

Desmistificando Algoritmos



Desmistificando Algoritmos

Thomas H. Cormen

© 2014, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei nº 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.

© 2013, Massachusetts Institute of Technology. All rights reserved.

Copidesque: Ivone Teixeira

Revisão: Tania Heglacy Moreira de Almeida

Editoração Eletrônica: Thomson Digital

Elsevier Editora Ltda.

Conhecimento sem Fronteiras

Rua Sete de Setembro, 111 – 16º andar
20050-006 – Centro – Rio de Janeiro – RJ – Brasil
Rua Quintana, 753 – 8º andar
04569-011 – Brooklin – São Paulo – SP

Serviço de Atendimento ao Cliente

0800-0265340

atendimento1@elsevier.com

ISBN original 978-0-262-51880-2

ISBN 978-85-352-7177-5

ISBN digital 978-85-352-7179-9

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação ao nosso Serviço de Atendimento ao Cliente, para que possamos esclarecer ou encaminhar a questão. Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

CIP-BRASIL. CATALOGAÇÃO NA PUBLICAÇÃO SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

C833d

Cormen, Thomas H.

Desmistificando algoritmos / Thomas H. Cormen ; tradução Arlete Simille Marques. - 1. ed. - Rio de Janeiro : Elsevier, 2014.

23 cm.

Tradução de: Algorithms unlocked

ISBN 978-85-352- 7177-5

1. Algoritmos. 2. Estruturas de dados (Computação). 3. Programação (Computadores). I. Título.

13-05070

CDD: 005.1

CDU: 004.42

Em memória da minha amada mãe, Renee Cormen.

Prefácio

Como os computadores resolvem problemas? Como o seu pequenino GPS consegue encontrar, entre gazilhões de possíveis rotas, o caminho mais rápido para o seu destino e fazer isso em meros segundos? Quando você compra algo pela Internet, como o número do seu cartão de crédito é protegido contra alguém que o intercepta? A resposta a essas perguntas e a uma tonelada de outras é *algoritmos*. Redigi este livro para revelar a você o mistério dos algoritmos.

Sou coautor do livro didático sobre algoritmos *Introduction to Algorithms*. É um livro maravilhoso (é claro que a minha opinião é preconceituosa), mas às vezes muito técnico.

Este livro não é *Introduction to Algorithms*. Não é nem mesmo um livro didático. Não se aprofunda nem se expande muito na área de algoritmos de computador, não ensina técnicas minuciosas para projetar algoritmos de computador e não contém absolutamente nenhum problema ou exercício para o leitor resolver.

Então, o que é este livro? É um lugar para você começar se:

- estiver interessado em saber como os computadores resolvem problemas,
- se quiser saber como avaliar a qualidade dessas soluções,
- gostaria de ver como os problemas em computação e as abordagens para resolvê-los estão relacionados com o mundo que não é do computador,
- souber um pouco de matemática e
- nunca escreveu um programa de computador (embora não fará mal se já tiver escrito algum).

Alguns livros sobre algoritmos de computador são conceituais, com poucos detalhes técnicos. Alguns estão repletos de precisão técnica. Alguns estão no meio desses dois. Cada tipo de livro tem seu lugar. Eu colocaria este livro na categoria intermediária. Sim, tem um pouco de matemática e, às vezes, torna-se um pouco preciso, porém eu evitei me aprofundar em detalhes (exceto talvez no final do livro, quando não consegui mais me controlar).

Eu imagino este livro como se fosse a entrada para uma refeição. Suponha que você vá a um restaurante italiano e peça uma entrada, adiando um pouco a decisão do prato que pedirá depois. A entrada chegou, você a come. Talvez não tenha gostado dela e decidiu não pedir nada mais. Quem sabe tenha gostado, mas é muito nutritiva e você achou que não precisa pedir mais nada. Ou, talvez, você tenha gostado da entrada, não achou que era muito nutritiva e está esperando ansiosamente o prato principal. Ao pensar neste livro como uma entrada para uma refeição, espero um dos dois últimos resultados: você lê o livro, fica satisfeito e acha que não precisa se aprofundar mais no mundo dos algoritmos ou você gostou tanto do que leu que quer aprender mais. Cada capítulo termina com uma seção intitulada “O que mais ler?”, que o guiará a livros e artigos que se aprofundam nos tópicos apresentados.

O que você aprenderá com este livro?

Não sei dizer o que você aprenderá com este livro. Eis o que eu *pretendo* que você aprenda com este livro:

- O que são algoritmos de computador, um modo de descrevê-los e como avaliá-los.
- Modos simples de procurar informações em um computador.
- Métodos para rearranjar informações em um computador, de modo que esteja em uma ordem prescrita (denominamos essa tarefa “ordenação”).
- Como resolver problemas básicos que podemos modelar em um computador com uma estrutura matemática conhecida como “grafo”. Entre muitas aplicações, grafos são ótimos para modelar redes rodoviárias (quais são as estradas que entram e saem de cruzamentos e quais são os comprimentos dessas estradas?), dependências entre tarefas (qual tarefa deve preceder quais outras tarefas?), relações financeiras (quais são as taxas de câmbio entre todas as moedas do mundo?) ou interações entre pessoas (quem conhece quem, quem odeia quem, qual ator apareceu em um filme com qual outro ator?).
- Como resolver problemas que fazem perguntas sobre cadeias de caracteres de texto. Alguns desses problemas têm aplicações em áreas como biologia, na qual os caracteres representam moléculas de base e as cadeias de caracteres representam as estruturas de DNA.
- Os princípios básicos que fundamentam a criptografia. Mesmo que você nunca tenha criptografado uma mensagem, o seu computador provavelmente criptografou (por exemplo, quando você compra mercadorias on-line).
- Ideias fundamentais de compressão de dados, que vão bem além de “f u cn rd ths u cn gt a gd jb n gd pay”.
- Que alguns problemas são difíceis de resolver em um computador em qualquer quantidade de tempo razoável ou, no mínimo, que ninguém conseguiu descobrir como fazê-lo.

O que você já precisa saber para entender o material deste livro?

Como eu disse antes, há um pouco de matemática neste livro. Se a matemática o assusta, você pode tentar saltá-la ou tentar um livro menos técnico. Porém, eu fiz o possível para tornar a matemática acessível.

Não dou por certo que você já tenha escrito ou até mesmo lido um programa de computador. Se você puder seguir instruções no formato de descrição, conseguirá entender como eu expresso as etapas que, juntas, formam um algoritmo. Se você entender a seguinte anedota, já está a meio caminho:

Você sabe a do cientista de computador que ficou preso no chuveiro? Ele¹ estava lavando os cabelos e seguindo as instruções na embalagem do xampu. Ele leu: “Faça espuma. Enxague. Repita.”

¹Ou ela. Dada a infeliz razão de gênero na ciência de computadores, a maior probabilidade é que seja ele.



Usei um estilo de escrever razoavelmente informal, esperando que uma abordagem pessoal o ajudará tornando o material acessível. Alguns capítulos dependem de material em capítulos anteriores, mas tais dependências são poucas. Alguns capítulos começam de maneira não técnica e tornam-se progressivamente mais técnicos. Mesmo que você ache que eu estou passando por cima da sua cabeça em um capítulo, provavelmente se beneficiará de ler, no mínimo, o início do próximo capítulo.

Comunique os erros

Se você encontrar um erro neste livro, favor comunicar por meio do endereço unlocked@mit.edu.

Agradecimentos

Grande parte do material deste livro foi retirado de *Introduction to Algorithms* e, portanto, devo muito aos meus coautores, Charles Leiserson, Ron Rivest e Cliff Stein. Você verá que em todo este livro eu me referi (leia: eu inseri) a *Introduction to Algorithms*, conhecido mais amplamente pelas iniciais CLRS dos quatro autores. Escrever este livro sozinho me fez perceber o quanto eu sinto falta da colaboração de Charles, Ron e Cliff. E também aproveito para agradecer a todos a quem eu agradeci no prefácio do CLRS.

Também usei material de cursos que lecionei em Dartmouth, especialmente Ciência de Computadores 1, 5 e 25. Agradeço a meus alunos por me deixarem perceber, por meio de suas perguntas perspicazes, quais abordagens pedagógicas funcionavam e, pelo seu silêncio de pedra, quais não funcionavam.

Este livro deve sua existência a uma sugestão de Ada Brunstein, que foi nossa editora na MIT Press quando preparamos a terceira edição de CLRS. Ada seguiu em frente e Jim DeWolf ocupou o lugar dela. Originalmente, este livro estava programado para ser parte da série “Essential Knowledge” da MIT Press, porém a editora achou que era muito técnico para a série (imagine só — eu escrevi um livro demasiadamente técnico para a MIT!). Jim tratou dessa situação potencialmente incômoda permitindo que eu escrevesse o livro que queria em vez do livro que a MIT Press achava que eu estava escrevendo. Agradeço também o apoio de Ellen Faran e Gita Devi Manaktala, da MIT Press.

Julie Sussman, PPA², foi nossa revisora técnica para a segunda e terceira edições de CLRS e, mais uma vez, adorei tê-la como editora deste livro. Melhor. Editora. Técnica. Que já existiu. Ela permitiu que eu me safasse com nada. Eis aqui uma prova, na forma de um e-mail que Julie me enviou sobre um primeiro rascunho do Capítulo 5:

Prezado Sr. Cormen,
Autoridades apreenderam um capítulo que escapou e que estava escondido no seu
livro. Não conseguimos determinar de qual livro ele escapou, mas não podemos
imaginar como ele poderia ter-se alojado em seu livro durante todos esses meses sem
o nosso conhecimento, portanto não temos nenhuma opção senão responsabilizá-lo.
Esperamos que se dê ao trabalho de reformar esse capítulo e eu lhe darei uma
oportunidade de tornar-se um cidadão produtivo do seu livro. Anexo envio um
relatório do policial que efetuou a prisão. Julie Sussman.

²Professional pain in the ASS = “pé no saco profissional”.

Caso você esteja imaginando o que “PPA” significa, as duas primeiras letras significam “*Professional Pain*”. Você provavelmente adivinhará o que “A” significa, mas quero salientar que Julie se orgulha desse título e com muita razão. Googols de agradecimentos, Julie!

Não sou nenhum criptógrafo, e o capítulo sobre princípios de criptografia beneficiou-se tremendamente de comentários e sugestões de Ron Rivest, Sean Smith, Rachel Miller e Huijia Rachel Lin. Esse capítulo traz uma citação de pé de página sobre sinais de beisebol, pela qual agradeço a Bob Whalen, o treinador de beisebol de Dartmouth, pela paciência que teve ao me explicar um pouco sobre os sistemas de sinais no beisebol. Ilana Arbisser verificou que os biólogos computacionais alinharam sequências de DNA do modo que eu expliquei no Capítulo 7. Jim DeWolf e eu fizemos várias combinações de palavras para o título deste livro, mas foi um aluno de Dartmouth, Chander Ramesh, que propôs *Algorithms Unlocked*.

O Departamento de Ciência de Computadores de Dartmouth College é um lugar incrível para trabalhar. Meus colegas são brilhantes e companheiros, e nosso pessoal profissional não deve nada a nenhum outro. Se você estiver procurando um programa de ciência de computadores no nível de graduação e pós-graduação ou se estiver procurando uma cargo universitário em ciência de computadores, tente Dartmouth.

Finalmente, agradeço à minha esposa, Nicole Cormen; aos meus pais, Renee e Perry Cormen; à minha irmã, Jane Maslin; e aos pais de Nicole, Colette e Paul Sage, por seu amor e apoio. Meu pai tem certeza de que a figura na página 2 é um 5, e não um S.

*TOM CORMEN
Hanover, New Hampshire
Novembro de 2012*

O que são algoritmos e por que você deve se importar com eles?

Vamos começar com a pergunta que sempre me fazem: “O que é um algoritmo?”¹

Uma resposta de âmbito geral seria “um conjunto de etapas para executar uma tarefa”. Você executa algoritmos na sua vida diária. Você tem um algoritmo para escovar os dentes: abrir o tubo de pasta dental, pegar a escova de dentes, apertar o tubo de pasta dental sobre a escova e aplicar a quantidade necessária de dentífrico, fechar o tubo, colocar a escova em um quadrante da boca, movimentá-la para cima e para baixo durante N segundos etc. Se você pega ônibus ou metrô para ir trabalhar, terá um algoritmo para isso. E assim por diante.

Mas este livro é sobre algoritmos executados em computadores ou, de modo mais geral, dispositivos de computação. Exatamente como os algoritmos que *você* executa, os algoritmos executados em computadores também afetam a sua vida diária. Você usa o seu GPS para determinar uma rota de viagem? O aparelho executa o que denominamos algoritmo de “caminho mínimo” para determinar essa rota. Você compra produtos pela Internet? Então você usa (ou deveria usar) um site seguro que executa um algoritmo criptográfico. Quando você compra produtos pela Internet, eles são entregues por um serviço de entrega privado? Esse serviço usa algoritmos para designar pacotes a caminhões individuais e então determinar a ordem em que cada motorista deve entregar esses pacotes. Algoritmos são executados em computadores em todos os lugares — no seu laptop, em servidores, no seu smartphone, em sistemas embutidos (como no seu carro, no seu forno de micro-ondas ou em sistemas de ar condicionado) — em todos os lugares!

O que distingue um algoritmo executado em um computador de um algoritmo que você executa? Você poderia tolerar quando um algoritmo não é descrito com precisão, mas um computador não pode. Por exemplo, se você vai de carro para o trabalho, o seu algoritmo de ir de carro para o trabalho poderia dizer “se o trâfego estiver ruim, pegue uma rota alternativa”. Embora você saiba o que quer dizer “trâfego ruim”, um computador não sabe.

¹ Ou, como um amigo que jogava hóquei comigo perguntaria: “O que é um mau goritmo?”

Portanto, um algoritmo de computador é um conjunto de etapas para executar uma tarefa descrita com precisão suficiente para que um computador possa executá-la. Se você já fez programação de computador, pouca que seja, em Java, C, C++, Python, Fortran, Matlab ou semelhantes, tem alguma ideia do que significa nível de precisão. Se você nunca escreveu um programa de computador, talvez perceba tal nível de precisão ao ver como eu descrevo algoritmos neste livro.

Vamos à próxima pergunta: “O que queremos de um algoritmo de computador?”

Algoritmos de computador resolvem problemas de computação. Queremos duas coisas de um algoritmo de computador: dada uma entrada para um problema, o algoritmo deve sempre produzir uma solução correta para o problema e usar recursos computacionais eficientemente ao fazê-lo. Vamos examinar esses dois desejos, um por vez.

CORREÇÃO

O que significa produzir uma solução correta para um problema? Normalmente, podemos especificar com precisão o que uma solução correta acarretaria. Por exemplo, se o seu GPS produzir uma solução correta para determinar a melhor rota de viagem, pode ser que essa rota, entre todas as rotas possíveis para você ir de onde está até onde deseja ir, seja a que o levará ao seu destino no menor tempo possível. Ou, talvez, essa rota seja a que tem a menor distância possível. Ou a que o levará ao seu destino desejado o mais rapidamente possível, mas sem pagar pedágio. É claro que as informações que o seu GPS usa para determinar uma rota podem não estar de acordo com a realidade. A menos que possa acessar informações de tráfego em tempo real, o seu GPS pode entender que o tempo para percorrer uma estrada é igual à distância total da estrada dividida pelo limite de velocidade nessa estrada. Todavia, se a estrada estiver congestionada, o GPS pode lhe dar um mau conselho se o que você estiver procurando for a estrada mais rápida. Ainda assim poderíamos dizer que o algoritmo de roteamento que o GPS executa está correto, mesmo que a entrada dada ao algoritmo não esteja; para a entrada que lhe foi dada, o algoritmo de roteamento produz a rota mais rápida.

Agora, para alguns problemas, pode ser difícil ou até impossível dizer se um algoritmo produz uma solução correta. Considere o reconhecimento de caracteres ópticos, por exemplo. Essa imagem de 11×6 pixels é um 5 ou um S?



Alguns dirão que é um 5, enquanto outros poderão achar que é um S; portanto, como podemos afirmar que a decisão de um computador é correta ou incorreta?

Não podemos. Neste livro, focalizaremos algoritmos de computador que têm soluções cognoscíveis.



Todavia, às vezes, podemos aceitar que um algoritmo de computador pode produzir uma resposta incorreta, desde que possamos controlar a frequência com que isso acontece. Criptografia é um bom exemplo. O criptossistema RSA comumente usado depende de determinar se números grandes — realmente grandes, significando os de centenas de dígitos — são primos. Se você já escreveu um programa de computador, provavelmente pode escrever um que determine se um número n é primo. O programa testaria todos os divisores candidatos de 2 até $n - 1$ e, se qualquer desses candidatos for de fato um divisor de n , então n é composto. Se nenhum número entre 2 e $n - 1$ for divisor de n , então n é primo. Porém, se n tiver centenas de dígitos, haverá muitos divisores candidatos, mais até do que um computador realmente veloz poderia verificar em qualquer quantidade de tempo razoável. É claro que você poderia fazer algumas otimizações, como eliminar todos os candidatos pares uma vez determinado que 2 não é um divisor, ou parar ao chegar a \sqrt{n} (visto que, se d é maior que \sqrt{n} e d é um divisor de n , então n/d é menor que \sqrt{n} e é também divisor de n ; por consequência, se n tem um divisor, você o encontrará no instante em que chegar a \sqrt{n}). Se n tiver centenas de dígitos, embora \sqrt{n} tenha aproximadamente somente metade do número de dígitos que n tem, ainda é um número grande. A boa notícia é que sabemos de um algoritmo que testa rapidamente se um número é primo. Porém, a má notícia é que ele pode cometer erros. Em particular, se ele declarar que n não é primo, então n definitivamente não é primo, mas se ele declarar que n é primo, há uma chance de n não sê-lo. A má notícia não é tão ruim assim: podemos controlar a taxa de erro de modo que seja realmente baixa, por exemplo, um erro a cada 2^{50} vezes. Isso é suficientemente raro — um erro em aproximadamente um milhão de bilhões de vezes — para que a maioria de nós se sinta confortável com a utilização desse método para determinar se um número é primo pelo método criptográfico RSA.

Correção é um assunto arriscado com outra classe de algoritmos, denominados algoritmos de aproximação. Algoritmos de aproximação aplicam-se a problemas de otimização, nos quais queremos determinar a melhor solução de acordo com alguma medida quantitativa. Determinar a rota mais rápida, como um GPS faz, é um exemplo no qual a medida quantitativa é o tempo de viagem. Para alguns problemas, não temos nenhum algoritmo que determine uma solução ótima em qualquer quantidade de tempo razoável, mas sabemos de um algoritmo de aproximação que, em quantidade razoável de tempo, pode encontrar uma solução que é quase ótima. Nesse caso, “quase ótima” normalmente quer dizer que a medida quantitativa da solução encontrada pelo algoritmo de aproximação está dentro de algum fator da medida quantitativa da solução ótima. Contanto que especifiquemos qual é o fator desejado, podemos dizer que uma solução correta dada por um algoritmo de aproximação é qualquer solução que esteja dentro daquele fator da solução ótima.

USO DE RECURSO

O que significa um algoritmo usar recursos computacionais eficientemente? Já aludimos a uma medida de eficiência quando discutimos algoritmos de aproximação: tempo. Um algoritmo que dá uma solução correta mas leva muito tempo para produzir essa solução correta poderia ser de pouco ou nenhum valor. Se o seu GPS demorou uma hora para

determinar qual rota ele recomenda, você se daria o trabalho de ligá-lo? Na verdade, o tempo é a medida principal de eficiência que usamos para avaliar um algoritmo, uma vez demonstrado que o algoritmo dá uma solução correta. Mas não é a única medida. Poderíamos nos preocupar com a quantidade de memória que o algoritmo exige (seu uso de memória), visto que um algoritmo tem de ser executado dentro da memória disponível. Outros possíveis recursos que um algoritmo poderia usar: comunicação em rede, bits aleatórios (porque algoritmos que fazem escolhas aleatórias precisam de uma fonte de números aleatórios) ou operações de disco (para algoritmos projetados para trabalhar com dados residentes em disco).

Neste livro, como na maioria da literatura de algoritmos, focalizaremos apenas um recurso: o tempo. Como julgamos o tempo exigido por um algoritmo? Diferentemente da correção, que não depende do computador particular no qual o algoritmo é executado, o tempo de execução propriamente dito de um algoritmo depende de diversos fatores extrínsecos ao algoritmo em si: da velocidade do computador, da linguagem de programação na qual o algoritmo foi implementado, do compilador ou interpretador que traduz o programa para código executado no computador, da habilidade do programador que escreve o programa e de outras atividades que ocorrem no computador ao mesmo tempo que a execução do programa. E tudo isso pressupondo que o algoritmo seja executado em apenas um computador com todos os seus dados na memória.

Se fôssemos avaliar a velocidade de um algoritmo implementando-o em uma linguagem de programação real, executando-o em um computador particular com uma entrada dada e medindo o tempo que o algoritmo gasta, nada saberíamos sobre a velocidade de execução do algoritmo com uma entrada de tamanho diferente ou, possivelmente, até mesmo com uma entrada diferente de igual tamanho. Se quiséssemos comparar a velocidade relativa do algoritmo com a de algum outro algoritmo em relação ao mesmo problema, teríamos de implementar ambos e executá-los com várias entradas de vários tamanhos. Então, como podemos avaliar a velocidade de um algoritmo?

A resposta é que fazemos isso por meio de uma combinação de duas ideias. A primeira é que determinamos quanto tempo o algoritmo leva em função do tamanho de sua entrada. Em nosso exemplo de determinação de rota, a entrada seria alguma representação de um mapa rodoviário, e seu tamanho dependeria do número de interseções e do número de estradas que se ligam às interseções no mapa. (O tamanho físico da rede rodoviária não importaria, visto que podemos caracterizar todas as distâncias por números, e todos os números ocupam o mesmo tamanho na entrada; o comprimento de uma estrada não tem nenhuma relação com o tamanho da entrada.) Em um exemplo mais simples, para pesquisar uma lista de itens dada e determinar se um item particular está presente na lista, o tamanho da entrada seria o número de itens na lista.

