_1, .., v_k\rangle$ forma um ***ciclo*** (***simples***) se k 3, $v_0 = v_k$ e $v_1, v_2, ..., v_k$ são distintos. Por exemplo, na Figura B.2(b), o caminho ⟨1, 2, 5, 1⟩ é um ciclo. Um grafo sem ciclos é ***acíclico***.

Um grafo não orientado é ***conectado*** se todo par de vértices está conectado por um caminho. Os ***componentes conectados*** de um grafo são as classes de equivalência de vértices sob a relação "é acessível a partir de". O grafo da Figura B.2(b) tem três componentes conectados: {1, 2, 5}, {3, 6} e {4}. Todo vértice em {1, 2, 5} é acessível a partir de cada um dos outros vértices em {1, 2, 5}. Um grafo não orientado é conectado se tem exatamente um componente conectado, isto é, se todo vértice é acessível a partir de todos os outros vértices.

Um grafo orientado é ***fortemente conectado*** se cada um de dois vértices quaisquer é acessível a partir do outro. Os ***componentes fortemente conectados*** de um grafo orientado são as classes de equivalência de vértices sob a relação "são mutuamente acessíveis". Um grafo orientado é fortemente conectado se ele só tem um componente fortemente conectado. O grafo da Figura B.2(a) tem três componentes fortemente conectados: $\{1, 2, 4, 5\}$, $\{3\}$ e $\{6\}$. Todos os pares de vértices em $\{1, 2, 4, 5\}$ são mutuamente acessíveis. Os vértices $\{3, 6\}$ não formam um componente fortemente conectado, pois o vértice 6 não pode ser alcançado a partir do vértice 3.

Dois grafos $G = (V, E)$ e $G' = (V', E')$ são ***isomórficos*** se existe uma bijeção $f : V \rightarrow V'$ tal que $(u, v) \in E$ se e somente se $(f(u), f(v)) \in E'$. Em outras palavras, podemos identificar novamente os vértices de G como vértices de G', mantendo as arestas correspondentes em G e G'. A Figura B.3(a) mostra um par de grafos isomórficos G e G' com os conjuntos de vértices respectivos $V = \{1, 2, 3, 4, 5, 6\}$ e $V' = \{u, v, w, x, y, z\}$. O mapeamento de V para V' dado por $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$ é a função bijetora exigida. Os grafos da Figura B.3(b) não são isomórficos. Embora ambos os grafos tenham 5 vértices e 7 arestas, o grafo superior tem um vértice de grau 4, e o grafo inferior não.

Dizemos que um grafo $G' = (V', E')$ é um ***subgrafo*** de $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$. Dado um conjunto $V' \subseteq V$, o subgrafo de G ***induzido*** por V' é o grafo $G' = (V', E')$, onde

$$E' = \{(u, v) \in E : u, v \in V'\} .$$

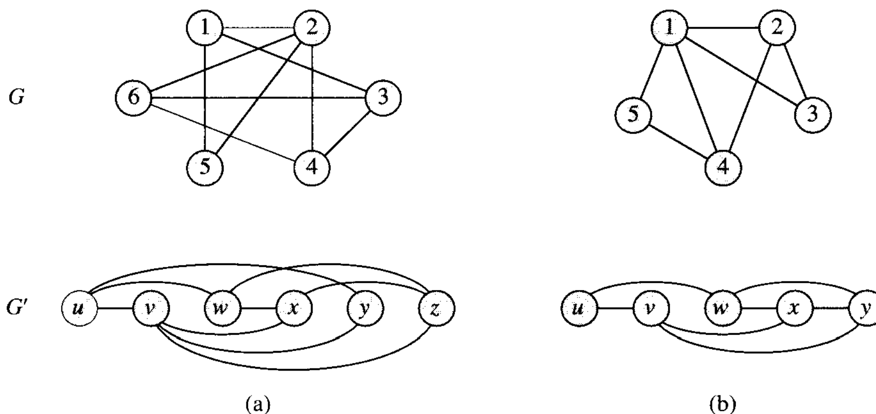


FIGURA B.3 **(a)** Um par de grafos isomórficos. Os vértices do grafo superior são mapeados para os vértices do grafo inferior por $f(1) = u, f(2) = v, f(3) = w, f(4) = x, f(5) = y, f(6) = z$. **(b)** Dois grafos que não são isomórficos, pois o grafo superior tem um vértice de grau 4 e o grafo inferior não tem

O subgrafo induzido pelo conjunto de vértices $\{1, 2, 3, 6\}$ na Figura B.2(a) aparece na Figura B.2(c) e tem o conjunto de arestas $\{(1, 2), (2, 2), (6, 3)\}$.

Dado um grafo não orientado $G = (V, E)$, a ***versão orientada*** de G é o grafo orientado $G' = (V, E')$, onde $(u, v) \in E'$ se e somente se $(u, v) \in E$. Ou seja, cada aresta não orientada (u, v) em G é substituída na versão orientada pelas duas arestas orientadas (u, v) e (v, u). Dado um grafo orientado $G = (V, E)$, a ***versão não orientada*** de G é o grafo não orientado $G' = (V, E')$, onde $(u, v) \in E'$ se e somente se $u \neq v$ e $(u, v) \in E$. Isto é, a versão não orientada contém as arestas de G "com suas orientações removidas" e com autoloops eliminados. (Como (u, v) e (v, u) são a mesma aresta em um grafo não orientado, a versão não orientada de um grafo orientado a contém somente uma vez, mesmo que o grafo orientado contenha ambas as arestas (u, v) e (v, u).) Em um grafo orientado $G = (V, E)$, um ***vizinho*** de um vértice u é qualquer vértice que seja adjacente a u na versão não orientada de G. Ou seja, v é um vizinho de u se $(u, v) \in E$ ou $(v, u) \in E$. Em um grafo não orientado, u e v são vizinhos se são adjacentes.

Vários tipos de grafos recebem nomes especiais. Um ***grafo completo*** é um grafo não orientado no qual todo par de vértices é adjacente. Um ***grafo bipartido*** é um grafo não orientado G

$= (V, E)$ em que V pode ser particionado em dois conjuntos V_1 e V_2 tais que $(u, v) \in E$ implica ou $u \in V_1$ e $v \in V_2$ ou $u \in V_2$ e $v \in V_1$. Ou seja, todas as arestas ficam entre os dois conjuntos V_1 e V_2. Um grafo acíclico não orientado é uma ***floresta***, e um grafo conectado acíclico não orientado é uma ***árvore (livre)*** (ver Seção B.5). Com freqüência, usaremos as primeiras letras de "grafo acíclico orientado", e chamaremos tal grafo um ***gao***.

Há duas variantes de grafos que você poderá encontrar ocasionalmente. Um ***multigrafo*** é semelhante a um grafo não orientado, mas pode ter várias arestas entre vértices e também auto-loops. Um ***hipergrafo*** é semelhante a um grafo não orientado, mas cada ***hiperaresta***, em lugar de conectar dois vértices, conecta um subconjunto arbitrário de vértices. Muitos algoritmos escritos para grafos orientados e não orientados comuns podem ser adaptados para funcionar nessas estruturas semelhantes a grafos.

A ***contração*** de um grafo não orientado $G = (V, E)$ por uma aresta $e = (u, v)$ é um grafo $G' = (V', E')$, onde $V' = V - \{u, v\} \cup \{x\}$ e x é um novo vértice. O conjunto de arestas E' é formado a partir de E pela eliminação da aresta (u, v) e, para cada vértice w incidente em u ou v, eliminando-se o par dentre (u, w) e (v, w) que esteja em E e adicionando-se a nova aresta (x, w).

Exercícios

B.4-1

Os participantes de uma festa na faculdade apertam as mãos para se cumprimentar, e cada professor memoriza quantas vezes tem de apertar as mãos de outras pessoas. No final da festa, o chefe de departamento efetua a soma do número de vezes que cada professor teve de apertar as mãos de outros. Mostre que o resultado é par, provando o ***lema do cumprimento***: se $G = (V, E)$ é um grafo não orientado, então

$$\sum_{v \in V} \text{grau}(v) = 2|E| .$$

B.4-2

Mostre que, se um grafo orientado ou não orientado contém um caminho entre dois vértices u e v, então ele contém um caminho simples entre u e v. Mostre que, se um grafo orientado contém um ciclo, então ele contém um ciclo simples.

B.4-3

Mostre que qualquer grafo conectado não orientado $G = (V, E)$ satisfaz a $|E| \geq |V| - 1$.

B.4-4

Verifique se, em um grafo não orientado, a relação "é acessível a partir de" é uma relação de equivalência sobre os vértices do grafo. Qual das três propriedades de uma relação de equivalência é válida em geral para a relação "é acessível a partir de" sobre os vértices de um grafo orientado?

B.4-5

Qual é a versão não orientada do grafo orientado da Figura B.2(a)? Qual é a versão orientada do grafo não orientado da Figura B.2(b)?

B.4-6 ★

Mostre que um hipergrafo pode ser representado por um grafo bipartido, se permitimos que a incidência no hipergrafo corresponda à adjacência no grafo bipartido. (*Sugestão*: Faça um conjunto de vértices no grafo bipartido corresponder a vértices do hipergrafo e faça o outro conjunto de vértices do grafo bipartido corresponder a hiperarestas.)

B.5 Árvores

Como ocorre no caso dos grafos, existem muitas noções de árvores relacionadas entre si, embora ligeiramente diferentes. Esta seção apresenta definições e propriedades matemáticas de vários tipos de árvores. As Seções 10.4 e 22.1 descrevem como as árvores podem ser representadas na memória de um computador.

B.5.1 Árvores livres

Como definimos na Seção B.4, uma ***árvore livre*** é um grafo acíclico não orientado conectado. Com freqüência, omitimos o adjetivo "livre" quando dizemos que um grafo é uma árvore. Se um grafo não orientado é acíclico mas possivelmente desconectado, ele é uma ***floresta***. Muitos algoritmos que funcionam para árvores também funcionam para florestas. A Figura B.4(a) mostra uma árvore livre e a Figura B.4(b) mostra uma floresta. A floresta da Figura B.4(b) não é uma árvore porque não é conectada. O grafo da Figura B.4(c) não é nem uma árvore nem uma floresta, porque contém um ciclo.

O teorema a seguir engloba muitos fatos importantes sobre árvores livres.

Teorema B.2 (Propriedades de árvores livres)
Seja $G = (V, E)$ um grafo não orientado. As declarações a seguir são equivalentes.

1. G é uma árvore livre.
2. Dois vértices quaisquer em G estão conectados por um caminho simples único.
3. G é conectado mas, se qualquer aresta for removida de E, o grafo resultante será desconectado.
4. G é conectado, e $|E| = |V| - 1$.
5. G é acíclico, e $|E| = |V| - 1$.
6. G é acíclico mas, se qualquer aresta for adicionada a E, o grafo resultante conterá um ciclo.

Prova (1) $\Rightarrow$ (2): Tendo em vista que uma árvore é conectada, dois vértices quaisquer em G são conectados por pelo menos um caminho simples. Sejam u e v vértices que estão conectados por dois caminhos simples distintos p_1 e p_2, como mostra a Figura B.5. Seja w o vértice no qual os caminhos divergem pela primeira vez; isto é, w é o primeiro vértice em p_1 e p_2 cujo sucessor em p_1 é x e cujo sucessor em p_2 é y, onde $x \neq y$. Seja z o primeiro vértice no qual os caminhos voltam a convergir; ou seja, z é o primeiro vértice seguinte a w em p_1 que também está em p_2. Seja p' o subcaminho de p_1 a partir de w através de x até z, e seja p'' o subcaminho de p_2 a partir de w através de y até z. Os caminhos p' e p'' não compartilham nenhum vértice, exceto seus pontos extremos. Portanto, o caminho obtido pela concatenação de p' com o inverso de p'' é um ciclo. Isso contradiz nossa hipótese de que G é uma árvore. Desse modo, se G é uma árvore, pode haver no máximo um caminho simples entre dois vértices.

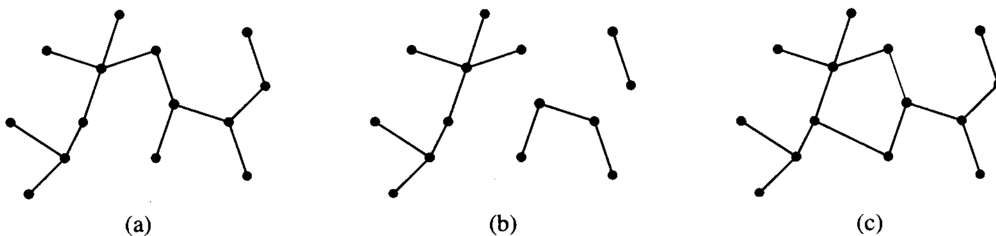


FIGURA B.4 **(a)** Uma árvore livre. **(b)** Uma floresta. **(c)** Um grafo que contém um ciclo e que, por essa razão, não é uma árvore nem uma floresta

(2) $\Rightarrow$ (3): Se dois vértices quaisquer em G estão conectados por um caminho simples único, então G é conectado. Seja (u, v) qualquer aresta em E. Essa aresta é um caminho de u até v e, portanto, deve ser o caminho único de u até v. Se removermos (u, v) de G, não haverá nenhum caminho de u até v e, conseqüentemente, sua remoção desconecta G.

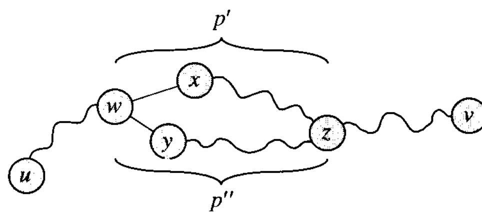


FIGURA B.5 Uma etapa na prova do Teorema B.2: se (1) G é uma árvore livre, então (2) dois vértices quaisquer em G estão conectados por um caminho simples único. Suponha para fins de contradição que os vértices u e v estejam conectados por dois caminhos simples distintos p_1 e p_2. Esses caminhos divergem primeiro no vértice w, e depois voltam a convergir primeiro no vértice z. O caminho p' concatenado com o inverso do caminho p'' forma um ciclo, o que produz a contradição

(3) ⇒ (4): Por hipótese, o grafo G é conectado e, pelo Exercício B.4-3, temos $|E| \geq |V| - 1$. Provaremos $|E| \leq |V| - 1$ por indução. Um grafo conectado com $n = 1$ ou $n = 2$ vértices tem $n - 1$ arestas. Suponha que G tenha $n \geq 3$ vértices e que todos os grafos que satisfazem a (3) com menos de n vértices também satisfazem a $|E| \leq |V| - 1$. A remoção de uma aresta arbitrária de G separa o grafo em $k \geq 2$ componentes conectados (na realidade, $k = 2$). Cada componente satisfaz a (3) ou, do contrário, G não satisfaria a (3). Assim, por indução, o número de arestas em todos os componentes combinados é no máximo $|V| - k \geq |V| - 2$. A adição da aresta removida produz $|E| \leq |V| - 1$.

(4) ⇒ (5): Suponha que G seja conectado e que $|E| = |V| - 1$. Devemos mostrar que G é acíclico. Vamos supor que G tenha um ciclo contendo k vértices $v_1, v_2, ..., v_k$ e, sem perda de generalidade, suponha que esse ciclo seja simples. Seja $G_k = (V_k, E_k)$ o subgrafo de G que consiste no ciclo. Observe que $|V_k| = |E_k| = k$. Se $k < |V|$, deve existir um vértice $v_{k+1} \in V - V_k$ que seja adjacente a algum vértice $v_i \in V_k$, pois G é conectado. Defina $G_{k+1} = (V_{k+1}, E_{k+1})$ como o subgrafo de G com $V_{k+1} = V_k \cup \{v_{k+1}\}$ e $E_{k+1} = E_k \cup \{v_i, v_{k+1}\}$. Observe que $|v_{k+1}| = |E_{k+1}|$. Se $k + 1 < n$, podemos continuar, definindo G_{k+2} da mesma maneira, e assim por diante até obtermos $G_n = (V_n, E_n)$, onde $n = |V|$, $V_n = V$ e $|E_n| = |V_n| = |V|$. Como G_n é um subgrafo de G, temos $E_n \subseteq E$ e, portanto, $|E| \geq |V|$, o que contradiz a hipótese de que $|E| = |V| - 1$. Desse modo, G é acíclico.

(5) ⇒ (6): Suponha que G seja acíclico e que $|E| = |V| - 1$. Seja k o número de componentes conectados de G. Cada componente conectado é uma árvore livre por definição e, como (1) implica (5), a soma de todas as arestas em todos os componentes conectados de G é $|V| - k$. Conseqüentemente, devemos ter $k = 1$, e G é de fato uma árvore. Como (1) implica (2), dois vértices quaisquer em G estão conectados por um caminho simples único. Portanto, a adição de qualquer aresta a G cria um ciclo.

(6) ⇒ (1): Suponha que G seja acíclico mas que, se qualquer aresta for adicionada a E, seja criado um ciclo. Devemos mostrar que G é conectado. Sejam u e v vértices arbitrários em G. Se u e v ainda não forem adjacentes, a adição da aresta (u, v) criará um ciclo no qual todas as arestas com exceção de (u, v) pertencerão a G. Desse modo, existe um caminho de u até v e, como u e v foram escolhidos arbitrariamente, G é conectado. ∎

B.5.2 Árvores enraizadas e ordenadas

Uma ***árvore enraizada*** é uma árvore livre na qual um dos vértices se distingue dos outros. O vértice distinto é chamado ***raiz*** da árvore. Com freqüência, faremos referência a um vértice de uma árvore enraizada como um ***nó***[4] da árvore. A Figura B.6(a) mostra uma árvore enraizada em um conjunto de 12 nós com raiz 7.

[4] O termo "nó" (ou "nodo") é usado com freqüência na literatura sobre a teoria dos grafos como sinônimo de "vértice". Reservaremos o termo "nó" para indicar um vértice de uma árvore enraizada.

Considere um nó x em uma árvore enraizada T com raiz r. Qualquer nó y no caminho único de r a x é chamado ***ancestral*** de x. Se y é um ancestral de x, então x é um ***descendente*** de y. (Todo nó é ao mesmo tempo um ancestral e um descendente de si próprio.) Se y é um ancestral de x e $x \neq y$, então y é um ***ancestral próprio*** de x, e x é um ***descendente próprio*** de y. A ***subárvore enraizada em x*** é a árvore induzida pelos descendentes de x, com raiz em x. Por exemplo, a subárvore enraizada no nó 8 na Figura B.6(a) contém os nós 8, 6, 5 e 9.

Se a última aresta no caminho da raiz r de uma árvore T até um nó x é (y, x), então y é o ***pai*** de x, e x é um ***filho*** de y. A raiz é o único nó em T que não tem pai. Se dois nós têm o mesmo pai, eles são ***irmãos***. Um nó sem filhos é um ***nó externo*** ou ***folha***. Um nó que não é uma folha é um ***nó interno***.

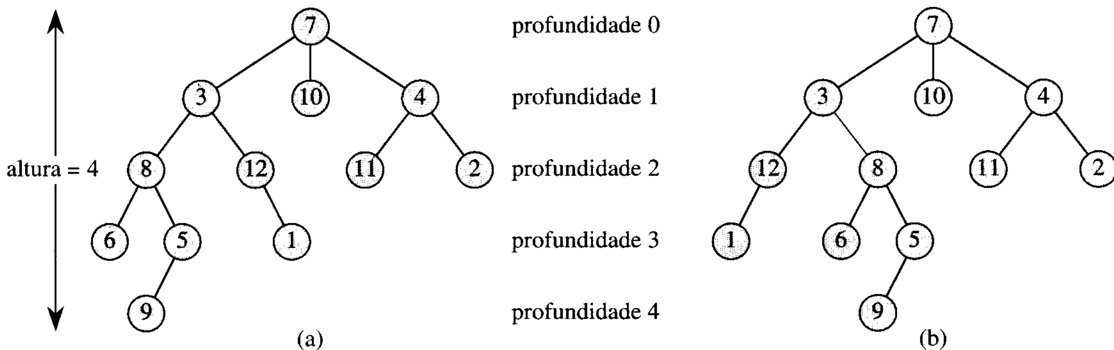


FIGURA B.6 Árvores enraizadas e ordenadas. **(a)** Uma árvore enraizada com altura 4. A árvore é desenhada de um modo padrão: a raiz (nó 7) está na parte superior, seus filhos (os nós com profundidade 1) estão abaixo dela, os filhos dos filhos (nós com profundidade 2) estão abaixo deles e assim por diante. Se a árvore é ordenada, a ordem relativa da esquerda para a direita dos filhos de um nó é importante; caso contrário, ela não é importante. **(b)** Outra árvore enraizada. Sendo uma árvore enraizada, ela é idêntica à árvore em (a) mas, como árvore ordenada é diferente, pois os filhos do nó 3 aparecem em uma ordem distinta

O número de filhos de um nó x em uma árvore enraizada T é chamado ***grau*** de x.[5] O comprimento do caminho desde a raiz r até um nó x é a ***profundidade*** de x em T. A ***altura*** de um nó em uma árvore é o número de arestas no caminho descendente simples mais longo desde o nó até uma folha, e a altura de uma árvore é a altura de sua raiz. A altura de uma árvore também é igual à maior profundidade de qualquer nó na árvore.

Uma ***árvore ordenada*** é uma árvore enraizada na qual os filhos de cada nó estão ordenados. Isto é, se um nó tem k filhos, então existe um primeiro filho, um segundo filho, ... e um k-ésimo filho. As duas árvores da Figura B.6 são diferentes quando consideradas árvores ordenadas, mas são idênticas quando são consideradas apenas árvores enraizadas.

B.5.3 Árvores binárias e árvores posicionais

As árvores binárias são definidas de modo recursivo. Uma ***árvore binária*** T é uma estrutura definida em um conjunto finito de nós que

- não contém nenhum nó, ou
- é formada por três conjuntos disjuntos de nós: um nó ***raiz***, uma árvore binária chamada sua ***subárvore da esquerda***, e uma árvore binária chamada sua ***subárvore da direita***.

[5] Observe que o grau de um nó depende de T ser considerada uma árvore enraizada ou uma árvore livre. O grau de um vértice em uma árvore livre é, como em qualquer grafo não orientado, o número de vértices adjacentes. Porém, em uma árvore enraizada, o grau é o número de filhos – o pai de um nó não conta para definir seu grau.

A árvore binária que não contém nenhum nó é chamada ***árvore vazia*** ou ***árvore nula***, algumas vezes denotada por NIL. Se a subárvore da esquerda é não vazia, sua raiz é chamada ***filho da esquerda*** da raiz da árvore inteira. Da mesma forma, a raiz de uma subárvore da direita não nula é o ***filho da direita*** da raiz da árvore inteira. Se uma subárvore é a árvore nula NIL, dizemos que o filho está ***ausente*** ou ***faltando***. A Figura B.7(a) mostra uma árvore binária.