A segunda é que avaliamos o quanto rapidamente a função que caracteriza o tempo de execução aumenta com o tamanho da entrada — a *taxa de crescimento* do tempo de execução. No Capítulo 2, veremos as notações que usamos para caracterizar o tempo de execução de um algoritmo, porém o mais interessante em nossa abordagem é que examinamos somente o termo dominante no tempo de execução e não consideramos coeficientes. Isto é, focamos a *ordem de crescimento* do tempo de execução. Por exemplo, suponha que pudéssemos determinar que uma implementação específica de



um algoritmo particular para pesquisar uma lista de n itens leva $50n + 125$ ciclos de máquina. O termo $50n$ domina o termo 125 assim que n se torna grande o suficiente, começando com $n \geq 3$ e crescendo em dominância para listas de tamanhos ainda maiores. Assim, não consideramos o termo de baixa ordem 125 quando descrevemos o tempo de execução desse algoritmo hipotético. O que poderia surpreender você é que também descartamos o coeficiente 50, caracterizando, desse modo, que o tempo de execução cresce linearmente com o tamanho da entrada n . Como outro exemplo, se um algoritmo levasse $20n^3 + 100n^2 + 300n + 200$ ciclos de máquina, diríamos que seu tempo de execução cresce segundo n^3 . Novamente, os termos de baixa ordem — $100n^2$, $300n$ e 200 — tornam-se cada vez menos significativos à medida que o tamanho da entrada n aumenta.

Na prática, os coeficientes que ignoramos não importam. Mas eles dependem tão fortemente de fatores extrínsecos que é inteiramente possível que, se estivéssemos comparando dois algoritmos, A e B, que têm a mesma ordem de crescimento e executam a mesma entrada, A pudesse executar mais rapidamente que B com uma combinação particular de máquina, linguagem de programação, compilador/interpretador e programador, ao passo que B executaria mais rapidamente que A com alguma outra combinação. É claro que, se os algoritmos A e B produzem soluções corretas e A sempre executa duas vezes mais rapidamente que B, se todo o resto for igual, vamos preferir sempre executar A em vez de B. Todavia, do ponto de vista da comparação de algoritmos no campo abstrato, focalizamos a ordem de crescimento, descartando coeficientes ou termos de baixa ordem.

Quanto à pergunta final que fazemos neste capítulo, “Por que eu deveria me importar com algoritmos de computador?”, a resposta depende de quem você seja.

ALGORITMOS DE COMPUTADOR PARA NÃO AFICIONADOS

Ainda que não se considere um aficionado de computadores, os algoritmos de computador importam para você. Afinal, a menos que esteja em uma expedição da vida selvagem sem GPS, provavelmente os usará todos os dias. Você já procurou algo na Internet hoje? O motor de busca que você usou — Google, Bing ou qualquer outro — empregou algoritmos sofisticados para pesquisar a Web e decidir em que ordem apresentar seus resultados. Você já dirigiu seu carro hoje? A menos que o seu carro seja um clássico dos automóveis, seus computadores de bordo tomaram milhares de decisões, todas baseadas em algoritmos, durante a sua viagem. Eu poderia continuar indefinidamente.

Como usuário final de algoritmos, é bom que você aprenda um pouco sobre como projetamos, caracterizamos e avaliamos os algoritmos. Entendo que você tenha no mínimo um leve interesse, já que pegou este livro e o leu até aqui. Muito bem! Agora vamos ver se despertamos o seu interesse rapidamente e se aguenta até a próxima festa na qual surja o assunto de algoritmos.²

² Sim, eu sei que, a menos que você more no Vale do Silício, raramente o assunto algoritmos surgirá nas festas que você frequenta; porém, por alguma razão, nós, os professores de ciência da computação, achamos que é importante que nossos alunos não nos envergonhem em festas com sua falta de conhecimento em áreas particulares da ciência da computação.

ALGORITMOS DE COMPUTADOR PARA AFICIONADOS

Se você gosta de computadores, é bom se interessar também por algoritmos de computador! Eles não somente estão no coração de tudo o que vai dentro do seu computador, mas também são uma tecnologia, exatamente como tudo o mais que está dentro do seu computador. Você pode até pagar mais caro por um computador equipado com o mais recente e melhor processador, mas precisará executar implementações de bons algoritmos nesse computador para que o dinheiro que gastou valha a pena.

Damos um exemplo que ilustra como os algoritmos são de fato uma tecnologia. No Capítulo 3, veremos alguns algoritmos diferentes que ordenam uma lista de n valores em ordem crescente. Alguns desses algoritmos terão tempos de execução que crescem segundo n^2 , mas alguns terão tempos de execução que crescem somente segundo $n \lg n$. O que é $\lg n$? É o logaritmo na base 2 de n , ou $\log_2 n$. Cientistas da computação usam logaritmos base 2 com tanta frequência que, exatamente como matemáticos e cientistas usam a abreviatura \ln para o logaritmo natural — $\log_e n$ —, os cientistas da computação usam sua própria abreviatura para logaritmos base 2. Agora, como a função $\lg n$ é o inverso de uma função exponencial, ela cresce muito lentamente com n . Se $n = 2^x$, então $x = \lg n$. Por exemplo, $2^{10} = 1024$, portanto $\lg 1024$ é apenas 10; de modo semelhante, $2^{20} = 1.048.576$ e, assim, $\lg 1.048.576$ é apenas 20; $2^{30} = 1.073.471.824$ significa que $\lg 1.073.471.824$ é somente 30. Portanto, o crescimento de $n \lg n$ versus n^2 troca um fator de n por um fator de apenas $\lg n$, e esse é um negócio que você aceitaria fazer a qualquer instante.

Vamos tornar esse exemplo mais concreto confrontando um computador mais rápido (computador A), que executa um algoritmo de ordenação cujo tempo de execução para n valores cresce segundo n^2 , com um computador mais lento (computador B), que executa um algoritmo de ordenação cujo tempo de execução cresce segundo $n \lg n$. Cada um deles deve ordenar um arranjo de 10 milhões de números. (Embora 10 milhões de números possa parecer muito, se os números forem inteiros de 8 bytes, a entrada ocupará cerca de 80 megabytes, o que cabe muitas e muitas vezes na memória até mesmo de um laptop baratinho.) Suponha que o computador A execute 10 bilhões de instruções por segundo (mais rapidamente que qualquer computador sequencial único à época da redação deste livro) e que o computador B execute somente 10 milhões de instruções por segundo, de modo que o computador A é mil vezes mais rápido do que o computador B em poder de computação bruto. Para tornar a diferença ainda mais drástica, suponha que o programador mais esperto do mundo codifique em linguagem de máquina para o computador A, e o código resultante exija $2n^2$ instruções para ordenar n números. Suponha ainda mais que um programador apenas médio escreva para o computador B usando linguagem de alto nível com um compilador ineficiente e que o código resultante tenha $50n \lg n$ instruções. Para ordenar 10 milhões de números, o computador A leva

$$\frac{2 \cdot (10^7)^2 \text{ instruções}}{10^{10} \text{ instruções / segundo}} = 20.000 \text{ segundos}$$



o que é mais de 5,5 horas, enquanto o computador B leva

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instruções}}{10^7 \text{ instruções / segundo}} \approx 1.163 \text{ segundos},$$

o que é menos de 20 minutos. Usando um algoritmo cujo tempo de execução cresce mais lentamente, mesmo com um compilador ruim, o computador B executa mais de 17 vezes mais rapidamente que o computador A! A vantagem do algoritmo $n \lg n$ é ainda mais pronunciada quando ordenamos 100 milhões de números: enquanto o algoritmo n^2 no computador A leva mais de 23 dias, o algoritmo $n \lg n$ no computador B leva menos de quatro horas. Em geral, à medida que o tamanho do problema aumenta, o mesmo ocorre com a vantagem relativa do algoritmo $n \lg n$.

Mesmo com os impressionantes avanços que vemos continuamente no hardware de computador, o desempenho total do sistema depende de escolher algoritmos eficientes, tanto quanto de escolher hardware rápido ou sistemas operacionais eficientes. Exatamente como há avanços rápidos em outras tecnologias de computador, também há esses mesmos avanços em algoritmos.

O QUE MAIS LER?

Em minha muitíssimo tendenciosa opinião, a fonte mais clara e mais útil sobre algoritmos de computador é *Introduction to Algorithms* [CLRS09], de autoria de quatro camaradas demoniacamente lindos. O livro é comumente denominado “CLRS”, as iniciais dos autores. Muito do material que apresento neste livro retirei do livro deles, que é muitíssimo mais completo que este. Porém, os autores supõem que você já fez ao menos um pouco de programação de computador, e não economizam na matemática. Se achar que o nível de matemática desse livro está bom e que você está pronto para ir mais a fundo no assunto, o CRLS será melhor (na minha humilde opinião, é claro).

O livro de John MacCormick, *Nine Algorithms That Changed the Future* [Mac12] descreve vários algoritmos e aspectos relacionados da computação que afetam nossa vida diária. O tratamento de MacCormick é menos técnico do que o deste livro. Se achar que a minha abordagem neste livro é demasiadamente matemática, recomendo que tente ler o livro de MacCormick. Você conseguirá entender grande parte dele, mesmo que seus conhecimentos de matemática sejam pífios.

No improvável evento de você achar que o CLRS é muito água com açúcar, poderá tentar os vários volumes do conjunto *The Art of Computer Programming*, de Donald Knuth [Knu97, Knu98a, Knu98b, Knu11]. Embora o título da série possa dar a entender que ela focaliza detalhes de escrita de código, esses livros contêm análises de algoritmos brilhantes e profundos. Porém, aviso: o material em *TAOCP* é intenso. A propósito, se você está imaginando de onde vem a palavra “algoritmo”, Knuth informa que ela é derivada do nome “al-Khowârizmî”, um matemático persa do século IX.

Além do CLRS, há vários outros textos excelentes sobre algoritmos de computador publicados ao longo dos anos. As notas referentes ao Capítulo 1 do CLRS apresentam uma lista de muitos desses livros. Em vez de reproduzir essa lista aqui, procure-a no CLRS.

Como descrever e avaliar algoritmos de computador

No capítulo anterior, você teve uma ideia de como expressamos em palavras o tempo de execução de um algoritmo de computador: focalizando o tempo de execução como função do tamanho da entrada e, especificamente, a ordem de crescimento do tempo de execução. Neste capítulo, voltaremos um pouco atrás e veremos como descrever algoritmos de computador. Então veremos as notações que usamos para caracterizar os tempos de execução de algoritmos. Encerraremos este capítulo examinando algumas técnicas que usamos para projetar e entender algoritmos.

COMO DESCREVER ALGORITMOS DE COMPUTADOR

Sempre temos a opção de descrever um algoritmo de computador como um programa executável em uma linguagem de programação comumente usada, como Java, C, C++, Python ou Fortran. Na verdade, é exatamente isso que fazem vários livros didáticos sobre algoritmos. O problema de usar linguagens de programação reais para especificar algoritmos é que você pode se atolar nos detalhes da linguagem e não perceber as ideias que fundamentam os algoritmos. Outra abordagem, que adotamos no livro *Introduction to Algorithms*, usa “pseudocódigo”, que parece uma mistura de várias linguagens; se você já usou uma linguagem de programação real, facilmente entenderá pseudocódigo. Porém, se nunca programou, o pseudocódigo poderá parecer um pouco misterioso.

A abordagem que adoto neste livro é a de não tentar descrever algoritmos para software ou hardware, mas para “massacinentaware”: aquilo que está entre as suas orelhas. Além disso, adotarei como premissa que você nunca escreveu um programa de computador e, portanto, não expressarei algoritmos em nenhuma linguagem de programação real ou nem mesmo em pseudocódigo. Em vez disso, eu os descreverei em inglês (nesta tradução, em português) usando analogias com cenários do mundo real, sempre que puder. Para indicar o que acontece quando (o que denominamos “fluxo de controle” em programação) usarei listas e listas dentro de listas. Se quiser implementar um algoritmo em uma linguagem de programação real, acreditei piamente que você será capaz de traduzir minha descrição em código executável.

Se bem que tentarei manter as descrições em nível um tanto não técnico quanto possível, este livro é sobre algoritmos para computadores, e por isso terei de usar terminologia de computação. Por exemplo, programas de computador contêm *procedimentos* (também conhecidos como funções ou métodos em linguagens de programação reais) que especificam como fazer algo. Para conseguir que o procedimento realmente faça o que deve fazer, nós o *chamamos*. Quando chamamos um procedimento, nós lhe fornecemos entrada (usualmente, no mínimo, uma, mas alguns procedimentos não precisam de nenhuma). Especificamos a entrada como *parâmetros* entre parênteses após o nome do procedimento. Por exemplo, para calcular a raiz quadrada de um número, podemos definir um procedimento *SQUARE-ROOT(x)*; nesse caso, nos referimos à entrada para o procedimento como parâmetro *x*. A chamada de um procedimento pode ou não produzir resultado, dependendo de como especificamos o procedimento. Se o procedimento produzir resultado (ou saída), usualmente consideramos que tal resultado é algo que é passado de volta ao seu chamador. Em jargão de computação, dizemos que o procedimento *retorna* um valor.

Muitos programas e algoritmos trabalham com vetores de dados. Um *arranjo* agrupa dados do mesmo tipo em uma entidade. Pense em um arranjo como se ele fosse uma tabela, na qual, dado o *índice* de uma *entrada*, podemos falar sobre o *elemento* do arranjo que está naquele índice. Por exemplo, veja a tabela a seguir com os cinco primeiros presidentes dos Estados Unidos:

Índice	Presidente
1	George Washington
2	John Adams
3	Thomas Jefferson
4	James Madison
5	James Monroe

Por exemplo, o elemento no índice 4 nessa tabela é James Madison. Pensamos nessa tabela não como cinco entidades separadas, mas como uma tabela com cinco entradas. Um arranjo é semelhante. Os índices para um arranjo são números consecutivos que podem começar em qualquer lugar, mas usualmente nós os começaremos em 1.¹ Dado o nome de um arranjo e um índice para o arranjo, nós os combinamos com colchetes para indicar um elemento particular do arranjo. Por exemplo, denotamos o *i*-ésimo elemento de um arranjo *A* por *A[i]*.

Arranjos em computadores têm outra característica importante: o tempo que se leva para acessar qualquer elemento de um arranjo é o mesmo. Uma vez dado ao computador um índice *i* para o arranjo, ele pode acessar o *i*-ésimo elemento tão rapidamente quanto pode acessar o primeiro elemento, independentemente do valor de *i*.

¹ Se você programa em Java, C ou C++, está acostumado a usar arranjos que começam em 0. Começar arranjos em 0 é bom para computadores, mas para a nossa *massacinzentaware* muitas vezes é mais intuitivo começar em 1.

Vamos ver nosso primeiro algoritmo: buscar um valor particular em um arranjo. Isto é, dado um arranjo, queremos saber qual entrada no arranjo, se houver alguma, contém um valor dado. Para ver como podemos fazer uma busca em um arranjo, vamos imaginar que ele seja uma longa prateleira cheia de livros e supor que você queira saber em que lugar da prateleira pode encontrar um livro escrito por Jonathan Swift. Os livros na prateleira podem estar organizados de algum modo, talvez em ordem alfabética por autor, em ordem alfabética por título ou, em uma biblioteca, pelo número de chamada. Talvez a prateleira de livros seja como a que eu tenho em casa, na qual meus livros não estão organizados de nenhum modo particular.

Se você não puder saber de antemão que os livros estão organizados na prateleira, como encontrará o livro de Jonathan Swift? Eis o algoritmo que eu seguiria. Eu começaria na extremidade esquerda da prateleira e examinaria o livro que está na extremidade esquerda. Se for de Swift, localizei o livro. Caso contrário, examinaria o próximo livro à direita e, se esse livro fosse de Swift, teria localizado o livro. Se não, continuaria indo para a direita e examinaria livro após livro até encontrar um livro de Swift ou até chegar à extremidade direita da prateleira, caso em que poderia concluir que ela não contém nenhum livro de Jonathan Swift. (No Capítulo 3, veremos como buscar um livro quando os livros *estão* organizados na prateleira.)

Agora veja como podemos descrever esse problema de busca em termos de computação. Vamos imaginar que os livros que estão na prateleira formam um arranjo. O livro na extremidade esquerda está na posição 1, o próximo livro à direita dele está na posição 2, e assim por diante. Se tivermos n livros na prateleira, o livro na extremidade direita estará na posição n . Queremos determinar o número de posição na prateleira de qualquer livro de Jonathan Swift.

Como um problema de computação geral, temos um arranjo A (a prateleira inteira cheia de livros na qual teremos de procurar) de n elementos (os livros individuais) e queremos determinar se um valor x (um livro de Jonathan Swift) está presente no arranjo A . Se estiver, queremos determinar um índice i tal que $A[i] = x$ (a i -ésima posição na prateleira contém um livro de Jonathan Swift). Também precisamos de algum modo de indicar que o arranjo A não contém x (a prateleira não contém nenhum livro de Jonathan Swift). Não supomos que x aparece no máximo uma vez no arranjo (talvez você tenha várias cópias de algum livro) e, portanto, se x estiver presente no arranjo x , ele pode aparecer várias vezes. Tudo o que queremos de um algoritmo de busca é *qualquer* índice no qual encontraremos x no arranjo. Vamos supor que os índices desse arranjo começam em 1, de modo que seus elementos são $A[1]$ até $A[n]$.

Se buscarmos um livro de Jonathan Swift começando na extremidade esquerda da prateleira, verificando livro por livro à medida que prosseguimos para a direita, denominaremos essa técnica **busca linear**. Em termos de um arranjo em um computador, começamos no início do arranjo, examinamos cada elemento do arranjo por vez ($A[1]$ depois $A[2]$, depois $A[3]$, e assim por diante até $A[n]$) e registramos o lugar onde encontramos x , caso o encontremos.

O procedimento a seguir, LINEAR-SEARCH, adota três parâmetros, que separamos por vírgulas na especificação.

Procedimento LINEAR-SEARCH (A, n, x)

Entrada:

- A : um arranjo.
- n : o número de elementos em A no qual procurar.
- x : o valor que buscamos.

Saída: Um índice i para o qual $A[i] = x$ ou o valor especial NOT-FOUND, que pode ser qualquer índice inválido no arranjo, por exemplo, 0 ou qualquer inteiro negativo.

1. Ajustamos *resposta* para NOT-FOUND.
 2. Para cada índice i , indo de 1 a n , em ordem:
 - a. Se $A[i] = x$, então ajuste *resposta* para o valor de i .
 3. Retorne o valor de *resposta* como saída.
-

Além dos parâmetros A , n e x , o procedimento LINEAR-SEARCH usa uma *variável* denominada *resposta*. O procedimento *designa* um valor inicial de NOT-FOUND à *resposta* na etapa 1. A etapa 2 verifica cada entrada de arranjo $A[1]$ até $A[n]$ para ver se a entrada contém o valor x . Sempre que a entrada $A[i]$ for igual a x , a etapa 2A designa o valor corrente de i à *resposta*. Se x aparecer no arranjo, o valor de saída retornando na etapa 3 é o último índice no qual x apareceu. Se x não aparecer no arranjo, o teste de igualdade na etapa 2A nunca será verdadeiro, e o valor de saída retornado é NOT-FOUND, como designado à *resposta* na etapa 1.

Antes de continuarmos discutindo busca linear, comentamos como especificar ações repetidas, como na etapa 2. É bastante comum em algoritmos executar alguma ação para uma variável adotando valores em alguma faixa. A execução de ações repetidas é denominada *laço* e, toda vez que o executamos, o laço é uma *iteração*. Para o laço da etapa 2, escrevi: “Para cada índice i , indo de 1 a n , na ordem.” Em vez disso, de agora em diante, escreverei “Para $i = 1$ até n ”, que é uma frase mais curta, porém transmite a mesma estrutura. Observe que, quando escrevo um laço desse modo, temos de dar à *variável do laço* (aqui, i) um valor inicial (aqui, 1), e em cada iteração do laço temos de testar o valor corrente da variável do laço em relação a um limite (aqui, n). Se o valor corrente da variável do laço for menor ou igual ao limite, fazemos tudo no *corpo* do laço, (aqui, a etapa 2A). Após uma iteração executar o corpo do laço, **incrementamos** a variável do laço — somando 1 a ela — e voltamos a comparar a variável do laço, agora com seu novo valor, com o limite. Testamos repetidamente a variável do laço em relação ao limite, executamos o corpo do laço e incrementamos a variável do laço até que ela ultrapasse esse limite. Então, a execução continua da etapa que vem imediatamente após o corpo do laço (aqui, etapa 3). Um laço da forma “Para $i = 1$ até n ”, executa n iterações e $n + 1$ testes em relação ao limite (porque a variável do laço ultrapassa o limite no teste $n + 1$).

Espero que ache óbvio que o procedimento LINEAR-SEARCH sempre retorna uma resposta correta. Todavia, você pode ter notado que esse procedimento é inefficiente: ele continua a pesquisar o arranjo mesmo após ter encontrado um índice i para o qual $A[i] = x$. Normalmente, você não continuaria a procurar um livro assim que o encontrasse em sua prateleira, continuaria? Em vez disso, podemos escrever nosso procedimento de busca linear de modo que ele pare assim que encontrar o valor x no arranjo. Presumimos que, quando dizemos retornar um valor, o procedimento imediatamente retorna o valor ao seu chamador, que então assume o controle.

Procedimento BETTER-LINEAR-SEARCH (A, n, x)

Entradas e Saída: As mesmas de LINEAR-SEARCH.

1. Para $i = 1$ até n :
 - a. Se $A[i] = x$, então retorne o valor de i como saída.
 2. Retorne NOT-FOUND como saída.
-

Acredite ou não, podemos tornar a busca linear ainda mais eficiente. Observe que, cada vez que passamos pelo laço da etapa 1, o procedimento BETTER-LINEAR-SEARCH faz dois testes: um teste na etapa 1 para determinar se $i \leq n$ (e, se for, executar mais uma iteração do laço) e o teste da igualdade na etapa 1A. Em termos de fazer uma busca em uma prateleira de livros, esses testes correspondem a ter de verificar duas coisas para cada livro: você passou do final da prateleira e, se não passou, o livro seguinte é de Jonathan Swift? É claro que você não sofrerá muito por ter passado do final da prateleira (se estiver examinando os livros muito de perto, o máximo que acontecerá é dar de cara com uma parede no final da prateleira), mas em um programa de computador, em geral, é muito ruim tentar acessar elementos do arranjo depois do final do arranjo. O seu programa pode falhar ou corromper dados.