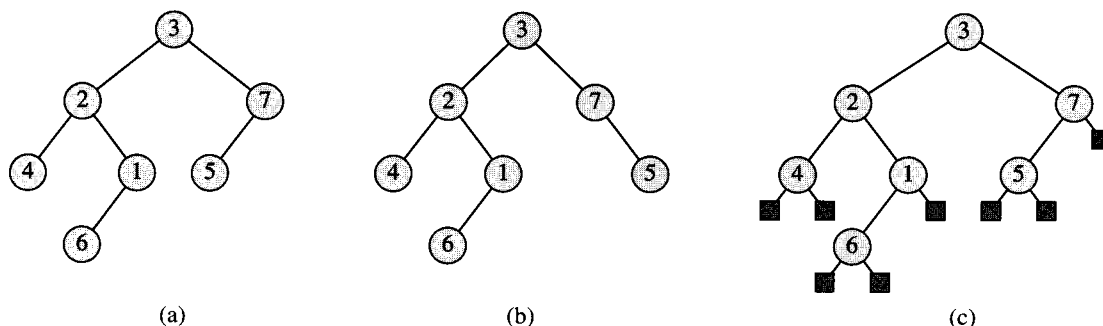


FIGURA B.7 Árvores binárias. **(a)** Uma árvore binária desenhada de um modo padrão. O filho da esquerda de um nó é desenhado abaixo do nó e à esquerda. O filho da direita é desenhado abaixo e à direita do nó. **(b)** Uma árvore binária diferente da que está em (a). Em (a), o filho da esquerda do nó 7 é 5, e o filho da direita está ausente. Em **(b)**, o filho da esquerda do nó 7 está ausente e o filho da direita é 5. Como árvores ordenadas, essas árvores são idênticas mas, como árvores binárias, elas são distintas. **(c)** A árvore binária em (a) representada pelos nós internos de uma árvore binária completa: uma árvore ordenada na qual cada nó interno tem grau 2. As folhas na árvore são mostradas como quadrados

Uma árvore binária não é simplesmente uma árvore ordenada na qual cada nó tem grau máximo 2. Por exemplo, em uma árvore binária, se um nó tem apenas um filho, a posição do filho – seja ele o ***filho da esquerda*** ou o ***filho da direita*** – é importante. Em uma árvore ordenada, não há distinção de um único filho como da esquerda ou da direita. A Figura B.7(b) mostra uma árvore binária que difere da árvore da Figura B.7(a) por causa da posição de um único nó. Contudo, considerando-se as árvores ordenadas, as duas árvores são idênticas.

As informações de posicionamento em uma árvore binária podem ser representadas pelos nós internos de uma árvore ordenada, como mostra a Figura B.7(c). A idéia é substituir cada filho que falta na árvore binária por um nó que não tem nenhum filho. Esses nós folhas estão desenhados como quadrados na figura. A árvore resultante é uma ***árvore binária completa***: cada nó é ou uma folha ou tem grau exatamente igual a 2. Não existe nenhum nó de grau 1. Em conseqüência disso, a ordem dos filhos de um nó preserva as informações de posição.

As informações de posicionamento que distinguem árvores binárias de árvores ordenadas podem ser estendidas a árvores com mais de 2 filhos por nó. Em uma ***árvore posicional***, os filhos de um nó são identificados com inteiros positivos distintos. O i-ésimo filho de um nó é ***ausente*** se nenhum filho é identificado com o inteiro i. Uma árvore ***k-ária*** é uma árvore posicional na qual, para todo nó, todos os filhos com rótulos maiores que k estão faltando. Desse modo, uma árvore binária é uma árvore k-ária com $k = 2$.

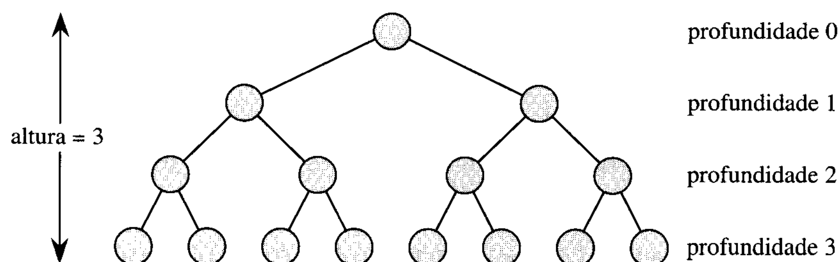


FIGURA B.8 Uma árvore binária completa de altura 3 com 8 folhas e 7 nós internos

Uma ***árvore k-ária completa*** é uma árvore k-ária na qual todas as folhas têm a mesma profundidade e todos os nós internos têm grau k. A Figura B.8 mostra uma árvore binária completa de altura 3. Quantas folhas tem uma árvore k-ária completa de altura h? A raiz tem k filhos à profundidade 1, cada um dos quais tem k filhos à profundidade 2 e assim por diante. Portanto, o número de folhas à profundidade h é k^h. Conseqüentemente, a altura de uma árvore k-ária completa com n folhas é $\log_k n$. O número de nós internos de uma árvore k-ária completa de altura h é

$$\begin{aligned} 1 + k + k^2 + \ldots + k^{h-1} &= \sum_{i=0}^{h-1} k^i \\ &= \frac{k^h - 1}{k - 1} \end{aligned}$$

pela equação (A.5). Assim, uma árvore binária completa tem $2^h - 1$ nós internos.

Exercícios

B.5-1

Desenhe todas as árvores livres compostas pelos 3 vértices A, B e C. Trace todas as árvores enraizadas com nós A, B e C, que têm A como raiz. Desenhe todas as árvores ordenadas com nós A, B e C que têm A como raiz. Trace todas as árvores binárias com nós A, B e C que têm A como raiz.

B.5-2

Seja $G = (V, E)$ um grafo acíclico orientado no qual existe um vértice $v_0 \in V$ tal que existe um caminho único de v_0 até todo vértice $v \in V$. Prove que a versão não orientada de G forma uma árvore.

B.5-3

Mostre por indução que o número de nós de grau 2 em qualquer árvore binária não vazia é uma unidade menor que o número de folhas.

B.5-4

Mostre por indução que uma árvore binária não vazia com n nós tem altura pelo menos igual a $\lfloor \lg n \rfloor$.

B.5-5 ★

O ***comprimento do caminho interno*** de uma árvore binária completa é a soma, considerada sobre todos os nós internos da árvore, da profundidade de cada nó. Da mesma forma, o ***comprimento do caminho externo*** é a soma, considerada sobre todas as folhas da árvore, da profundidade de cada folha. Seja uma árvore binária completa com n nós internos, comprimento de caminho interno i e comprimento de caminho externo e. Prove que $e = i + 2n$.

B.5-6 ★

Vamos associar um "peso" $w(x) = 2^{-d}$ a cada folha x de profundidade d em uma árvore binária T. Prove que $\sum_x w(x) \leq 1$, onde a soma é considerada sobre todas as folhas x de T. (Isso é conhecido como ***desigualdade de Kraft***.)

B.5-7 ★

Mostre que toda árvore binária com L folhas contém uma subárvore que tem entre $L/3$ e $2L/3$ folhas, inclusive.

Problemas

B-1 Coloração de grafos

Uma ***k-coloração*** de um grafo não orientado $G = (V, E)$ é uma função $c : V \to \{0, 1, \ldots, k - 1\}$ tal que $c(u) \neq c(v)$ para toda aresta $(u, v) \in E$. Em outras palavras, os números $0, 1, \ldots, k - 1$ representam as k cores, e vértices adjacentes devem ter cores diferentes.

a. Mostre que qualquer árvore pode ter 2 cores.

b. Mostre que os itens seguintes são equivalentes:

1. G é bipartido.
2. G pode ter duas cores.
3. G não tem nenhum ciclo de comprimento ímpar.

c. Seja d o grau máximo de qualquer vértice em um grafo G. Prove que G pode ser colorido com $d + 1$ cores.

d. Mostre que, se G tem $O(|V|)$ arestas, então G pode ser colorido com $O(\sqrt{|V|})$ cores.

B-2 Grafos amistosos

Reformule cada uma das instruções a seguir como um teorema sobre grafos não orientados, e depois prove-o. Suponha que esse caráter amistoso seja simétrico, mas não reflexivo.

a. Em qualquer grupo de $n \geq 2$ pessoas, existem duas pessoas com o mesmo número de amigos no grupo.

b. Todo grupo de seis pessoas contém três amigos mútuos ou três estranhos mútuos.

c. Qualquer grupo de pessoas pode ser particionado em dois subgrupos tais que pelo menos metade dos amigos de cada pessoa pertence ao subgrupo do qual essa pessoa *não* é um membro.

d. Se toda pessoa em um grupo é amiga de pelo menos metade das pessoas no grupo, então o grupo pode se sentar em torno de uma mesa de tal modo que toda pessoa fique sentada entre dois amigos.

B-3 Bisseção de árvores

Muitos algoritmos de dividir e conquistar que operam sobre grafos exigem que o grafo seja dividido em dois subgrafos de tamanho praticamente igual, que são induzidos por uma partição dos vértices. Este problema investiga a bisseção de árvores formadas pela remoção de um pequeno número de arestas. Exigimos que, sempre que dois vértices terminarem na mesma subárvore depois que as arestas forem removidas, eles estejam obrigatoriamente na mesma partição.

a. Mostre que, removendo uma única aresta, podemos particionar os vértices de qualquer árvore binária de n vértices em dois conjuntos A e B tais que $|A| \leq 3n/4$ e $|B| \leq 3n/4$.

b. Mostre que a constante 3/4 na parte (a) é ótima no pior caso, dando um exemplo de uma árvore simples cuja partição balanceada de modo mais uniforme após a remoção de uma única aresta tem $|A| = 3n/4$.

c. Mostre que, removendo no máximo $O(\lg n)$ arestas, podemos particionar os vértices de qualquer árvore binária de n vértices em dois conjuntos A e B tais que $|A| = \lfloor n/2 \rfloor$ e $|B| = \lceil n/2 \rceil$.

Notas do capítulo

G. Boole foi o pioneiro no desenvolvimento da lógica simbólica e introduziu muitas das notações básicas de conjuntos em um livro publicado em 1854. A moderna teoria dos conjuntos foi criada por G. Cantor durante o período de 1874 a 1895. Cantor focalizou principalmente os conjuntos de cardinalidade infinita. O termo "função" é atribuído a G. W. Leibnitz, que o usou para se referir a várias espécies de fórmulas matemáticas. Sua definição limitada foi generalizada várias vezes. A teoria dos grafos teve origem em 1736, quando L. Euler provou que era impossível cruzar cada uma das sete pontes da cidade de Königsberg exatamente uma vez e retornar ao ponto de partida.

Um compêndio útil com muitas definições e resultados da teoria dos grafo é o livro de Harary [138].

Bibliografia

1. Milton Abramowitz e Irene A. Stegun, editores. *Handbook of Mathematical Functions*. Dover, 1965.
2. G. M. Adel'son-Vel'skiĭ e E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259-1263, 1962.
3. Leonard M. Adleman, Carl Pomerance e Robert S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics*, 117:173-206, 1983.
4. Alok Aggarwal e Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116-1127, 1988.
5. Alfred V. Aho, John F. Hopcroft e Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
6. Alfred V. Aho, John E. Hopcroft e Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
7. Ravindra K. Ahuja, Thomas L. Magnanti e James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
8. Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin e Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37:213-223, 1990.
9. Ravindra K. Ahuja e James B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748-759, 1989.
10. Ravindra K. Ahuja, James B. Orlin e Robert E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5):939-954, 1989.
11. Miklos Ajtai, János Komlós e Endre Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1-19, 1983.
12. Miklos Ajtai, Nimrod Megiddo e Orli Waarts. Improved algorithms and analysis for secretary problems and generalizations. Em *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pp. 473-482, 1995.
13. Mohamad Akra e Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195-210, 1998.
14. Noga Alon. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters*, 35:201-204, 1990.
15. Arne Andersson. Balanced search trees made simple. Em *Proceedings of the Third Workshop on Algorithms and Data Structures*, número 709 em Lecture Notes in Computer Science, pp. 60-71. Springer-Verlag, 1993.
16. Arne Andersson. Faster deterministic sorting and searching in linear space. Em *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pp. 135-141, 1996.
17. Arne Andersson, Torben Hagerup, Stefan Nilsson e Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57:74-93, 1998.
18. Tom M. Apostol. *Calculus*, volume 1. Blaisdell Publishing Company, segunda edição, 1967.
19. Sanjeev Arora. *Probabilistic checking of proofs and the hardness of approximation problems*. Tese de doutorado, University of California, Berkeley, 1994.
20. Sanjeev Arora. The approximability of NP-hard problems. Em *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pp. 337-348, 1998.
21. Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753-782, 1998.
22. Sanjeev Arora e Carsten Lund. Hardness of approximations. Em Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pp. 399-446. PWS Publishing Company, 1997.
23. Javed A. Aslam. A simple bound on the expected height of a randomly built binary search tree. Relatório técnico TR2001-387, Dartmouth College Department of Computer Science, 2001.
24. Mikhail J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.
25. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela e M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, 1999.
26. Sara Baase e Alan Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, terceira edição, 2000.
27. Eric Bach. Comunicação particular, 1989.
28. Eric Bach. Number-theoretic algorithms. Em *Annual Review of Computer Science*, volume 4, pp. 119-172. Annual Reviews, Inc., 1990.
29. Eric Bach e Jeffery Shallit. *Algorithmic Number Theory – Volume I: Efficient Algorithms*. The MIT Press, 1996.
30. David H. Bailey, King Lee e Horst D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4):357-371, 1990.
31. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290-306, 1972.
32. R. Bayer e E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173-189, 1972.
33. Pierre Beauchemin, Gilles Brassard, Claude Crépeau, Claude Goutier e Carl Pomerance. The generation of random numbers that are probably prime. *Journal of Cryptology*, 1:53-64, 1988.
34. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

35. Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics,* 16(1):87-90, 1958.
36. Michael Ben-Or. Lower bounds for algebraic computation trees. Em *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing,* pp. 80-86, 1983.
37. Michael A. Bender, Erik D. Demaine e Martin Farach-Colton. Cache-oblivious B-trees. Em *Proceedings of the 41st Annual Symposium on Foundations of Computer Science,* pp. 399-409, 2000.
38. Samuel W. Bent e John W. John. Finding the median requires 2n comparisons. Em *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing,* pp. 213- 216, 1985.
39. Jon L. Bentley. *Writing Efficient Programs.* Prentice-Hall, 1982.
40. Jon L. Bentley. *Programming Pearls.* Addison-Wesley, 1986.
41. Jon L. Bentley, Dorothea Haken e James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News,* 12(3):36-44, 1980.
42. Patrick Billingsley. *Probability and Measure.* John Wiley & Sons, segunda edição, 1986.
43. Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest e Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences,* 7(4):448-461, 1973.
44. Béla Bollobás. *Random Graphs.* Academic Press, 1985.
45. J. A. Bondy e U. S. R. Murty. *Graph Theory with Applications.* American Elsevier, 1976.
46. Gilles Brassard e Paul Bratley. *Algorithmics: Theory and Practice.* Prentice-Hall, 1988.
47. Gilles Brassard e Paul Bratley. *Fundamentals of Algorithmics.* Prentice Hall, 1996.
48. Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT,* 20(2):176-184, 1980.
49. Mark R. Brown. *The Analysis of a Practical and Nearly Optimal Priority Queue.* Tese de PhD, Computer Science Department, Stanford University, 1977. Relatório técnico STAN-CS-77-600.
50. Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing,* 7(3):298-319, 1978.
51. J. P. Buhler, H. W. Lenstra, Jr. e Carl Pomerance. Factoring integers with the number field sieve. Em A. K. Lenstra e H. W. Lenstra, Jr., editores, *The Development of the Number Field Sieve,* volume 1554 of *Lecture Notes in Mathematics,* pp. 50-94. Springer-Verlag, 1993.
52. J. Lawrence Carter e Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences,* 18(2): 143-154, 1979.
53. Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM,* 47(6): 1028-1047, 2000.
54. Joseph Cheriyan e Torben Hagerup. A randomized maximum-flow algorithm. *SIAM Journal on Computing,* 24(2):203-226, 1995.
55. Joseph Cheriyan e S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing,* 18(6):1057-1086, 1989.
56. Boris V. Cherkassky e Andrew V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica,* 19(4):390-410, 1997.
57. Boris V. Cherkassky, Andrew V. Goldberg e Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming,* 73(2): 129-174, 1996.
58. Boris V. Cherkassky, Andrew V. Goldberg e Craig Silverstein. Buckets, heaps, lists and monotone priority queues. *SIAM Journal on Computing,* 28(4): 1326-1346, 1999.
59. H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics,* 23:493-507, 1952.
60. Kai Lai Chung. *Elementary Probability Theory with Stochastic Processes.* Springer-Verlag, 1974.
61. V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research,* 4(3):233-235, 1979.
62. V. Chvátal. *Linear Programming.* W. H. Freeman and Company, 1983.
63. V. Chvátal, D. A. Klarner e D. E. Knuth. Selected combinatorial research problems. Relatório técnico STAN-CS-72-292, Computer Science Department, Stanford University, 1972.
64. Alan Cobham. The intrinsic computational difficulty of functions. Em *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science,* pp. 24-30. North-Holland, 1964.
65. H. Cohen e H. W. Lenstra, Jr. Primality testing and Jacobi sums. *Mathematics of Computation,* 42(165):297-330, 1984.
66. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys,* 11(2): 121-137, 1979.
67. Stephen Cook. The complexity of theorem proving procedures. Em *Proceedings of the Third Annual ACM Symposium on Theory of Computing,* pp. 151-158, 1971.
68. James W. Cooley e John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation,* 19(90):297-301, 1965.
69. Don Coppersmith. Modifications to the number field sieve. *Journal of Cryptology,* 6:169-180, 1993.
70. Don Coppersmith e Shmuel Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation,* 9(3):251-280, 1990.
71. Thomas H. Cormen. *Virtual Memory for Data-Parallel Computing.* Tese de doutorado, Department of Electrical Engineering and Computer Science, MIT, 1992.
72. Eric V. Denardo e Bennett L. Fox. Shortest-route methods: 1. Reaching, pruning, and buckets. *Operations Research,* 27(1):161-186, 1979.
73. Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert e Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing,* 23(4):738-761, 1994.
74. Whitfield Diffie e Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory,* IT-22(6):644-654, 1976.
75. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik,* 1:269-271, 1959.
76. E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady,* 11(5):1277-1280, 1970.

77. Brandon Dixon, Monika Rauch e Robert E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184-1192, 1992.
78. John D. Dixon. Factorization and primality tests. *The American Mathematical Monthly*, 91(6):333-352, 1984.
79. Dorit Dor e Un Zwick. Selecting the median. Em *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms*, pp. 28-37, 1995.
80. Alvin W. Drake. *Fundamentals of Applied Probability Theory*. McGraw-Hill, 1967.
81. James R. Driscoll, Harold N. Gabow, Ruth Shrairman e Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343-1354, 1988.
82. James R. Driscoll, Neil Sarnak, Daniel D. Sleator e Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86-124, 1989.
83. Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1987.
84. Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449-467, 1965.
85. Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126-136, 1971.
86. Jack Edmonds e Richard M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248-264, 1972.
87. Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.
88. William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, terceira edição, 1968.
89. Robert W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.
90. Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7:701, 1964.
91. Robert W. Floyd. Permuting information in idealized two-level storage. Em Raymond E. Miller e James W. Thatcher, editores, *Complexity of Computer Computations*, pp. 105- 109. Plenum Press, 1972.
92. Robert W. Floyd e Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165-172, 1975.
93. Lestor R. Ford, Jr. e D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
94. Lestor R. Ford, Jr. e Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66:387-389, 1959.
95. Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83-89, 1976.
96. Michael L. Fredman, János Komlós e Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538-544, 1984.
97. Michael L. Fredman e Michael E. Saks. The cell probe complexity of dynamic data structures. Em *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 1989.
98. Michael L. Fredman e Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596-615, 1987.
99. Michael L. Fredman e Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424-436, 1993.
100. Michael L. Fredman e Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533-551, 1994.
101. Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74:107-114, 2000.
102. Harold N. Gabow, Z. Galil, T. Spencer e Robert E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109-122, 1986.
103. Harold N. Gabow e Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209-221, 1985.
104. Harold N. Gabow e Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5): 1013-1036, 1989.
105. Zvi Galil e Oded Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2): 103-139, 1997.
106. Zvi Galil e Oded Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54(2):243-254, 1997.
107. Zvi Galil e Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280-294, 1983.
108. Igal Galperin e Ronald L. Rivest. Scapegoat trees. Em *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pp. 165-174, 1993.
109. Michael R. Garey, R. L. Graham e J. D. Ullman. Worst-case analysis of memory allocation algorithms. Em *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, pp. 143-150, 1972.
110. Michael R. Garey e David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
111. Saul Gass. *Linear Programming: Methods and Applications*. International Thomson Publishing, quarta edição, 1975.
112. Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180-187, 1972.
113. Alan George e Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.
114. E. N. Gilbert e E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38(4):933-967, 1959.
115. Michel X. Goemans e David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6): 1115-1145, 1995.