É possível dar um jeito de executar somente uma verificação para cada livro que examinar. E se você tivesse certeza de que sua prateleira contém um único livro de Jonathan Swift? Então também teria certeza de que o encontraria e, portanto, nunca teria de verificar se chegou ao final da prateleira. Bastaria verificar cada livro por vez para ver se é de Swift.

Talvez você tenha emprestado todos os seus livros de Jonathan Swift ou, então, achou que tinha livros dele mas nunca teve; portanto, poderia não ter certeza de que a sua prateleira contém qualquer livro desse autor. Eis o que você pode fazer. Pegue uma caixa vazia do tamanho de um livro e escreva no lado estreito da caixa (onde seria a lombada de um livro) “*As viagens de Gulliver*” de Jonathan Swift. Então, quando estiver procurando da esquerda para a direita ao longo da prateleira, só precisará verificar se está vendo alguma coisa escrita por Swift; não terá de se preocupar em passar do final da prateleira porque *sabe* que encontrará algo de Swift. A única pergunta é se você realmente encontrou um livro de Swift ou se encontrou a caixa vazia que identificou como se fosse um livro dele. Se encontrou a caixa vazia, na realidade você não tem um livro de Swift. Todavia, isso é fácil de verificar, e você só precisa fazê-lo uma única vez, ao final de sua busca, em vez de uma vez para cada livro na prateleira.

Há mais um detalhe do qual você precisa estar ciente: e se o único livro de Jonathan Swift que você tinha em sua prateleira fosse o livro na extrema direita? Se substituí-lo pela caixa vazia, a sua busca terminará na caixa vazia e você poderá concluir que não tinha o livro. Portanto, terá de fazer mais uma verificação para essa possibilidade, mas é apenas uma verificação, em vez de uma verificação para cada livro na prateleira.

Em termos de um algoritmo de computador, colocaremos o valor x que estávamos procurando na última posição, $A[n]$, depois de salvar o conteúdo de $A[n]$ em outra variável. Assim que encontrarmos x , testaremos para ver se *realmente* o encontramos. Denominamos o valor que pusemos no arranjo **sentinela**, mas você pode imaginá-lo como se fosse a caixa vazia.

Procedimento SENTINEL-LINEAR-SEARCH (A, n, x)

Entradas e saída: As mesmas de LINEAR-SEARCH.

1. Salve $A[n]$ em *último* e então ponha x em $A[n]$.
 2. Iguele i a 1.
 3. Enquanto $A[i] \neq x$, faça o seguinte:
 - a. Incremente i .
 4. Restaure $A[n]$ de *último*.
 5. Se $i < n$ ou $A[n] = x$, retorne o valor de i como saída.
 6. Caso contrário, retorne NOT-FOUND como saída.
-

A etapa 3 é um laço, mas não um laço que conta alguma variável de laço. Em vez disso, o laço itera enquanto a condição se mantiver; aqui, a condição é que $A[i] \neq x$. O modo de interpretar tal laço é realizar o teste (aqui, $A[i] \neq x$) e, se o teste for verdadeiro, fazer tudo no corpo do laço (aqui, etapa 3A, que incrementa i). Então volte e execute o teste, e, se o teste der verdadeiro, execute o corpo. Continue assim, executando o teste e o corpo, até o teste dar falso. Então continue da etapa seguinte após o corpo do laço (aqui, continue da etapa 4).

O procedimento SENTINEL-LINEAR-SEARCH é um pouco mais complicado que os dois primeiros procedimentos de busca linear. Como ele coloca x em $A[n]$ na etapa 1, temos a garantia de que $A[i]$ será igual a x para algum teste na etapa 3. Quando isso acontecer, sairemos do laço da etapa 3, e o índice i não mudará dali em diante. Antes de fazermos qualquer outra coisa, a etapa 4 restaura o valor original em $A[n]$ (minha mãe me ensinou a pôr as coisas de volta em seus lugares depois de usá-las). Então, temos de determinar se realmente encontramos x no arranjo. Como colocamos x no último elemento, $A[n]$, sabemos que, se encontrarmos x em $A[i]$ onde $i < n$, realmente encontramos x e queremos retornar o índice i . E se encontrarmos x em $A[n]$? Isso significa que não encontramos x antes de $A[n]$ e, portanto, precisamos determinar se $A[n]$ é igual a x . Se for, temos de retornar o índice n , que é igual a i nesse ponto, mas se não for temos de retornar NOT-FOUND. A etapa 5 faz esses testes e retorna o índice correto se x estava originalmente no arranjo. Se x foi encontrado só porque a etapa 1 o colocou no arranjo, a etapa 6 retorna NOT-FOUND. Embora SENTINEL-LINEAR-SEARCH tenha de executar dois testes depois de seu laço terminar, ele realiza somente um teste em cada iteração do laço, o que o torna mais eficiente do que LINEAR-SEARCH ou BETTER-LINEAR-SEARCH.

CÓMO CARACTERIZAR OS TEMPOS DE EXECUÇÃO

Vamos voltar ao procedimento LINEAR-SEARCH da página 12 e entender seu tempo de execução. Lembre-se de que queremos caracterizar o tempo de execução em função do tamanho da entrada. Aqui, nossa entrada é um arranjo A de n elementos, juntamente com o número n e o valor x que estamos buscando. O tamanhos de n e x são insignificantes à medida que o arranjo fica grande — afinal, n é apenas um inteiro isolado e x é apenas tão grande quanto um dos n elementos do arranjo —, portanto diremos que o tamanho da entrada é n , o número de elementos em A .

Temos de adotar algumas premissas simples sobre o tempo que as coisas demoram. Presumiremos que cada operação individual — seja uma operação aritmética (como adição,

subtração, multiplicação ou divisão), seja uma comparação, uma designação a uma variável, uma indexação a um arranjo ou uma chamada ou retorno de um procedimento — demora alguma quantidade de tempo fixa que é independente do tamanho da entrada.² O tempo poderia variar de operação a operação, por exemplo, uma divisão poderia levar mais tempo que uma adição, porém quando uma etapa comprehende apenas operações simples, cada execução individual daquela etapa leva alguma quantidade de tempo constante. Como as operações executadas são diferentes de etapa a etapa, e em razão dos fatores extrínsecos que apresentamos na página 3, o tempo para executar uma etapa poderia variar de etapa a etapa. Vamos dizer que cada execução da etapa i leva tempo t_i , onde t_i é alguma constante que não depende de n .

É claro que temos de levar em conta que algumas etapas são executadas várias vezes. As etapas 1 e 3 são executadas apenas uma vez, mas e a etapa 2? Temos de testar i em relação a n um total de $n + 1$ vezes: n vezes nas quais $i \leq n$ e uma vez quando i é igual a $n + 1$ para sairmos do laço. A etapa 2A é executada exatamente n vezes, uma vez para cada valor de i de 1 a n . Não sabemos de antemão quantas vezes igualaremos *resposta* ao valor de i ; poderia ser qualquer número de vezes, de 0 (se x não estiver presente no arranjo) a n (se todo valor de n no arranjo for igual a x). Se quisermos ser precisos em nossas contas — e normalmente não seremos assim tão precisos —, teremos de reconhecer que a etapa 2 faz duas coisas diferentes que são executadas um número diferente de vezes: o teste de i em relação a n ocorre $n + 1$ vezes, mas incrementar i acontece somente i vezes. Vamos separar o tempo para a linha 2 em t'_2 para o teste e t''_2 para incrementar. De modo semelhante, separaremos o tempo para a etapa 2A em t'_{2A} para testar se $A[i] = x$ e t''_{2A} para igualar *resposta* a i . Portanto, o tempo de execução de LINEAR-SEARCH está em algum lugar entre

$$t_1 + t'_2 \cdot (n + 1) + t''_2 \cdot n + t'_{2A} \cdot n + t''_{2A} \cdot 0 + t_3$$

e

$$t_1 + t'_2 \cdot (n + 1) + t''_2 \cdot n + t'_{2A} \cdot n + t''_{2A} \cdot n + t_3.$$

Agora reescrevemos novamente esses limites, reunimos os termos que são multiplicados por n , reunimos o resto dos termos e vemos que o tempo de execução está em algum lugar entre o ***limite inferior***

$$(t'_2 + t''_2 + t'_{2A}) \cdot n + (t_1 + t'_2 + t_3)$$

e o ***limite superior***

$$(t'_2 + t''_2 + t'_{2A} + t''_{2A}) \cdot n + (t_1 + t'_2 + t_3).$$

² Se você conhece um pouco de arquitetura de computadores, saberá que o tempo para acessar uma variável ou elemento de arranjo dado não é necessariamente fixo, já que pode depender de uma variável ou elemento de arranjo estar na memória cache, na memória principal ou fora da memória, em um disco ou sistema de memória virtual. Alguns modelos sofisticados de computadores levam essas questões em conta, mas muitas vezes é suficiente presumir que todas as variáveis e entradas de arranjo estão na memória principal e que o acesso a todas elas leva a mesma quantidade de tempo.

Observe que ambos os limites são da forma $c \cdot n + d$, onde c e d são constantes que não dependem de n . Isto é, eles são *funções lineares* de n . O tempo de execução de LINEAR-SEARCH é limitado por baixo por uma função linear de n e por cima por uma função linear de n .

Usamos uma notação especial para indicar que um tempo de execução é limitado por cima por alguma função linear de n e por baixo por alguma função linear (possivelmente diferente) de n . Escrevemos que o tempo de execução é $\Theta(n)$. Esta é a letra grega teta, e dizemos “teta de n ” ou apenas “teta n ”. Como prometido no Capítulo 1, essa notação descarta o termo de ordem baixa ($t_1 + t'_2 + t_3$) e os coeficientes de $n(t'_2 + t''_2 + t'_{2A})$ para o limite inferior e $t'_2 + t''_2 + t'_{2A} + t''_{2A}$ para o limite superior). Embora percamos precisão por caracterizarmos o tempo de execução como $\Theta(n)$, ganhamos as vantagens de destacar a ordem de crescimento do tempo de execução e suprimir detalhes tediosos.

Essa notação Θ aplica-se a funções em geral, e não apenas às que descrevem tempos de execução de algoritmos, aplicando-se a outras funções que não as lineares. A ideia é que, se temos duas funções, $f(n)$ e $g(n)$, dizemos que $f(n)$ é $\Theta(g(n))$ se $f(n)$ estiver dentro de um fator constante em relação a $g(n)$ para n suficientemente grande. Portanto, podemos dizer que o tempo de execução de LINEAR-SEARCH está dentro de um fator constante em relação a n , tão logo n torne-se grande o suficiente.

Há uma definição técnica assustadora da notação Θ , mas felizmente é raro que tenhamos de recorrer a ela para usar a notação. Simplesmente focalizamos o termo dominante descartando termos de ordens mais baixas e fatores constantes. Por exemplo, a função $n^2/4 + 100n + 50$ é $\Theta(n^2)$; nesse caso descartamos os termos de baixa ordem $100n$ e 50 , e descartamos o fator constante $1/4$. Embora os termos de baixa ordem dominem $n^2/4$ para valores pequenos de n , assim que n passar de 400 , o termo $n^2/4$ ultrapassará $100n + 50$. Quando $n = 1.000$, o termo dominante $n^2/4$ é igual a 250.000 , ao passo que os termos de baixa ordem $100n + 50$ somam somente 100.050 ; para $n = 2.000$, a diferença torna-se $1.000.000$ versus 200.050 . No mundo dos algoritmos, abusamos um pouco da notação e escrevemos $f(n) = \Theta(g(n))$, de modo que podemos escrever $n^2/4 + 100n + 50 = \Theta(n^2)$.

Agora vamos examinar o tempo de execução de BETTER-LINEAR-SEARCH da página 13. Esse é um pouco mais complicado que o de LINEAR-SEARCH porque não sabemos de antemão quantas vezes o laço iterará. Se $A[1]$ igual a x , ele iterará apenas uma vez. Se x não estiver presente no arranjo, o laço iterará todas as n vezes, que é o máximo possível. Cada iteração do laço leva alguma quantidade de tempo constante; portanto, podemos dizer que, *no pior caso*, BETTER-LINEAR-SEARCH leva o tempo $\Theta(n)$ para buscar um arranjo de n elementos. Por que “pior caso”? Como queremos que os algoritmos tenham baixos tempos de execução, o pior caso ocorre quando um algoritmo leva o tempo máximo para qualquer entrada possível.

No melhor caso, quando $A[1]$ é igual a x , BETTER-LINEAR-SEARCH leva apenas uma quantidade de tempo constante: ele iguala i a 1 , verifica se $i \leq n$, o teste $A[1] = x$ resulta verdadeiro e o procedimento retorna o valor de i , que é 1 . Essa quantidade de tempo não depende de n . Escrevemos que o *tempo de execução do melhor caso* de BETTER-LINEAR-SEARCH é $\Theta(1)$ porque, no melhor caso, seu tempo de execução está dentro de um fator constante de 1 . Em outras palavras, o tempo de execução do melhor caso é uma constante que não depende de n .

Portanto, vemos que não podemos usar a notação Θ como uma afirmação abrangente que se aplica a todos os casos do tempo de execução de BETTER-LINEAR-SEARCH. Não podemos dizer que o tempo de execução é sempre $\Theta(n)$ porque, no melhor caso, ele é $\Theta(1)$. E não podemos dizer que o tempo de execução é sempre $\Theta(1)$ porque, no pior caso, ele é $\Theta(n)$. Todavia, podemos dizer que uma função linear de n é um *limite superior* em todos os casos e que temos uma notação para isso: $O(n)$. Quando falamos nessa notação, dizemos “Ó maiúsculo de n ” (“big-oh of n ”) ou apenas “ó de n ” (“oh of n ”). Uma função $f(n)$ é $O(g(n))$ se, logo que n tornar suficientemente grande, $f(n)$ é limitada por cima por alguma constante vezes $g(n)$. Novamente, abusamos um pouco da notação e escrevemos $f(n) = O(g(n))$. Para BETTER-LINEAR-SEARCH, podemos usar a declaração abrangente que diz que seu tempo de execução em todos os casos é $O(n)$; embora o tempo de execução possa ser melhor que uma função linear de n , ele nunca é pior.

Usamos a notação O para indicar que um tempo de execução nunca é *pior* que uma constante vezes alguma função de n , mas e se quiséssemos indicar que um tempo de execução nunca é *melhor* que uma constante vezes alguma função de n ? Esse é um limite inferior, e usamos a notação Ω , que é a imagem especular da notação O : uma função $f(n)$ é $\Omega(g(n))$ se, logo que n tornar-se suficientemente grande, $f(n)$ é limitada por baixo por alguma constante vezes $g(n)$. Dizemos que “ $f(n)$ é ômega maiúsculo (ômega grande) de $g(n)$ ” ou apenas que “ $f(n)$ é ômega de $g(n)$ ”, e podemos escrever $f(n) = \Omega(g(n))$. Visto que a notação O dá um limite superior, a notação Ω dá um limite inferior, e a notação Θ dá ambos os limites, superior e inferior, podemos concluir que uma função $f(n)$ é $\Theta(g(n))$ se e somente se $f(n)$ for $O(g(n))$ e $\Omega(g(n))$.

Podemos fazer uma declaração abrangente sobre um limite inferior para o tempo de execução de BETTER-LINEAR-SEARCH: em todos os casos ele é $\Omega(1)$. É claro que essa declaração é pateticamente fraca, visto que seria de esperar que qualquer algoritmo aplicado a qualquer entrada levasse, no mínimo, tempo constante. Não usaremos muito a notação Ω , mas ocasionalmente ela virá a calhar.

O termo abrangente para a notação, Θ a notação O e a notação Ω é **notação assintótica**. Isso porque essas notações capturam o crescimento de uma função à medida que seu argumento aproxima-se assintoticamente de infinito. Todas essas notações assintóticas nos dão o luxo de descartar termos de baixa ordem e fatores constantes de modo que possamos ignorar detalhes tediosos e focalizar o que é importante: como a função cresce com n .

Agora vamos voltar a SENTINEL-LINEAR-SEARCH, da página 14. Exatamente como BETTER-LINEAR-SEARCH, cada iteração de seu laço leva uma quantidade de tempo constante e pode haver qualquer número de 1 a n interações. A diferença fundamental entre SENTINEL-LINEAR-SEARCH e BETTER-LINEAR-SEARCH é que o tempo por iteração de SENTINEL-LINEAR-SEARCH é menor que o tempo por iteração de BETTER-LINEAR-SEARCH. Ambos levam uma quantidade de tempo linear no pior caso, mas o fator constante para SENTINEL-LINEAR-SEARCH é melhor. Embora pudéssemos esperar que SENTINEL-LINEAR-SEARCH fosse mais rápido na prática, ele o seria apenas por um fator constante. Quando expressamos o tempo de execução de BETTER-LINEAR-SEARCH e SENTINEL-LINEAR-SEARCH usando notação assintótica, eles são equivalentes: $\Theta(n)$ no pior caso, $\Theta(1)$ no melhor caso e $O(n)$ em todos os casos.

INVARIANTES DE LAÇO

Para os nossos três tipos de busca linear, foi fácil ver que cada um dá uma resposta correta. Às vezes, é um pouco mais difícil. Há uma ampla gama de técnicas, mas do que eu poderia comentar aqui.

Um método comum de mostrar correção usa uma **invariante de laço**: uma afirmativa que demonstramos ser verdadeira cada vez que iniciamos uma iteração de laço. Para que uma invariante de laço nos ajude a questionar a correção, temos de mostrar três coisas sobre ela:

Inicialização: É verdadeira antes da primeira iteração do laço.

Manutenção: Se é verdadeira antes de uma iteração do laço, permanecerá verdadeira antes da próxima iteração.

Terminação: O laço termina e, quando termina, a invariante do laço, juntamente com a razão do término do laço, nos dá uma propriedade útil.

Como exemplo, damos aqui uma invariante de laço para BETTER-LINEAR-SEARCH:

No início de cada iteração da etapa 1, se x estiver presente no arranjo A , estará presente no **subvetor** (uma porção contígua de um arranjo) de $A[i]$ a $A[n]$.

Nem precisamos dessa invariante de laço para mostrar que, se o procedimento retornar um índice que não seja NOT-FOUND, o índice retornado é correto: o único modo de o procedimento poder retornar um índice i na etapa 1A é porque x é igual a $A[i]$. Em vez disso, usaremos essa invariante de laço para mostrar que, se o procedimento retornar NOT-FOUND na etapa 2, x não está em nenhum lugar no arranjo:

Inicialização: Inicialmente, $i = 1$ de modo que o subvetor na invariante de laço é $A[i]$ até $A[n]$, que é o arranjo inteiro.

Manutenção: Considere que, no início de uma iteração para um valor de i , se x estiver presente no arranjo A , ele estará presente na forma do subvetor de $A[i]$ a $A[n]$. Se passarmos por essa iteração sem retornar, saberemos que $A[i] \neq x$ e poderemos dizer com segurança que, se x estiver presente no arranjo A , ele estará presente no subvetor de $A[i + 1]$ a $A[n]$. Como i é incrementado antes da próxima iteração, a invariante de laço continua a valer antes da próxima iteração.

Terminação: Esse laço deve terminar, seja porque o procedimento retorna na etapa 1A seja porque $i > n$. Já tratamos do caso em que o laço termina porque o procedimento retorna na etapa 1A.

Para tratar do caso em que o laço termina porque $i > n$, recorremos ao contrapositivo da invariante de laço. O **contrapositivo** da declaração “se A então B ” é “se não B então não A ”. O contrapositivo de uma declaração é verdadeiro se e somente se a declaração for verdadeira. O contrapositivo da invariante de laço é “se x não está presente no subvetor de $A[i]$ a $A[n]$, então ele não está presente no arranjo A ”.

Agora, quando $i > n$, o subvetor de $A[i]$ a $A[n]$ é vazio e, portanto, esse subvetor não pode conter x . Portanto, pelo contrapositivo da invariante do laço, x não está presente em nenhum lugar no arranjo A e, portanto, é adequado retornar NOT-FOUND na etapa 2.

Uau! É muito raciocínio para o que, na realidade, é um simples laço! Temos que passar por tudo isso todas as vezes que escrevermos um laço? Eu não, mas há alguns cientistas da computação que insistem em tal raciocínio rigoroso para cada laço que



seja. Quando estou escrevendo código real, na maioria das vezes que escrevo um laço constato que tenho uma invariante de laço em algum lugar no fundo da memória. Ela pode estar tão fundo na minha mente que nem mesmo percebo que ela está lá, mas eu poderia declará-la caso fosse necessário. Embora a maioria de nós concorde que uma invariante de laço já é demais para entender o simples laço em BETTER-LINEAR-SEARCH, invariantes de laço podem vir muito bem a calhar quando queremos entender por que laços mais complexos fazem a coisa certa.

RECURSÃO

Com a técnica de recursão, resolvemos um problema resolvendo instâncias menores do mesmo problema. Dou aqui o meu exemplo canônico favorito de recursão: computar $n!$ (“fatorial de n ”), que é definido por valores não negativos de n como $n! = 1$ se $n = 0$ e

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot (n - 3) \cdots 3 \cdot 2 \cdot 1$$

se $n \geq 1$. Por exemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Observe que

$$(n - 1)! = (n - 1) \cdot (n - 2) \cdot (n - 3) \cdots 3 \cdot 2 \cdot 1,$$

e, portanto,

$$n! = n \cdot (n - 1)$$

para $n \geq 1$. Definimos $n!$ em termos de um problema “menor”, a saber, $(n - 1)!$. Poderíamos escrever um procedimento recursivo para calcular $n!$ da seguinte maneira:

Procedimento FATORIAL (n)

Entrada: Um inteiro $n \geq 0$.

Saída: O valor de $n!$.

1. Se $n = 0$, então retorne 1 como saída.
 2. Caso contrário, retorne n vezes o valor retornado chamando FACTORIAL($n - 1$) recursivamente.
-

O modo como escrevi a etapa 2 é bem atrapalhado. Em vez disso, eu poderia escrever apenas “Caso contrário, retorne $n \cdot \text{FACTORIAL}(n - 1)$ ” usando o valor de retorno da chamada recursiva dentro de uma expressão aritmética maior.