116. Michel X. Goemans e David P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. Em Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pp. 144-191. PWS Publishing Company, 1997.
117. Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. Tese de doutorado, Department of Electrical Engineering and Computer Science, MIT, 1987.
118. Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494-504, 1995.
119. Andrew V. Goldberg, Éva Tardos e Robert E. Tarjan. Network flow algorithms. Em Bernhard Korte, László Lovász, Hans Jurgen Prömel e Alexander Schrijver, editores, *Paths, Flows, and VLSI-Layout*, pp. 101-164. Springer-Verlag, 1990.
120. Andrew V. Goldberg e Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45:783-797, 1998.
121. Andrew V. Goldberg e Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921-940, 1988.
122. D. Goldfarb e M. J. Todd. Linear programming. Em G. L. Nemhauser, A. H. G. Rinnooy-Kan e M. J. Todd, editores, *Handbook in Operations Research and Management Science, Vol. 1, Optimization*, pp. 73-170. Elsevier Science Publishers, 1989.
123. Shafi Goldwasser e Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270-299, 1984.
124. Shafi Goldwasser, Silvio Micali e Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281-308, 1988.
125. Gene H. Golub e Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, terceira edição, 1996.
126. G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.
127. Rafael C. Gonzalez e Richard E. Woods. *Digital Image Processing*. Addison-Wesley, 1992.
128. Michael T. Goodrich e Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.
129. Ronald L. Graham. Bounds for certain multiprocessor anomalies. *Bell Systems Technical Journal*, 45:1563-1581, 1966.
130. Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132-133, 1972.
131. Ronald L. Graham e Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43-57, 1985.
132. Ronald L. Graham, Donald E. Knuth e Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, segunda edição, 1994.
133. David Gries. *The Science of Programming*. Springer-Verlag, 1981.
134. M. Grötschel, László Lovász e Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1988.
135. Leo J. Guibas e Robert Sedgewick. A dichromatic framework for balanced trees. Em *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pp. 8-21. IEEE Computer Society, 1978.
136. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
137. Yijie Han. Improved fast integer sorting in linear space. Em *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pp. 793-796, 2001.
138. Frank Harary. *Graph Theory*. Addison-Wesley, 1969.
139. Gregory C. Harfst e Edward M. Reingold. A potential-based amortized analysis of the union-find data structure. *SIGACT News*, 31(3):86-95, 2000.
140. J. Hartmanis e R. E. Steams. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285-306, 1965.
141. Michael T. Heideman, Don H. Johnson e C. Sidney Burrus. Gauss and the history of the Fast Fourier Transform. *IEEE ASSP Magazine*, pp. 14-21, 1984.
142. Monika R. Henzinger e Valerie King. Fully dynamic biconnectivity and transitive closure. Em *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pp. 664-672, 1995.
143. Monika R. Henzinger e Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502-516, 1999.
144. Monika R. Henzinger, Satish Rao e Harold N. Gabow. Computing vertex connectivity: New bounds from old techniques. *Journal of Algorithms*, 34(2):222-250, 2000.
145. Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transactions on Mathematical Software*, 16(4):352-368, 1990.
146. C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321-322, 1961.
147. C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10-15, 1962.
148. Dorit S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Math*, 6:243-254, 1983.
149. Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1997.
150. W. Hoeffding. On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics*, 27:713-721, 1956.
151. Micha Hofri. *Probabilistic Analysis of Algorithms*. Springer-Verlag, 1987.
152. John E. Hopcroft e Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225-231, 1973.

153. John E. Hopcroft, Rajeev Motwani e Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, segunda edição, 2001.
154. John E. Hopcroft e Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372-378, 1973.
155. John E. Hopcroft e Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294-303, 1973.
156. John E. Hopcroft e Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
157. Ellis Horowitz e Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.
158. Ellis Horowitz, Sartaj Sahni e Sanguthevar Rajasekaran. *Computer Algorithms*. Computer Science Press, 1998.
159. T. C. Hu e M. T. Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing*, 11(2):362-373, 1982.
160. T. C. Hu e M. T. Shing. Computation of matrix chain products. Part II. *SIAM Journal on Computing*, 13(2):228-251, 1984.
161. T. C. Hu e A. C. Tucker. Optimal computer search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics*, 21(4):514-532, 1971.
162. David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098-1101, 1952.
163. Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao e Thomas Turnbull. Implementation of Strassen's algorithm for matrix multiplication. Em *SC96 Technical Papers*, 1996.
164. Oscar H. Ibarra e Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463-468, 1975.
165. R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*. 2:18-21, 1973.
166. David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256-278, 1974.
167. David S. Johnson. The NP-completeness column: An ongoing guide – the tale of the second prover. *Journal of Algorithms*, 13(3):502-524, 1992.
168. Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1-13, 1977.
169. David R. Karger, Philip N. Klein e Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321-328, 1995.
170. David R. Karger, Daphne Koller e Steven J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199-1217, 1993.
171. Howard Karloff. *Linear Programming*. Birkäuser, 1991.
172. N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373-395, 1984.
173. Richard M. Karp. Reducibility among combinatorial problems. Em Raymond E. Miller e James W. Thatcher, editores, *Complexity of Computer Computations*, pp. 85-103. Plenum Press, 1972.
174. Richard M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34:165-201, 1991.
175. Richard M. Karp e Michael O. Rabin. Efficient randomized pattern-matching algorithms. Relatório técnico TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.
176. A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434-437, 1974.
177. Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*. 18(2):263-270, 1997.
178. Valerie King, Satish Rao e Robert E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17:447-474, 1994.
179. Jeffrey H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis*. Addison-Wesley, 1990.
180. D. G. Kirkpatrick e R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(2):287-.299, 1986.
181. Philip N. Klein e Neal E. Young. Approximation algorithms for NP-hard optimization problems. Em *CRC Handbook on Algorithms*, pp. 34-1-34-19. CRC Press, 1999.
182. Donald E. Knuth. *Fundamental Algorithms*, volume 1 de *The Art of Computer Programming*. Addison-Wesley, 1968. Segunda edição, 1973.
183. Donald E. Knuth. *Seminumerical Algorithms*, volume 2 de *The Art of Computer Programming*. Addison-Wesley, 1969. Segunda edição, 1981.
184. Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14-25, 1971.
185. Donald E. Knuth. *Sorting and Searching*, volume 3 de *The Art of Computer Programming*. Addison-Wesley, 1973.
186. Donald E. Knuth. Big omicron and big omega and big theta. *ACM SIGACT News*, 8(2):18-23, 1976.
187. Donald E. Knuth, James H. Morris, Jr. e Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323-350, 1977.
188. J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57-65, 1985.
189. Bernhard Korte e László Lovász. Mathematical structures underlying greedy algorithms. Em F. Gecseg, editor, *Fundamentals of Computation Theory*, número 117 em Lecture Notes in Computer Science, pp. 205-209. Springer-Verlag, 1981.
190. Bernhard Korte e László Lovász. Structural properties of greedoids. *Combinatorica*, 3:359-374, 1983.
191. Bernhard Korte e László Lovász. Greedoids – a structural framework for the greedy algorithm. Em W. Pulleybank, editor, *Progress in Combinatorial Optimization*, pp. 221-24. Academic Press, 1984.
192. Bernhard Korte e László Lovász. Greedoids and linear objective functions. *SIAM Journal on Algebraic and Discrete Methods*, 5(2):229-238, 1984.
193. Dexter C. Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, 1992.

194. David W. Krumme, George Cybenko e K. N. Venkataraman. Gossiping in minimal time. *SIAM Journal on Computing*, 21(1):111-139, 1992.
195. J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48-50, 1956.
196. Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.
197. Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan e D. B. Shmoys, editores. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.
198. C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346-365, 1961.
199. Tom Leighton e Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. Em *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pp. 422-431, 1988.
200. Debra A. Lelewer e Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261-296, 1987.
201. A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse e J. M. Pollard. The number field sieve. Em A. K. Lenstra e H. W. Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volume 1554 de *Lecture Notes in Mathematics*, pp. 11-42. Springer-Verlag, 1993.
202. H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649- 673, 1987.
203. L. A. Levin. Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3):265-266, 1973. Em russo.
204. Harry R. Lewis e Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, segunda edição, 1998.
205. C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.
206. László Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383-390, 1975.
207. László Lovász e M. D. Plummer. *Matching Theory*, volume 121 of *Annals of Discrete Mathematics*. North Holland, 1986.
208. Bruce M. Maggs e Serge A. Plotkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291-293, 1988.
209. Michael Main. *Data Structures and Other Objects Using Java*. Addison-Wesley, 1999.
210. Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.
211. Conrado Martínez e Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288-323, 1998.
212. William J. Masek e Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1): 18-31, 1980.
213. H. A. Maurer, Th. Ottmann e H.-W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1): 11-14, 1976.
214. Ernst W. Mayr, Hans Jürgen Prömel e Angelika Steger, editores. *Lectures on Proof Verification and Approximation Algorithms*. Número 1367 em Lecture Notes in Computer Science. Springer-Verlag, 1998.
215. C. C. McGeoch. All pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5):426-441, 1995.
216. M. D. McIlroy. A killer adversary for quicksort. *Software – Practice and Experience*, 29(4):341-344, 1999.
217. Kurt Mehlhorn. *Sorting and Searching*, volume 1 de *Data Structures and Algorithms*. Springer-Verlag, 1984.
218. Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 de *Data Structures and Algorithms*. Springer-Verlag, 1984.
219. Kurt Mehlhorn. *Multidimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms*. Springer-Verlag, 1984.
220. Alfred J. Menezes, Paul C. van Oorschot e Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
221. Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300-317, 1976.
222. John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
223. Joseph S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST. and related problems. *SIAM Journal on Computing*, 28(4): 1298-1309, 1999.
224. Louis Monier. *Algorithmes de Factorisation D'Enriers*. Tese de doutorado, L'Université Paris-Sud, 1980.
225. Louis Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, 12(1):97-108, 1980.
226. Edward F Moore. The shortest path through a maze. Em *Proceedings of the International Symposium on the Theory of Switching*, pp. 285-292. Harvard University Press, 1959.
227. Rajeev Motwani, Joseph (Seffi) Naor e Prabakhar Raghavan. Randomized approximation algorithms in combinatorial optimization. Em Dorit Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pp. 447-481. PWS Publishing Company, 1997.
228. Rajeev Motwani e Prabhakar Raghavan. *Randomized Algorithms*. Cambridge Unveristy Press, 1995.
229. J. I. Munro e V. Raman. Fast stable in-place sorting with $O(n)$ data moves. *Algorithmica*, 16(2):151-160, 1996.
230. J. Nievergelt e E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33-43, 1973.
231. Ivan Niven e Herbert S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, quarta edição, 1980.
232. Alan V. Oppenheim e Ronald W. Schafer, with John R. Buck. *Discrete-Time Signal Processing*. Prentice-Hall, segunda edição, 1998.
233. Alan V. Oppenheim e Alan S. Willsky, com S. Hamid Nawab. *Signals and Systems*. Prentice-Hall, segunda edição, 1997.
234. James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78(1): 109-129, 1997.

235. Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1993.
236. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
237. Christos H. Papadimitriou e Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.
238. Michael S. Paterson, 1974. Conferência não publicada, Ile de Berder, França.
239. Michael S. Paterson. Progress in selection. Em *Proceedings of the Fifth Scandinavian Workshop on Algorithm Theory*, pp. 368-379, 1996.
240. Pavel A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000.
241. Steven Phillips e Jeffery Westbrook. Online load balancing and network flow. Em *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pp. 402-411, 1993.
242. J. M. Pollard. A Monte Carlo method for factorization. *BIT*, 15:331-334, 1975.
243. J. M. Pollard. Factoring with cubic integers. Em A. K. Lenstra e H. W. Lenstra, Jr., editores, *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*, pp. 4-10. Springer, 1993.
244. Carl Pomerance. On the distribution of pseudoprimes. *Mathematics of Computation*, 37(156):587-593, 1981.
245. Carl Pomerance, editor. *Proceedings of the AMS Symposia in Applied Mathematics: Computational Number Theory and Cryptography*. American Mathematical Society, 1990.
246. William K. Pratt. *Digital Image Processing*. John Wiley & Sons, segunda edição, 1991.
247. Franco P. Preparata e Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.
248. William H. Press, Brian P. Flannery, Saul A. Teukolsky e William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1986.
249. William H. Press, Brian P. Flannery, Saul A. Teukolsky e William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.
250. R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389-1401, 1957.
251. William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668-676, 1990.
252. Paul W. Purdom, Jr. e Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Rinehart, and Winston, 1985.
253. Michael O. Rabin. Probabilistic algorithms. Em J. F Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pp. 21-39. Academic Press, 1976.
254. Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128-138, 1980.
255. P. Raghavan e C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365-374, 1987.
256. Rajeev Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2):81-87, 1997.
257. Edward M. Reingold, Jurg Nievergelt e Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.
258. Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. Progress in Mathematics. Birkhäuser, 1985.
259. Ronald L. Rivest, Adi Shamir e Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120-126, 1978. Consulte também U.S. Patent 4,405,829.
260. Herbert Robbins. A remark on Stirling's formula. *American Mathematical Monthly*, 62(1):26-29, 1955.
261. D. J. Rosenkrantz, R. E. Stearns e P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6:563-581, 1977.
262. Salvador Roura. An improved master theorem for divide-and-conquer recurrences. Em *Proceedings of Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, pp. 449-459, 1997.
263. Y. A. Rozanov. *Probability Theory: A Concise Course*. Dover, 1969.
264. S. Sahni e T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23:555-565, 1976.
265. A. Schönhage, M. Paterson e N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13(2): 184-199, 1976.
266. Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.
267. Alexander Schrijver. Paths and flows – a historical survey. *CWI Quarterly*, 6:169-183, 1993.
268. Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847-857, 1978.
269. Robert Sedgewick. *Algorithms*. Addison-Wesley, segunda edição, 1988.
270. Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400-403, 1995.
271. Raimund Seidel e C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464-497, 1996.
272. João Setubal e João Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.
273. Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall, segunda edição, 2001.
274. Jeffrey Shallit. Origins of the analysis of the Euclidean algorithm. *Historia Mathematica*, 21(4):401-419, 1994.
275. Michael I. Shamos e Dan Hoey. Geometric intersection problems. Em *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pp. 208-215, 1976.
276. M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7:67-72, 1981.
277. David B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. Em William Cook, László Lovász e Paul Seymour, editores, *Combinatorial Optimization*, volume 20 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1995.
278. Avi Shoshan e Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. Em *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pp. 605-614, 1999.

279. Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
280. Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998.
281. Daniel D. Sleator e Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362-391, 1983.
282. Daniel D. Sleator e Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652-686, 1985.
283. Joel Spencer. *Ten Lectures on the Probabilistic Method*. Regional Conference Series on Applied Mathematics (No. 52). SIAM, 1987.
284. Daniel A. Spielman e Shang-Hua Teng. The simplex algorithm usually takes a polynomial number of steps. Em *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, 2001.
285. Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.
286. Gilbert Strang. *Linear Algebra and Its Applications*. Harcourt Brace Jovanovich, terceira edição, 1988.
287. Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354- 356, 1969.
288. T. G. Szymanski. A special case of the maximal common subsequence problem. Relatório técnico TR-170, Computer Science Laboratory, Princeton University, 1975.
289. Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146-160, 1972.
290. Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215-225, 1975.
291. Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2): 110-127, 1979.
292. Robert E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.
293. Robert F. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306-318, 1985.
294. Robert E. Tarjan. Class notes: Disjoint set union. COS 423, Princeton University, 1999.
295. Robert E. Tarjan e Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245-281, 1984.
296. George B. Thomas, Jr. e Ross L. Finney. *Calculus and Analytic Geometry*. Addison-Wesley, sétima edição, 1988.
297. Mikkel Thorup. Faster deterministic sorting and priority queues in linear space. Em *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pp. 550-555, 1998.
298. Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362-394, 1999.
299. Mikkel Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86-109, 2000.
300. Richard Tolimieri, Myoung An e Chao Lu. *Mathematics of Multidimensional Fourier Transform Algorithms*. Springer-Verlag, segunda edição, 1997.
301. P. van Emde Boas. Preserving order in a forest in less than logarithmic time. Em *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pp. 75-84. IEEE Computer Society, 1975.
302. Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier Science Publishers e The MIT Press, 1990.
303. Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.
304. Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, 1996.
305. Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.
306. Rakesh M. Verma. General techniques for analyzing recursive algorithms with applications. *SIAM Journal on Computing*, 26(2):568-581, 1997.
307. Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309-315, 1978.
308. Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1): 11-12, 1962.
309. Michael S. Waterman. *Introduction to Computational Biology, Maps, Sequences and Genomes*. Chapman & Hall, 1995.
310. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, 1994.
311. Mark Allen Weiss. *Algorithms, Data Structures and Problem Solving with C++*. Addison-Wesley, 1996.
312. Mark Allen Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998.
313. Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-Wesley, 1999.
314. Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509-533, 1935.
315. Herbert S. Wilf. *Algorithms and Complexity*. Prentice-Hall, 1986.
316. J. W. J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7:347-348, 1964.
317. S. Winograd. On the algebraic complexity of functions. Em *Actes du Congres International des Mathématiciens*, volume 3, pp. 283-288, 1970.
318. Andrew C.-C. Yao. A lower bound to finding convex hulls. *Journal of the ACM*, 28(4):780-787, 1981.
319. Yinyu Ye. *Interior Point Algorithms: Theory and Analysis*. John Wiley & Sons, 1997.
320. Daniel Zwillinger, editor. *CRC Standard Mathematical Tables and Formulae*. CRC Press, 30ª edição, 1996.

Índice

Este índice usa as convenções a seguir. Os números estão em ordem alfabética como são lidos; por exemplo, "2-3-4, árvore" é indexado como se fosse "dois-três-quatro, árvore". Quando uma entrada se referir a um local que não seja o texto principal, o número da página será seguido por uma identificação: ex. para exercício, pr. para problema, fig. para figura e n. para nota de rodapé. Um número de página identificado dessa maneira indica com freqüência a primeira página de um exercício, um problema, uma figura ou uma nota de rodapé; essa não é necessariamente a página na qual a referência aparece de fato. Por exemplo, se "linear, pesquisa" fosse definida na página 6 em um exercício que começasse na página 5, a entrada de índice correspondente à expressão "pesquisa linear" seria "5 ex."

$\alpha(n)$, 408
ε-denso, grafo, 507 pr.
ε-universal, função hash, 191 ex.
o, notação, 37-38
O, notação, 44 fig., 35
O', notação, 48 pr.
$\tilde{O}$, notação, 48 pr.
ω, notação, 38
Ω, notação, 34 fig., 35-36
$\rho(n)$-aproximação, algoritmo, 1022
Θ, notação, 32-34, 34 fig.
$\tilde{\Theta}$, notação, 48 pr.
$\{\ \}$ (conjunto), 845
$\in$ (pertence a conjunto), 845
$\notin$ (não pertence a conjunto), 845
$\emptyset$ (conjunto vazio), 845
$\subseteq$ (subconjunto), 845
$\subset$ (subconjunto próprio), 845
: (tal que), 846
$\cap$ (interseção de conjuntos), 846
$\cup$ (união de conjuntos), 846
$-$ (diferença de conjuntos), 846
$|\ |$
  (cardinalidade de conjunto), 848
  (comprimento de uma cadeia), 718
  (valor de fluxo), 510
$\times$
  (produto cartesiano), 848
  (produto cruzado), 934
$\langle\ \rangle$
  (codificação padrão), 770
  (seqüência), 852
! (fatorial), 43
$\lceil\ \rceil$ (teto), 40
$\lfloor\ \rfloor$ (piso), 40
Σ (somatório), 835
Π (produto), 838
$\rightarrow$ (relação de adjacência), 853
$\rightsquigarrow$ (relação de acessibilidade), 854
$\wedge$ (AND), 500, 779
$\neg$ (NOT), 779
$\vee$ (OR), 500, 779
$\oplus$ (operador de grupo), 682
$\otimes$ (operador de convolução), 653-654
$*$ (operador de fechamento), 771
$|$ (relação divide), 673
$\nmid$ (relação não divide), 673
$\equiv$ (equivalente, módulo n), 71, 1077 ex.
$\not\equiv$ (não equivalente, módulo n), 71
$[a]_n$ (classe de equivalência, módulo n), 674
$+_n$ (adição, módulo n), 683
$\cdot_n$ (multiplicação, módulo n), 683
$(\frac{a}{p})$ (símbolo de Legendre), 714 pr.
ε (cadeia vazia), 716, 771
$\sqsupset$ (relação prefixo), 718
$\sqsubset$ (relação sufixo), 718
$>_x$ (relação acima), 744
$\triangleright$ (símbolo de comentário), 14
$\leq_P$ (relação de redutibilidade de tempo polinomial), 776-777, 784 ex.
Z_n^* (elementos de grupo multiplicativo, módulo n), 684
Z_n^+ (elementos diferentes de zero de, 703-704

AA, árvore, 241
abeliano, grupo, 682
aberto, intervalo, 249
ABOVE, 745
absolutamente convergente, série, 836
absorção, leis para conjuntos, 847
abstrato, problema, 768
aceitação
  por um algoritmo, 771
  por uma automação finita, 725
aceitação, estado, 725
aceitável, par de inteiros, 707-708
acessibilidade em um grafo, 854
acesso, espúrio, 722
acesso aleatório, máquina, 16-17
  versus redes de comparação, 555
acíclico, grafo, 855
  relação a matróides, 322 pr.
acima, relação, 744
Ackermann, função, 715
add, instrução, 16-17
adiantada, tarefa, 319
adição
  de inteiros binários, 16 ex.
  de matrizes, 574
  de polinômios, 651
  módulo n $(+_n)$, 683
adjacentes, vértices, 854
admissível, aresta, 538
admissível, rede, 538-540
agregado, fluxo, 625
agregados, análise, 325-328
  para algoritmo de Dijkstra, 473
  para busca de Graham, 754
  para caminhos mais curtos em um grafo, 468
  para contadores binários, 326-327
  para estruturas de dados de conjuntos disjuntos, 402, 403 ex., 406 ex.
  para heaps de Fibonacci, 393 ex.
  para operações de pilhas, 325-326
  para pesquisa primeiro na extensão, 424-425
  para pesquisa primeiro na profundidade, 431
  para tabelas dinâmicas, 334-335
agrupamento, 193-194
Akra-Bazzi, método para resolver uma recorrência, 72
AKS, rede de ordenação, 570
aleatória, amostragem, 124
aleatória, permutação, 81-83
  uniforme, 74-75, 81
aleatória, variável, 873-878
  indicador, *ver* indicador, variável aleatória
aleatório, algoritmo, 75, 79-84
  arredondamento aleatório, 830-831
  e análise probabilística, 79-81
  e entradas médias, 20
  hash universal, 188-191
  ordenação de comparação, 143-144 pr.
  para inserção em uma árvore de pesquisa binária com chaves iguais, 216 pr.
  para o problema da contratação, 80
  para particionamento, 124, 128 ex., 129 pr., 130-131 pr.
  para permutação de um arranjo, 81-83
  para pesquisa em uma lista compacta, 177 pr.
  para satisfabilidade de MAX-3-CNF, 820
  para seleção, 149-152
  pior caso, desempenho do, 124 ex.
  Pollard, heurística rho, 710-713, 713 ex.
  quicksort, 124, 128 ex., 129 pr., 130-131 pr.
  teste de caráter primo de Miller-Rabin, 704-709
aleatório, arredondamento, 830-831
alfabeto, 725, 770
algoritmo, 3
  como uma tecnologia, 8-9
  correção de, 4
  origem da palavra, 31
  tempo de execução de, 17