Para a recursão funcionar, duas propriedades devem valer. A primeira é que deve haver um ou mais **casos-bases**, nos quais computamos a solução diretamente sem recursão. A segunda é que cada chamada recursiva do procedimento deve ser a uma *instância menor do mesmo problema* que eventualmente chegará a um caso-base. Para o procedimento FACTORIAL, o caso-base ocorre quando n é igual a 0, e cada chamada recursiva é sobre uma instância na qual o valor de n é reduzido de 1. Desde que o valor original de n seja não negativo, a certa altura as chamadas recursivas se reduzirão ao caso-base.

Questionemos que o trabalho de um algoritmo recursivo pode parecer exageradamente simples à primeira vista. A chave é acreditar que cada chamada recursiva produz o resultado correto. Desde que estejamos dispostos a acreditar que chamadas recursivas fazem a coisa certa, questionar a correção muitas vezes é fácil. Eis como

poderíamos questionar que o procedimento FACTORIAL retorna a resposta correta. É claro que, quando $n = 0$, o valor retornado, 1, é igual a $n!$. Presumimos que, quando $n \geq 1$, a chamada recursiva FACTORIAL($n - 1$) faz a coisa certa: ela retorna o valor de $(n - 1)!$. Então o procedimento multiplica esse valor por n e, com isso, computa o valor de $n!$, que ele retorna.

Damos aqui um exemplo no qual as chamadas recursivas não são instâncias menores do mesmo problema, embora a matemática esteja correta. É de fato verdadeiro que, se $n \geq 0$, então $n! = (n + 1)!/(n + 1)$. Porém, o seguinte procedimento recursivo, que tira proveito dessa fórmula, nunca conseguiria dar uma resposta quando $n \geq 1$:

Procedimento BAD-FACTORIAL (n)

Entrada e Saída: As mesmas de FACTORIAL.

1. Se $n = 0$, então retorne 1 como saída.
2. Caso contrário, retorne BAD-FACTORIAL($n + 1$)/($n + 1$).

Se chamássemos BAD-FACTORIAL(1), ele geraria uma chamada recursiva de BAD-FACTORIAL(2), que geraria uma chamada recursiva de BAD-FACTORIAL(3), e assim por diante, e jamais chegaria até o caso-base quando n é igual a 0. Se você implementasse esse procedimento em uma linguagem de programação real e o executasse em um computador real, veria rapidamente algo como “erro de estouro de pilha”.

Muitas vezes podemos reescrever algoritmos que usam um laço em estilo recursivo. Damos a seguir busca linear, sem uma sentinela, escrita recursivamente:

Procedimento RECURSIVE-LINEAR-SEARCH (A, n, i, x)

Entradas: As mesmas de LINEAR-SEARCH, mas com um parâmetro adicional i .

Saída: O índice de um elemento igualando x no subvetor de $A[i]$ a $A[n]$ ou NOT-FOUND se x não aparece nesse subvetor.

1. Se $i > n$, então retorne NOT-FOUND.
2. Caso contrário, ($i \leq n$), se $A[i] = x$, então retorne i .
3. Caso contrário, ($i \leq n$ e $A[i] \neq x$), retorne
RECURSIVE-LINEAR-SEARCH ($A, n, i + 1, x$).

Aqui, o subproblema é buscar x no subvetor, indo de $A[i]$ a $A[n]$. O caso-base ocorre na etapa 1 quando esse subvetor está vazio, isto é, quando $i > n$. Como o valor de i aumenta em cada uma das chamadas recursivas da etapa 3, se nenhuma chamada recursiva nunca retornar um valor de i na etapa 2 a certa altura i ficará maior que n e chegaremos ao caso-base.

O QUE MAIS LER?

Os Capítulos 2 e 3 de CLRS [CLRS09] abrangem muito do material deste capítulo. Um dos primeiros livros didáticos sobre algoritmos de autoria de Aho, Hopcroft e Ullman [AHU74] influenciou a área ao utilizar notação assintótica para analisar algoritmos. Tem dado muito trabalho provar que os programas estão corretos; se você quiser ir mais fundo nessa área, experimente ler os livros de Gries [Gri81] e Mitchell [Mit96].

Algoritmos para ordenar e buscar

No Capítulo 2, vimos três variações de busca linear em um vetor. Podemos fazer algo melhor? Resposta: depende. Se nada soubermos sobre a ordem dos elementos no vetor, a resposta é não, não podemos fazer algo melhor. No pior caso, teríamos de examinar todos os b elementos porque, se não encontrarmos o valor que estamos procurando nos primeiros $n - 1$ elementos, ele poderia estar no último ou n -ésimo elemento. Portanto, não poderemos conseguir um tempo de execução do pior caso melhor que $\Theta(n)$ se nada soubermos sobre a ordem dos elementos no vetor.

Todavia, suponha que o vetor esteja ordenado em ordem não decrescente: cada elemento é menor ou igual ao seu sucessor no vetor, de acordo com alguma definição de “menor que”. Neste capítulo veremos que, se um vetor está ordenado, podemos usar uma técnica simples conhecida como busca binária para fazer uma busca em um vetor de n elementos no tempo de apenas $O(\lg n)$. Como vimos no Capítulo 1, o valor de $\lg n$ cresce muito lentamente em comparação com n e, portanto, a busca binária ganha da busca linear no pior caso.¹

O que significa um elemento ser menor que outro? Quando os elementos são números, é óbvio. Quando os elementos são cadeias de caracteres de texto, podemos pensar em uma *ordenação lexicográfica*: um elemento é menor que outro se vier antes do outro elemento em um dicionário. Quando os elementos são alguma outra forma de dados, temos de definir o que significa “menor que”. Desde que tenhamos alguma noção clara de “menor que”, podemos determinar se um vetor é ordenado.

Lembrando o exemplo de livros em uma prateleira, no Capítulo 2, poderíamos ordenar os livros em ordem alfabética por autor, em ordem alfabética por título ou, em uma biblioteca, por número de chamada. Neste capítulo, diremos que os livros estão ordenados na prateleira se aparecerem em ordem alfabética por autor, da esquerda para a direita. Todavia, a prateleira pode conter mais de um livro do mesmo autor; talvez você tenha várias obras de William Shakespeare. Se quisermos buscar não apenas qualquer livro de Shakespeare, mas um livro específico de Shakespeare, diremos que, se dois livros têm o mesmo autor, o livro cujo título vem antes na ordem alfabética deve ficar à esquerda. Alternativamente, poderemos dizer que só o que nos

¹ Se você não é aficionado de computadores e pulou a seção “Algoritmos de computador para aficionados”, no Capítulo 1, deve ler o material sobre logaritmos na página 6.

importa é o nome do autor; portanto, quando fizermos uma busca, qualquer coisa de Shakespeare servirá. Denominamos **chave** a informação que serve de comparação. Em nosso exemplo da prateleira de livros, a chave é apenas o nome do autor, em vez de uma combinação baseada primeiro no nome do autor e depois no título, caso haja duas obras do mesmo autor.

Então, como conseguimos ordenar o vetor, antes de mais nada? Neste capítulo, veremos quatro algoritmos — ordenação por seleção, ordenação por inserção, ordenação por intercalação e ordenação rápida (quicksort) — para ordenar um vetor, aplicando cada um deles ao nosso exemplo da prateleira de livros. Cada algoritmo de ordenação terá suas vantagens e suas desvantagens, e, no final do capítulo, faremos uma revisão e comparação desses algoritmos de ordenação. Todos os algoritmos de ordenação que veremos neste capítulo levam tempo $\Theta(n^2)$ ou $\Theta(n \lg n)$ no pior caso. Portanto, se você for executar somente algumas buscas, será melhor executar apenas busca linear. Mas, se você for executar muitas buscas, o melhor será primeiro ordenar o vetor e depois usar busca binária.

Ordenar é por si só um problema importante, e não somente como etapa prévia de processamento na busca binária. Pense em todos os dados que devem ser ordenados, como as entradas em uma lista telefônica, por nome; cheques no extrato mensal do banco por números e/ou datas nos quais eles foram processados pelo banco; ou até resultados de um motor de busca na Web, por relevância em relação à consulta. Ordenar é, frequentemente, uma etapa em algum outro algoritmo. Por exemplo, em computação gráfica, frequentemente os objetos são dispostos em camadas, uns sobre os outros. Um programa que apresenta objetos na tela teria de ordená-los de acordo com uma relação “acima” para poder desenhá-los de baixo para cima.

Antes de continuarmos, um comentário sobre o que é que ordenamos. Além da chave (que denominaremos **chave de ordenação** quando estamos ordenando), os elementos que ordenamos usualmente incluem também o que denominamos **dados satélites**. Embora dados satélites possam vir de um satélite, em geral não vêm. Dados satélites são as informações associadas à chave de ordenação, e devem acompanhá-la quando movimentamos elementos. Em nosso exemplo da prateleira de livros, a chave de ordenação é o nome do autor e os dados satélites são o livro em si.

Expliquei dados satélites aos meus alunos de um modo que eu tenho certeza de que eles entenderão. Mantenho uma planilha com as notas dos alunos, cujas linhas são ordenadas em ordem alfabética por nome de aluno. Para determinar as notas finais do curso no final do semestre, eu ordeno as linhas, sendo a chave de ordenação a coluna que contém a porcentagem de pontos obtidos no curso, e o resto das colunas, incluindo os nomes dos alunos, os dados satélites. Em seguida, ordeno a planilha em ordem decrescente de porcentagem, de modo que as linhas no topo correspondem a notas A, e as linhas de baixo, a notas D e E.² Suponha que eu rearranjasse somente a coluna que contém a porcentagem sem mover a coluna inteira que contém a porcentagem. Isso deixaria os nomes dos alunos em ordem alfabética independentemente das

² Dartmouth usa E, não F, para indicar uma nota que reprovaria o aluno. Não sei bem por que, mas imagino que isso tenha simplificado o programa de computador que converte notas representadas por letras em uma escala de 4 a 0.

porcentagens. Então, os alunos cujos nomes aparecem em primeiro lugar no alfabeto ficariam felizes, enquanto os que viesssem no final do alfabeto, nem tanto.

Damos aqui outros exemplos de chaves de ordenação e dados satélites. Em uma lista telefônica, a chave de ordenação seria o nome, e os dados satélites seriam o endereço e o número do telefone. Em um extrato bancário, a chave de ordenação seria o número do cheque, e os dados satélites incluiriam o montante do cheque e a data em que foi compensado. Em um motor de busca, a chave de ordenação seria a medida da relevância para a consulta, e os dados satélites seriam o URL da página Web mais quaisquer outros dados sobre a página que estão armazenados no motor de busca.

Quando trabalharmos com vetores neste capítulo, agiremos como se cada elemento contivesse somente uma chave de ordenação. Se você estiver implementando qualquer dos algoritmos de ordenação dados aqui, terá de certificar-se de estar movendo os dados satélites associados a cada elemento ou, no mínimo, um ponteiro para os dados satélites sempre que mover a chave de ordenação.

Para que a analogia da prateleira de livros aplique-se a vetores em um computador, precisamos pressupor que a prateleira e seus livros têm dois aspectos adicionais, que eu admito que não são terrivelmente realistas. O primeiro é que todos os livros na prateleira são do mesmo tamanho porque, em um vetor de computador, todas as entradas de vetor são do mesmo tamanho. O segundo é que podemos numerar as posições dos livros na prateleira de 1 a n , e denominaremos cada posição **espaço (slot)**. Espaço 1 é o espaço na extrema esquerda e espaço n é o espaço na extrema direita. Como você provavelmente já percebeu, cada espaço na prateleira corresponde a uma entrada no vetor.

Quero também comentar a palavra “ordenação”. Em linguagem comum, ordenação pode significar algo diferente do que aquilo que usamos em computação.

O dicionário on-line My Mac define “ordenar” como “arranjar sistematicamente em grupos; separar de acordo com tipo, classe etc.”: o modo como poderíamos “ordenar” roupas, por exemplo, camisas em um lugar, calças em outro, e assim por diante. No mundo dos algoritmos de computador, ordenar significa pôr em alguma ordem bem definida, e “arranjar sistematicamente em grupos” significa “colocar em um balde” (“bucketing”, “bucketizing”) ou em uma “cesta” (“binning”) ou, ainda, “classificar”.

BUSCA BINÁRIA

Antes de vermos alguns algoritmos de ordenação, vamos ver o que é busca binária, que requer que o vetor a ser submetido à busca já esteja ordenado. A busca binária tem a vantagem de levar somente tempo $O(\lg n)$ para fazer uma busca em um vetor de n elementos.

Em nosso exemplo da prateleira de livros, começamos com os livros já ordenados por nome de autor, da esquerda para a direita na prateleira. Usaremos o nome do autor como a chave e buscaremos qualquer livro de Jonathan Swift. Agora você pode entender que, como o sobrenome do autor começa com “S”, que é a décima nona letra do alfabeto, pode percorrer três quartos do caminho na prateleira (visto que $19/26$ é aproximadamente $3/4$) e fazer a busca ali. Porém, se você tiver todas as obras de Shakespeare, tem vários livros de um autor cujo sobrenome vem antes de Swift, o que pode empurrar os livros de Swift mais para a direita do que você esperava.

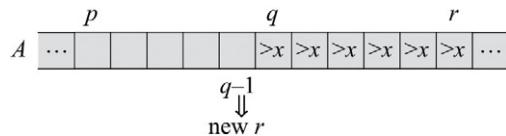
Em vez disso, veja como você poderia aplicar a busca binária para encontrar um livro de Jonathan Swift. Vá até o espaço que está exatamente no meio da prateleira, ache o livro que está ali e examine o nome do autor. Vamos dizer que você encontrou um livro de Jack London. Não somente esse não é o livro que você está procurando, mas, como você sabe que os livros estão ordenados em ordem alfabética de autor, também sabe que todos os livros à esquerda do livro de London não podem ser o livro que você está procurando. Examinando apenas um livro, você não precisou considerar metade dos livros que estão na prateleira! Qualquer livro de Swift deve estar na metade direita da prateleira. Portanto, agora você achará o espaço que está no ponto a meio caminho na metade direita da prateleira, e procurará o livro ali. Suponha que seja um livro de Leon Tolstói. Novamente, esse não é o livro que procura, mas você sabe que pode eliminar todos os livros à direita dele: metade dos livros que continuaram como possibilidades. Nesse ponto, você sabe que, se sua prateleira contiver qualquer livro de Swift, eles estarão no quarto de livros que está à direita do livro de London e à esquerda do livro de Tolstói. Em seguida, você acha o livro no espaço que está a meio caminho nesse quarto em consideração. Se for de Swift, a rodada terminou. Caso contrário, novamente você pode eliminar metade dos livros restantes. A certa altura, ou você acha o livro de Swift ou chega ao ponto no qual nenhum espaço é uma possibilidade. Neste último caso, você conclui que a prateleira não contém nenhum livro de Jonathan Swift.

Em um computador, fazemos busca binária em um vetor. A qualquer ponto, estamos considerando apenas um subvetor, isto é, a porção do vetor entre dois índices, incluindo os dois índices; vamos denominá-los p e r . Inicialmente, $p = 1$ e $r = n$, de modo que o subvetor começa como o vetor inteiro. Dividimos repetidamente ao meio o tamanho do subvetor que estamos considerando até que uma de duas coisas aconteça: ou encontramos o valor que estamos procurando ou o subvetor está vazio (isto é, p torna-se maior que r). A divisão repetitiva do tamanho do subvetor ao meio é o que dá origem ao tempo de execução $O(\lg n)$.

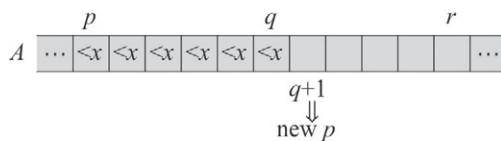
Com um pouco mais de detalhes, eis como uma busca binária funciona. Vamos dizer que estejamos buscando o valor x no vetor A . Em cada etapa, estamos considerando somente o subvetor que começa em $A[p]$ e termina em $A[r]$. Como estaremos trabalhando bastante com subvetores, vamos denotar esse subvetor por $A[p..r]$. A cada etapa, computamos o ponto médio q do subvetor em consideração, calculando a média entre p e r , e descartando a parte fracionária, se houver alguma: $q = \lfloor (p + r) / 2 \rfloor$. (Aqui, usamos a operação “piso”, $\lfloor \rfloor$, para descartar a parte fracionária. Se você estiver implementando essa operação em linguagem como Java, C ou C++, poderá apenas usar divisão inteira para descartar a parte fracionária.) Verificamos se $A[q]$ é igual a x ; se for, terminamos, porque podemos apenas retornar q como um índice em que o vetor A contém x .

Se, ao contrário, constatarmos que $A[q] \neq x$, aproveitamos a vantagem de supor que o vetor A já está ordenado. Visto que $A[q] \neq x$, há duas possibilidades: $A[q] > x$ ou $A[q] < x$. Em primeiro lugar tratamos do caso em que $A[q] > x$. Como o vetor está ordenado, sabemos não somente que $A[q]$ é maior que x , mas também — considerando que o vetor está disposto da esquerda para a direita — que todo elemento do vetor à direita de $A[q]$ é maior que x . Portanto, podemos eliminar de consideração todos os

elementos à direita de $A[q]$. Iniciaremos nossa próxima etapa sem mudar p , mas com r igual a $q - 1$:



Se, em vez disso, constatarmos que $A[q] < x$, saberemos que cada elemento do vetor à esquerda de $A[q]$ é menor que x e, portanto, podemos eliminar esses elementos de consideração. Começaremos nossa próxima etapa sem mudar r , mas com p igual a $q + 1$:



Damos a seguir o exato procedimento para busca binária:

Procedimento BINARY-SEARCH (A, n, x)

Entradas e Saída: As mesmas de LINEAR-SEARCH.

1. Iguale p a 1 e r a n .
2. Enquanto $p \leq r$, faça o seguinte:
 - a. Iguale q a $\lfloor (p+r)/2 \rfloor$.
 - b. Se $A[q] = x$, então retorne q .
 - c. Caso contrário, ($A[q] \neq x$), se $A[q] > x$, então iguale r a $q - 1$.
 - d. Caso contrário, ($A[q] < x$), iguale p a $q + 1$.
3. Retorne NOT-FOUND.

O laço na etapa 2 não termina necessariamente porque p tornou-se maior que r . Ele pode terminar na etapa 2B porque constata que $A[q]$ é igual a x e retorna q como um índice em A , onde x ocorre.

Para mostrar que o procedimento BINARY-SEARCH funciona corretamente, basta mostrar que x não está presente em nenhum lugar no vetor se BINARY-SEARCH retornar NOT-FOUND na etapa 3. Usamos a seguinte invariante de laço:

No início de cada iteração do laço da etapa 2, se x estiver em algum lugar no vetor A , está em algum lugar no subvetor $A[p..r]$.

E um breve argumento usando a invariante de laço:

Inicialização: A etapa 1 inicializa os índices p e r para 1 e n , respectivamente, e portanto a invariante do laço é verdadeira quando o procedimento entra pela primeira vez no laço.

Manutenção: Argumentamos acima que as etapas 2C e 2D ajustam p ou r corretamente.

Término: Se x não está no vetor, a certa altura o procedimento chega ao ponto onde p e r são iguais. Quando isso acontece, a etapa 2A computa q como o mesmo

p e r . Se a etapa 2C igualar r a $q - 1$, no início da próxima iteração r será igual a $p - 1$, de modo que p será maior que r . Se a etapa 2D igualar p a $q + 1$, no início da próxima iteração p será igual a $r + 1$, e novamente p será maior que r . De qualquer modo, o teste do laço na etapa 2 resultará falso e o laço terminará. Como $p > r$, o subvetor $p > r$ estará vazio e, portanto, o valor x não pode estar presente nele. Tomar o contrapositivo da invariante do laço (veja página 18) nos diz que, se x não está presente no subvetor $A[p..r]$, então não está presente em nenhum lugar no vetor A . Portanto, o procedimento está correto ao retornar NOT-FOUND na etapa 3. Podemos também escrever busca binária como um procedimento recursivo:

Procedimento RECURSIVE-BINARY-SEARCH (A, p, r, x)

Entradas e Saída: As entradas A e x são as mesmas de LINEAR-SEARCH, assim como a saída. As entradas p e r delimitam o subvetor $A[p..r]$ em consideração.

1. Se $p > r$, então retorne NOT-FOUND.
2. Caso contrário $p \leq r$, faça o seguinte:
 - a. Iguale q para $\lfloor (p+r)/2 \rfloor$.
 - b. Se $A[q] = x$, então retorne q .
 - c. Caso contrário, ($A[q] \neq x$), se $A[q] > x$, então retorne
RECURSIVE-BINARY-SEARCH $A, p, q - 1, x$.
 - d. Caso contrário, ($A[q] < x$), retorne
RECURSIVE-BINARY-SEARCH $(A, q + 1, r, x)$.

A chamada inicial é RECURSIVE-BINARY-SEARCH $(A, 1, n, x)$.

Agora vamos ver como é que a busca binária leva tempo $O(\lg n)$ em um vetor de n elementos. A observação fundamental é que o tamanho $r - p + 1$ do subvetor em consideração é dividido aproximadamente ao meio em cada iteração do laço (ou em cada chamada recursiva da versão recursiva, mas vamos focalizar a versão iterativa em BINARY-SEARCH). Se você tentar todos os casos constatará que, se uma iteração começa com um subvetor de s elementos, a próxima iteração terá $\lfloor s/2 \rfloor$ ou $s/2 - 1$ elementos, dependendo de s ser par ou ímpar e de $A[q]$ ser maior ou menor que x . Já vimos uma vez que logo que o tamanho do subvetor chega a 1, o procedimento termina na próxima iteração. Portanto, podemos perguntar de quantas iterações do laço precisamos para dividir repetidamente ao meio um subvetor desde o seu tamanho original n até o tamanho 1. Isso seria o mesmo que o número de vezes que, começando com um subvetor de tamanho 1, precisaríamos para dobrar seu tamanho para chegar a um tamanho de n . Mas isso é apenas exponenciação: multiplicar repetidamente por 2. Em outras palavras, para qual valor de x 2^x chega a n ? Se n fosse uma potência exata de 2, já vimos na página 6 que a resposta seria $\lg n$. É claro que n poderia não ser uma potência exata de 2, caso em que a resposta estaria dentro de 1 de $\lg n$. Finalmente, observamos que cada iteração do laço leva uma quantidade de tempo constante, isto é, o tempo para uma única iteração não depende do tamanho n do vetor original ou do tamanho do subvetor sob consideração. Vamos usar notação assintótica para suprimir os fatores constantes e o termo de baixa ordem. (O número de iterações do laço é $\lg n$ ou $\lfloor \lg n + 1 \rfloor$? Quem se importa?) Entendemos que o tempo de execução de busca binária é $O(\lg n)$.

Aqui usei notação O porque queria fazer uma declaração abrangente que cobrisse todos os casos. No pior caso, quando o valor x não está presente no vetor, dividimos ao meio e dividimos ao meio e dividimos ao meio até o subvetor sob consideração ficar vazio, o que dá um tempo de execução de $\Theta(\lg n)$. No melhor caso, quando x é encontrado na primeira iteração do laço, o tempo de execução é $\Theta(1)$. Nenhuma notação Θ cobre todos os casos, mas um tempo de execução de $O(\lg n)$ é sempre correto para busca binária, desde que o vetor já esteja ordenado.

É possível bater o tempo $\Theta(\lg n)$ do pior caso para busca, mas somente se organizarmos os dados de modos mais elaborados e adotarmos certas premissas sobre as chaves.

ORDENAÇÃO POR SELEÇÃO

Agora voltamos nossa atenção à *ordenação*: rearranjar os elementos do vetor — também conhecido como *permutar* o vetor — de modo que cada elemento seja menor ou igual ao seu sucessor. O primeiro algoritmo de ordenação que veremos, a ordenação por seleção, é o que eu considero o mais simples porque é o algoritmo que me veio à mente na primeira vez em que precisei projetar um algoritmo de ordenação. Está longe de ser o mais rápido.

Eis como a ordenação por seleção funcionaria para ordenar livros em uma prateleira de acordo com os nomes dos autores. Examine a prateleira inteira e ache o livro cujo nome do autor vem antes no alfabeto. Vamos dizer que seja um livro de Louisa May Alcott (se a prateleira contiver dois ou mais livros dessa autora, escolha qualquer um deles). Troque a localização desse livro com o livro que está no espaço 1. Agora o livro no espaço 1 é um livro de um autor cujo nome vem antes na ordem alfabética. Percorra a prateleira de livros da esquerda para a direita, começando com o livro no espaço 2 para encontrar o livro nos espaços 2 a n cujo nome do autor vem antes no alfabeto. Suponha que seja Jane Austen. Troque a localização desse livro com o livro no espaço 2, de modo que agora os espaços 1 e 2 tenham o primeiro e o segundo livros na ordenação alfabética geral. Faça o mesmo para o espaço 3, e assim por diante. Uma vez colocado o livro correto no espaço $n - 1$ (talvez seja de H. G. Wells), terminamos porque sobrou apenas um livro à esquerda (digamos, um livro de Oscar Wilde) e ele está no espaço n , onde deve estar.

Para transformar essa abordagem em um algoritmo de computador, troque a prateleira de livros por um vetor e os livros por elementos do vetor. Aqui está o resultado:

Procedimento **SELECTION-sort** (A, n)

Entradas:

- A : um vetor.
- n : o número de elementos em A a ordenar.

Resultado: Os elementos de A são ordenados em ordem não decrescente.

1. Para $i = 1$ a $n - 1$:
 - a. Iguele *menor* ao índice do menor elemento no subvetor $A[i..n]$.
 - b. Troque $A[i]$ por $A[menor]$.
-

Encontrar o menor elemento em $A[i..n]$ é uma variante da busca linear. Em primeiro lugar, declare $A[i]$ como o menor elemento visto no subvetor até esse ponto e então

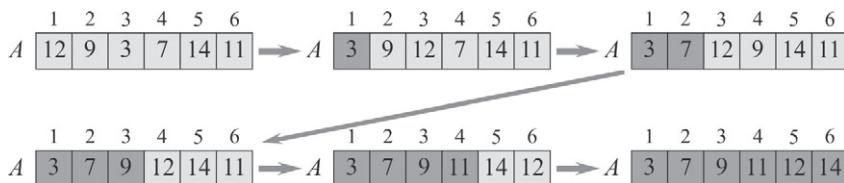
percorra o resto do subvetor, atualizando o índice do menor elemento toda vez que encontrar um elemento menor que o menor atual. Aqui está o procedimento refinado:

Procedimento SELECTION-sort (A, n)

Entradas e Resultado: Os mesmos de antes.

1. Para $i = 1$ a $n - 1$:
 - a. Iguale *menor* a i .
 - b. Para $j = i + 1$ a n :
 - Se $A[j] < A[\text{menor}]$, então iguale *menor* a j .
 - c. Troque $A[i]$ por $A[\text{menor}]$.
-

Esse procedimento tem laços “aninhados”, sendo que o laço da etapa 1B está aninhado no interior do laço da etapa 1. O laço interno executa todas as suas iterações para cada iteração individual do laço externo. Observe que o valor inicial de j no laço interno depende do valor corrente de i no laço externo. Essa ilustração mostra como a ordenação por seleção funciona em um vetor de seis elementos:



O vetor inicial aparece na esquerda superior, e cada etapa mostra o vetor após uma iteração do laço externo. Os elementos em cinza mais escuro contêm o subvetor que sabemos estar ordenado.

Se você quisesse usar uma invariante de laço para questionar se o procedimento SELECTION-SORT ordena o vetor corretamente, precisaria de um para cada um dos laços. Esse procedimento é tão simples que não precisaremos percorrer todos os argumentos da invariante do laço, mas damos aqui as invariantes do laço:

No início de cada iteração do laço da etapa 1, o subvetor $A[1..i - 1]$ contém os $i - 1$ menores elementos de todo o vetor A , e eles estão ordenados em ordem.

No início de cada iteração do laço da etapa 1B, $A[\text{menor}]$ é o menor elemento no subvetor $A[i..j - 1]$.

Qual é o tempo de execução de SELECTION-SORT? Mostraremos que é $\Theta(n^2)$. A chave é analisar quantas iterações o laço interno executa, observando que cada iteração leva o tempo $\Theta(1)$. (Aqui, os fatores constantes nos limites inferior e superior na notação Θ podem ser diferentes porque a atribuição a *menor* pode ou não ocorrer em uma iteração dada.) Vamos contar o número de iterações, com base no valor da variável de laço i no laço externo. Quando i é igual a 1, o laço interno itera para j de 2 a n ou $n - 1$ vezes. Quando i é igual a 2, o laço interno itera para j de 3 a n ou $n - 2$ vezes. Cada vez que o laço externo incrementa i , o laço interno executa uma vez menos. Em geral, o laço interno executa $n - i$ vezes. Na última iteração do laço externo, quando

i é igual a $n - 1$, o laço interno itera durante somente uma vez. Portanto, o número total de iterações do laço interno é

$$(n - 1) + (n - 2) + (n - 3) + \cdots + 2 + 1.$$

Essa soma é conhecida como **progressão aritmética**, e aqui está um fato básico sobre progressões aritméticas: para qualquer inteiro não negativo k ,

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

Substituindo $n - 1$ por k , vemos que o número total de iterações do laço interno é $(n - 1)n/2$ ou $(n^2 - n)/2$. Vamos usar notação assintótica para nos livrarmos do termo de baixa ordem ($-n$) e do fator constante ($1/2$). Então podemos dizer que o número total de iterações do laço interno é $\Theta(n^2)$. Portanto, o tempo de execução de SELECTION-SORT é $\Theta(n^2)$. Observe que esse tempo de execução é uma declaração abrangente que abarca todos os casos. Independentemente dos valores reais dos elementos, o laço interno executa $\Theta(n^2)$ vezes.

Damos aqui outro modo de ver que o tempo de execução é $\Theta(n^2)$, sem usar as séries aritméticas. Mostraremos separadamente que o tempo de execução é tanto $O(n^2)$ quanto $\Omega(n^2)$; juntando os limites assintóticos superior e inferior, temos $\Theta(n^2)$. Para ver que o tempo de execução é $O(n^2)$, observe que cada iteração do laço externo executa o laço interno no máximo $n - 1$ vezes, o que é $O(n)$ porque cada iteração do laço interno leva uma quantidade de tempo constante. Visto que o laço externo itera $n - 1$ vezes, o que também é $O(n)$, o tempo total gasto no laço interno é $O(n)$ vezes $O(n)$, ou $O(n^2)$. Para ver que o tempo de execução é $\Omega(n^2)$, observe que, em cada uma das primeiras $n/2$ iterações do laço externo, executamos o laço interno no mínimo $n/2$ vezes, para um total de no mínimo $n/2$ vezes $n/2$, ou $n^2/4$ vezes. Visto que cada iteração do laço interno leva uma quantidade de tempo constante, vemos que o tempo de execução é, no mínimo, uma constante vezes $n^2/4$, ou $\Omega(n^2)$.

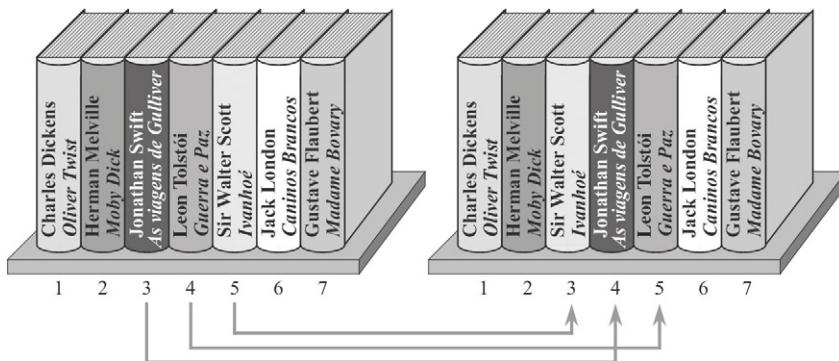
Duas observações finais sobre ordenação por seleção. A primeira é que veremos que seu tempo de execução assintótico de $\Theta(n^2)$ é o pior dos algoritmos de ordenação que examinaremos. A segunda é que, se você examinar cuidadosamente como a ordenação por seleção funciona, verá que o tempo de execução $\Theta(n^2)$ vem das comparações na etapa 1Bi. Mas o número de vezes que ele *move* elementos do vetor é somente $\Theta(n)$ porque a etapa 1C executa somente $n - 1$ vezes. Se mover elementos do vetor for uma operação particularmente consumidora de tempo — talvez porque eles são grandes ou estão armazenados em um dispositivo lento, como um disco — a ordenação por seleção pode ser um algoritmo razoável para usar.

ORDENAÇÃO POR INSERÇÃO

A ordenação por inserção é um pouco diferente da ordenação por seleção, embora tenha aparência semelhante. Na ordenação por seleção, quando decidimos qual livro colocar no i -ésimo espaço, os livros nos primeiros i espaços eram os primeiros i livros *do total*, ordenados em ordem alfabética por nome de autor. Em ordenação por inserção,

os livros nos primeiros i espaços serão os mesmos livros *que estavam originalmente nos primeiros i espaços*, porém agora ordenados por nome de autor.

Por exemplo, vamos supor que os livros nos quatro primeiros espaços já estejam ordenados por nome de autor e que, em ordem, eles são os livros de Charles Dickens, Herman Melville, Jonathan Swift e Leon Tolstói. Vamos dizer que o livro que começa no espaço 5 seja de Sir Walter Scott. Com a ordenação por inserção, deslocamos os livros de Swift e Tolstói um espaço para a direita, transportando-os dos espaços 3 e 4 para os espaços 4 e 5, e depois colocamos o livro de Scott no espaço vago 3. No momento em que trabalhamos com o livro de Scott, não nos importamos com quais livros estão à sua direita (os livros de Jack London e Gustave Flaubert na figura a seguir); trataremos deles mais adiante.



Para deslocar os livros de Swift e Tolstói, em primeiro lugar compararmos o nome de autor Tolstói com Scott. Constatando que Tolstói vem depois de Scott, deslocamos o livro de Tolstói um espaço para a direita, do espaço 4 para o espaço 5. Então compararmos o nome de autor Swift com Scott. Constatando que Swift vem depois de Scott, deslocamos o livro de Swift um espaço para a direita, do espaço 3 para o espaço 4, que ficou vago quando deslocamos o livro de Tolstói. Em seguida compararmos o nome de autor Herman Melville com Scott. Dessa vez, constatamos que Melville *não* vem depois de Scott. Nesse ponto, paramos de comparar nomes de autores porque constatamos que o livro de Scott deve estar à direita do livro de Melville e à esquerda do livro de Swift. Podemos colocar o livro de Scott no espaço 3, que ficou vago quando deslocamos o livro de Swift.

Para traduzir essa ideia para a ordenação de um vetor por ordenação por inserção, o subvetor $A[1..i - 1]$ conterá somente os elementos que estavam originalmente nas primeiras $i - 1$ posições do vetor, ordenados em ordem alfabética. Para determinar para onde vai o elemento que originalmente estava em $A[i]$, a ordenação por inserção percorre $A[1..i - 1]$, começando em $A[i - 1]$ e indo para a esquerda, deslocando cada elemento maior que ele uma posição para a direita. Tão logo encontramos um elemento que não é maior que $A[i]$ ou chegamos à extremidade esquerda do vetor, colocamos o elemento que estava originalmente em $A[i]$ em sua nova posição no vetor.

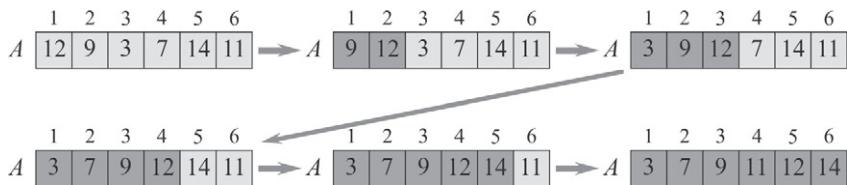
Procedimento INSERTION-sort (A, n)

Entradas e Resultado: Os mesmos de SELECTION-SORT.

1. Para $i = 2$ a n :
 - a. Iguale *chave* a $A[i]$ e j a $i - 1$.
 - b. Enquanto $j > 0$ e $A[j] > \text{chave}$, faça o seguinte:
 - Iguale $A[j + 1]$ a $A[j]$.
 - Decrementa j (isto é, iguale j a $j - 1$).
 - c. Iguale $A[j + 1]$ a *chave*.

O teste na etapa 1B depende de o operador “e” ter um **curto-circuito**: se a expressão à esquerda, $j > 0$, é falsa, ele não avalia a expressão à direita, $A[j] > \text{chave}$. Se ele realmente tentasse acessar $A[j]$ quando $j \leq 0$, ocorreria um erro de indexação de vetor.

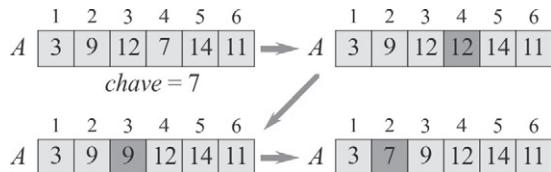
Eis como a ordenação por inserção funciona no mesmo vetor que vimos na página 28 para ordenação por seleção:



Mais uma vez, o vetor inicial aparece na esquerda superior e cada etapa mostra o vetor depois de uma iteração do laço externo da etapa 1. Os elementos em cinza mais escuro contêm o subvetor que sabe-se que está ordenado. A invariante de laço para o laço externo (novamente não provaremos isso) é a seguinte:

No início de cada iteração do laço da etapa 1, o subvetor $A[1..i - 1]$ consiste nos elementos originalmente em $A[1..i - 1]$, mas ordenados em ordem alfabética.

A próxima ilustração demonstra como o laço interno da etapa 1B funciona no exemplo anterior quando i é igual a 4. Consideraremos que o subvetor $A[1..3]$ contém os elementos que originalmente estavam nas três primeiras posições do vetor, porém agora eles estão ordenados. Para determinar onde colocar o elemento que estava originalmente em $A[4]$, nós o salvamos em uma variável denominada *chave* e deslocamos cada elemento em $A[1..3]$, que é maior que *chave* uma posição para a direita:



As posições em cinza mais escuro mostram para onde os elementos foram transportados. Na última etapa mostrada, o valor de $A[1]$, 3 não é maior que o valor de *chave*, 7, e portanto o laço interno termina. O valor de *chave* cai na posição logo à direita de $A[1]$, como mostra a última etapa. É claro que temos de salvar o valor que

estava originalmente em $A[1]$, na *chave* na etapa 1A, porque a primeira iteração do laço interno sobrescreve $A[1]$.

É também possível que o laço interno termine porque o teste $j > 0$ resulta falso. Essa situação ocorre se *chave* é menor que todos os elementos em $A[1..i - 1]$. Quando j torna-se 0, cada elemento em $A[1..i - 1]$ foi deslocado para a direita e, portanto, 1C coloca *chave* em $A[1]$, exatamente onde a queremos.

Quando analisamos o tempo de execução de INSERTION-SORT, vemos que ele é um pouco mais complicado que SELECTION-SORT. O número de vezes que o laço interno itera no procedimento SELECTION-SORT depende somente do índice i do laço e, absolutamente, não de todos os elementos em si. Todavia, no caso do procedimento INSERTION-SORT, o número de vezes que o laço interno itera depende de ambos, do índice i do laço externo e dos valores nos elementos do vetor.

O melhor caso de INSERTION-SORT ocorre quando o laço interno faz zero iteração toda vez. Para isso acontecer, o teste $A[j] > chave$ deve resultar falso na primeira vez para cada valor de i . Em outras palavras, devemos ter $A[i - 1] \leq A[i]$ toda vez que a etapa 1B executa. Como essa situação pode ocorrer? Somente se o vetor A já estiver ordenado quando o procedimento começa. Nesse caso, o laço externo itera $n - 1$ vezes, e cada iteração do laço externo leva uma quantidade de tempo constante, de modo que INSERTION-SORT leva somente tempo $\Theta(n)$.

O pior caso ocorre quando o laço interno faz o número máximo possível de iterações toda vez. Agora o teste $A[j] > chave$ deve sempre resultar verdadeiro, e o laço deve terminar porque o teste $j > 0$ resulta falso. Cada elemento $A[1]$ deve percorrer todo o caminho até a esquerda do vetor. Como essa situação pode acontecer? Somente se o vetor A começar na ordem inversa, isto é, ordenado em ordem *decrescente*. Nesse caso, para toda vez que o laço externo itera, o laço interno itera $i - 1$ vezes. Visto que o laço externo executa com i crescendo de 2 até n , o número de iterações do laço interno forma uma progressão aritmética:

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1),$$

que, como vimos para ordenação por seleção, é $\Theta(n^2)$. Visto que cada iteração do laço interno leva tempo constante, o tempo de execução do pior caso de ordenação por inserção é $\Theta(n^2)$. Portanto, no pior caso, a ordenação por seleção e a ordenação por inserção têm tempos de execução que são assintoticamente os mesmos.

Teria sentido tentar entender o que acontece na média com a ordenação por inserção? Depende de como uma entrada “média” aparece. Se a ordenação de elementos no vetor de entrada for verdadeiramente aleatória, esperaremos que cada elemento seja maior que aproximadamente a metade dos elementos que o precedem e menor que aproximadamente a metade dos elementos que o precedem, de modo que cada vez que o laço interno executar, ele fará aproximadamente $(i - 1)/2$ iterações. Isso reduzirá o tempo de execução à metade, em comparação com o pior caso. Porém, $1/2$ é apenas um fator constante e, portanto, assintoticamente não seria diferente do tempo de execução do pior caso: ainda $\Theta(n^2)$.

A ordenação por inserção é uma excelente escolha quando o vetor está “quase ordenado” desde o início. Suponha que cada elemento do vetor comece dentro de k

posições de onde acabará no vetor ordenado. Então, o número total de vezes que um dado elemento é deslocado, considerando-se todas as iterações do laço interno, é no máximo k . Portanto, o número total de vezes que todos os elementos são deslocados, considerando-se todas as iterações do laço interno, é no máximo kn , o que por sua vez nos diz que o número total de iterações do laço interno é no máximo kn (visto que cada iteração do laço interno desloca exatamente um elemento de uma posição). Se k é uma constante, então o tempo total de execução de ordenação por inserção será somente $\Theta(n)$, porque a notação Θ integra o fator constante k . Na verdade, podemos até mesmo tolerar alguns elementos que se movem uma longa distância no vetor, desde que não haja um número demasiadamente grande de tais elementos. Em particular, se l elementos puderem mover-se para qualquer lugar no vetor (de modo que cada um desses elementos possa mover-se até $n - 1$ posições) e os remanescentes $n - l$ elementos puderem mover-se no máximo k posições, o número total de deslocamentos será, no máximo, $l(n - 1) + (n - l)k = (k + l)n - (k + 1)l$, que é $\Theta(n)$ se k e l forem constantes.

Se compararmos os tempos de execução assintóticos de ordenação por inserção e ordenação por seleção veremos que, no pior caso, eles são os mesmos. A ordenação por inserção é melhor se o vetor estiver quase ordenado. Todavia, a ordenação por seleção tem uma vantagem em relação à ordenação por inserção: a ordenação por seleção move elementos $\Theta(n)$ vezes, não importa o que aconteça, mas a ordenação por inserção poderia mover elementos até $\Theta(n^2)$ vezes, visto que cada execução da etapa 1Bi de INSERTION-SORT move um elemento. Como observamos na página 27 para ordenação por seleção, se mover um elemento for uma operação que consome bastante tempo e você não tiver nenhuma razão para esperar que as entradas para ordenação por inserção aproximam-se da situação do melhor caso, será melhor executar a ordenação por seleção, em vez da ordenação por inserção.

ORDENAÇÃO POR INTERCALAÇÃO

Nosso próximo algoritmo de ordenação, a ordenação por intercalação, tem tempo de execução de apenas $\Theta(n \lg n)$ em todos os casos. Quando comparamos seu tempo de execução com os tempos de execução de ordenação por seleção e ordenação por inserção do pior caso, $\Theta(n^2)$, estamos trocando um fator de n por um fator de apenas $\lg n$. Como observamos na página 6, no Capítulo 1, essa é uma troca que você deve fazer sem pensar duas vezes.