Alice, 697
ALLOCATE-NODE, 355
ALLOCATE-OBJECT, 172
alocação de objetos, 171-172
alta, extremidade de um intervalo, 249
alterando
uma chave em um heap de Fibonacci, 396-397 pr.
variáveis no método de substituição, 53
altura
de um heap, 104-105, 104-105 ex.
de um heap *d*-ário, 115 pr.
de um nó em um heap, 104-105, 109 ex.
de um nó em uma árvore, 859
de uma árvore, 859
de uma árvore B, 353-354
de uma árvore binomial, 367
de uma árvore de decisão, 134-135
de uma árvore vermelho-preto, 222
do preto, 222
exponencial, 213
altura, função em algoritmos de push-relabel, 531
altura balanceada, árvore, 237 pr.
amortizada, análise, 324-344
análise agregada, 325-328
método de contabilidade de, 328-329
método potencial de, 329-333
para algoritmo de Dijkstra, 473
para algoritmo de Knuth-Morris-Pratt, 733
para algoritmo genérico de push-relabel, 536-537
para árvores de peso balanceado, 341 pr.
para busca de Graham, 754
para caminhos mais curtos em um grafo, 468
para criação de dinâmica de pesquisa binária, 341 pr.
para estruturas de dados de conjuntos disjuntos, 402-403, 403 ex., 406 ex., 408-412, 413 ex.
para heaps de Fibonacci, 383-386, 389-390, 393, 399 ex.
para permutação de reversão de bits, 340 pr.
para pesquisa primeiro na extensão, 424-425
para pesquisa primeiro na profundidade, 431
para pilhas em armazenamento secundário, 363 pr.
para reestruturação de árvores vermelho-preto, 342-343 pr.
para tabelas dinâmicas, 333-340
amortizado, custo
em análise agregada, 325
no método de contabilidade, 328
no método potencial, 330
amostragem, 124
amostral, espaço, 868
ampliando caminho, 517-518, 549 pr.
ampliando estruturas de dados, 242-255
amplitude, árvore, 315, 445
gargalo, 457 pr.
verificação de, 458
ver também mínima, árvore de amplitude
análise, árvore, 788
análise de algoritmos, 16-21
ver também amortizada, análise; probabilística, análise
ancestral, 859
menos comum, 415 pr.
AND, função ($\wedge$), 501, 779
AND, porta, 779
ângulo polar, 743 ex.
aninhando caixas, 486 pr.
aniversário, paradoxo, 85-87
anti-simetria, 1076
ANY-SEGMENTS-INTERSECT, 746
aos pares, conjuntos disjuntos, 848
aos pares, independência, 870-871
aos pares, primos relativos, 676
APPROX-MIN-WEIGHT-VC, 821
APPROX-SUBSET-SUM, 825
APPROX-TSP-TOUR, 811
APPROX-VERTEX-COVER, 809
aproximação
por mínimos quadrados, 603-606
por somatório de integrais, 842
aproximação, algoritmo, 806-831
aleatória, 819
para cobertura de vértices de peso mínimo, 820-822
para correspondência máxima, 829 pr.
para empacotamento de caixas, 828 pr.
para o problema da cobertura de conjuntos ponderada, 828 pr.
para o problema da cobertura de vértices, 807-810
para o problema da programação de máquinas paralelas, 829 pr.
para o problema da satisfabilidade de MAX-CNF, 823 ex.
para o problema de cobertura de conjunto, 815-819
para o problema do caixeiro-viajante, 810-815
para o problema do clique máximo, 828 pr.
para o problema do somatório de subconjuntos, 823-827
para o problema MAX-CUT, 823 ex.
para satisfabilidade de MAX-3-CNF, 819-820
aproximação, esquema, 807
aproximação, razão, 806
aproximação de 1, algoritmo, 807
arbitragem, 486 pr.
arco, *ver* aresta
aresta, 853
admissível, 538
árvore, 427, 429, 433-434
capacidade de, 510
crítica, 524
cruzada, 433-434
de peso negativo, 461
dianteira, 433-434
inadmissível, 538
luz, 447
ordenação em pesquisa primeiro na extensão, 442-443 pr.
ordenação em pesquisa primeiro na profundidade, 434
peso de, 420-421
ponte, 443 pr.
residual, 516
saturada, 531
segura, 446
traseira, 433-434
arestas, conectividade, 525 ex.
arestas, conjunto, 853
argumento de uma função, 852
aritmética, modular, 41, 683-687
aritmética, série, 836
aritmética com infinidades, 464-465
aritméticas, instruções, 16-17
armazenamento, gerenciamento, 103, 171-172, 173 ex., 185 ex.
armazenar, instrução, 16-17
arranjo, 14
Monge, 71 pr.
arredondamento, 822
aleatório, 830-831
articulação, ponto, 443 pr.
árvore, 857-861
altura de, 859
amplitude, *ver* amplitude mínima, árvore; amplitude, árvore
amplitude mínima, *ver* amplitude mínima, árvore
análise, 788
árvores AA, 242
árvores B, 349-364
AVL, 237-238 pr.
binária, *ver* binária, árvore
binomial, 366-368, 383
bisseção de, 862 pr.
bode expiatório, 241
caminhos mais curtos, 462, 482-484
de altura balanceada, 237-238 pr.
de pesquisa binária ótima, 285-291, 295
decisão, 134-135
diâmetro de, 428 ex.
dinâmica, 346
2-3, 241, 364
2-3-4, 353, 363 pr.
enraizada, 174-176, 858-859
fusão, 146, 346-347
heap, 103-116
intervalo, 249-254
k-vizinhos, 241
livre, 856, 857-858
oblíqua, 241, 346
ordem estatística, 242-247
percurso, *ver* árvore, percurso
percurso completo de, 812
peso balanceado, árvores, 241
pesquisa, *ver* pesquisa, árvore
primeiro na extensão, 423, 427-428
primeiro na profundidade, 429
recursão, 27-28, 54-58
treap, 237-238 pr.
vermelho-preto, *ver* vermelho-preto, árvore
árvore, aresta, 427, 429, 433-434
árvore, percurso, 205, 209 ex., 244-245, 812-813
árvore de pesquisa binária, propriedade, 205
versus propriedade de heap mínimo, 207 ex.
assinatura, 698-699
assintótica, eficiência, 32
assintótica, notação, 32-40, 47-48 pr.
e algoritmos de grafos, 417-418
e linearidade de somatórios, 836
assintoticamente maior, 39
assintoticamente menor, 39

assintoticamente não negativo, 33
assintoticamente positivo, 33
assintoticamente restrito, limite, 33
assintótico inferior, limite, 35
assintótico superior, limite, 35
associativa, operação, 382
associativas, leis para conjuntos, 847
atrasada, tarefa, 319
atribuição
múltipla, 14
satisfatória, 780, 785-786
verdade, 780, 785-786
atributo de um objeto, 15
aumentando uma chave em um heap máximo, 112-113
áurea, razão, 45, 69 pr.
ausente, filho, 860
autenticação, 203 pr.
autoloop, 853
automação
de correspondência de cadeias, 726-730
finita, 725
auxiliar, função hash, 193
auxiliar, programa linear, 643
avaliação de um polinômio, 30 pr., 653, 657 ex.
e suas derivadas, 669 pr.
em vários pontos, 670 pr.
AVL, árvore, 237 pr.
AVL-INSERT, 237 pr.
axiomas, para probabilidade, 868

B, árvore, 350-364
altura de, 353-354
árvores 2-3-4, 353
chave mínima de, 360 ex.
criando, 355
dividindo um nó em, 356-358
eliminação de, 360-363
grau mínimo de, 353
inserção em, 356-360
nó total em, 353
pesquisando, 355
propriedades de, 352-355
B*, árvore, 353
B+, árvore, 352
BAD-SET-COVER-INSTANCE, 819 ex.
BALANCE, 237 pr.
balanceada, árvore de pesquisa
árvores AA, 241
árvores AVL, 237 pr.
árvores B, 350-364
árvores de bode expiatório, 241
árvores de *k* vizinhos, 241
árvores de peso balanceado, 241, 341 pr.
árvores 2-3, 341, 364
árvores 2-3-4, 353, 363 pr.
árvores oblíquas, 241, 346
árvores vermelho-preto, 220-241
treaps, 237 pr.
balde, 140
base, 280-281
base, função, 603
base *a*, pseudoprimo, 703-704
básica, solução, 628
básica, solução viável, 628
básica, variável, 620
básico, caso, 52
Batcher, rede de intercalação ímpar-par, 568 pr.
Bayes, teorema, 871-872
BELLMAN-FORD, 465
Bellman-Ford, algoritmo, 465-468
e funções objetivas, 479-480 ex.
no algoritmo de Johnson, 505-506
otimização de Yen para, 485 pr.
para caminhos mais curtos de todos os pares, 490, 505-506
para resolver sistemas de restrições de diferença, 478
BELOW, 745
Bernoulli, experiência, 878
e bolas e caixas, 88-89
e seqüências, 89-92
BFS, 423
BIASED-RANDOM, 75 ex.
biconectado, componente, 442 pr.
bijetora, função, 852
binária, árvore, 859
completa, 860
número de tipos diferentes, 218 pr.
representação de, 174
ver também pesquisa binária, árvore
binária, pesquisa, 28 ex.
com inserção rápida, 341 pr.
em árvores B de pesquisa, 360-361 ex.
em ordenação por inserção, 28 ex.
binária, relação, 849
binário, algoritmo mdc, 714 pr.
binário, contador
analisado pelo método de contabilidade, 329
analisado pelo método potencial, 331-332
analisado por análise agregada, 327
de inversão de bits, 340 pr.
e heaps binomiais, 378 ex.
binário, heap, *ver* heap
binários, código de caracteres, 308
binomial, árvore, 366-368
não ordenada, 383
binomial, coeficiente, 865-866
binomial, distribuição, 879-882
e bolas e caixas, 88
finais de, 883-889
valor máximo de, 882 ex.
binomial, expansão, 864-865
binomial, heap, 365-380
chave mínima de, 370
criando, 370
diminuindo uma chave em, 377
e contador binário e adição binária, 378 ex.
eliminação de, 377
em algoritmo de árvore de amplitude mínima, 379-380 pr.
extraindo a chave mínima de, 376
inserção em, 375
propriedades de, 368
tempos de execução de operações sobre, 366 fig.
unificando, 371-376
BINOMIAL-HEAP-DECREASE-KEY, 377
BINOMIAL-HEAP-DELETE, 377
BINOMIAL-HEAP-EXTRACT-MIN, 375
BINOMIAL-HEAP-INSERT, 375
BINOMIAL-HEAP-MERGE, 372
BINOMIAL-HEAP-MINIMUM, 371
BINOMIAL-HEAP-UNION, 371
BINOMIAL-LINK, 371
bipartida, correspondência, 396-397, 525-529
Hopcroft-Karp, algoritmo para, 549-550 pr.
bipartido, grafo, 856
d-regular, 529 ex.
e hipergrafos, 856 ex.
fluxo em rede correspondente de, 526
bisbilhotando, 343
biscoito a ser mantido fora do cesto, 511
bissecção de uma árvore, 862 pr.
bit, operação, 673
no algoritmo de Euclides, 714 pr.
bitônica, rede de ordenação, 561-564
bitônica, seqüência, 561
e caminhos mais curtos, 488 pr.
bitônico, percurso, 291 pr.
BITONIC-SORTER, 563
BIT-REVERSE-COPY, 667
BIT-REVERSED-INCREMENT, 340 pr.
bits, vetor, 180 ex.
Bob, 697
bode expiatório, árvore, 241
bolas e caixas, 88-89, 889 pr.
bom sujeito, 428 ex.
Boole, desigualdade, 793 ex.
booleana, fórmula, 764, 776 ex., 785-786, 791 ex.
booleana, função, 866 ex.
booleana, multiplicação de matrizes
e fechamento transitivo, 600 ex.
booleano, circuito combinacional, 780
booleano, conectivo, 785-786
booleano, elemento combinacional, 779
borboleta, operação, 664-665
vka, algoritmo, 458
braço, 350
branco, vértice, 422, 429
B-TREE-CREATE, 355
B-TREE-DELETE, 360
B-TREE-INSERT, 357-358
B-TREE-INSERT-NONFULL, 359
B-TREE-SEARCH, 355, 360-361 ex.
B-TREE-SPLIT-CHILD, 356-357
BUBBLESORT, 29 pr.
bucket sort, 140-143
BUCKET-SORT, 140
BUILD-MAX-HEAP, 107
BUILD-MAX-HEAP', 115 pr.
BUILD-MIN-HEAP, 109

cabeça, em uma unidade de disco, 350
cache, 16-17
cache, sem memória, 364
cadeia, 717, 864
cadeia de matrizes, multiplicação, 266-272
cadeia de uma envoltória convexa, 754-755
cadeias, correspondência, 717-737
algoritmo de Knuth-Morris-Pratt, 730-736
algoritmo de Rabin-Karp, 721-725
algoritmo simples, 720-721
baseada em fatores de repetição, 736 pr.
com caracteres de intervalos, 720 ex., 730 ex.
por autômatos finitos, 725-730

caixa, aninhamento, 486 pr.
caixas, empacotamento, 828 pr.
camadas
convexas, 760 pr.
máximas, 760 pr.
caminho, 854
ampliando, 517-518, 549-550 pr.
crítico, 470
hamiltoniano, 776 ex.
localização, 404
mais curto, *ver* mais curtos, caminhos
mais longo, 274, 763
peso de, 459
caminho, cobertura, 546 pr.
caminho, compactação, 404
caminho branco, teorema, 433
caminho externo, comprimento, 861 ex.
caminho hamiltoniano, problema, 802 ex.
caminho total, comprimento, 217-218 pr.
caminhos mais curtos, 459-508
algoritmo de Bellman-Ford, 465-468
algoritmo de Dijkstra, 470-475
algoritmo de escalonamento de Gabow, 486 pr.
algoritmo de Floyd-Warshall, 497-500
algoritmo de Johnson, 503-506
árvore de, 462, 482-484
com arestas de peso negativo, 461
com caminhos bitônicos, 488 pr.
como um programa linear, 623-624
de destino único, 460
de par único, 273-274, 460
de única origem, 460-489
e caminhos mais longos, 763
e ciclos de peso negativo, 460-461
e pesquisa primeiro na extensão, 424-427, 460
e relaxação, 463-465
e restrições de diferença, 475-480
em grafos *g*-densos, 507 pr.
em um grafo acíclico orientado, 468-470
em um grafo não ponderado, 273-274, 425
em um grafo ponderado, 459
estimativa de, 463
limite superior, propriedade de, 464, 480-481
nenhum caminho, propriedade de, 465, 481
por elevação ao quadrado repetida, 494-496
por multiplicação de matrizes, 492-497
propriedade de convergência de, 464, 481
relaxação de caminho, propriedade de, 465, 481-482
subestrutura ótima de, 460-461
subgrafo de predecessor, propriedade de, 465, 484
todos os pares, 460, 490-508
triângulos, desigualdade de, 464, 480-481
variantes de problemas, 460
campo de um objeto, 15
campo numérico, peneira, 716
cancelamento, lema, 659
cancelamento de fluxo, 511
capacidade
de um corte, 518
de uma aresta, 510
residual, 515, 517-518
capacidade, restrição, 510
caracteres, código, 308
caráter primo, testes, 702-709, 715
caráter pseudoprimo, testes, 704
teste de Miller-Rabin, 704-709
cardinalidade de um conjunto (| |), 848
carga, fator
de uma tabela dinâmica, 333-334
de uma tabela hash, 183
carga, instrução, 16-177
carimbo de tempo, 429, 485 ex.
Carmichael, número, 704, 709 ex.
cartesiana, soma, 657 ex.
cartesiano, produto (×), 848
CASCADING-CUT, 392
cascata, corte, 392
caso médio, tempo de execução, 20
catalães, números, 218-219 pr., 267
certificado
em um sistema de criptografia, 701
para algoritmos de verificação, 774
certo, evento, 868
CHAINED-HASH-DELETE, 183
CHAINED-HASH-INSERT, 182
CHAINED-HASH-SEARCH, 182
chamada
de uma sub-rotina, 16, 17 n.
por valor, 15
chapelaria, problema, 79 ex.
chave, 11, 99, 111-112, 159
estática, 198
mediana, de um nó de árvore B, 356
pública, 697, 699
secreta, 697, 699
chave pública, sistema de criptografia, 697
chinês, teorema do resto, 691-693
cíclica, rotação, 736 ex.
cíclico, grupo, 694
ciclo de um grafo, 854-855
de peso negativo, *ver* peso negativo, ciclo
e caminhos mais curtos, 462
hamiltoniano, 764, 773
peso médio mínimo, 487 pr.
ciclo hamiltoniano, problema, 773, 795-799
cilindro, 350
cinza, vértice, 422, 429
circuito
combinacional booleano, 780
para transformação rápida de Fourier, 667
circuito, satisfabilidade, 779-784
CIRCUIT-SAT, 781
circular, lista duplamente ligada com uma sentinela, 168
circular, lista ligada, 166
ver também ligada, lista
classe
complexidade, 771-772
equivalência, 849
classificação de arestas
em pesquisa primeiro na extensão, 442 pr.
em pesquisa primeiro na profundidade, 433-434, 435 ex.
cláusula, 788
clique, 791
CLIQUE, 791
clique, problema, 791-794
algoritmo de aproximação para, 828 pr.
CNF (forma normal conjuntiva), 764, 768
CNF, satisfabilidade, 823 ex.
cobertura
caminho, 547 pr.
por um subconjunto, 815
vértice, 794, 807-808, 820-823
cobertura de conjunto, problema, 815-818
ponderado, 828 pr.
cobertura de vértices, problema
algoritmo de aproximação para, 807-810
caráter NP-completo de, 794-795
codificação de instâncias de problemas, 768-770
código, 308
Huffman, 308-314
co-domínio, 851
coeficiente
binomial, 865
de um polinômio, 41, 651
em forma relaxada, 621
coeficiente, representação, 653
e multiplicação rápida, 655-657
co-fator, 577
coincidente, vértice, 525-526
colecionador de cupons, problema, 89
colinearidade, 739
colisão, 181
resolução por encadeamento, 182-185
resolução por endereçamento aberto, 192-198
colocação entre parênteses de um produto de cadeia de matrizes, 266
coloração de grafo, problema, 804 pr.
colorindo, 804 pr., 861 pr.
coluna, ordem, 576
coluna, vetor, 572
com inversão de bits, contador binário, 340 pr.
combinação, 1864-865
combinacional, circuito, 780
combinacional, elemento, 779
comentário em pseudocódigo (▷), 14
compacta, lista, 177 pr.
COMPACTIFY-LIST, 173 ex.
COMPACT-LIST-SEARCH, 177 pr.
COMPACT-LIST-SEARCH', 178 pr.
comparação, ordenação, 133
aleatória, 144 pr.
e árvores de pesquisa binária, 206 ex.
e heaps intercaláveis, 390-391 ex.
e seleção, 154
comparação, rede, 555-559
comparador, 556
comparáveis, segmentos de linhas, 744
compatíveis, atividades, 297
compatíveis, matrizes, 266-267, 574
complementar, relaxação 648 pr.
complemento
de um conjunto, 647
de um evento, 869
de um grafo, 794-795
de uma linguagem, 771
de Schur, 591, 602

complemento de Schur, lema, 602
completa, árvore *k*-ária, 861
ver também heap
completeza de uma linguagem, 784 ex.
completo, grafo, 856
complexa, raiz de unidade, 658
interpolação de, 663
complexidade, classe, 771-772
co-NP, 775
NP, 764, 775
NPC, 764, 778
P, 764, 768
complexidade, medida, 771-772
componente
biconectado, 442 pr.
conectado, 855
fortemente conectado, 855
componente, grafo, 439
comprimento
de um caminho, 854
de uma cadeia, 718, 864
de uma espinha, 237-238 pr.
de uma seqüência, 852
comprimento de caminho, de uma árvore, 218 pr., 861 ex.
comprimento fixo, código, 308
comprimento variável, código, 308
computacional, geometria, 738-762
computacional, problema, 3-4
COMPUTE-PREFIX-FUNCTION, 733
COMPUTE-TRANSITION-FUNCTION, 730
comum, divisor, 674-675
máximo, *ver* máximo divisor comum
comum, moeda, 869
comum, subexpressão, 664-665
comum, subseqüência, 281
mais longa, 281-285, 295
comum múltiplo, 681 ex.
comutativas, leis para conjuntos, 846
comutatividade sob um operador, 682
concatenação
de cadeias, 718
de linguagens, 771
conclusão, tempo, 321 pr., 829 pr.
concreto, problema, 768
condicional, independência, 873 ex.
condicional, probabilidade, 871
conectado, componente, 1855
identificado com o uso de estruturas de dados de conjuntos disjuntos, 399
identificado com o uso de pesquisa primeiro na profundidade, 436 ex.
conectado, grafo, 855
conectivo, 785-786
configuração, 782
conjugada, transposição, 600 ex.
conjunctive normal form (CNF), *ver* forma normal conjuntiva
conjunta, função de densidade de probabilidades, 874
conjuntiva, forma normal, *ver* forma normal conjuntiva
conjunto ({ }), 845-849
convexo, 514 ex.
independente, 803 pr.
conjunto vazio, leis, 846
conjuntos disjuntos, estrutura de dados de, 398-416
análise de, 408-412, 413 ex.
caso especial de tempo linear de, 416
determinação em profundidade, 413 pr.
em ancestrais mínimos comuns off-line, 415 pr.
em mínimo off-line, 413 pr.
em programação de tarefas, 322 pr.
implementação de floresta de conjuntos disjuntos, 403-406
implementação de lista ligada de, 400-403
no logaritmo de Kruskal, 451
conjuntos disjuntos, floresta de, 403-406
análise de, 408-412, 413 ex.
propriedades de ordem de, 408, 413 ex.
ver também conjuntos disjuntos, estrutura de dados de
CONNECTED-COMPONENTS, 399-400
co-NP, 775
conservação do fluxo, 510
consistência de literais, 792
consolidando uma lista raiz de um heap de Fibonacci, 386-387
CONSOLIDATE, 387
construída aleatoriamente, árvore de pesquisa binária, 213-216, 217-218 pr.
consulta, 160
contabilidade, método, 328-329
para contadores binários, 329
para operações de pilha, 328-329, 329 ex.
para tabelas dinâmicas, 335
contador, *ver* binário, contador
contagem, 863-868
probabilística, 95 pr.
contagem, ordenação, 135-137
em radix sort, 172
contavelmente infinito, conjunto, 848
contém em um caminho, 854
contínua, distribuição uniforme de *probabilidades*, 870
contração
de um grafo não orientado por uma aresta, 856
de um matróide, 317
de uma tabela dinâmica, 336-338
contratação, problema, 73-74
análise probabilística de, 78-79
on-line, 93-95
controle, instruções, 16-17
convergência, propriedade, 465, 481-482
convergente, série, 836
convexa, combinação de pontos, 739
convexa, envoltória, 749-756, 761 pr.
convexa, função, 876
convexas, camadas, 760 pr.
convexo, conjunto, 514 ex.
convexo, polígono, 742 ex.
convolução (⊗), 653
convolução, teorema, 663
cópia, instrução, 16-17
cor de um nó de árvore vermelho-preto, 220
corda, 247 ex.
correção de um algoritmo, 4
correspondência
bipartida ponderada, 396-397
de cadeias, 717-737
de máximo, 829 pr.
e fluxo máximo, 525-529
máxima, 809, 829 pr.
correspondência de cadeias, autômato, 726-730, 730 ex.
corte
de um fluxo em rede, 517-520
de um grafo não orientado, 447
em cascata, 391
peso de, 823 ex.
COUNTING-SORT, 135-136
co-vertical, 745
crédito, 328
criptografia, sistema, 697-702
crítica, aresta, 524
crítico, caminho, 470
cruzada, aresta, 433-434
cruzado, produto (×), 739
cruzando um corte, 446-447
cúbica, curva, 607 pr.
curto-circuito, operador, 15
curva, 607 pr.
curva elíptica, método de fatoração, 716
curvas, ajuste, 603-606
custo mínimo, árvore de amplitude, *ver* mínima, árvore de amplitude
custo mínimo, fluxo, 779-780
custo mínimo, fluxo de várias mercadorias, 781 ex.
CUT, 391