A ordenação por intercalação tem um par de desvantagens em comparação com os outros dois algoritmos de ordenação que vimos. A primeira é que o fator constante que ocultamos na notação assintótica é mais alto que para os outros dois algoritmos. É claro que tão logo o tamanho n do vetor fique suficientemente grande, na realidade isso não importa. A segunda é que a ordenação por intercalação não funciona *no lugar*: ela tem de fazer cópias completas do vetor de entrada. Compare esse aspecto com ordenação por seleção e ordenação por inserção, que a qualquer tempo mantêm uma cópia extra de apenas uma entrada de vetor, e não cópias de todas as entradas de vetor. Se o espaço tiver alto preço, sem dúvida você não optará pela ordenação por intercalação.

Empregamos um paradigma algorítmico comum conhecido como *dividir e conquistar* na ordenação por intercalação. Em dividir e conquistar, desmembramos o problema

em subproblemas semelhantes ao problema original, resolvemos os subproblemas recursivamente e então combinamos as soluções para os subproblemas para resolver o problema original. Lembre-se de que dissemos no Capítulo 2 que, para a recursão funcionar, cada chamada recursiva deve ser para uma instância menor do mesmo problema que a certa altura chegará ao caso-base. Damos aqui um esboço geral do algoritmo dividir e conquistar:

- 1.** *Divida* o problema em vários subproblemas que são instâncias menores do mesmo problema.
- 2.** *Conquiste* os subproblemas resolvendo-os recursivamente. Se eles não forem suficientemente pequenos, resolva os subproblemas como casos-bases.
- 3.** *Combine* as soluções para os subproblemas na solução para o problema original. Quando ordenamos os livros em nossa prateleira com ordenação por intercalação, cada subproblema consiste em ordenar os livros em espaços consecutivos na prateleira. Inicialmente, queremos ordenar todos os n livros, nos espaços 1 até n , mas em um subproblema geral o que queremos é ordenar todos os livros em espaços p até r . Eis como aplicamos o método dividir e conquistar:
 - 1.** *Divida* determinando o número q do espaço a meio caminho entre p e r . Fazemos isso do mesmo modo que determinamos o ponto do meio em busca binária: somamos p e q , dividimos por 2 e tomamos o piso.
 - 2.** *Conquiste* ordenando recursivamente os livros em cada um dos dois subproblemas criados pela etapa de dividir: ordene recursivamente os livros que estão nos espaços p até q e ordene recursivamente os livros que estão nos espaços $q + 1$ até r .
 - 3.** *Combine* intercalando os livros ordenados que estão nos espaços p até q e espaços $q + 1$ até r , de modo que todos os livros nos espaços p até r sejam ordenados. Veremos como intercalar livros mais adiante.

O caso-base ocorre quando menos que dois livros precisam ser ordenados (isto é, quando $p \geq r$, visto que um conjunto de livros sem nenhum livro ou com apenas um livro já está trivialmente ordenado).

Para converter essa ideia em ordenação de um vetor, os livros nos espaços p até r correspondem ao subvetor $A[p..r]$. Damos a seguir o procedimento de ordenação por intercalação, que chama um procedimento $\text{MERGE}(A, p, q, r)$ para intercalar os subvetores ordenados $A[p..q]$ e $A[q + 1 .. r]$ no único subvetor ordenado $A[p..r]$.

Procedimento MERGE-SORT (A, p, r)

Entradas:

- A : um vetor.
- p, r : índices iniciais e finais de um subvetor de A .

Resultado: Os elementos do subvetor $A[p..r]$ estão ordenados em ordem crescente.

1. Se $p \geq r$, o subvetor $A[p..r]$ tem, no máximo, um elemento e, portanto, já está ordenado. Apenas retorne sem fazer nada.
2. Caso contrário, faça o seguinte:
 - a. Iguale q a $(p + r) / 2$.
 - b. Chame recursivamente $\text{MERGE-SORT}(A, p, q)$.
 - c. Chame recursivamente $\text{MERGE-SORT}(A, q + 1, r)$.
 - d. Chame $\text{MERGE}(A, p, q, r)$.

Embora ainda tenhamos de ver como o procedimento MERGE funciona, podemos examinar um exemplo de como o procedimento MERGE-SORT funciona. Vamos começar com este vetor:

1	2	3	4	5	6	7	8	9	10
12	9	3	7	14	11	6	2	10	5

A chamada inicial é MERGE-SORT($A, 1, 10$). A etapa 2A computa q como 5, de modo que as chamadas recursivas nas etapas 2B e 2C são MERGE-SORT($A, 1, 5$) e MERGE-SORT($A, 6, 10$).

1	2	3	4	5	6	7	8	9	10
12	9	3	7	14	11	6	2	10	5

Depois que as duas chamadas recursivas retornam, esses dois subvetores estão ordenados:

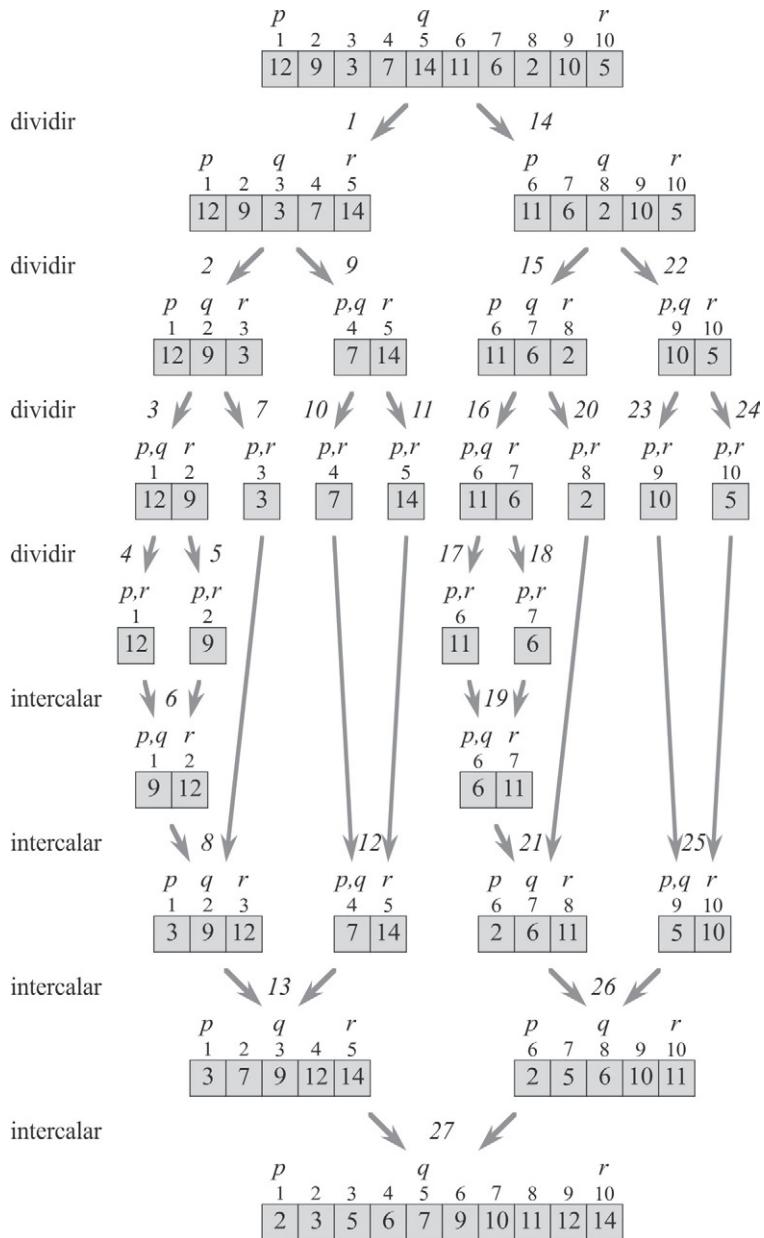
1	2	3	4	5	6	7	8	9	10
3	7	9	12	14	2	5	6	10	11

Finalmente, a chamada MERGE($A, 1, 5, 10$) na etapa 2D intercala os dois subvetores ordenados em um único subvetor ordenado, que é o vetor inteiro nesse caso:

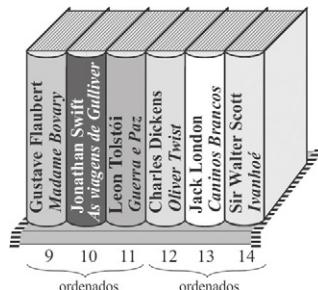
1	2	3	4	5	6	7	8	9	10
2	3	5	6	7	9	10	11	12	14

Se desdobrarmos a recursão, obteremos a figura apresentada na próxima página. Setas divergentes indicam etapas de divisão e setas convergentes indicam etapas de intercalação. As variáveis p , q e r que aparecem acima de cada subvetor estão localizadas nos índices aos quais elas correspondem em cada chamada recursiva. Os números em *italico* dão a ordem na qual as chamadas de procedimento ocorrem depois da chamada inicial MERGE-SORT ($A, 1, 10$). Por exemplo, a chamada MERGE ($A, 1, 3, 5$) é a décima terceira chamada de procedimento após a chamada inicial, e a chamada MERGE-SORT ($A, 6, 7$) é a décima sexta chamada.

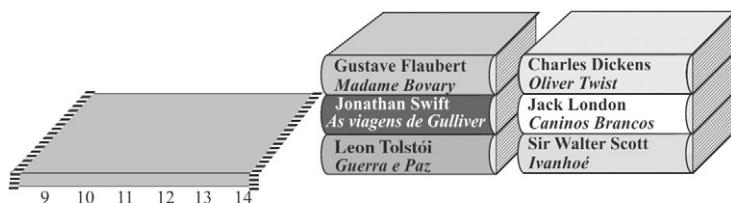
O trabalho para valer acontece no procedimento MERGE. Portanto, o procedimento MERGE não somente tem de funcionar corretamente, mas também tem de ser rápido. Se estivermos intercalando um total de n elementos, o melhor que podemos esperar é o tempo $\Theta(n)$, visto que cada elemento tem de ser intercalado em seu lugar adequado e, na verdade, podemos conseguir intercalação em tempo linear.



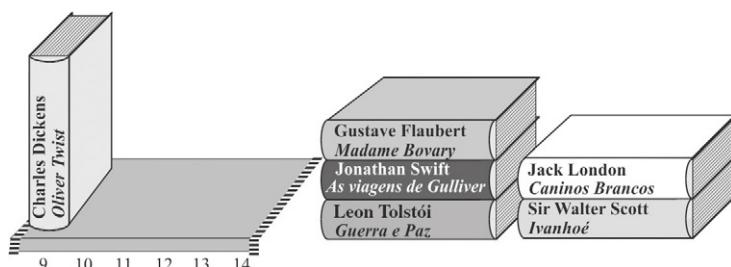
Voltando ao nosso exemplo do livro, vamos examinar apenas a porção da prateleira do espaço 9 até o espaço 14. Suponha que já ordenamos os livros nos espaços 9–11 e nos espaços 12–14:



Retiramos os livros nos espaços 9–11 e os colocamos em uma pilha na qual o livro cujo autor é o primeiro em ordem alfabética está no topo e fazemos o mesmo com os livros nos espaços 12–14, em uma pilha separada:

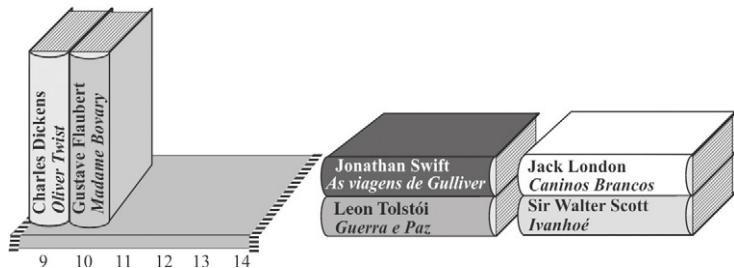


Como as duas pilhas já estão ordenadas, o livro que deve voltar para o espaço 9 será um dos que estão no topo de sua pilha: o livro de Gustave Flaubert ou o livro de Charles Dickens. Na verdade, vemos que o livro de Dickens vem antes do livro de Flaubert, então o movemos para o espaço 9:

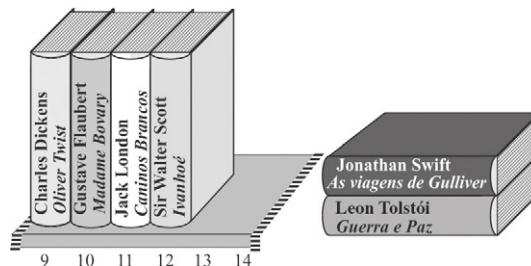


Depois de passarmos o livro de Dickens para o espaço 9, o livro que deve ir para o espaço 10 deve ser o livro que ainda está no topo da primeira pilha, de Flaubert, ou o

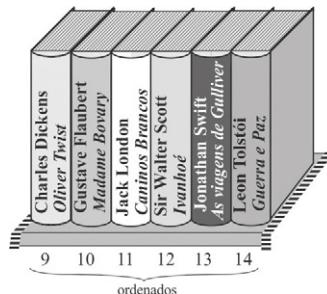
livro que agora está no topo da segunda pilha, de Jack London. Passamos o livro de Flaubert para o espaço 10:



Em seguida, comparamos os livros que agora estão no topo de suas pilhas, que são de Jonathan Swift e London, e passamos o livro de London para o espaço 11. Isso deixa o livro de Sir Walter Scott em cima da pilha da direita e, quando o comparamos com o livro de Swift, passamos o livro de Scott para o espaço 12. Nesse ponto, a pilha da direita está vazia:



Basta passar os livros que estão na pilha da esquerda para os espaços remanescentes, em ordem. Agora todos os livros nos espaços 9–14 estão ordenados:



Quão eficiente é esse procedimento de intercalação? Movemos cada livro exatamente duas vezes: uma vez para retirá-lo da prateleira e colocá-lo em uma pilha, e uma vez para movê-lo do topo de uma pilha e colocá-lo de volta na prateleira. Além

do mais, sempre que estamos decidindo qual livro colocar de volta na prateleira, precisamos comparar apenas dois livros: os que estão no topo de suas pilhas. Portanto, para intercalar n livros, movemos os livros $2n$ vezes e comparamos os pares de livros, no máximo, n vezes.

Por que retirar os livros da prateleira? E se tivéssemos deixado os livros na prateleira e apenas rastreássemos quais livros tínhamos colocado em seus espaços corretos na prateleira e quais não? Isso poderia muito bem dar muito mais trabalho. Por exemplo, suponha que todo livro na metade direita devesse vir antes de todo livro na metade esquerda. Antes que pudéssemos passar o primeiro livro da metade direita para o primeiro espaço da metade esquerda, teríamos de deslocar todo livro que começasse na metade esquerda um espaço para a direita, de modo a criar um espaço. Então teríamos de fazer o mesmo para colocar o próximo livro que começasse na metade direita no segundo espaço da metade esquerda. E o mesmo para todos os outros livros que começassem na metade direita. Teríamos de deslocar metade dos livros — todos os livros que começassem na metade esquerda — cada vez que quiséssemos colocar um livro que começasse na metade direita em seu espaço correto.

Esse argumento explica por que não intercalamos no lugar.³ Voltando ao caso em que intercalamos os subvetores ordenados $A[p..q]$ e $A[q + 1..r]$ no subvetor $A[p..r]$, começamos copiando os elementos a serem intercalados do vetor A para vetores temporários e os intercalamos de volta em A . Seja $n_1 = q - p + 1$ o número de elementos em $A[p..q]$ e $n_2 = r - q$ o número de elementos em $A[q + 1..r]$. Criamos vetores temporários B com n_1 elementos e C com n_2 elementos, e copiamos os elementos em $A[p..q]$, em ordem, em B , e do mesmo modo os elementos $A[q + 1..r]$, em ordem, em C . Agora podemos intercalar esses elementos de volta em $A[p..q]$ sem medo de sobrescrever as únicas cópias que temos deles.

Intercalamos os elementos do vetor do mesmo modo que intercalamos livros. Copiamos elementos dos vetores B e C de volta ao subvetor $A[p..r]$, mantendo índices para rastrear o menor elemento ainda não copiado de volta para B e C , e copiando de volta o menor dos dois. Em tempo constante, podemos determinar qual elemento é o menor, copiá-lo de volta à posição correta de $A[p..r]$ e atualizar os índices nos vetores.

A certa altura, todos os elementos de um dos dois vetores terão sido copiados de volta a $A[p..r]$. Esse momento corresponde ao momento em que resta somente uma pilha de livros. Porém usamos um truque para evitar ter de verificar cada vez se um dos vetores foi esvaziado: colocamos na extremidade direita de cada um dos vetores B e C um elemento extra que é maior que qualquer outro elemento. Lembra-se do truque da sentinelas que usamos em SENTINEL-LINEAR-SEARCH no Capítulo 2? Essa ideia é semelhante. Aqui, usamos ∞ (infinito) como a chave de ordenação da sentinelas, de modo que sempre que um elemento com uma chave de ordenação ∞ for o menor elemento restante em seu vetor, é garantido que ele “perderá” a disputa para

³ Na verdade, é possível intercalar no lugar em tempo linear, mas o procedimento para tal é bem complicado.

ver qual vetor tem o menor elemento restante.⁴ Uma vez que todos os elementos dos vetores B e C tenham sido copiados de volta, as sentinelas de ambos os vetores serão seus menores elementos remanescentes. Mas não há nenhuma necessidade de comparar as sentinelas nesse ponto porque a essa altura já teremos copiado todos os elementos “reais” (os que não são sentinelas) de volta a $A[p..r]$. Visto que sabemos de antemão que teremos de copiar elementos de volta para $A[p]$ até $A[r]$, podemos parar assim que tivermos copiado um elemento de volta a $A[r]$. Podemos apenas executar um laço com um índice em A que executa de p a r .

Damos a seguir o procedimento MERGE. Parece longo, mas apenas segue o método que acabamos de descrever.

Procedimento MERGE (A, p, q, r)

Entradas:

- A : um vetor.
- p, q, r : índices para A . Considera-se que cada um dos subvetores $A[p..q]$ e $A[q + 1..r]$ já está ordenado.

Resultado: O subvetor $A[p..r]$ contém os elementos originalmente em $A[p..q]$ e $A[q + 1..r]$, mas agora o subvetor $A[p..r]$ inteiro está ordenado.

1. Iguale n_1 a $q - p + 1$ e iguale n_2 a $r - q$.
2. Sejam $B[1..n_1 + 1]$ e $C[1..n_2 + 1]$ novos vetores.
3. Copie $A[p..q]$ para $B[1..n_1]$ e $A[q + 1..r]$ para $C[1..n_2]$.
4. Iguale $B[n_1 + 1]$ e $C[n_2 + 1]$ a ∞ .
5. Iguale i e j a ∞ .
6. Para $k = p$ a r :
 - a. Se $B[i] \leq C[j]$, então iguale $A[k]$ a $B[i]$ e incremente i .
 - b. Caso contrário, $B[i] > C[j]$, iguale $A[k]$ a $C[j]$ e incremente j .

Depois que as etapas 1–4 alocaram os vetores B e C , copiaram $A[p..q]$ para B e $A[q + 1..r]$ para C , e inseriram as sentinelas nesses vetores, cada iteração do laço principal na etapa 6 copia de volta o menor elemento remanescente para a próxima posição em $A[p..r]$, terminando assim que tenha copiado de volta todos os elementos em B e C . Nesse laço, i indexa o menor elemento remanescente em B , j indexa o menor elemento remanescente em C e k indexa o local em A para onde o elemento será copiado de volta.

Se estivermos intercalando n elementos no total (de modo que $n = n_1 + n_2$), copiar os elementos para os vetores B e C leva tempo $\Theta(n)$ e copiá-los de volta para $A[p..r]$ leva tempo constante por elemento, o que dá um tempo total de intercalação de apenas $\Theta(n)$.

Afirmamos antes que o algoritmo de ordenação por intercalação inteiro leva tempo $\Theta(n \lg n)$. Adotaremos a premissa simplificadora de que o tamanho do vetor n é uma potência de 2; portanto, toda vez que dividirmos o vetor, os tamanhos dos subvetores serão iguais. (Em geral, n poderia não ser uma potência de 2 e, portanto, os tamanhos dos subvetores poderiam não ser iguais em uma chamada recursiva dada. Uma análise rigorosa pode dar conta dessa tecnicidade, mas não vamos nos preocupar com ela.)

⁴ Na prática, representamos ∞ por um valor que, por comparação, é muito maior que qualquer chave de ordenação. Por exemplo, se as chaves de ordenação forem nomes de autores, ∞ poderia ser ZZZZ — supondo, é claro, que nenhum autor real tenha tal nome.

Eis como analisamos a ordenação por intercalação. Vamos dizer que ordenar um subvetor de n elementos leva tempo $T(n)$, que é uma função que aumenta com n (visto que, presumivelmente, leva mais tempo para ordenar mais elementos). O tempo $T(n)$ vem das três componentes do paradigma dividir e conquistar, cujos tempos somamos:

1. Dividir leva tempo constante porque equivale apenas a computar o índice q .
2. Conquistar consiste nas duas chamadas recursivas em subvetores, cada uma com $n/2$ elementos. Pela definição que demos para o tempo que leva para ordenar um subvetor, cada uma das duas chamadas recursivas leva tempo $T(n/2)$.
3. Combinar os resultados das duas chamadas recursivas intercalando os subvetores ordenados leva tempo $\Theta(n)$.

Como o tempo constante para dividir é um termo de baixa ordem em comparação com o tempo $\Theta(n)$ para combinar, podemos absorver o tempo para dividir no tempo para combinar e dizer que dividir e combinar, juntas, leva tempo $\Theta(n)$. A etapa conquistar custa $T(n/2) + T(n/2)$, ou $2T(n/2)$. Agora podemos escrever uma equação para $T(n)$:

$$T(n) = 2T(n/2) + f(n),$$

onde $f(n)$ representa o tempo para dividir e combinar que, como acabamos de observar, é $\Theta(n)$. Uma prática comum no estudo de algoritmos é apenas colocar a notação assintótica na equação e permitir que ela fique no lugar de alguma função à qual não vale a pena dar um nome e, portanto, reescrevemos essa equação como

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

Mas, espere! Parece haver algo errado aqui. Definimos a função T que descreve o tempo de execução de ordenação por intercalação em termos dessa mesma função! Denominamos tal equação *equação de recorrência* ou apenas *recorrência*. O problema é que queremos expressar $T(n)$ de maneira não recursiva, isto é, não em termos dela mesma. Pode ser uma grande dor de cabeça converter uma função expressa como recorrência para uma forma não recursiva. Porém, para ampla classe de equações de recorrência podemos aplicar um método prático conhecido como “método mestre”. O método mestre aplica-se a muitas recorrências (mas não a todas) da forma $T(n) = aT(n/b) + f(n)$, onde a e b são constantes inteiras positivas. Felizmente, ela se aplica à nossa recorrência intercalar-ordenar e dá o resultado de que $T(n)$ é $\Theta(n \lg n)$.

Esse tempo de execução $\Theta(n \lg n)$ aplica-se a todos os casos de ordenação por intercalação — melhor caso, pior caso e todos os casos entre esses dois. Cada elemento é copiado $\Theta(n \lg n)$ vezes. Como você pode ver examinando o método MERGE, quando ele é chamado com $p = 1$ e $r = n$, faz cópias de todos os n elementos e, portanto, a ordenação por intercalação definitivamente não executa no lugar.

QUICKSORT

Como a ordenação por intercalação, o quicksort (ordenação rápida) usa o paradigma dividir e conquistar (e, por consequência, usa recursão). Todavia, o quicksort usa dividir e conquistar de um modo ligeiramente diferente que a ordenação por

intercalação. Ele tem algumas outras diferenças significativas em relação à ordenação por intercalação:

- O quicksort funciona no lugar.
- O tempo de execução assintótico do quicksort é diferente para o pior caso e para o caso médio. Em particular, o tempo de execução do pior caso do quicksort é $\Theta(n^2)$, mas seu tempo de execução para o caso médio é melhor: $\Theta(n \lg n)$.

O quicksort também tem bons fatores constantes (melhores que os da ordenação por intercalação) e frequentemente é um bom algoritmo de ordenação para usar na prática.

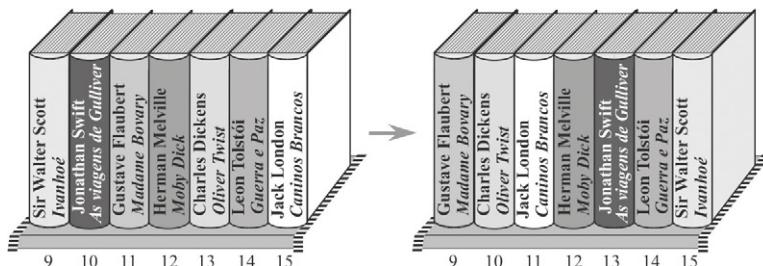
Mostramos a seguir como o quicksort usa o dividir e conquistar. Novamente vamos pensar em ordenação de livros em uma prateleira. Como ocorreu com a ordenação por intercalação, inicialmente queremos ordenar todos os n livros em espaços de 1 até n , e consideraremos o problema geral de ordenar livros em espaços p até r .

1. Divida

escolhendo em primeiro lugar qualquer livro que esteja nos espaços p até r .

Chame esse livro de **pivô**. Rearrange os livros na prateleira de modo que todos os outros livros cujos nomes de autor vêm antes do nome do autor do pivô ou foram escritos pelo mesmo autor fiquem à esquerda do pivô, e todos os livros cujos nomes de autor vêm depois do nome do autor do pivô fiquem à direita do pivô.

Nesse exemplo, escolhemos o livro que está na extrema direita, de Jack London, como o pivô quando rearranjamos os livros nos espaços 9 até 15:



Depois de rearranjar — o que denominamos **particionar** no quicksort —, os livros de Flaubert e Dickens, que vêm antes de London em ordem alfabética, estarão à esquerda do livro de London, e todos os outros livros, de autores que vêm depois de London em ordem alfabética, estarão à direita. Observe que, depois de particionar, os livros à esquerda do livro de London não estarão em nenhuma ordem particular, e o mesmo vale para os livros à direita.

2. Conquiste

ordenando recursivamente os livros à esquerda do pivô e à direita do pivô. Isto é, se a etapa dividir mover o pivô para o espaço q (espaço 11 em nosso exemplo), ordene recursivamente os livros nos espaços p até $q - 1$ e ordene recursivamente os livros nos espaços $q + 1$ até r .

3. Combine, fazendo nada!

Depois que a etapa conquistar ordenar recursivamente, terminamos. Por quê? Todos os livros à esquerda do pivô (nos espaços p até $q - 1$) vêm antes do pivô ou têm o mesmo autor que o pivô e estão ordenados, e todos os livros à direita do pivô (nos espaços $q + 1$ até r) vêm depois do pivô e estão ordenados. Os livros nos espaços p até r só podem estar ordenados!

Se você trocar a prateleira pelo vetor e os livros pelos elementos do vetores, terá a estratégia para o quicksort. Como aconteceu na ordenação por intercalação, o caso-base ocorre quando o subvetor a ser ordenado tem menos de dois elementos.

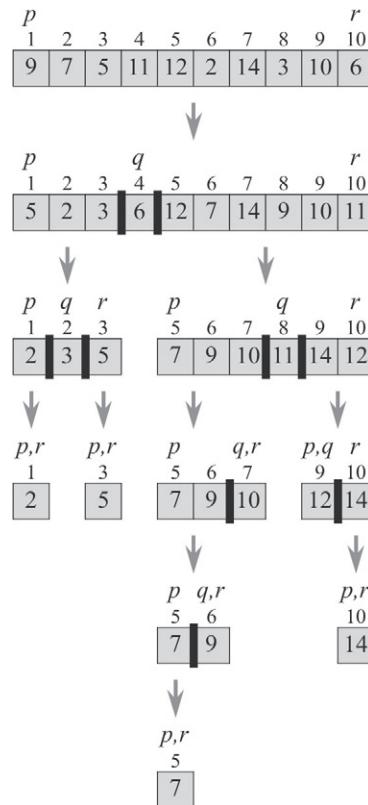
O procedimento para o quicksort pressupõe que podemos chamar um procedimento PARTITION (A, p, r) que particiona o subvetor $A[p..r]$ retornando o índice q onde ele substituiu o pivô.

Procedimento QUICKSORT (A, p, r)

Entradas e Resultado: Os mesmos de MERGE-SORT.

1. Se $p \geq r$, então apenas retorne sem fazer nada.
2. Caso contrário, faça o seguinte:
 - a. Chame PARTITION (A, p, r) e iguale q ao seu resultado.
 - b. Chame recursivamente QUICKSORT ($A, p, q - 1$).
 - c. Chame recursivamente QUICKSORT ($A, q + 1, r$).

A chamada inicial é $\text{QUICKSORT}(A, 1, n)$, semelhante ao procedimento MERGE-SORT. Damos aqui um exemplo de como a recursão se desenvolve, mostrando os índices p , q e r para cada subvetor no qual $p \leq r$:



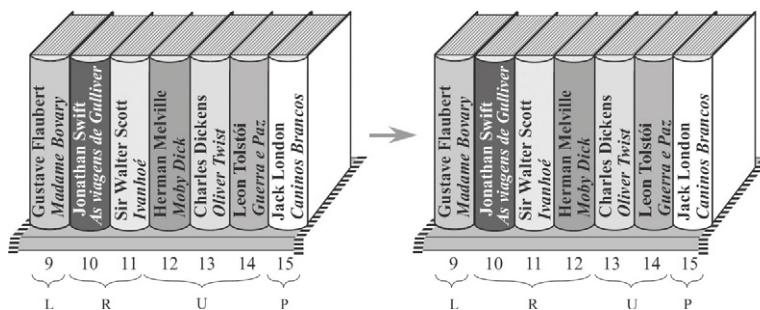
O valor que está mais embaixo em cada posição do vetor dá o elemento final armazenado ali. Quando você lê o vetor da esquerda para a direita, examinando o valor que está mais embaixo em cada posição, vê que esse vetor está de fato ordenado.

A chave do quicksort é o particionamento. Exatamente como conseguimos intercalar n elementos em tempo $\Theta(n)$, podemos particionar n elementos em tempo $\Theta(n)$. Mostramos aqui como particionamos os livros que estão nos espaços p até r na prateleira. Escolhemos o livro da extrema direita do conjunto — o livro no espaço r — como o pivô. A qualquer tempo, cada livro estará exatamente em um de quatro grupos, e esses grupos estarão em espaços p até r , da esquerda para a direita:

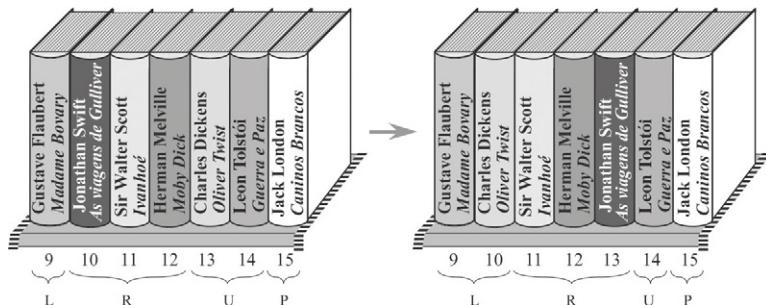
- grupo L (grupo da esquerda): livros cujos autores sabemos que vêm antes do autor do pivô em ordem alfabética ou que foram escritos pelo autor do pivô, seguidos de
- grupo R (grupo da direita): livros cujos autores sabemos que vêm depois do autor do pivô em ordem alfabética, seguidos de
- grupo U (grupo desconhecido): livros que ainda não examinamos e que, portanto, não sabemos como os seus autores se comparam com o autor do pivô, seguidos de
- grupo P (pivô): apenas um livro, o pivô

Percorremos os livros no grupo U, da esquerda para a direita, comparando cada um com o pivô e passando-o para o grupo L ou para o grupo R, parando quando chegamos ao pivô. O livro que comparamos com o pivô é sempre o livro que está na extrema esquerda no grupo U.

Se o autor do livro vier depois do autor do pivô, o livro torna-se o livro na extrema direita no grupo R. Visto que o livro era o da extrema esquerda no grupo U, e o grupo U vem imediatamente após o grupo R, temos apenas que deslocar a linha divisória entre os grupos R e U um espaço para a direita, sem mover nenhum livro:



- Se o autor do livro vier antes do autor do pivô ou for o autor do pivô, faremos desse o livro da extrema direita no grupo L. Trocamos esse livro com o livro na extrema esquerda no grupo R e deslocamos as linhas divisórias entre os grupos L e R, e entre os grupos R e U, um espaço para a direita:



Assim que chegamos ao pivô, nós o trocamos pelo livro da extrema esquerda no grupo R. Em nosso exemplo, acabamos com o vetor de livros mostrado na página 42.

Compararmos cada livro com o pivô uma vez, e cada livro cujo autor vem antes do autor do pivô ou é o autor do pivô provoca uma troca. Portanto, para particionar n livros, fazemos no máximo $n - 1$ comparações (visto que não temos que comparar o pivô com ele mesmo) e no máximo n trocas. Observe que, diferentemente da intercalação, podemos particionar os livros sem removê-los todos da prateleira. Isto é, podemos particionar no lugar.

Para converter partição de livros em partição de um subvetor $A[p..r]$, em primeiro lugar escolhemos $A[r]$ (o elemento da extrema direita) como o pivô. Então percorremos o subvetor da esquerda para a direita, comparando cada elemento com o pivô. Mantemos índices q e u no subvetor que dividimos como a seguir:

- O subvetor $A[p..q - 1]$ corresponde ao grupo L: cada elemento é menor ou igual ao pivô.
- O subvetor $A[q..u - 1]$ corresponde ao grupo R: cada elemento é maior que o pivô.
- O subvetor $A[u..r - 1]$ corresponde ao grupo U: ainda não sabemos como eles se comparam com o pivô.
- O elemento $A[r]$ corresponde ao grupo P: ele contém o pivô.

Essas divisões são, de fato, invariantes de laço (mas não provaremos isso aqui).

Em cada etapa, compararmos $A[u]$, o elemento da extrema esquerda no grupo U, com o pivô. Se $A[u]$ for maior que o pivô, incrementamos u ou deslocamos a linha divisória entre os grupos R e U para a direita. Se, em vez disso, $A[u]$ for menor ou igual ao pivô, trocamos os elementos em $A[q]$ (o elemento na extrema esquerda no grupo R) e $A[u]$ e incrementamos q e u para deslocar as linhas divisórias entre os grupos L e R e os grupos R e U para a direita. Damos aqui o procedimento PARTITION:

Procedimento PARTITION (A, p, r)

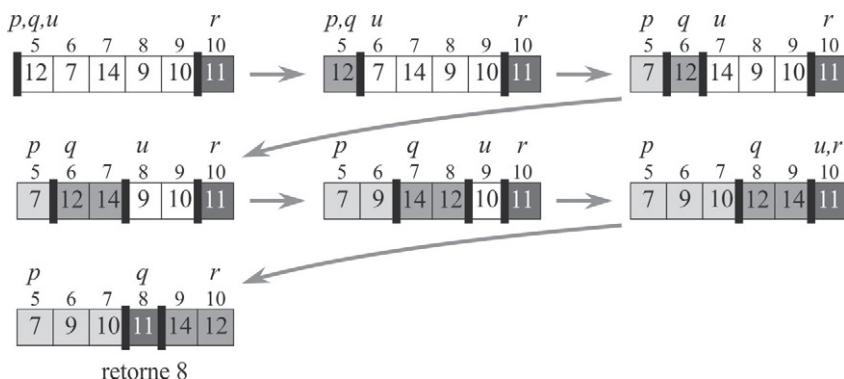
Entradas: As mesmas de MERGE-SORT.

Resultado: Rearranja os elementos de $A[p..r]$ de modo que todo elemento em $A[p..q - 1]$ é menor ou igual a $A[q]$ e todo elemento em $A[q + 1..r]$ é maior que q . Retorna o índice q ao chamador.

1. Iguale q a p .
2. Para $u = p$ a $r - 1$ faça:
 - a. Se $A[u] \leq A[r]$, troque $A[q]$ por $A[u]$ e incremente q .
3. Troque $A[q]$ por $A[r]$ e retorne q .

Iniciando os índices q e u em p , os grupos L ($A[p..q - 1]$) e R ($A[q..u - 1]$) estão inicialmente vazios, e o grupo U ($A[u..r - 1]$) contém todo elemento exceto o pivô. Em algumas instâncias, por exemplo, se $A[p] \leq A[r]$, um elemento poderia ser trocado por ele mesmo, o que resulta em nenhuma mudança no vetor. A etapa 3 termina trocando o elemento pivô com o elemento na extrema esquerda no grupo R e, com isso, move o pivô para o seu lugar correto no vetor particionado e retorna o novo índice do pivô, q .

Eis como o procedimento PARTITION funciona, etapa por etapa, no subvetor $A[5..10]$ criado no primeiro particionamento no exemplo de quicksort na página 44. O grupo U é mostrado em branco, o grupo L em cinza-claro, o grupo R em cinza-médio e o elemento em cinza-escuro é o pivô, grupo P. A primeira parte da figura mostra o vetor inicial e índices, as próximas cinco partes mostram o vetor e índices depois de cada iteração do laço da etapa 2 (incluindo incrementar o índice u ao final de cada iteração), e a última parte mostra o vetor final particionado:



Do mesmo modo que quando particionamos livros, compararmos cada elemento com o pivô uma vez e executarmos, no máximo, uma troca para cada elemento que comparamos com o pivô. Visto que cada comparação leva tempo constante e cada troca leva tempo constante, o tempo total para PARTITION de um subvetor com n elementos é $\Theta(n)$.

Portanto, quanto tempo o procedimento QUICKSORT leva? Como fizemos para a intercalação, vamos dizer que ordenar subvetor de n elementos leva tempo $T(n)$, uma função aumenta com n . Dividir, executada pelo procedimento PARTITION, leva tempo $\Theta(n)$. Mas o tempo do QUICKSORT depende da eventual equitatividade do particionamento.

No pior caso, os tamanhos da partição estão realmente desequilibrados. Se todo elemento exceto o pivô for menor que ele, então PARTITION acaba deixando o pivô em $A[r]$ e retorna o índice r a QUICKSORT, que o guarda na variável q . Nesse caso, a partição $A[q + 1..r]$ é vazia e a partição $A[p..q - 1]$ é apenas um elemento menor que $A[p..r]$. A chamada recursiva ao subvetor vazio leva tempo $\Theta(1)$ (o tempo para fazer a chamada e determinar que o subvetor está vazio na etapa 1). Podemos apenas englobar esse tempo $\Theta(1)$ ao tempo $\Theta(n)$ para particionamento. Mas, se $A[p..r]$ tem n

elementos, $A[p..q - 1]$ tem $n - 1$ elementos e, assim, a chamada recursiva a $A[p..q - 1]$ leva tempo $T(n - 1)$. Obtemos a recorrência

$$T(n) = T(n - 1) + \Theta(n).$$

Não podemos resolver essa recorrência usando o método mestre, mas ela tem a solução $T(n) = \Theta(n^2)$, que não é melhor que a ordenação por seleção! Como podemos obter tal divisão tão desequilibrada? Se todo pivô for maior que todos os outros elementos, então o vetor deve ter começado já ordenado. Além disso, acontece que obtemos uma divisão não equitativa toda vez que o vetor começar em ordem reversa ordenada.

Por outro lado, se obtivermos uma divisão equitativa toda vez, cada um dos subvetores terá no máximo $n/2$ elementos. A recorrência será a mesma recorrência da página 42 para ordenação por intercalação,

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

com a mesma solução: $T(n) = \Theta(n \lg n)$. É claro que teríamos de ter muita sorte ou o vetor de entrada teria de ser arquitetado de modo a obter uma divisão perfeitamente equitativa toda vez.

O caso usual está em algum lugar entre o melhor e o pior caso. A análise técnica é confusa e não vou impingi-la a você. Porém, se os elementos do vetor de entrada vierem em ordem aleatória, na média obteremos divisões próximas o suficiente da equitativa para o QUICKSORT levar tempo $\Theta(n \lg n)$.

Agora vamos ficar paranoicos. Suponha que o seu pior inimigo tenha lhe dado um vetor para ordenar, sabendo que você sempre escolhe o último elemento em cada subvetor como pivô, e tenha disposto o vetor de modo que você sempre obtenha o pior caso de divisão. Como você frustraria o seu inimigo? Você poderia primeiro verificar para ver se o vetor começa ordenado em ordem direta ou em ordem inversa e fazer algo especial nesses casos. Então, novamente, o seu inimigo poderia arquitetar o vetor de modo tal que as divisões sejam sempre ruins, mas não maximamente ruins. Seria interessante você verificar todo caso ruim possível.

Felizmente, há uma solução muito mais simples: não escolha sempre o último elemento como o pivô. O lindo procedimento PARTITION não funcionará porque os grupos não estão onde deveriam estar. Isso também não é um problema: antes de executar o procedimento PARTITION, troque $A[r]$ por um elemento escolhido aleatoriamente em $A[p..r]$. Agora você escolheu o seu pivô aleatoriamente e pode executar o procedimento PARTITION.

Na verdade, com um pouco mais de esforço, você pode melhorar sua chance de obter uma divisão próxima da equitativa. Em vez de escolher um elemento em $A[p..r]$ aleatoriamente, escolha três elementos aleatoriamente e troque a mediana dos três com $A[r]$. Aqui, mediana de três quer dizer o valor que está entre os outros dois (se dois ou mais dos elementos escolhidos aleatoriamente forem iguais, retire um deles arbitrariamente). Novamente, não vou aborrecê-lo com a análise, mas você terá de ser realmente azarado sempre que escolher os elementos aleatórios para que o

QUICKSORT leva mais tempo que $\Theta(n \lg n)$. Além disso, a menos que o seu inimigo tenha acesso ao seu gerador de números aleatórios, ele não terá nenhum controle sobre a eventual equitatividade resultante da sua divisão.

Quantas vezes o QUICKSORT troca elementos? Depende de você contar “trocar” um elemento para a mesma posição que ele começou como uma troca. Você certamente pode verificar para ver se é esse o caso e evitar a troca se realmente for esse o caso. Portanto, vamos denominar uma troca como troca somente quando um elemento realmente se mover no vetor como resultado de uma troca, isto é, quando $q \neq u$ na etapa 2A ou quando $q \neq r$ na etapa 3 de PARTITION. O melhor caso para minimizar trocas é também um dos piores casos para tempo de execução assintótico: quando o vetor já está ordenado. Então, não ocorre nenhuma troca. A maioria das trocas ocorre quando n é par e o vetor de entrada é parecido com $n, n - 2, n - 4, \dots, 4, 2, 1, 3, 5, \dots, n - 3, n - 1$. Então ocorrem $n^2/4$ trocas, e o tempo de execução assintótico ainda é o pior caso $\Theta(n^2)$.

RECAPITULANDO

Neste capítulo e no anterior, vimos quatro algoritmos para buscar e quatro para ordenar. Vamos resumir suas propriedades em duas tabelas. Como os três algoritmos de busca do Capítulo 2 eram apenas variações sobre o mesmo tema, podemos considerar BETTER-LINEAR-SEARCH ou SENTINEL-LINEAR-SEARCH como representantes da busca linear.

Algoritmos de busca

Algoritmo	Tempo de execução do pior caso	Tempo de execução do melhor caso	Requer vetor ordenado?
Busca linear	$\Theta(n)$	$\Theta(1)$	não
Busca binária	$\Theta(\lg n)$	$\Theta(1)$	sim

Algoritmos de ordenação

Algoritmo	Tempo de execução do pior caso	Tempo de execução do melhor caso	Trocas no pior caso	No lugar?
Ordenação por seleção	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	sim
Ordenação por inserção	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	sim
Ordenação por intercalação	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	não
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n^2)$	sim

Essas tabelas não mostram tempos de execução do caso médio porque, com a notável exceção do quicksort, eles são iguais aos tempos de execução do pior caso. Como vimos, o tempo de execução do caso médio do quicksort, supondo que o vetor comece em ordem aleatória, é apenas $\Theta(n \lg n)$.