dados, estrutura, 6, 159-255, 345-416
ampliação de, 242-255
árvores 2-3, 241, 364
árvores 2-3-4, 353, 363 pr.
árvores AA, 241
árvores AVL, 237-238 pr.
árvores B, 350-364
árvores de bode expiatório, 241
árvores de fusão, 146, 346
árvores de intervalos, 249-254
árvores de *k* vizinhos, 241
árvores de ordem estatística, 242-247
árvores de peso balanceado, 241
árvores de pesquisa binária, 204-219
árvores de pesquisa exponencial, 146, 347
árvores dinâmicas, 346
árvores enraizadas, 174-176
árvores oblíquas, 241, 346
árvores vermelho-preto, 220-241
bits, vetores, 180 ex.
conjuntos dinâmicos, 159-160
deques, 166 ex.
dicionários, 159
filas, 163-166
filas de prioridades, 111-114
heaps, 103-116
heaps 2-3-4, 379 pr.
heaps binomiais, 365-380
heaps de Fibonacci, 381-397
heaps relaxados, 397
listas de saltos, 241
listas ligadas, 166-170
no armazenamento secundário, 349-351
para conjuntos disjuntos, 398-416
para grafos dinâmicos, 347
persistente, 236 pr., 346
pilhas, 163-164
potencial de, 330
raiz de árvores, 217 pr.
tabelas de endereço direto, 180

tabelas hash, 181-185
treaps, 238 pr.
van Emde Boas, 346
DAG-SHORTEST-PATHS, 468
d-ário, heap, 115 pr.
em algoritmos de caminhos mais curtos, 507 pr.
decisão, árvore, 134-135
princípio zero-um para, 561 ex.
decisão, problema, 765, 767-768
e problemas de otimização, 765
decisão por um algoritmo, 771
DECREASE-KEY, 111-112, 365
DECREMENT, 327 ex.
definida como positiva, matriz, 377
degeneração, 636
deixando um vértice, 853
deixando variável, 629
DELETE, 160, 365
DELETE-LARGER-HALF, 333 ex.
demanda, paginação, 16-17
DeMorgan, leis, 847
densidade de números primos, 702-703
densidade de probabilidades, função, 874
denso, grafo, 419
g-denso, 507 pr.
dependência
e indicadores de variáveis aleatórias, 77
linear, 576
ver também independência
depósito, 421 ex., 510, 512
deque, 166 ex.
DEQUEUE, 165
derivada de série, 837
desalocação de objetos, 171-172
descarga de um vértice de transbordamento, 539-540
descendente, 858
descoberto, vértice, 422, 429
descritor, 112, 366, 382
desigualdade, restrição, 616-617
e restrições de igualdade, 618
desigualdade linear, 612
desigualdade linear, problema de viabilidade, 648-649 pr.
deslocamento, instrução, 16-17
deslocamento em correspondência de cadeias, 717
desmembrando
árvores 2-3-4, 363 pr.
nós de árvores B, 356-357
destino, 1013
destino, vértice, 460
destino único, caminhos mais curtos, 460
desvio, instruções, 16-17
desvio condicional, instrução, 16-17
desvio padrão, 877
det, *ver* determinante
determinação da profundidade, problema, 519 pr.
determinante, 577
e multiplicação de matrizes, 600 ex.
determinística, 79-80
DETERMINISTIC-SEARCH, 95 pr.
DFS, 429-430
DFS-VISIT, 430
DFT (Discrete Fourier Transform), 660
diagonal, matriz, 572
decomposição de LUP de, 596 ex.
diâmetro de uma árvore, 428 ex.
dianteira, aresta, 433-434
dicionário, 159
diferença, equação, *ver* recorrência
diferença, restrições, 475-480
diferença de conjuntos (-), 846
simétrica, 549-550 pr.
diferenciação de séries, 837
digital, assinatura, 698
digrafo, *ver* orientado, grafo
DIJKSTRA, 470
Dijkstra, algoritmo, 470-475
com pesos de arestas inteiros, 474-475 ex.
implementado com um heap de Fibonacci, 473-474
implementado com um heap mínimo, 473-474
no algoritmo de Johnson, 504-506
para caminhos mais curtos de todos os pares, 490, 504-506
semelhança com a pesquisa primeiro na extensão, 473, 474, 474 ex.
semelhança com o algoritmo de Prim, 452, 473-474
diminuindo uma chave
em heaps binomiais, 376-377
em heaps de Fibonacci, 390-393
em heaps 2-3-4, 379 pr.
dinâmica, árvore, 346
dinâmica, tabela, 333-340
analisada por análise agregada, 334-335
analisada por método de contabilidade, 335
analisada por método potencial, 335-336, 338-339
fator de carga de, 333-334
dinâmico, conjunto, 159-161
ver também estrutura de dados
dinâmico, grafo, 399 n.
algoritmo de árvore de amplitude mínima, 455 ex.
estruturas de dados para, 347
fechamento transitivo de, 507 pr.
DIRECT-ADDRESS-DELETE, 180
DIRECT-ADDRESS-INSERT, 180
DIRECT-ADDRESS-SEARCH, 180
DIRECTION, 741
direita, conversão, 225 ex.
direita, espinha, 237-238 pr.
direita, filho da, 860
direita, rotação, 223
direita, subárvore da, 860
direta, substituição, 588
direto, endereçamento, 180-181
direto, raio horizontal, 743-744 ex.
DISCHARGE, 539-540
disco, 350
disco, 749 ex.
ver também secundário, armazenamento
discreta, variável aleatória, 873-878
discreto, logaritmo, 694
disjuntiva, forma normal, 789
disjuntos, conjuntos, 847
DISK-READ, 351
DISK-WRITE, 351
dispositivo, 795
distância
de um caminho mais curto, 424
edição, 291 pr.
euclidiana, 756-757
L_m, 760 ex.
Manhattan, 156 pr., 760 ex.
distribuição
binomial, 879-882
de entradas, 74-75, 79-80
de envoltória esparsa, 762 pr.
de números primos, 703
geométrica, 872
probabilidades, 868-869
distribuição de probabilidades, função, 143 ex.
distributivas, leis para conjuntos, 847
divergente, série, 836
divide, instrução, 16-17
divide, relação (|), 673
dividir e conquistar, método, 21-25
análise de, 25
e árvores de recursão, 54
para algoritmo de Strassen, 579-585
para conversão de binário em decimal, 677 ex.
para Fast Fourier Transform, 661-663
para inversão de matrizes, 598-600
para localizar a envoltória convexa, 749-750
para localizar o par de pontos mais próximos, 757-759
para multiplicação, 668 pr.
para ordenação por intercalação, 21-28
para pesquisa binária, 28 ex.
para quicksort, 117-132
para seleção, 149-155
relação com a programação dinâmica, 259
resolvendo recorrências para, 50-72
divisão, método, 186-187
divisão em duas partes iguais, lema, 659
divisão teorema, 674
divisor, 673-674
comum, 674
ver também máximo divisor comum
DNA, 281, 291-292 pr.
DNF (disjunctive normal form), 789
ver também disjuntiva, forma normal
2-3, árvore, 241, 364
2-3-4, árvore, 353
e árvores vermelho-preto, 354-355 ex.
desmembrando, 363 pr.
junção, 363 pr.
2-3-4, heap, 379 pr.
2-CNF, satisfabilidade, 791 ex.
e satisfabilidade de 3-CNF, 764
2-CNF-SAT, 791-792 ex.
domina, relação, 760 pr.
domínio, 851
d-regular, grafo, 529 ex.
dualidade, 638-642
fraca, 638
duas passagens, método, 405
duplamente ligada, lista, 166
ver também ligada, lista
duplo, hash, 194-195, 197 ex.
duplo linear, programa, 658

e (and), em pseudocódigo, 15
e, 42
E[ ] (valor esperado), 875

edição, distância, 292 pr.
Edmonds-Karp, algoritmo, 521-525
eixo, 350
elementar, evento, 868
elemento de um conjunto ($\in$), 845
elemento máximo de um conjunto parcialmente ordenado, 850
elevação ao quadrado, repetida
  para caminhos mais curtos de todos os pares, 494-496
  para elevar um número a uma potência, 696
eliminação
  de árvores B, 360-363
  de árvores de intervalos, 251
  de árvores de ordem estatística, 246
  de árvores de pesquisa binária, 211-212
  de árvores vermelho-preto, 231-236
  de filas, 164
  de heaps, 114 ex.
  de heaps 2-3-4, 379 pr.
  de heaps binomiais, 376-377
  de heaps de Fibonacci, 393, 396 pr.
  de listas ligadas, 168
  de pilhas, 163
  de status de linhas de varredura, 745
  de tabelas dinâmicas, 338
  de tabelas hash de endereço aberto, 192-193
  de tabelas hash encadeadas, 183
else, em pseudocódigo, 14
em grau, 854
em ordem, percurso de árvore, 205, 209-210 ex., 244-245
empurrão, operação (em algoritmos de push-relabel), 531
empurrão sobre uma pilha de tempo de execução, 130-131 pr.
encadeamento
  de árvores binomiais, 366-367
  de raízes de heap de Fibonacci, 386
encadeando, 182-184, 202 pr.
endereçamento, aberto, *ver* aberto, endereçamento
endereço aberto, tabela hash, 192-198
  duplo, hash, 194-195, 197-198 ex.
  linear, sondagem, 193
  quadrática, prova, 193-194, 202 pr.
endereço direto, tabela, 179-181
enraizada, árvore, 858
  representação de, 174-176
entrada
  distribuição de, 74, 79-80
  para um algoritmo, 3
  para um circuito combinacional, 780
  para uma porta lógica, 779
  tamanho da, 17
entrada, alfabeto, 725
entrada, fio, 556
entrada, seqüência, 556
entropia, função, 866
entropia binária, função, 866
envoltória convexa, 749-756, 762 pr.
envoltória esparsa, distribuição, 762 pr.
equação
  e notação assintótica, 36-37
  normal, 605
  recorrência, *ver* recorrência
equivalência, classe, 849
  módulo n ($[a]_n$), 674
equivalência, modular, 41, 851 ex.
equivalência, relação, 849-850
  e equivalência modular ($\equiv$), 851 ex.
equivalentes, programas lineares, 617-618
escalar, múltiplo, 574
escalar, produto de fluxo, 514 ex.
escalonamento
  em caminhos mais curtos de uma única origem, 486 pr.
  em fluxo máximo, 548 pr.
escape, problema, 547 pr.
escolha $\binom{n}{k}$, 864-865
esparso, grafo, 419
espelhamento, 660 n.
esperado, tempo de execução, 20
esperado, valor, 875-876
  de uma distribuição binomial, 880
  de uma distribuição geométrica, 878
  de um indicador de variável aleatória, 76
espinha, 237-238 pr.
espúrio, acesso, 722
esquerda, espinha, 237-238 pr.
esquerda, filho, 860
esquerda, rotação, 223
esquerda, subárvore, 860
essencial, termo, 582
essencialidade, teorema, 528
estabilidade
  numérica, 571, 587, 608-609
  de algoritmos de ordenação, 137, 140 ex.
estado de um autômato finito, 725
estado final, função, 726
estático, conjunto de chaves, 198
estático, grafo, 399 n.
estrelado, polígono, 756 ex.
estritamente crescente, 40
estritamente decrescente, 40
estrutura de blocos em pseudocódigo, 14
estrutura de parênteses de pesquisa primeiro na profundidade, 431
EUCLID, 679
Euclides, algoritmo, 679-682, 714 pr.
euclidiana, distância, 756
euclidiana, norma, 575
euclidiano bitônico, problema do *caixeiro-viajante*, 291-292 pr.
Euler, função phi, 685
Euler, teorema, 694, 709 ex.
Euler, viagem, 763
  e ciclos hamiltonianos, 763
evento, 868
evento, ponto, 745
evidência do caráter composto de um número, 704-705
EXACT-SUBSET-SUM, 824
excesso, fluxo, 529-530
exclusão e inclusão, 849 ex.
exclusiva, fatoração de inteiros, 676
execução, tempo, 17
  caso médio, 20
  melhor caso, 20 ex., 36
  de uma rede de comparação, 57
  esperado, 20
  de um algoritmo de grafo, 417-418
  ordem de crescimento, 20
  taxa de crescimento, 20
  pior caso, 20, 36
executar uma sub-rotina, 17 n.
expansão de uma tabela dinâmica, 334
expectativa, *ver* esperado, valor
experiência, Bernoulli, 878
experiência, divisão, 703
explorado, vértice, 731
exponenciação
  modular, 696
exponenciação, instrução, 16-17
exponencial, altura, 213
exponencial, árvore de pesquisa, 146, 347
exponencial, função, 41-42
exponencial, série, 837
EXTENDED-EUCLID, 681
EXTEND-SHORTEST-PATHS, 493
extensão de um conjunto, 315
exterior de um polígono, 743 ex.
externo, nó, 859
externo, produto, 575
extração
  de uma pilha em tempo de execução, 130-131 pr.
  operação de pilha, 164
EXTRACT-MAX, 112
EXTRACT-MIN, 112, 365
extraindo a chave máxima
  de heaps *d*-ários, 115 pr.
  de heaps máximos, 112
extraindo a chave mínima
  de heaps binomiais, 375-376
  de heaps de Fibonacci, 385-390
  de heaps 2-3-4, 379 pr.
  de quadros de Young, 115 pr.
extremidade baixa de um intervalo, 249

falha em uma experiência de Bernoulli, 878
fan-out, 780
Farkas, lema, 649 pr.
FASTER-ALL-PAIRS-SHORTEST-PATHS, 496, 496 ex.
FASTEST-WAY, 264
fator, 674
  giro, 662
fator de desvio, em árvores B, 651-652
fator de repetição de uma cadeia, 737 pr.
fatoração, 896-901, 716
  única, 676
fatorial, função (!), 43-44
fechado, intervalo, 249
fechado, semi-anel, 508
fechamento
  de uma linguagem, 771
  operador (*), 771
  propriedade de grupo, 682
  transitivo, *ver* transitivo, fechamento
Fermat, teorema, 694
FFT (Fast Fourier Transform), *ver* transformação rápida de Fourier (FFT)
FIB-HEAP-CHANGE-KEY, 397 pr.
FIB-HEAP-DECREASE-KEY, 390-391
FIB-HEAP-DELETE, 393
FIB-HEAP-EXTRACT-MIN, 386
FIB-HEAP-INSERT, 384
FIB-HEAP-LINK, 387
FIB-HEAP-PRUNE, 397 pr.
FIB-HEAP-UNION, 385-386
Fibonacci, heap, 381-397
  alterando uma chave em, 397 pr.
  chave mínima de, 385
  criando, 384

diminuindo uma chave em, 391-393
eliminação de, 393, 396 pr.
extraindo a chave mínima de, 386-390
função potencial para, 383
grau máximo de, 383, 394-396
inserção em, 384-385
no algoritmo de Dijkstra, 473-474
no algoritmo de Johnson, 506
no algoritmo de Prim, 454
podando, 397 pr.
tempos de execução de operações sobre, 366 fig.
unificando, 385-386
Fibonacci, números, 45, 69 pr., 394-395
computação de, 714 pr.
FIFO ("first in, first out", primeiro a entrar, primeiro a sair), 163
ver também fila
fila, 163-165
em algoritmos de push-relabel, 546 ex.
em pesquisa primeiro na extensão, 423
implementação de lista ligada de, 169 ex.
implementada por pilhas, 166 ex.
prioridades. *ver* prioridades, fila
filha, 858-859, 860
filho da esquerda, representação de irmão da direita, 173-174, 176 ex.
final
de uma distribuição binomial, 883-889
de uma fila, 164-165
de uma lista ligada, 166
final, recursão, 130-131 pr., 301
FIND-DEPTH, 413 pr.
FIND-SET, 399
implementação de, com floresta de conjuntos disjuntos, 405, 416
implementação de, com lista ligada, 400
finita, seqüência, 852
FINITE-AUTOMATON-MATCHER, 727-728
finito, autômato, 725
para correspondência de cadeias, 726-730
finito, conjunto, 847-848
finito, grupo, 682
fio, 556, 780
floresta, 856, 857
de conjuntos disjuntos, 403-406
primeiro na profundidade, 429
FLOYD-WARSHALL, 498
FLOYD-WARSHALL', 502 ex.
Floyd-Warshall, algoritmo, 497-500, 501 ex.
fluxo, 510-514
agregado, 625
bloqueio, 550-551
de valor inteiro, 527
excesso, 529
líquido total, 510-511
positivo total, 510-511
soma, 514 ex.
valor de, 510
fluxo, conservação, 51
fluxo, rede, 510-515
com capacidades negativas, 549 pr
correspondendo a um grafo bipartido, 526
corte de, 518-520
fluxo de bloqueio, 550
fluxo máximo, corte mínimo, teorema, 520
folha, 859
for
e loops invariantes, 13 n.
em pseudocódigo, 14
FORD-FULKERSON, 520-521
Ford-Fulkerson, método, 515-526
FORD-FULKERSON-METHOD, 515
forma canônica para programação de tarefas, 319
forma normal conjuntiva, 764, 788
forma normal disjuntiva, *ver* disjuntiva, forma normal
formal, série de potências, 69 pr.
fortemente conectado, componente, 85
decomposição em, 438-441
fortemente conectado, grafo, 855
fraca, dualidade, 638
FREE-OBJECT, 172
freqüência, domínio, 651
função, 1077-1080
convexa, 876
de Ackermann, 415
de base, 603
linear, 19, 612
objetivo, *ver* objetivo, função
prefixo, 728-732
quadrática, 19
sufixo, 726
transição, 725
função sufixo, desigualdade, 728
função sufixo, lema de recursão, 728
funcional, iteração, 44
fundo de uma pilha, 163
fusão, árvore, 146

Gabow, algoritmo de escalonamento para caminhos mais curtos de única origem, 486 pr.
grafo, *ver* orientado, grafo acíclico
gargalo, árvore de amplitude, 457 pr.
gargalo, problema do caixeiro-viajante, 1033 ex.
gaussiana, eliminação, 590
GENERIC-MST, 446
GENERIC-PUSH-RELABEL, 533
geométrica, distribuição, 878
e bolas e caixas, 88
geométrica, série, 837
gerador
de um subgrupo, 686
geral, peneira de campo numérico, 716
gerando função, 69 pr.
giro, fator, 662
global, variável, 14
Goldberg, algoritmo, *ver* push-relabel, algoritmo
grade, 547 pr.
gráfico, matróide, 314, 458
grafo, 853-856
algoritmos para, 417-551
caminho mais curto em, 425
complemento de, 794-795
componente, 439-440
denso, 419
dinâmico, 499 n.
g-denso, 507 pr.
esparso, 419
estático, 399 n.
hamiltoniano, 773
intervalo, 303 ex.
matriz de incidência de, 322 pr., 422 ex.
não hamiltoniano, 773
pesquisa primeiro na extensão de, 442-428
pesquisa primeiro na profundidade de, 429-436
ponderado, 420-421
por lista de adjacência, representação de, 420
por matriz de adjacência, representação de, 420-421
unicamente conectado, 436 ex.
viagem de, 798
ver também orientado, grafo acíclico; orientado, grafo; fluxo, rede; não orientado, grafo; árvore
GRAFT, 413 pr.
Graham, varredura, 750-754
GRAHAM-SCAN, 750
GRAPH-ISOMORPHISM, 775 ex.
grau
de um nó, 859
de um polinômio, 41
de um vértice, 854
de uma raiz de árvore binomial, 366-367
máximo, de um heap de Fibonacci, 383, 390 ex., 394-396
mínimo, de uma árvore B, 353
grau, limite, 651
greedoid, 322-323
GREEDY, 316-317
GREEDY-ACTIVITY-SELECTOR, 302-303
GREEDY-SET-COVER, 816
grupo, 682-687
cíclico, 694
grupo aditivo, módulo *n*, 6383
grupo multiplicativo, módulo *n*, 684
gulosa, propriedade de escolha, 304-305
de códigos de Huffman, 310-312
de um matróide ponderado, 317
guloso, algoritmo, 296-323
algoritmo de Dijkstra, 470-475
algoritmo de Kruskal, 470-452
algoritmo de Prim, 452-454
comparação com programação dinâmica, 273-274, 280-281 ex., 299-300, 304, 305-306
e matróides, 314-318
elementos de, 303-307
em um matróide ponderado, 316-318
para árvore de amplitude mínima, 445
para código de Huffman, 308-314
para o problema da cobertura de conjunto ponderado, 828 pr.
para o problema da cobertura de conjuntos, 815-818
para problema da mochila fracionária, 305-306
para programação de tarefas, 319-321, 321 pr., 322 pr.
para seleção de atividade, 297-303
para troca de moeda, 321 pr.
propriedade de escolha gulosa em, 304-305
subestrutura ótima em, 305

HALF-CLEANER, 562
Hall, teorema, 529-530 ex.
HAM-CYCLE, 774
hamiltoniano, caminho, 776 ex.
hamiltoniano, ciclo, 764, 773-774
hamiltoniano, grafo, 773-774
HAM-PATH, 776 ex.
handshake, lema, 856 ex.
harmônica, série, 837
harmônico, número, 837
hash, 179-203
duplo, 194-195, 197 ex.
encadeamento, 182-184, 202 pr.
endereçamento aberto, 192-198
k-universal, 203 pr.
perfeito, 198-201
universal, 188-191
hash, função, 181, 185-192
auxiliar, 193
divisão, método, 186-187
g-universal, 191 ex.
multiplicação, método, 187
resistente a colisões, 701
universal, 188-191
hash, tabela, 181-185
dinâmica, 339 ex.
secundária, 198
ver também hash
hash, valor, 181
HASH-DELETE, 197 ex.
HASH-INSERT, 192, 197 ex.
HASH-SEARCH, 193, 197 ex.
heap, 103-116
altura de, 104-105
analisado pelo método potencial, 333 ex.
aumentando uma chave em, 112-113
binomial, *ver* binomial, heap
chave máxima de, 112
como armazenamento do lixo coletado, 103
como uma fila de prioridades, 111-114
construindo, 107-109, 115 pr.
d-ário, 115 pr., 507 pr.
2-3-4, 379 pr.
e treaps, 238 pr.
extraindo a chave máxima de, 112
Fibonacci, *ver* heap de Fibonacci
heap máximo, 104
heap mínimo, 104-105
inserção em, 113
intercalável, *ver* intercalável, heap
no algoritmo de Dijkstra, 473-474
no algoritmo de Huffman, 310
no algoritmo de Johnson, 506
no algoritmo de Prim, 454
para implementar um heap intercalável, 365
relaxado, 397
tempos de execução de operações sobre, 366 fig.
heap, propriedade, 104
manutenção de, 105-107
versus árvore de pesquisa binária, propriedade, 207 ex.
heap máximo, propriedade, 104
manutenção de, 105-107
heap mínimo, ordenado, 368
heap mínimo, propriedade, 104, 368
manutenção de, 107 ex.
versus propriedade de árvore de pesquisa binária, 207 ex.
HEAP-DECREASE-KEY, 114 ex.
HEAP-DELETE, 114 ex.
HEAP-EXTRACT-MAX, 112
HEAP-EXTRACT-MIN, 113 ex.
HEAP-INCREASE-KEY, 113
HEAP-MAXIMUM, 112
HEAP-MAXIMUM, 113 ex.
heapsort, 103-116
HEAPSORT, 110
hereditários, família de subconjuntos, 314
hermitiana, matriz, 600 ex.
hiperaresta, 856
hipergrafo, 856
e grafos bipartidos, 856 ex.
HIRE-ASSISTANT, 73-74
HOARE-PARTITION, 129 pr.
HOPCROFT-KARP, 549-550 pr.
Hopcroft-Karp, algoritmo de correspondência bipartida, 549-550 pr.
hora de término, em seleção de atividade, 297
hora de término, na pesquisa primeiro em profundidade, 429
e componentes fortemente conectado, 440
horizontal, raio, 506 ex.
Horner, regra, 30 pr., 653
no algoritmo de Rabin-Karp, 721
HUFFMAN, 310
Huffman, código, 308-314

idempotência, leis para conjuntos, 846
identidade, 682
identidade, matriz, 572-573
if, em pseudocódigo, 14
igualdade
de conjuntos, 845
de funções, 852
linear, 612
igualdade, restrição, 479 ex., 616-617
e restrições de desigualdade, 618
rígida, 627
violação de, 627
imagem, 852
ímpar-par, rede de intercalação, 568 pr.
ímpar-par, rede de ordenação, 568 pr.
inadmissível, aresta, 538
incidência, 853
incidência, matriz
de um grafo não orientado, 322 pr.
de um grafo orientado, 322 pr., 422 ex.
e restrições de diferença, 477
inclinada, simetria, 510
inclusão e exclusão, 848 ex.
incondicional, instrução de desvio, 16-17
INCREASE-KEY, 111-112
INCREMENT, 326
incremental, método de projeto, 20
para localizar a envoltória convexa, 749-750
independência
de eventos, 870, 873 ex.
de subproblemas em programação dinâmica, 275-276
de variáveis aleatórias, 874
independente, conjunto, 803 pr.
de tarefas, 320
independente, família de subconjuntos, 314
indicador, variável aleatória, 76-79
em análise da altura esperada de uma árvore de pesquisa binária construída aleatoriamente, 213-215
em análise de inserção em um treap, 237 pr.
em análise de seleção aleatória, 150-152, 152 ex.
em análise de seqüências, 91-92
em análise do paradoxo do aniversário, 87-88
na análise da bucket sort, 141-143
na análise de hash, 184
na análise de hash universal, 188-189
na análise de quicksort, 127-128, 129 pr.
na análise do problema de contratação, 78-79
na limitação do final direito da distribuição binomial, 887
no algoritmo de aproximação para satisfabilidade de MAX-3-CNF, 820
índice de um elemento de Z_n^*, 694
induzido, subgrafo, 855
inferior, matriz triangular, 573
inferior, mediana, 147
inferiores, limites
e funções potenciais, 343
para cobertura de vértices de peso mínimo, 822
para envoltória convexa, 756 ex.
para intercalação, 145 pr.
para localização da mediana, 156-157
para ordenação, 133-136
para ordenação média, 145 pr.
para tamanho de cobertura ótima de vértices, 809
para tamanho de uma rede de intercalação, 566 ex.
infinidade, aritmética com, 464-465
infinita, seqüência, 852
infinita, soma, 835
infinito, conjunto, 847-848
inicial, estado, 725
inicial, submatriz, 601
início
de uma fila, 164-165
de uma lista ligada, 166
início, hora, 297
INITIALIZE-PREFLOW, 532
INITIALIZE-SIMPLEX, 631, 644
INITIALIZE-SINGLE-SOURCE, 463
injetora, função, 852
INORDER-TREE-WALK, 205-206
inserção
em árvores B, 356-359
em árvores de intervalos, 251
em árvores de ordem estatística, 246
em árvores de pesquisa binária, 210-311
em árvores vermelho-preto, 226-230
em filas, 164
em heaps, 113
em heaps 2-3-4, 379 pr.
em heaps binomiais, 375
em heaps *d*-ários, 115 pr.

em heaps de Fibonacci, 384-385
em listas ligadas, 167-168
em pilhas, 163
em quadros de Young, 115 pr.
em status de linhas de varredura, 745
em tabelas de endereço direto, 180
em tabelas dinâmicas, 334-335
em tabelas hash de endereço aberto, 192-193
em tabelas hash encadeadas, 1873
inserção, ordenação, 8, 11-14, 18-19
árvore de decisão para, 134 fig.
comparação com a ordenação por intercalação, 9 ex.
comparação com quicksort, 123 ex.
em bucket sort, 140-143
em ordenação por intercalação, 28 pr.
em quicksort, 128 ex.
usando pesquisa binária, 28 ex.
inserindo séries, 837-838
inserindo soma, 838
INSERT, 111, 160, 333 ex., 365
INSERTION-SORT, 13, 18
instância
de um problema, 3
de um problema abstrato, 765, 768
instruções do modelo de RAM, 16
integração de séries, 837
integral para aproximação de somatórios, 842
inteiro, tipo de dados, 16-17
inteiros (Z), 845
intercalação
de dois arranjos ordenados, 21
de *k* listas ordenadas, 114 ex.
limites inferiores para, 145 pr.
usando uma rede de comparação, 564-565
intercalação, ordenação por, 8, 21-38
comparação com ordenação por inserção, 9 ex.
rede de ordenação, implementação de, 566-568
uso de ordenação por inserção em, 28 pr.
intercalação, rede, 564-565
ímpar-par, 568 pr.
intercalável, heap, 345, 365
e ordenações por comparação, 391 ex.
heaps 2-3-4, 379 pr.
lista ligada, implementação de, 176 pr.
relaxados, heaps, 397
tempos de execução de operações sobre, 366 fig.
ver também binomial, heap; Fibonacci, heap
intercalável, heap máximo, 176 n., 345 n., 365 n.
interceptação, 740
interior de um polígono, 743 ex.
intermediário, vértice, 497
interno, comprimento de caminho, 861 ex.
interno, nó, 859
interno, produto, 575
interpolação por uma curva cúbica, 607 pr.
interseção
de conjuntos (∩), 846
de cordas, 247 ex.
de linguagens, 770
determinando, para dois segmentos de linhas, 740-742
determinando, para um conjunto de segmentos de linhas, 743-749
INTERVAL-DELETE, 250
INTERVAL-INSERT, 250
intervalo, 852
intervalo, 249
ordenação nebulosa de, 131 pr.
intervalo, árvore, 249-254
intervalo, caráter, 720 ex., 730 ex.
intervalo, heurística, 546 ex.
intervalo, tricotomia, 250
intervalos, problema de coloração grafo de, 303 ex.
INTERVAL-SEARCH, 250, 251-252
INTERVAL-SEARCH-EXACTLY, 254 ex.
intratabilidade, 763
introduzindo um vértice, 853
introduzindo variável, 629
inválido, deslocamento, 717
inversa, substituição, 588
inversão de bits, permutação, 340 pr., 666
inversão em uma seqüência, 30 pr., 79-80 ex.
inverso
de uma função bijetora, 852
de uma matriz, 575, 578 ex.
de uma matriz a partir de uma decomposição de LUP, 597-598
em um grupo, 683
multiplicativo, módulo *n*, 690
inverso multiplicativo, módulo *n*, 690
inversor, 779
invertível, matriz, 575
inviável, programa linear, 617
inviável, solução, 617
irmão, 859
isolado, vértice, 854
isomórficos, grafos, 855
isomorfismo de subgrafo, problema, 805 ex.
iteração de função de prefixo, lema, 734
ITERATIVE-FFT, 666
ITERATIVE-TREE-SEARCH, 207-208

Jarvis, marcha, 754-755
Jensen, desigualdade, 876
JOHNSON, 505-506
Johnson, algoritmo, 503-506
Josephus, permutação, 255 pr.
junção
de árvores 2-3-4, 363 pr.
de árvores vermelho-preto, 237 pr.

k-ária, árvore, 860
Karmarkar, algoritmo, 616, 650
Karp, algoritmo do ciclo de peso médio mínimo, 487 pr.
k-cadeia, 864
k-CNF, 764
k-coloração, 804 pr., 861 pr.
k-combinação, 864
k-conjuntiva, forma normal, 764
k-ésima, potência, 677 ex.
Kleene, estrela (*), 771
KMP, algoritmo, 731-736
KMP-MATCHER, 733
Knuth-Morris-Pratt, algoritmo, 731-736
k-ordenado, 145 pr.
k-permutação, 864
Kraft, desigualdade, 861 ex.
Kruskal, algoritmo, 450-452
k-subcadeia, 864
k-subconjunto, 847-848
k-universal, hash, 203 pr.
k-vizinhos, árvore, 239-240

lado de um polígono, 742 ex.
Lagrange, fórmula, 654
Lagrange, teorema, 686
Lamé, teorema, 680
LCA, 415 pr.
LCS, *ver* subseqüência comum mais longa
LCS-LENGTH, 283
LEFT, 104
LEFT-ROTATE, 224, 253 ex.
Legendre, símbolo $(\frac{a}{p})$, 714 pr.
leve, aresta, 447
lexicográfica, ordenação, 217 pr.
lexicograficamente menor que, 217 pr.
lg (logaritmo binário), 42
lg lg (composição de logaritmos), 42
lg* (função logaritmo repetido), 44-45
lg^k (exponenciação de logaritmos), 42
liberação, hora, 321 pr.
liberação de objetos, 171-172
LIFO ("last in, first out", último a entrar, primeiro a sair), 163
ver também pilha
ligada, lista, 166-170
compacta, 173 ex., 177 pr.
eliminação de, 167-168
inserção em, 167
lista de vizinhos, 539-540
para implementar conjuntos disjuntos, 400-403
pesquisando, 167, 191 ex.
limitando um somatório, 838-844
limite
assintoticamente restrito, 33
assintótico inferior, 35
assintótico superior, 34
em coeficientes binomiais, 865-806
em distribuições binomiais, 881
nas extremidades de uma distribuição binomial, 883-889
polilogarítmico, 43
limite, condição, 51-52
limite de um polígono, 742 ex.
limite superior, propriedade, 465, 481
limpa, seqüência, 562
linear, dependência, 576
linear, desigualdade, 612
linear, função, 19, 612
linear, igualdade, 612
linear, independência, 576
linear, ordem, 851
linear, pesquisa, 16 ex.
linear, programação, 610-650
algoritmo de Karmarkar para, 616, 649
algoritmo simplex para, 626-638
algoritmos para, 615-616
aplicações de, 615
dualidade em, 638-643
e caminho mais curto de par único, 623
e caminhos mais curtos de única origem, 475-480

e fluxo de custo mínimo, 624-625
e fluxo de várias mercadorias, 625-626
e fluxo de várias mercadorias de custo mínimo, 626 ex.
e fluxo máximo, 623
encontrando uma solução inicial para, 643-647
forma padrão para, 616-619
forma relaxada para, 619-621
métodos de pontos interiores para, 615, 650
teorema fundamental da, 647
ver também inteiros, problema de programação linear; 0-1, problema de programação de inteiros
linear, restrição, 612
linear, sondagem, 193
lineares, equações
resolvendo, modulares, 688-690
resolvendo sistemas de, 585-597
resolvendo sistemas tridiagonais de, 607 pr.
linearidade de expectativa, 875
linearidade de somatórios, 836
linguagem, 770
completeza de, 787 ex.
provando o caráter NP-completo de, 785-786
verificação de, 774
linha, segmento, 739
descobrindo a mudança de direção de, 740
descobrindo se dois se interceptam, 740-742
descobrindo se quaisquer se interceptam, 743-749
linha, vetor, 572
linha de montagem, programação, 260-265
linha de varredura, status, 745-746
linhas, ordem, 576
LINK, 405
líquido, fluxo através de um corte, 518
lista, *ver* ligada, lista
lista de adjacência, representação, 420
lista filha em um heap de Fibonacci, 382
LIST-DELETE, 167-168
LIST-DELETE', 167-168
LIST-INSERT, 167-168
LIST-INSERT', 168-169
LIST-SEARCH, 167
LIST-SEARCH', 168-169
literal, 788
livre, árvore, 856, 857-858
livre, lista, 172
lixo, coleta, 103, 171
L_m-distância, 760 ex.
ln (logaritmo natural), 42
local, variável, 14
localização, caminho, 404
localização da agência postal, problema, 155-156 pr.
logaritmo, função (log), 42-43
discreta, 694
repetida (lg*), 44-45
logaritmo discreto, teorema, 694
lógica, porta, 779
LONGEST-PATH, 772 ex.
LONGEST-PATH-LENGTH, 772 ex.
LOOKUP-CHAIN, 279
loop, invariante, 13
e loops **for**, 12 n.
e término, 13
inicialização de, 13
manutenção de, 13
origem de, 31
para algoritmo de Prim, 454
para algoritmo simplex, 632
para aumentar uma chave em um heap, 114 ex.
para autômatos de correspondência de cadeias, 727-728, 729
para consolidar a lista de raízes na extração do nó mínimo de um heap de Fibonacci, 387
para construção de um heap, 108-109
para determinar a ordem de um elemento em uma árvore de ordem estatística, 244
para exponenciação modular, 696-697
para heapsort, 110 ex.
para inserção em árvore vermelho-preto, 227-228
para intercalação, 23
para o algoritmo de Dijkstra, 472
para o algoritmo de Rabin-Karp, 724
para o algoritmo de relabel-to-front, 543
para o algoritmo genérico de árvore de amplitude mínima, 446
para o algoritmo genérico de push-relabel, 534
para ordenação por inserção, 13-14
para particionamento, 118
para permutação aleatória de um arranjo, 82-83
para pesquisa em uma árvore de intervalos, 252
para pesquisa primeiro na extensão, 424-425
para regra de Horner, 30 pr.
para unificação de heaps binomiais, 378 ex.
loops, construções em pseudocódigo, 14
LU, decomposição, 590-593
LU-DECOMPOSITION, 592
LUP, decomposição, 587
computação de, 592-596
de uma matriz de permutação, 595 ex.
de uma matriz diagonal, 595 ex.
e multiplicação de matrizes, 600 ex.
em inversão de matrizes, 597-598
uso de, 587-590
LUP-DECOMPOSITION, 595
LUP-SOLVE, 589

mais longa, subseqüência comum, 281-285, 295
mais longo, caminho simples, 763
em um grafo não-ponderado, 274
mais longo, problema do ciclo simples, 802 ex.
MAKE-BINOMIAL-HEAP, 370
MAKE-HEAP, 365
MAKE-SET, 398
floresta de conjuntos disjuntos, implementação de, 405
lista ligada, implementação de, 400-401
MAKE-TREE, 414 pr.
Manhattan, distância, 156 pr., 760 ex.
mão única, função hash, 701
marcado, nó, 383, 391
Markov, desigualdade, 877-878 ex.
matric, matróide, 314
MATRIX-CHAIN-MULTIPLY, 271
MATRIX-CHAIN-ORDER, 270
MATRIX-MULTIPLY, 266, 494
matriz, 571-579
adjacência, 421
hermitiana, 600 ex.
incidência, 322 pr., 422 ex.
ordenando as entradas de, 568 ex.
predecessora, 491
pseudo-inversa de, 605
simétrica definida como positiva, 601-603
Toeplitz, 669 pr.
transposição conjugada de, 600 ex.
transposição de, 421, 572
ver também matrizes, inversão; matrizes, multiplicação
matriz de adjacência, representação, 421
matrizes, inversão, 597-600
matrizes, multiplicação
booleana, 600 ex.
e cálculo do determinante, 600 ex.
e decomposição de LUP, 600 ex.
e inversão de matrizes, 598-599
Pan, método, 585 ex.
para caminhos mais curtos de todos os pares, 491-497
Strassen, algoritmo, 579-586
matróide, 314-318, 322 pr., 458
mau sujeito, 428 ex.
MAX-3-CNF, satisfabilidade, 820
MAX-CNF, satisfabilidade, 823 ex.
MAX-CUT, problema, 823 ex.
MAX-FLOW-BY-SCALING, 548 pr.
MAX-HEAPIFY, 105-106
MAX-HEAP-INSERT, 113
construindo um heap com, 114 pr.
máxima, correspondência, 809, 829 pr.
máxima, correspondência bipartida, 526-529, 537 ex.
algoritmo de Hopcroft-Karp para, 549 pr.
máximas, camadas, 760 pr.
maximização, programa linear, 612
e programas lineares de minimização, 617
máximo, 147
de uma distribuição binomial, 882 ex.
em árvores de ordem estatística, 248 ex.
em árvores de pesquisa binária, 208
em árvores vermelho-preto, 222-223
em heaps, 112
localizando, 148
máximo, correspondência de, 829 pr.
máximo, fluxo, 509-551
algoritmo de Edmonds-Karp para, 523-525
algoritmo de relabel-to-front, 538-546
algoritmos de push-relabel para, 529-546
atualizando, 548 pr.
com capacidades negativas, 548-549 pr.

como um programa linear, 623
e correspondência bipartida máxima, 526-529
escalonamento, algoritmo, 548 pr.
Ford-Fulkerson, método, 515-526
máximo, grau em um heap de Fibonacci, 384, 390 ex., 394-396
máximo, heap, 104
aumentando uma chave em, 112-113
chave máxima de, 112
como uma fila de prioridade máxima, 111-114
construindo, 107-109
eliminação de, 115 ex.
em heapsort, 109-112
extraindo a chave máxima de, 112
inserção em, 113
intercalável, *ver* intercalável, heap máximo
máximo, ponto, 760 pr.
máximo divisor comum (mdc), 675
calculado por algoritmo binário de mdc, 713 pr.
calculado por algoritmo de Euclides, 677-682
com mais de dois argumentos, 681 ex.
teorema de recursão para, 678
MAXIMUM, 111, 160
MAYBE-MST-A, 457 pr.
MAYBE-MST-B, 457 pr.
MAYBE-MST-C, 458 pr.
mdc, 674-675, 677 ex.
ver também máximo divisor comum
mediana, 147-157
de listas ordenadas, 155 ex.
ponderada, 155 pr.
mediana, chave de um nó de árvore B, 356
mediana de 3, método, 130-131 pr.
médio, peso de um ciclo, 487 pr.
médio, *ver* esperado, valor
meio aberto, intervalo, 249
melhor caso, tempo de execução, 20 ex., 36
membro de um conjunto (∈), 845
MEMOIZED-MATRIX-CHAIN, 279
memória, 349
memória, hierarquia, 16-17
memorização, 278-280
menor ancestral comum, 415 pr.
menor de uma matriz, 577
mercadoria, 625
MERGE, 22
MERGE-LISTS, 824
MERGER, 565
MERGE-SORT, 24-25
árvore de recursão para, 280 ex.
mestre, método para resolver uma recorrência, 59-61
mestre, teorema, 59
prova de, 61-68
metade 3-CNF, satisfabilidade, 803 ex.
MILLER-RABIN, 706
Miller-Rabin, teste de caráter primo, 704-709
MIN-GAP, 254 ex.
MIN-HEAPIFY, 107 ex.
MIN-HEAP-INSERT, 114 ex.
mínima, árvore de amplitude, 445-458
algoritmo genérico, 446-450
algoritmo de Boruvka para, 458
construída com o uso de heaps binomiais, 379 pr.
em algoritmo de aproximação para problema do caixeiro-viajante, 811
em grafos dinâmicos, 455 ex.
Kruskal, algoritmo, 450-452
Prim, algoritmo, 452-454
relação com matróides, 314, 315-316
segundo melhor, 455 pr.
mínima, chave em heaps 2-3-4, 379 pr.
mínima, cobertura de caminho, 546 pr.
minimização, programa linear, 612
e programas lineares de maximização, 617
mínimo, 147
em árvores B, 359 ex.
em árvores de ordem estatística, 248 ex.
em árvores de pesquisa binária, 208
em árvores vermelho-preto, 222-223
em heaps binomiais, 370-371
em heaps de Fibonacci, 385
localizando, 148
off-line, 413 pr.
mínimo, ciclo de peso médio, 487 pr.
mínimo, corte, 518
mínimo, grau de uma árvore B, 353
mínimo, heap, 104
analisado por método potencial, 333
como uma fila de prioridade mínima, 114 ex.
construindo, 107-109
intercalável, *ver* intercalável, heap mínimo
no algoritmo de Dijkstra, 473-474
no algoritmo de Huffman, 310
no algoritmo de Johnson, 506
no algoritmo de Prim, 454
mínimo, nó de um heap de Fibonacci, 383
mínimo múltiplo comum, 661 ex.
mínimos quadrados, aproximação, 603-606
MINIMUM, 111-112, 147-148, 160
mmc (mínimo múltiplo comum), 682 ex.
mochila, problema
fracionária, 269, 307 ex.
0-1, 305-306, 307 ex.
mochila fracionária, problema, 305-306, 307 ex.
mod, 40, 674
modular, aritmética, 41, 682-687
modular, exponenciação, 696
modulares, equações lineares, 688-690
MODULAR-EXPONENTIATION, 696
MODULAR-LINEAR-EQUATION-SOLVER, 689-690
módulo, 40, 674
moeda, troca, 321 pr.
Monge, arranjo, 71 pr.
monotônica, seqüência, 116
monotonicamente crescente, 40
monotonicamente decrescente, 40
movimentação de dados, instruções, 16-17
MST, 380 pr.
MST-KRUSKAL, 450
MST-PRIM, 453-454
MST-REDUCE, 456-457 pr.
multidimensional, transformação rápida de Fourier, 669 pr.
multigrafo, 856
convertendo no grafo não orientado equivalente, 421 ex.
múltipla, atribuição, 14
múltiplas, origens e depósitos, 512
multiplicação
de matrizes, 574, 578-579 ex.
de números complexos, 585 ex.
de polinômios, 652
de uma cadeia de matrizes, *ver* cadeia de matrizes, multiplicação
dividir e conquistar, método para, 669 pr.
módulo n ($\cdot_n$), 683
multiplicação, instrução, 16-17
multiplicação, método, 187-188
múltiplo, 574, 673
de um elemento, módulo n, 688-690
mínimo comum, 682 ex.
múltiplo, número, 673
testemunha para, 704
MULTIPOP, 325
MULTIPUSH, 327 ex.
mutuamente exclusivos, eventos, 868
mutuamente independentes, eventos, 870

N (conjunto de números naturais), 845
n elementos, conjunto, 848
NAIVE-STRING-MATCHER, 717
não básica, variável, 620
não correspondente, vértice, 525
não determinístico, tempo de polinômio, 774 n.
ver também NP
não divide, relação, 673
não enumerável, conjunto, 848
não hamiltoniano, grafo, 773
não instância, 769 n.
não invertível, matriz, 575
não limitado, programa linear, 616-617
não negatividade, restrição, 616, 617
não ordenada, árvore binomial, 383
não ordenada, lista ligada, 166
ver também ligada, lista
não orientada, versão de um grafo orientado, 855
não orientado, grafo, 853
biconectado, componente de, 442 pr.
calculando uma árvore de amplitude mínima em, 445-458
clique em, 791
cobertura de vértices de, 794, 807-808
coloração de, 804 pr., 861 pr.
conjunto independente de, 803 pr.
convertendo um multigrafo em, 421 ex.
correspondência de, 525-526
d-regular, 529 ex.
grade, 546 pr.
hamiltoniano, 773
não hamiltoniano, 773
ponte de, 442 pr.
ponto de articulação de, 442 pr.
ver também grafo
não ponderados, caminhos mais curtos, 273-274
não ponderados, caminhos simples mais longos, 274

não saturante, empurrão, 532, 536
não singular, matriz, 575
não sobreposto, padrão de cadeia, 730 ex.
não trivial, potência, 676-677 ex.
não trivial, raiz quadrada de 1, módulo *n*, 695
naturais, números (N), 845
natural, curva cúbica, 607 pr.
nebulosa, ordenação, 131 pr.
negativa de uma matriz, 574
nenhum caminho, propriedade, 464-465, 481-482
NEXT-TO-TOP, 750
NIL, 15
nível de uma função, 407
nó, 858
  ver também vértice
nó, de uma curva, 607 pr.
norma de um vetor, 575
normal, equação, 605
NOT, função (¬), 779
NOT, porta, 779
novamente identificado, vértice, 532
NP (classe de complexidade), 764, 775, 776 ex.
NPC (classe de complexidade), 765, 768
caráter NP-completo, 763-785
  de determinar se uma fórmula booleana é uma tautologia, 790 ex.
  de programação com lucros e prazos finais, 804 pr.
  do problema da cobertura de conjuntos, 818 ex.
  do problema da cobertura de vértices, 794-795
  do problema da partição de conjuntos, 802 ex.
  do problema da programação de inteiros de 0 a 1, 802 ex.
  do problema da satisfabilidade de 3-CNF, 788-790
  do problema da satisfabilidade de circuito, 779-784
  do problema da satisfabilidade de fórmula, 785-787
  do problema da satisfabilidade de metade 3-CNF, 803 ex.
  do problema da soma de subconjuntos, 799-802
  do problema de coloração de grafos, 804 pr.
  do problema de programação linear de inteiros, 802 ex.
  do problema do caixeiro-viajante, 799
  do problema do caminho hamiltoniano, 802 ex.
  do problema do ciclo hamiltoniano, 795-798
  do problema do ciclo simples mais longo, 802 ex.
  do problema do clique, 791-794
  do problema do conjunto independente, 803 pr.
  do problema do isomorfismo de subgrafos, 802 ex.
  prova, de uma linguagem, 785
NP-completo, 764, 778
NP-difícil, 778

n-tupla, 848-849
núcleo de um polígono, 756 ex.
nula, árvore, 860
nulo, evento, 668
nulo, vetor, 576
numérica, estabilidade, 571, 587, 608-609
números aleatórios, gerador, 75
números complexos, multiplicação de, 585 ex.
números primos, teorema, 703
números pseudo-aleatórios, gerador, 75

O, notação, 34 fig., 34-35
o, notação, 37-38
O', notação, 48 pr.
Õ, notação, 48 pr.
O maiúsculo, notação, 34 fig., 34-35
o minúsculo, notação, 37-38
objetivo, função, 475, 479-480 ex., 613, 616
objetivo, valor, 614, 617
  ótimo, 617
objeto, 15
  alocação e liberação de, 171-172
  implementação de arranjo de, 170-173
  passagem como parâmetro, 15
oblíqua, árvore, 346
ocorrência de um padrão, 717
ocorrência em ciclo, de algoritmo simplex, 636
off-line, problema
  ancestrais comuns mínimos, 415 pr.
  mínimo, 413 pr.
OFF-LINE-MINIMUM, 414 pr.
Ômega, notação, 34 fig., 36
ômega maiúsculo, notação, 34 fig., 36
ômega minúsculo, notação, 38
on-line, problema da contratação, 93-95
on-line, problema da envoltória convexa, 756 ex.
ON-LINE-MAXIMUM, 93
ON-SEGMENT, 741
OPTIMAL-BST, 289
OR, função (∨), 779
OR, porta, 779
ordem
  colunas, 576
  de um nó em uma floresta de conjuntos disjuntos, 404, 408, 413 ex.
  de um número em um conjunto ordenado, 242
  de uma matriz, 576, 578-579 ex.
  em árvores de ordem estatística, 244-245, 246 ex.
  linhas, 576
  total, 576
ordem
  de um grupo, 686
  linear, 851
  parcial, 850
  total, 851
ordem, estatísticas, 147-157
  dinâmicas, 242-247
ordem de crescimento, 20
ordem dinâmica, estatística, 242-247
ordem estatística, árvore, 242-247
  consultando, 249 ex.
ordenação, rede, 557
  AKS, 570
  baseada na ordenação por inserção, 558 ex.
  baseada na ordenação por intercalação, 566-568
  bitônica, 561-564
  ímpar-par, 568 pr.
  profundidade, 567 ex.
  tamanho, 558 ex.
ordenada, árvore, 859
ordenada, lista ligada, 166
  ver também ligada, lista
ordenado, par, 848
ordenando, 11-14, 21-28, 99-146
  bubblesort, 29 pr.
  bucket sort, 140-143
  de pontos por ângulo polar, 743 ex.
  de uma matriz, 568 ex.
  em tempo linear, 135-143, 143 pr.
  heapsort, 103-116
  itens de comprimento variável, 144 pr.
  lexicográfica, 217 pr.
  limite inferior do caso médio para ordenação, 144 pr.
  limites inferiores para, 133-136
  nebulosa, 131 pr.
  no local, 12, 100
  ordenação de seleção, 20 ex.
  ordenação de Shell, 31
  ordenação por comparação, 133
  ordenação por contagem, 136-137
  ordenação por inserção, 8, 11-14
  ordenação por intercalação, 8, 21-28
  problema de, 3, 11, 99
  quicksort, 117-132
  radix sort, 137-139
  rede para, *ver* ordenação, rede
  topológica, *ver* topológica, ordenação
  usando redes, *ver* ordenação, rede
  usando uma árvore de pesquisa binária, 212 ex.
orientada, versão de um grafo não orientado, 855
orientado, grafo, 853
  caminho mais curto em, 459
  caminhos mais curtos de todos os pares em, 490-508
  caminhos mais curtos de única origem em, 459-489
  cobertura de caminho de, 546 pr.
  conectados isoladamente, 436 ex.
  e caminhos mais longos, 763
  fechamento transitivo de, 500
  hamiltoniano, ciclo de, 764
  PERT, diagrama, 470, 470 ex.
  quadrado de, 421 ex.
  restrição, grafo, 477
  semiconectado, 442 ex.
  transposição de, 421 ex.
  viagem de Euler de, 443 pr., 763
  ver também circuito; orientado, grafo acíclico; grafo; rede
orientado, grafo acíclico (grafo), 856
  algoritmo para caminhos mais curtos de única origem, 468-470
  e arestas traseiras, 436
  e grafos componentes, 439
  e problema do caminho hamiltoniano, 776 ex.
  ordenação topológica de, 436
orientado, segmento, 739

origem, 422, 460, 510, 512
origem, 739
ortonormal, 608-609
OS-KEY-RANK, 246 ex.
OS-RANK, 244
OS-SELECT, 244
ótima, árvore de pesquisa binária, 285-290, 295
ótima, cobertura de vértices, 807-808
ótima, solução, 617
ótima, subestrutura
de árvores de pesquisa binária, 287-288
de caminhos mais curtos, 460-461, 492, 497
de caminhos mais curtos não ponderados, 274
de códigos de Huffman, 312-313
de matróides ponderados, 317
de multiplicação de cadeias de matrizes, 267-268
de programação de linha de montagem, 261-262
de seqüencial de atividade, 371-373
de subseqüências comuns mais longas, 282
do problema da mochila 0-1, 305-306
do problema da mochila fracionária, 305-306
em algoritmos gulosos, 305
em programação dinâmica, 272-276
otimização, problema, 259, 764-765, 768
algoritmos de aproximação para, 806-831
e problemas de decisão, 765
ótimo, subconjunto de um matróide, 315-316
ótimo, valor de objetivo, 617
ou (or), em pseudocódigo, 15

P (classe de complexidade), 764, 768, 771-772, 773 ex.
pacote, embalagem, 754
padrão, correspondência, *ver* cadeias, correspondência
padrão, forma, 612-613, 616-619
padrão em correspondência de cadeias, 717
não sobrepostas, 730 ex.
página em um disco, 350-351 pr.
paginando, 16-17
pai, 858-859
em uma árvore de primeiro na extensão, 423
Pan, método para multiplicação de matrizes, 585 ex.
par, ordenado, 847-848
par mais distante, problema, 749-750
par mais próximo, localizando, 756-760
par único, caminho mais curto, 273-274
como um programa linear, 622-623
parâmetro, 15
custos da passagem, 68 pr.
parcial, ordem, 850
PARENT, 104
parênteses, teorema, 431
partição de conjunto, problema, 802 ex.
partição de um conjunto, 847-848, 850
particionamento, algoritmo, 118-120
aleatório, 124
em torno da mediana de 3 elementos, 128 ex.
PARTITION, 118
Pascal, triângulo, 867 ex.
PATH, 765, 771
percurso de uma árvore, *ver* árvore, percurso
perdido, filho, 860
perfeita, correspondência, 529 ex.
perfeito, hash, 198-201
permutação, 852
aleatória, 81-83
aleatória uniforme, 74-75, 81
de um conjunto, 864
inversão de bits, 340 pr., 666
Josephus, 255 pr.
no local, 82
permutação, matriz, 574, 578 ex.
decomposição de LUP de, 596 ex.
permutação, rede, 569 pr.
PERMUTE-BY-CYCLIC, 84 ex.
PERMUTE-BY-SORTING, 81
PERMUTE-WITH-ALL, 84 ex.
PERMUTE-WITHOUT-IDENTITY, 84 ex.
persistente, estrutura de dados, 236 pr., 346
PERSISTENT-TREE-INSERT, 236 pr.
PERT, diagrama, 470, 470 ex.
peso
de um caminho, 459
de um corte, 823 ex.
de uma aresta, 421
médio, 488 pr.
peso, função
em um matróide ponderado, 315
para um grafo, 421
peso balanceado, árvore, 241, 342 pr.
peso mínimo, árvore de amplitude, *ver* mínima, árvore de amplitude
peso mínimo, cobertura de vértices, 820-823
peso negativo, arestas, 461
peso negativo, ciclo
e caminhos mais curtos, 461
e relaxação, 485 ex.
e restrições de diferença, 477
pesquisa, árvore, *ver* pesquisa balanceada, árvore; pesquisa binária, árvore; B, árvore; exponencial, árvore de pesquisa; intervalos, árvore; ótima, árvore de pesquisa binária; ordem estatística, árvore; vermelho-preto, árvore; oblíqua, árvore; 2-3, árvore; 2-3-4, árvore
pesquisa binária, árvore, 204-219
árvores AA, 241
árvores AVL, 237-238 pr.
árvores de bode expiatório, 241
árvores de *k* vizinhos, 241
árvores de peso balanceado, 241
árvores oblíquas, 241
chave máxima de, 208
chave mínima de, 208
com chaves iguais, 216 pr.
construída aleatoriamente, 213-216, 217-218 pr.
consultando, 207-210
e treaps, 237-238 pr.
eliminação de, 211-212
inserção em, 210
ótima, 285-291, 295
para ordenação, 212 ex.
pesquisando, 207-208
predecessor em, 208-209
sucessor em, 208-209
ver também vermelho-preto, árvore
pesquisando
em árvores B, 354-355
em árvores de intervalos, 251-253
em árvores de pesquisa binária, 207-208
em árvores vermelho-preto, 222-223
em listas compactas, 177 pr.
em listas ligadas, 167
em tabelas de endereço direto, 180
em tabelas hash de endereço aberto, 192-193
em tabelas hash encadeadas, 193
em um arranjo não ordenado, 95 pr.
para um intervalo exato, 254 ex.
pesquisa binária, 28 ex.
pesquisa linear, 15 ex.
problema de, 15 ex.
phi, função, 685
pilha, 163-164
em busca de Graham, 750
implementação de lista ligada de, 169 ex.
implementada por filas, 166 ex.
no armazenamento secundário, 363 pr.
operações analisadas pelo método de contabilidade, 328-329
operações analisadas pelo método potencial, 331
operações analisadas por análise agregada, 325-327
para execução de procedimento, 130-131 pr.
pior caso, tempo de execução, 20, 36
PISANO-DELETE, 396 pr.
piso, função, $\lfloor\rfloor$, 40
em teorema mestre, 65-68
piso, instrução, 16-17
pivô
em decomposição LU, 592
em programação linear, 629-631, 637 ex.
em quicksort, 118
PIVOT, 630
podando um heap de Fibonacci, 396-397 pr.
podando uma lista, 824
podar e pesquisar, método, 749-750
polígono, 743 ex.
estrelado, 756 ex.
núcleo de, 756 ex.
polilogaritmicamente limitado, 43
polinomialmente limitado, 41
polinomialmente relacionado, 769
polinômio, 41, 651
adição de, 651
assintótico, comportamento de, 46 pr.
avaliação de, 30 pr., 653, 657 ex., 669-670 pr.
interpolação por, 653, 658 ex.
multiplicação de, 652, 655-657, 669 pr.
representação de coeficientes de, 653
representação de valor de ponto de, 653-654
Pollard, heurística rho, 710-713, 713 ex.
POLLARD-RHO, 710

ponderada, cobertura de vértices, 820-822
ponderada, correspondência bipartida, 397
ponderada, heurística de união, 402
ponderada, mediana, 155 pr.
ponderado, matróide, 315-318
ponderado, problema de cobertura de conjuntos, 828 pr.
ponte, 442 pr.
ponteiro, 15
ponto de evento, programação, 745
ponto extremo
de um intervalo, 249
de um segmento de linha, 739
ponto flutuante, tipo de dados, 16-17
ponto interior, método, 615
ponto mais próximo, heurística, 815 ex.
POP, 164
porta, 779
posição, 179-180
posicional, árvore, 861
positiva simétrica, matriz definida, 601-603
positivo, fluxo, 510
pós-ordem, percurso de árvore, 205
potência
de um elemento, módulo *n*, 693-698
k-ésima, 677 ex.
potencial, função, 330
para limites inferiores, 343
potencial, método, 330-333
para algoritmo de Knuth-Morris-Pratt, 733-734
para algoritmo genérico de push-relabel, 536-537
para contadores binários, 331-332
para estruturas de dados de conjuntos disjuntos, 408-412
para heaps de Fibonacci, 384-386, 389-390, 393
para heaps mínimos, 333 ex.
para operações de pilhas, 330-331
para reestruturação de árvores vermelho-preto, 342 pr.
para tabelas dinâmicas, 335-336, 338-340
potencial de uma estrutura de dados, 330
potências, conjunto, 848
potências, série, 69 pr.
Pr{ } (distribuição de probabilidades), 868
prazo final, 319
predecessor
em árvores B, 359-360 ex.
em árvores de caminhos mais curtos, 462
em árvores de ordem estatística, 248 ex.
em árvores de pesquisa binária, 208-209
em árvores vermelho-preto, 222
em listas ligadas, 166
PREDECESSOR, 160
predecessor, subgrafo
em caminhos mais curtos de todos os pares, 491
em caminhos mais curtos de única origem, 462
em pesquisa primeiro na extensão, 426
em pesquisa primeiro na profundidade, 429
predecessora, matriz, 491
preempção, 321 pr.
prefixo
de uma cadeia ($\sqsubset$), 718
de uma seqüência, 281
prefixo, código, 308
prefixo, função, 730-732
pré-ordem, percurso de árvore, 205
pré-ordenação, 759
presente, embalagem, 754-755
preto, altura, 222
preto, vértice, 422, 429
Prim, algoritmo, 452-454
com pesos de arestas inteiros, 455 ex.
com uma matriz de adjacência, 454 ex.
em algoritmo de aproximação para o problema do caixeiro-viajante, 811
implementado com um heap de Fibonacci, 454
implementado com um heap mínimo, 454
para grafos esparsos, 456 pr.
semelhança com o algoritmo de Dijkstra, 452, 473-474
primal, programa linear, 638
primeiro a entrar, primeiro a sair, 163
ver também fila
primeiro ajuste, heurística, 828 pr.
primeiro na extensão, árvore, 423, 427
primeiro na extensão, pesquisa, 422-428
e caminhos mais curtos, 425-426, 460
semelhança com o algoritmo de Dijkstra, 473, 474 ex.
primeiro na profundidade, árvore, 429
primeiro na profundidade, floresta, 429
primeiro na profundidade, pesquisa, 429-436
na localização de componentes fortemente conectados, 438-441
na localização de pontos de articulação, pontes e componentes biconectados, 443 pr.
na ordenação topológica, 436-438
primeiro-adiantada, forma, 319
primo, número, 674
densidade de, 702-703
primos, função de distribuição, 703
principal, agrupamento, 193
principal, memória, 349
principal, raiz da unidade, 658-659
princípio de inclusão e exclusão, 849 ex.
PRINT-ALL-PAIRS-SHORTEST-PATH, 491
PRINT-INTERSECTING-SEGMENTS, 748 ex.
PRINT-LCS, 284
PRINT-OPTIMAL-PARENS, 271, 271 ex.
PRINT-STATIONS, 265
prioridade máxima, fila, 111-112
prioridade mínima, fila, 111-112, 114 ex.
na construção de códigos de Huffman, 310
no algoritmo de Dijkstra, 473
no algoritmo de Prim, 454
prioridades, fila, 111-114
com extrações monotônicas, 116
fila de prioridade máxima, 111-112
fila de prioridade mínima, 111-112, 114 ex.
implementação de heap de, 111-114
na construção de códigos de Huffman, 310
no algoritmo de Dijkstra, 473
no algoritmo de Prim, 454
ver também pesquisa binária, árvore; binomial, heap; Fibonacci, heap
probabilidade, 868-873
probabilidades, distribuição, 868
probabilidades, distribuição discreta, 869
probabilística, análise, 74-75, 85-95
de algoritmo de aproximação para satisfabilidade de MAX-3-CNF, 820
de algoritmo de Rabin-Karp, 724
de altura de uma árvore de pesquisa binária construída aleatoriamente, 213-216
de bolas e caixas, 88-89
de bucket sort, 140-143, 143 ex.
de colisões, 185 ex., 201 ex.
de comparação de arquivos, 724 ex.
de contagem probabilística, 95 pr.
de envoltória convexa em uma distribuição de envoltória esparsa, 964 pr.
de hash com encadeamento, 183-184
de hash de endereço aberto, 195-197, 197 ex.
de hash perfeito, 199-201
de hash universal, 188-190
de heurística rho de Pollard, 711-713
de inserção em uma árvore de pesquisa binária comchavesiguais, 216 pr.
de limite de prova mais longo para hash, 201 pr.
de limite de tamanho de posição para encadeamento, 202 pr.
de limite inferior do caso médio para ordenação, 145-146 pr.
de paradoxo do aniversário, 85-87
de particionamento, 123 ex., 128 ex., 129 pr., 131 pr.
de pesquisa em uma lista compacta, 177 pr.
de pontos de ordenação por distância a partir da origem, 143 ex.
de profundidade de nó médio em uma árvore de pesquisa binária construída aleatoriamente, 217 pr.
de quicksort, 126-128, 129 pr., 131 pr., 216 ex.
de seleção aleatória, 149-152
de seqüências, 89-92
do problema da contratação, 78-79
do teste de caráter primo de Miller-Rabin, 707-709
e algoritmos aleatórios, 79-81
e entradas médias, 19-20
probabilística, contagem, 95 pr.
problema
abstrato, 767-768
computacional, 3-4

concreto, 768
decisão, 765, 767-768
intratável, 763
otimização, 259, 764, 767-768
solução para, 4, 768
tratável, 763
problema de satisfabilidade de fórmula, 786-787
problema do caixeiro-viajante
algoritmo de aproximação para, 810-815
caráter NP-completo de, 799
com desigualdade de triângulos, 811-813
euclidiano bitônico, 291 pr.
gargalo, 814-815 ex.
sem desigualdade de triângulos, 813-814
procedimento, 4, 11-12
produto
cartesiano, 848
cruzado, 738-739
de matrizes, 574-575, 578-579 ex.
de polinômios, 652
externo, 575
interno, 575
regra de, 864
profissional, lutador, 428 ex.
profundidade
de árvore de recursão de quicksort, 123 ex.
de SORTER, 567 ex.
de um nó em uma árvore enraizada, 859
de uma pilha, 131 pr.
de uma rede de comparação, 557-558
de uma rede de ordenação, 558 ex.
média, de um nó em uma árvore de pesquisa binária construída aleatoriamente, 217 pr.
programa, contador, 781-782
programação, 829 pr.
ponto de evento, 745
programação (agendamento), 295 pr., 321 pr., 804 pr., 829 pr.
programação, *ver* dinâmica, programação; linear, programação
programação de máquina paralela, problema, 829 pr.
programação dinâmica, método, 259-295
comparação com algoritmos gulosos, 273-274, 280 ex., 299-300, 304, 305-307
e memorização, 278-280
elementos de, 272-280
para algoritmo de Floyd-Warshall, 497-500
para algoritmo de Viterbi, 294 pr.
para árvores de pesquisa binária ótimas, 285-291
para caminhos mais curtos de todos os pares, 492-500
para distância de edição, 291 pr.
para fechamento transitivo, 500-502
para impressão esmerada, 291 pr.
para multiplicação de cadeias de matrizes, 266-272
para problema de mochila de 0-1, 307 ex.
para problema euclidiano bitônico do caixeiro-viajante, 260 pr.
para programação, 295 pr.
para programação de linha de montagem, 260-265
para seleção de atividade, 303 ex.
para subseqüência comum mais longa, 281-285
reconstruindo uma solução ótima em, 278
sobrepondo subproblemas em, 276-278
subestrutura ótima em, 272-276
programação linear, relaxação, 821
programação linear inteira, problema, 616, 649 pr., 802 ex.
pronto, vértice, 429
próprio, ancestral, 858
próprio, descendente, 858
próprio, subconjunto ($\subset$), 846
próprio, subgrupo, 686
prova, 192, 201 pr.
prova, seqüência, 192
pseudocódigo, 11, 14-15
pseudo-inverso, 605
PSEUDOPRIME, 704
pseudoprimo, 704
pública, chave, 697, 699-700
PUSH
pilha, operação, 194
push-relabel, operação, 531
push-relabel, algoritmo, 529-546
algoritmo genérico, 532-538
algoritmo relabel-to-front, 538-546
com uma fila de vértices de sobrecarga, 546 ex.
descarregando um vértice de sobrecarga de altura máxima, 546 ex.
heurística de intervalos para, 546 ex.
operações básicas em, 531-532
para encontrar uma correspondência bipartida máxima, 537 ex.
push-relabel, algoritmo genérico, 532-538

quadrada, matriz, 572
quadrado de um grafo orientado, 421 ex.
quadrática, função, 19
quadrática, sondagem, 194, 202 pr.
quadrático, resíduo, 714 pr.
quantil, 154 ex.
quicksort, 117-132
análise de, 120-124, 125-128
análise do pior caso de, 125
boa implementação do pior caso de, 154 ex.
caso médio, análise de, 125-128
com método de mediana de 1, 130 pr.
descrição de, 117-120
em comparação com a ordenação por inserção, 123 ex.
em comparação com a radix sort, 139
profundidade de pilha de, 130 pr.
uso de ordenação por inserção em, 128 ex.
versão aleatória de, 124, 129 pr.
versão de extremidade recursiva de, 130 pr.
QUICKSORT, 118
QUICKSORT', 130 pr.
quociente, 673-674

R (conjunto de números reais), 845
Rabin-Karp, algoritmo, 721-725
RABIN-KARP-MATCHER, 723-724
radix sort, 137-139
comparação com quicksort, 139
RADIX-SORT, 139
raio, 743 ex.
raiz
de uma árvore, 858
de unidade, 658
raiz, árvore, 217 pr.
raiz quadrada, módulo primo, 714-715 pr.
raízes, lista
de um heap binomial, 368
de um heap de Fibonacci, 383
RAM, *ver* acesso aleatório, máquina
RANDOM, 75, 75 ex.
RANDOMIZED-HIRE-ASSISTANT, 80
RANDOMIZED-PARTITION, 124
RANDOMIZED-QUICKSORT, 124, 216 ex.
relação com árvores de pesquisa binária construídas aleatoriamente, 217 pr.
RANDOMIZED-SELECT, 186
RANDOMIZE-IN-PLACE, 82-83
RANDOM-SEARCH, 95-96 pr.
RB-DELETE, 231
RB-DELETE-FIXUP, 232
RB-ENUMERATE, 249 ex.
RB-INSERT, 225
RB-INSERT-FIXUP, 226
RB-JOIN, 237 pr.
reais, números (R), 845
reconstruindo uma solução ótima em programação dinâmica, 278
recorrência, 25, 50-72
solução pelo método de Akra-Bazzi, 72
solução pelo método de árvore de recursão, 54-58
solução pelo método de substituição, 51-54
solução pelo método mestre, 59-61
recorrência, equação, *ver* recorrência
recortando, em um heap de Fibonacci, 391-392
recuo em pseudocódigo, 14
recursão, 21
recursão, árvore, 27-28, 54-58
e o método de substituição, 57-58
em prova de teorema mestre, 61-63
para ordenação por intercalação, 280 ex.
RECURSIVE-ACTIVITY-SELECTOR, 301
RECURSIVE-FFT, 662
RECURSIVE-MATRIX-CHAIN, 277
rede
admissível, 538-540
bitônica, ordenação, 561-564
comparação, 555-559
fluxo, *ver* fluxo, rede
ímpar-par, intercalação, 568 pr.
ímpar-par, ordenação, 568 pr.
ordenação, 565-570
para intercalação, 564-566
permutação, 569 pr.
residual, 515-517
transposição, 568 pr.

redefinindo o peso
em caminhos mais curtos de todos os pares, 503
em caminhos mais curtos de única origem, 486 pr.
redução, algoritmo, 766, 777
redução, função, 777
redutibilidade, 777-779
referência a um conjunto de arestas, 447
reflexiva, relação, 849
reflexividade de notação assintótica, 39
região, viável, 612
registro (dados), 99
regra da soma, 863
regra do produto, 863-864
rejeição
por um algoritmo, 771
por uma automação finita, 726
RELABEL, 532
relabel, operação (em algoritmos push-relabel), 532, 535
RELABEL-TO-FRONT, 543
relabel-to-front, algoritmo, 538-546
relação, 849-851
relativamente, primo, 676
RELAX, 464
relaxação,
de uma aresta, 463-465
programação linear, 821
relaxação, 619
relaxação de caminho, propriedade, 465, 482
relaxada, árvore vermelho-preto, 223 ex.
relaxada, forma, 612-613, 619-621
unicidade de, 635
relaxada, variável, 619
relaxado, heap, 396-397
repeat, em pseudocódigo, 14
repetida, elevação ao quadrado
para caminhos mais curtos de todos os pares, 494-496
para elevar um número a uma potência, 696
repetida, função, 48 pr.
repetida, função logaritmo, 44-45
REPETITION-MATCHER, 737 pr.
representação de um conjunto, 398
RESET, 330 ex.
residual, aresta, 516
residual, capacidade, 515, 517
residual, rede, 515-517
resíduo, 40, 673-674, 714 pr.
resto, 40, 673-674
resto, instrução, 16-17
restrição, 616
desigualdade, 616-617, 618
diferença, 476
igualdade, 479 ex., 617, 618
linear, 612
não negatividade, 616, 618
rígida, 627
violação de, 627
restrição, grafo, 477-478
retângulo, 254 ex.
retorno, instrução, 16-17
rho, heurística, 710-713, 713 ex.
RIGHT, 104
RIGHT-ROTATE, 224
rígida, restrição, 627
rotação
cíclica, 736 ex.
em uma árvore vermelho-preto, 223-225
rotacional, varredura, 749, 750-754
RSA, sistema de criptografia de chave pública, 697-702

saída
de um algoritmo, 3
de um circuito combinacional, 780
de uma porta lógica, 779
saída, fio, 556
saída, grau, 854
saída, seqüência, 556
saltos, lista, 241
SAME-COMPONENT, 400
SAT, 786
satélite, dados, 99, 159
satisfabilidade, 780, 785-788, 819-820, 823 ex.
satisfação, atribuição, 780, 785
satisfatória, fórmula, 764, 785
saturação, empurrão, 531, 536
saturada, aresta, 531
Schur, complemento, 591, 602
SCRAMBLE-SEARCH, 96 pr.
SEARCH, 160
secreta, chave, 697, 699-700
secundária, tabela hash, 198
secundário, agrupamento, 194
secundário, armazenamento
árvore de pesquisa para, 349-364
pilhas em, 363 pr.
segmento, *ver* orientado, segmento; linha, segmento
SEGMENTS-INTERSECT, 741
segunda melhor, árvore de amplitude mínima, 456 pr.
segura, aresta, 446
seleção
de atividades, *ver* seleção de atividades, problema
e ordenações de comparação, 154
em árvores de ordem estatística, 243-244
em tempo linear esperado, 149-152
problema de, 147
tempo linear no pior caso, 152-155
seleção, ordenação, 20 ex.
seleção de atividade, problema, 297-303
SELECT, 152-153
seletor, vértice, 796
semiconectado, grafo, 442 ex.
sentinela, 22, 167-169, 222
seqüência (⟨⟩)
bitônica, 488 pr., 561
entrada, 556
finita, 852
infinita, 852
inversão em, 30 pr., 79-80 ex.
limpa, 562
prova, 192
saída, 556
seqüências, 89-92
série, 69 pr., 836-837
Shell, ordenação, 31
SHORTEST-PATH, 765
símbolos, tabela, 179, 186, 188
simetria de notação de Θ, 39
simetria de transposição de notação assintótica, 39
simétrica, diferença, 549-550 pr.
simétrica, matriz, 573-574, 578 ex.
simétrica, relação, 849-850
simples, algoritmo para correspondência de cadeias, 719-721
simples, caminho, 854
mais longo, 274, 763
simples, ciclo, 854
simples, grafo, 855
simples, hash uniforme, 183
simples, polígono, 743 ex.
simplex, 614
SIMPLEX, 632
simplex, algoritmo, 614, 626-638, 650
singular, decomposição de valores, 608-609
singular, matriz, 575
sistemas de equações lineares, 586-597, 607 pr.
sistemas de restrições de diferença, 475-480
SLOW-ALL-PAIRS-SHORTEST-PATHS, 494
sobre, 852
sobrecarga
de uma fila, 164
de uma pilha, 164
sobrecarregando vértice, 530
sobrejeção, 852
solução
básica, 628
inviável, 616-617
ótima, 616-617
para um problema abstrato, 768
para um problema computacional, 4
para um problema concreto, 468
para um sistema de equações lineares, 586
viável, 475, 613, 617
soma
cartesiana, 658 ex.
de matrizes, 573-574
de polinômios, 651
fluxo, 514 ex.
infinita, 835
inserindo, 837-838
regra da, 863
soma de subconjuntos, problema
algoritmo de aproximação para, 823-827
caráter NP-completo de, 800-802
com destino unário, 1802 ex.
somatório, 835-844
em notação assintótica, 37, 836
fórmulas e propriedades de, 835-838
implícito, 512
limites, 838-844
linearidade de, 836
somatório, lema, 659
somatório implícito, notação, 512
somatórios, desmembrando, 841-842
sombra de um ponto, 756 ex.
sondagem, *ver* linear, sondagem; quadrática sondagem
SORTER, 566, 567 ex.
STACK-EMPTY, 164
Stirling, aproximação, 44
STOOGE-SORT, 130 pr.
Strassen, algoritmo, 579-585
STRONGLY-CONNECTED-COMPONENTS, 439
subárvore, 858-859

mantendo os tamanhos de, em
árvores de ordem estatística,
245-247
subcadeia, 864
subcaminho, 854
subcarga
de uma fila, 164
de uma pilha, 164
subconjunto
hereditários, família de, 314
independentes, família de, 314
subconjunto (⊆), 846
subconjunto máximo em um matróide,
315
subdeterminado, sistema de equações
lineares, 587
subgrafo, 855
predecessor, *ver* predecessor,
subgrafo
subgrafo de predecessor, propriedade,
464-465, 484
subgrupo, 686-687
sub-rotina
chamando, 15, 16, 17 n.
executando, 17 n.
subseqüência, 280
SUBSET-SUM, 799
substituição, método, 51-54
e árvores de recursão, 57-58
subtração de matrizes, 574
subtrair, instrução, 16
SUCCESSOR, 160
sucesso em uma experiência de
Bernoulli, 878
sucessor
em árvores de ordem estatística, 248
ex.
em árvores de pesquisa binária,
208-209
em árvores vermelho-preto, 222-223
em listas ligadas, 166
localizando o *i*-ésimo, de um nó em
uma árvore de ordem estatística,
246 ex.
sufixo (§), 718
sufixo, função, 726
superdepósito, 512
superdeterminado, sistema de equações
lineares, 587
superior, matriz triangular, 573
superior, mediana, 147
superorigem, 512
superpolinomial, tempo, 763
superpostos, intervalos, 249
localizando todos, 254 ex.
ponto de superposição máxima, 255
pr.
superpostos, lema de sufixos, 718-719
superpostos, retângulos, 254 ex.
superpostos, subproblemas, 276-278
suspensão, problema, 763
SVD, 609

TABLE-DELETE, 338
TABLE-INSERT, 334
tamanho
da entrada de um algoritmo, 17,
672-673, 768-770
de um circuito combinacional
booleano, 780-781
de um clique, 791
de um conjunto, 847-848
de uma árvore binomial, 366-367
de uma cobertura de vértices, 793,
807-808
de uma rede de comparação, 557
de uma rede de ordenação, 558 ex.
de uma subárvore em um heap de
Fibonacci, 395
tarefa, 319
tarefas, programação, 319-321, 323 pr.
Tarjan, algoritmo de ancestrais menos
comuns off-line de, 415 pr.
tautologia, 776 ex., 790-791 ex.
taxa de crescimento, 20
Taylor, séries, 218 pr.
tempo, domínio, 651
tempo, *ver* execução, tempo
tempo de descoberta em pesquisa
primeiro na profundidade, 429-430
tempo polinomial, aceitação, 771
tempo polinomial, algoritmo, 673
tempo polinomial, computabilidade,
769
tempo polinomial, decisão, 771
tempo polinomial, esquema de
aproximação, 807
tempo polinomial, redutibilidade ($\leq_P$),
776-777, 784 ex.
tempo polinomial, resolubilidade, 768
tempo polinomial, verificação, 773-776
tempo unitário, tarefa, 319
teorema fundamental de programação
linear, 647
testes
de caráter primo, 702-709, 715
de caráter pseudoprimo, 704
teto, função ($\lceil\,\rceil$), 40
em teorema mestre, 65-68
teto, instrução, 16-17
texto cifrado, 698-699
texto em correspondência de cadeias,
717
then, em pseudocódigo, 14
todos os pares, caminhos mais curtos,
460, 490-508
em grafos g-densos, 507 pr.
algoritmo de Floyd-Warshall, 497-500
algoritmo de Johnson, 503-506
por multiplicação de matrizes,
492-497
por elevação ao quadrado repetida,
494-496
Toeplitz, matriz, 669 pr.
TOP, 750
topo de uma pilha, 163
topológica, ordenação, 436-438
no cálculo de caminhos mais curtos
de única origem em um grafo,
468
TOPOLOGICAL-SORT, 436
total, árvore binária, 860, 861 ex.
relação com o código ótimo, 308-309
total, fluxo líquido, 511
total, fluxo positivo, 510
total, nó, 353
total, ordem, 851
total, ordem, 576
total, percurso de uma árvore, 812-813
totalmente entre parênteses, 266
totalmente polinomial, esquema de
aproximação de tempo, 807
para o problema da soma de
subconjuntos, 823-827
para o problema do clique máximo,
828 pr.
transformação discreta de Fourier, 660
transformação rápida de Fourier (FFT)
circuito para, 667
implementação iterativa de, 665-667
implementação recursiva de,
661-663
multidimensional, 669 pr.
usando aritmética modular, 670 pr.
transição, função, 725-726, 729-730
transitiva, relação, 849
TRANSITIVE-CLOSURE, 501
transitividade de notação assintótica, 30
transitivo, fechamento, 500-502
e multiplicação de matrizes
booleanas, 600 ex.
de grafos dinâmicos, 507 pr.
transposição
conjugada, 600 ex.
de um grafo orientado, 421 ex.
de uma matriz, 420-421, 572
transposição, rede, 568 pr.
traseira, aresta, 433-434, 437
tratabilidade, 763
travessia de uma árvore, *ver* árvore,
percurso
treap, 238 pr.
TREAP-INSERT, 238 pr.
TREE-DELETE, 211, 231-232
TREE-INSERT, 210, 225
TREE-MAXIMUM, 208
TREE-MINIMUM, 208
TREE-PREDECESSOR, 209
TREE-SEARCH, 207
TREE-SUCCESSOR, 208-209
3-CNF, 788
3-CNF, satisfabilidade, 788-1791
algoritmo de aproximação para,
819-820
e satisfabilidade de 2-CNF, 764
3-CNF-SAT, 788
3-COLOR, 804 pr.
3-forma normal conjuntiva, 788
triangular, matriz, 573-574, 578 ex.
triângulos, desigualdade, 811
e arestas de peso negativo, 814 ex.
para caminhos mais curtos, 464-465,
480-481
tricotomia, intervalo, 249
tricotomia, propriedade de números
reais, 39
tridiagonais, sistemas lineares,
607 pr.
tridiagonal, matriz, 573
trie, *ver* raiz, árvore
trilha, 350
TRIM, 825
trinado, transformação, 664 ex.
trivial, divisor, 673-674
troca, propriedade, 314
TSP, 799
tupla, 848

último a entrar, primeiro a sair, 163
ver também pilha
um para um, correspondência, 852
um para um, função, 852
uma passagem, método, 416
unário, 769
união
de conjuntos (∪), 846

de conjuntos dinâmicos, *ver* unificando
de linguagens, 770-771
união por ordem, 404
única origem, caminhos mais curtos, 459-489
algoritmo de escalonamento de Gabow, 486 pr.
Bellman-Ford, algoritmo, 465-468
com caminhos bitônicos, 488 pr.
Dijkstra, algoritmo, 471-475
e caminhos mais longos, 763
e restrições de diferença, 475-480
em grafos g-densos, 507 pr.
em um grafo acíclico orientado, 468-471
unicamente conectado, grafo, 436 ex.
unicamente ligada, lista, 166
ver também ligada, lista
unidade (1), 673-674
unificando
heaps, 365
heaps 2-3-4, 379
heaps binomiais, 371-375
heaps de Fibonacci, 385-386
listas ligadas, 169
uniforme, distribuição de probabilidades, 869
uniforme, hash, 193
uniforme, permutação aleatória, 74-75, 81
UNION, 365, 399
implementação de floresta de conjuntos disjuntos de, 405
implementação de lista ligada de, 401-403, 403 ex.
unitária, matriz triangular inferior, 573
unitária, matriz triangular superior, 573
unitário, 848
unitário, vetor, 572
universal, depósito, 421 ex.
universal, hash, 188-191
universo, 847

válido, deslocamento, 717
valor
de um fluxo, 510
de uma função, 852
objetivo, 614, 617
valor de ponto, representação, 653-654
valor inteiro, fluxo, 527
van Emde Boas, estrutura de dados, 116, 346
Vandermonde, matriz, 579 ex.
Var[ ] (variância), 877
variância, 876
de uma distribuição binomial, 880-881
de uma distribuição geométrica, 878
várias mercadorias, fluxo, 625-623
custo mínimo, 626 ex.
variável
aleatória, 873-877
básica, 620
deixando, 629
em pseudocódigo, 14
entrando, 629
não básica, 620
relaxada, 619
ver também indicador, variável aleatória
vários conjuntos, 845 n.
varredura, 743-749, 760 pr.
varredura, linha, 743
vazia, árvore, 1860
vazia, cadeia (ε), 718, 770
vazia, linguagem, 770-771
vazia, pilha, 164
vazio, conjunto, 845
Venn, diagrama, 847
verdade, atribuição, 780, 786
verdade, tabela, 779
verificação, 773-776
de árvores de amplitude, 458
verificação, algoritmo, 774
vermelho-preto, árvore, 220-240
altura de, 221
ampliação de, 248-249
chave máxima de, 222
chave mínima de, 222
comparação com árvores B, 354
e árvores 2-3-4, 354-355 ex.
eliminação de, 231-236
junção, 237 pr.
para determinar se quaisquer segmentos de linhas se interceptam, 745-746
para enumeração de chaves em um intervalo, 249 ex.
pesquisando em, 222
predecessor em, 222
propriedades de, 220-223
reestruturando, 342-343 pr.
relaxada, 223 ex.
rotação em, 223-225
sucessor em, 222
ver também intervalo, árvore; ordem estatística, árvore
VERTEX-COVER, 794
vértice
de um polígono, 742 ex.
em um grafo, 853
intermediário, 497
isolado, 854
ponto de articulação, 443 pr.
seletor, 796
vértices, cobertura, 794, 807-808, 820-823
vértices, conjunto, 853
vetor, 572, 575-576
convolução de, 653
produto cruzado de, 739
ortonormal, 609
no plano, 739
viabilidade, problema, 475, 648 pr.
viagem
bitônico, 291 pr.
de Euler, 443 pr., 763
de um grafo, 798
viável, programa linear, 616
viável, região, 613
viável, solução, 475, 613, 616
violação de uma restrição de igualdade, 627
virtual, memória, 16-17
vizinhança, 529 ex.
vizinho, 855-856
vizinhos, lista, 540
VLSI (very-large-scale integration – integração em escala muito grande), 70 n.

while, em pseudocódigo, 14
WITNESS, 705

Yen, aperfeiçoamento para o algoritmo de Bellman-Ford, 485
Young, quadro, 115

Z (conjunto de inteiros), 845
0-1, problema da mochila, 305-306, 307
0-1, problema de programação de inteiros, 802, 821
zero, matriz, 572
zero de um polinômio, módulo primo, 690
zero-um, princípio, 559-561, 564-565
Z_n (classes de equivalência, módulo n), 674

Este livro reúne qualidades insuperáveis: uma enciclopédica amplitude, didática clara e análise em profundidade de vários temas. Ele começa abordando os fundamentos matemáticos de algoritmos e mantém este rigor em toda a extensão do trabalho. As ferramentas e os conceitos desenvolvidos nas seções iniciais são então aplicados a questões de estruturas de dados, grafos, geometria computacional, modelos de computação paralela e a uma grande variedade de algoritmos. Considerando que algoritmos são utilizados em várias áreas – jogos, aplicativos gráficos e de simulação, além de engenharia elétricas –, este livro está destinado a atender com precisão e brilhantismo a estudantes e profissionais envolvidos com programação.

www.therebels.biz

ISBN 85-352-0926-