Como esses algoritmos de ordenação se comparam na prática? Eu os codifiquei em C++ e os executei em vetores de inteiros de 4 bytes em duas máquinas diferentes: o meu MacBook Pro (no qual escrevi este livro), com processador Intel Core 2 Duo de 2,4 GHz e 4 GB de RAM que executa Mac OS 10.6.8, e um Dell PC (meu servidor da Web) com processador Intel Pentium 4 de 3,2 GHz e 1 GB de RAM que executa Linux versão 2.6.22.14. Compilei o código com g++ e nível de otimização -O3. Executei cada algoritmo em vetores de tamanhos que vão até 50.000, sendo que cada vetor estava inicialmente em ordem inversa. Calculei as médias dos tempos de execução para 20 rodadas de cada algoritmo para cada tamanho de vetor.

Começando com cada vetor em ordem inversa, deduzi os tempos de execução assintóticos do pior caso de ambas, ordenação por inserção e quicksort. Portanto, executei duas versões do quicksort: quicksort “regular”, que sempre escolhe o pivô como o último elemento $A[r]$ do subvetor $A[p..r]$ particionado, e o quicksort aleatorizado, que troca um elemento escolhido aleatoriamente em $A[p..r]$ por $A[r]$ antes do particionamento. (Não executei o método da mediana de três.) A versão “regular” do quicksort é também conhecida como **determinística** porque não é aleatorizada; tudo o que ela faz é predeterminado uma vez, dado um vetor de entrada a ordenar.

O quicksort aleatorizado foi o campeão para $n \geq 64$ em ambos os computadores. Dou aqui as razões entre os tempos de execução dos outros algoritmos e os tempos de execução do quicksort aleatorizado para vários tamanhos da entradas.

MacBook Pro

Algoritmo	<i>n</i>						
	50	100	500	1.000	5.000	10.000	50.000
Ordenação por seleção	1,34	2,13	8,04	13,31	59,07	114,24	537,42
Ordenação por inserção	1,08	2,02	6,15	11,35	51,86	100,38	474,29
Ordenação por intercalação	7,58	7,64	6,93	6,87	6,35	6,20	6,27
Quicksort determinístico	1,02	1,63	6,09	11,51	52,02	100,57	475,34

Dell PC

Algoritmo	<i>n</i>						
	50	100	500	1.000	5.000	10.000	50.000
Ordenação por seleção	0,76	1,60	5,46	12,23	52,03	100,79	496,94
Ordenação por inserção	1,01	1,66	7,68	13,90	68,34	136,20	626,44
Ordenação por intercalação	3,21	3,38	3,57	3,33	3,36	3,37	3,15
Quicksort determinístico	1,12	1,37	6,52	9,30	47,60	97,45	466,83

O quicksort aleatorizado parece ser bem bom, mas podemos vencê-lo. Lembre-se de que a ordenação por inserção funciona bem quando nenhum elemento tem de se mover para muito longe no vetor. Logo que os tamanhos dos subproblemas nos algoritmos recursivos baixam para algum tamanho k , nenhum elemento tem de se mover mais que $k - 1$ posições. Em vez de continuar a chamar recursivamente o quicksort aleatorizado assim que os tamanhos do subproblema tornam-se pequenos, o que acontece se em vez disso executarmos a ordenação por inserção, adequadamente modificada para ordenar um subvetor em vez do vetor inteiro? De fato, com tal método híbrido, podemos ordenar até mais rapidamente que o quicksort aleatorizado. Constatei que, no meu MacBook Pro, um subvetor de tamanho 22 era o ponto ótimo de transição, e um subvetor de tamanho 17 foi o ponto de transição ótimo no meu PC. Apresento aqui as razões entre os tempos de execução do algoritmo híbrido e do quicksort aleatorizado em ambas as máquinas, para problemas do mesmo tamanho:

Máquina	n						
	50	100	500	1.000	5.000	10.000	50.000
MacBook Pro	0,55	0,56	0,60	0,60	0,62	0,63	0,66
PC	0,53	0,58	0,60	0,58	0,60	0,64	0,64

É possível bater o tempo $\Theta(n \lg n)$ para ordenação? Depende. Veremos no Capítulo 4 que, se o único modo possível de determinar onde colocar elementos é comparar elementos, fazendo coisas diferentes com base nos resultados das comparações, então não, não podemos bater o tempo $\Theta(n \lg n)$. Todavia, se soubermos alguma coisa sobre os elementos dos quais possamos tirar proveito, podemos nos sair melhor.

O QUE MAIS LER?

CLRS [CLRS09] abrange ordenação por inserção, ordenação por intercalação, quicksort determinístico e quicksort aleatorizado. Mas o avô dos livros sobre ordenação e busca continua sendo o volume 3 de *The Art of Computer Programming*, de Knuth [Knu98b]; o conselho que demos no Capítulo 1 se aplica — *TAOCP* é profundo e intenso.

Um limite inferior para ordenação e como batê-lo

No capítulo anterior, vimos quatro algoritmos para ordenar n elementos em um vetor. Dois deles, ordenação por seleção e ordenação por inserção, têm tempos de execução do pior caso $\Theta(n^2)$, que não é muito bom. Um deles, quicksort, também tem um tempo de execução do pior caso de $\Theta(n^2)$, mas leva em média apenas o tempo $\Theta(n \lg n)$. A ordenação por intercalação leva o tempo $\Theta(n \lg n)$ em todos os casos. Na prática, o quicksort é o mais rápido dos quatro, mas, se você tiver mesmo que se resguardar contra comportamento de pior caso ruim, escolherá a ordenação por intercalação.

O tempo $\Theta(n \lg n)$ é o melhor que podemos ter? É possível arquitetar um algoritmo de ordenação que bata o tempo $\Theta(n \lg n)$ no pior caso? A resposta depende das regras do jogo: como é permitido ao algoritmo de ordenação usar as chaves de ordenação quando se realiza a ordenação?

Neste capítulo, veremos que, sob certo conjunto de regras, não podemos bater $\Theta(n \lg n)$. Então veremos dois algoritmos de ordenação — a ordenação por contagem e a ordenação digital —, que desvirtuam as regras e com isso conseguem ordenar no tempo $\Theta(n)$ apenas.

REGRAS PARA ORDENAÇÃO

Se você examinar como os quatro algoritmos do capítulo anterior usam as chaves de ordenação, verá que eles realizam a ordenação com base apenas em comparação de pares de chaves de ordenação. Todas as decisões que eles tomam são da forma “se a chave de ordenação desse elemento for menor que a chave de ordenação desse outro elemento, faça alguma coisa e, caso contrário, faça alguma coisa ou não faça nada”. Você pode pensar que um algoritmo de ordenação toma *apenas* decisões dessa forma; quais outros tipos de decisões um algoritmo de ordenação poderia possivelmente tomar?

Para ver quais outros tipos de decisões são possíveis, vamos pegar uma situação realmente simples. Suponha que saibamos duas coisas sobre os elementos que estamos ordenando: cada chave de ordenação é 1 ou 2, e os elementos consistem em chaves de

ordenação apenas — nenhum dado satélite. Nessa situação simples, podemos ordenar n elementos em tempo $\Theta(n)$ apenas, o que ganha dos algoritmos $\Theta(n \lg n)$ do capítulo anterior. Como? Em primeiro lugar, vá até cada elemento e conte quantos deles têm o valor 1s; vamos dizer que k elementos tenham o valor 1. Então podemos percorrer o vetor, preenchendo o valor 1 nas primeiras k posições e o valor 2 nas últimas $n - k$ posições. Eis um procedimento:

Procedimento REALLY-SIMPLE-sort (A, n)

Entradas:

- A : um vetor no qual cada elemento é 1 ou 2.
- n : o número de elementos em A a ordenar.

Resultado: Os elementos de A são ordenados em ordem crescente.

1. Iguele k a 0.
 2. Para $i = 1$ a n :
 - a. Se $A[i] = 1$, incremente k .
 3. Para $i = 1$ a k :
 - a. Iguele $A[i]$ a 1.
 4. Para $i = k + 1$ a n :
 - a. Iguele $A[i]$ a 2.
-

As etapas 1 e 2 contam o número de 1s, incrementando a contagem k para todo elemento $A[i]$ que for igual a 1. A etapa 3 preenche $A[1..k]$ com 1s, e a etapa 4 preenche as posições remanescentes, $A[k + 1..n]$, com 2s. É bem fácil ver que esse procedimento executa em tempo $\Theta(n)$: o primeiro laço itera n vezes, os últimos dois laços juntos iteram n vezes e cada iteração de cada laço leva tempo constante.

Observe que REALLY-SIMPLE-SORT nunca compara dois elementos do vetor *um com o outro*. Ele compara cada elemento do vetor com o valor 1, mas nunca com outro elemento do vetor. Portanto, você percebe que nessa situação restrita podemos ordenar sem comparar pares de chaves de ordenação.

O limite inferior em ordenação por comparação

Agora que você tem uma ideia sobre como as regras do jogo podem variar, vamos ver um limite inferior para a velocidade com que podemos ordenar.

Definimos uma **ordenação por comparação** como qualquer algoritmo de ordenação que realize a ordenação somente comparando pares de elementos. Os quatro algoritmos de ordenação do capítulo anterior são ordenações por comparação, mas REALLY-SIMPLE-SORT não é.

Eis aqui o limite inferior:

No pior caso, qualquer algoritmo de ordenação por comparação para n elementos requer $\Omega(n \lg n)$ comparações entre pares de elementos.

Lembre-se de que a notação Ω dá um limite inferior, portanto o que estamos dizendo é “para n suficientemente grande, qualquer algoritmo de ordenação por comparação requer no mínimo $c n \lg n$ comparações no pior caso, para alguma constante c ”. Visto que cada comparação leva no mínimo tempo constante, isso nos dá um limite inferior $\Omega(n \lg n)$ para o tempo para ordenar n elementos, considerando que estamos usando o algoritmo de ordenação por comparação.

É importante entender um par de coisas sobre esse limite inferior. A primeira é que ele está dizendo algo somente sobre o pior caso. Você sempre pode fazer um algoritmo de ordenação executar em tempo linear no melhor caso: declare que o melhor caso é quando o vetor já está ordenado e apenas verifique que cada elemento (exceto o último) é menor ou igual ao seu sucessor no vetor. Isso é fácil de fazer em tempo $\Theta(n)$ e, se constatar que cada elemento é menor ou igual ao seu sucessor, você terminou. Todavia, no *pior caso* são necessárias $\Omega(n \lg n)$ comparações. Esse limite inferior é denominado limite inferior **existencial** porque ele diz que existe uma entrada que requer $\Omega(n \lg n)$ comparações. Outro tipo de limite inferior é um limite inferior universal, que se aplica a todas as entradas. Para ordenar, o único limite inferior universal que temos é $\Omega(n)$, visto que temos de examinar cada elemento, no mínimo, uma vez. Observe que, na sentença anterior, eu não disse $\Omega(n)$ o quê. Eu quis dizer $\Omega(n)$ comparações ou $\Omega(n)$ tempo? Eu quis dizer $\Omega(n)$ tempo, visto que faz sentido que temos de examinar cada elemento, mesmo que não estejamos comparando pares de elementos.

A segunda coisa importante é verdadeiramente notável: esse limite inferior não depende do algoritmo particular, desde que seja um algoritmo de ordenação por comparação. O limite inferior aplica-se a *todas* algoritmos de ordenação por comparação, não importando quanto simples ou complexo ele seja. O limite inferior aplica-se a algoritmos de ordenação por comparação que já foram inventados ou serão inventados no futuro. Aplica-se até mesmo a algoritmos de ordenação por comparação que nunca serão descobertos pela raça humana!

Bater o limite inferior com ordenação por contagem

Já vimos como bater o limite inferior em um ambiente altamente restrito: há somente dois valores possíveis para as chaves de ordenação e cada elemento consiste em apenas uma chave de ordenação, sem nenhum dado satélite. Nesse caso restrito, podemos ordenar n elementos em tempo $\Theta(n)$ sem comparar pares de elementos.

Podemos generalizar o método de REALLY-SIMPLE-SORT para manipular m diferentes valores possíveis para as chaves de ordenação, desde que elas sejam inteiros em uma faixa de m inteiros consecutivos, digamos, 0 a $m - 1$, e também podemos permitir que os elementos tenham dados satélites.

A ideia é essa: suponha que sabemos que as chaves de ordenação são inteiros na faixa 0 a $m - 1$, e vamos supor ainda mais que sabemos que exatamente três elementos têm chaves de ordenação igual a 5 e que exatamente seis elementos têm chaves de ordenação menores que 5 (isto é, na faixa 0 a 4). Então sabemos que, no vetor ordenado, os elementos com chaves de ordenação igual a 5 devem ocupar as posições 7, 8 e 9. Generalizando, se soubermos que k elementos têm chaves de ordenação iguais a x e que l elementos têm chaves de ordenação menores que m , então sabemos que os elementos que têm chaves de ordenação iguais a m devem ocupar as posições $l + 1$ até $l + k$ no vetor ordenado. Portanto, queremos computar, para cada valor possível da chave de ordenação, quantos elementos têm chaves de ordenação menores que esse valor e quantos elementos têm chaves de ordenação iguais a esse valor.

Podemos computar quantos elementos têm chaves de ordenação menores que cada valor possível da chave de ordenação computando em primeiro lugar quantos elementos têm chaves de ordenação iguais àquele valor; portanto, vamos começar com aquele:

Procedimento COUNT-KEYS-EQUAL (A, n, m)

Entradas:

- A : um vetor de inteiros na faixa 0 a $m - 1$.
- n : o número de elementos em A .
- m : define a faixa dos valores em A .

Saída: Um vetor $equal[0..m - 1]$ tal que $equal[j]$ contém o número de elementos de A que são iguais a $[j]$, para $j = 0, 1, 2, \dots, m - 1$.

1. Seja $equal [0..m - 1]$ um novo vetor.
 2. Iguale todos os valores em $equal$ a 0.
 3. Para $i = 1$ a n :
 - a. Iguale key a $A[i]$.
 - b. Incremente $equal[key]$.
 4. Retorne o vetor $equal$.
-

Observe que COUNT-KEYS-EQUAL nunca compara chaves de ordenação umas com as outras. Ele usa chaves de ordenação somente para indexar o vetor $equal$. Visto que o primeiro laço (implícito na etapa 2) faz m iterações, o segundo laço (etapa 3) faz n iterações, e cada iteração de cada laço leva tempo constante, COUNT-KEYS-EQUAL leva tempo $\Theta(m + n)$. Se m é uma constante, então COUNT-KEYS-EQUAL leva tempo $\Theta(n)$.

Agora podemos usar o vetor $equal$ para computar uma soma contínua para descobrir quantos elementos têm chaves de ordenação menores que cada valor:

Procedimento COUNT-KEYS-LESS ($equal, m$)

Entradas:

- $equal$: o vetor retornado por COUNT-KEYS-EQUAL.
- m : define a faixa de índices de $equal$: 0 a $m - 1$.

Saída : Um vetor $less[0..m - 1]$ tal que para $j = 0, 1, 2, \dots, m - 1$, $less[j]$ contém a soma $equal[0] + equal[1] + \dots + equal[j - 1]$.

1. Seja $less[0..m - 1]$ um novo vetor.
 2. Iguale $less[0]$ a 0.
 3. Para $j = 1$ a $m - 1$:
 - a. Iguale $less[j]$ a $less[j - 1] + equal[j - 1]$.
 4. Retorne o vetor $less$.
-

Considerando que $equal[j]$ dá uma contagem precisa de quantas chaves de ordenação são iguais a j , para $j = 0, 1, \dots, m - 1$, você poderia usar a seguinte invariante de laço para mostrar que, quando COUNT-KEYS-LESS retorna, $less[j]$ diz quantas chaves de ordenação são menores que j :

No início de cada iteração do laço da etapa 3,

$$less[j - 1]$$

é igual ao número de chaves de ordenação menores que $j - 1$.

Deixarei para você a tarefa de preencher as partes de inicialização, manutenção e término. É fácil ver que o procedimento COUNT-KEYS-LESS executa em tempo $\Theta(m)$. E ele certamente não compara chaves de ordenação umas com as outras.

Vamos ver um exemplo. Suponha que $m = 7$, de modo que todas as chaves de ordenação são inteiros na faixa 0 a 6 e que temos o seguinte vetor A com $n = 10$ elementos: $A = \langle 4, 1, 5, 0, 1, 6, 5, 1, 5, 3 \rangle$. Então $equal = \langle 1, 3, 0, 1, 1, 3, 1 \rangle$ e $less = \langle 0, 1, 4, 4, 5, 6, 9 \rangle$. Como $less[5] = 6$ e $equal[5] = 3$ (lembre-se de que indexamos os vetores *less* e *equal* começando em 0, e não em 1), quando terminarmos a ordenação as posições 1 até 6 devem conter valores de chave menores que 5, e as posições 7, 8 e 9 devem conter o valor de chave 5.

Tão logo tenhamos o vetor *less*, podemos criar um vetor ordenado, embora não no lugar:

Procedimento REARRANGE ($A, less, n, m$)

Entradas:

- A : um vetor de inteiros na faixa 0 a $m - 1$.
- $less$: o vetor retornado de COUNT-KEYS-LESS.
- n : o número de elementos em A .
- m : define a faixa dos valores em A .

Saída: Um vetor B contendo os elementos de A , ordenados.

1. Seja $B[1..n]$ e $next[0..m - 1]$ novos vetores.
 2. Para $j = 0$ a $m - 1$:
 - a. Iguale $next[j]$ a $less[j] + 1$.
 3. Para $i = 1$ a n :
 - a. Iguale *key* a $A[i]$.
 - b. Iguale *index* a $next[key]$.
 - c. Iguale $B[index]$ a $A[i]$.
 - d. Incremente $next[key]$.
 4. Retorne o vetor B .
-

A figura na próxima página ilustra como REARRANGE move os elementos do vetor A para o vetor B de modo que eles terminem ordenados em B . A parte de cima mostra os vetores *less*, *next*, A e B antes da primeira iteração do laço da etapa 3, e cada parte subsequente mostra *next*, A e B depois de cada iteração. Os elementos em A estão em cinza, já que são copiados para B .

A ideia é que, à medida que percorremos o vetor A do início ao fim, $next[j]$ dá o índice no vetor B para onde deve ir o próximo elemento de A cuja chave é j . Lembre-se de que dissemos antes que, se l elementos têm chaves de ordenação menores que x , então os k elementos cujas chaves de ordenação são iguais a x devem ocupar as posições $l + 1$ até $l + k$. O laço da etapa 2 estabelece o vetor *next* de modo que, a princípio, $next[j] = l + 1$, onde $l = less[j]$. O laço da etapa 3 percorre o vetor A do início ao fim. Para cada elemento $A[i]$, a etapa 3A armazena $A[i]$ em *key*, a etapa 3B computa *index* como o índice no vetor B para onde $A[i]$ deve ir e a etapa 3C move $A[i]$ para essa posição em B . Como o próximo elemento no vetor A que tem a mesma chave de ordenação que $A[i]$ (se houver alguma) deve ir para a próxima posição de B , a etapa 3D incrementa $next[key]$.

<i>less</i>	0 1 2 3 4 5 6	0 1 4 4 5 6 9	A 1 2 3 4 5 6 7 8 9 10
<i>next</i>	1 2 5 5 6 7 10		B
<i>next</i>	0 1 2 3 4 5 6	1 2 5 5 7 7 10	A 1 2 3 4 5 6 7 8 9 10
<i>next</i>	1 3 5 5 7 7 10		B
<i>next</i>	0 1 2 3 4 5 6	1 3 5 5 7 8 10	A 1 2 3 4 5 6 7 8 9 10
<i>next</i>	2 3 5 5 7 8 10		B
<i>next</i>	0 1 2 3 4 5 6	2 3 5 5 7 8 10	A 1 2 3 4 5 6 7 8 9 10
<i>next</i>	2 4 5 5 7 8 10		B
<i>next</i>	0 1 2 3 4 5 6	2 4 5 5 7 8 10	A 1 2 3 4 5 6 7 8 9 10
<i>next</i>	2 4 5 5 7 8 11		B
<i>next</i>	0 1 2 3 4 5 6	2 4 5 5 7 9 11	A 1 2 3 4 5 6 7 8 9 10
<i>next</i>	2 5 5 5 7 9 11		B
<i>next</i>	0 1 2 3 4 5 6	2 5 5 5 7 10 11	A 1 2 3 4 5 6 7 8 9 10
<i>next</i>	2 5 5 6 7 10 11		B

Quanto tempo REARRANGE leva? O laço da etapa 2 executa no tempo $\Theta(m)$ e o laço da etapa 3 executa no tempo $\Theta(n)$. Portanto, como COUNT-KEYS-EQUAL, REARRANGE executa no tempo $\Theta(m + n)$, que é $\Theta(n)$ se m for uma constante.

Agora podemos juntar os três procedimentos e criar uma *ordenação por contagem*:

Procedimento COUNTING-SORT (A, n, m)

Entradas:

A : um vetor de inteiros na faixa 0 a $m - 1$.

n : o número de elementos em A .

m : define a faixa dos valores em A .

Saída: Um vetor B que contém os elementos de A , ordenados.

1. Chame COUNT-KEYS-EQUAL (A, n, m) e designe seu resultado a *equal*.
 2. Chame COUNT-KEYS-LESS (*equal, m*) e designe seu resultado a *equal*.
 3. Chame REARRANGE ($A, less, n, m$) e designe seu resultado a B .
 4. Retornar o vetor B .
-

Pelos tempos de execução de COUNT-KEYS-EQUAL ($\Theta(m + n)$), COUNT-KEYS-LESS ($\Theta(m)$) e REARRANGE ($\Theta(m + n)$), você pode ver que COUNTING-SORT executa no tempo $\Theta(m + n)$ ou $\Theta(n)$ quando m é uma constante. A ordenação por contagem ganha do limite inferior de $\Omega(n \lg n)$ para ordenação por comparação porque nunca compara chaves de ordenação em relação umas com as outras. Em vez disso, usa chaves de ordenação para indexar vetores, o que pode fazer porque as chaves de ordenação são inteiros pequenos. Se as chaves de ordenação fossem números reais com partes fracionárias ou cadeias de caracteres, não poderíamos usar ordenação por contagem.

Você pode notar que o procedimento pressupõe que os elementos contêm somente chaves de ordenação e nenhum dado satélite. No entanto, eu prometi que, diferentemente de REALLY-SIMPLE-SORT, COUNTING-SORT permite dados satélites. E permite, desde que você modifique a etapa 3C de REARRANGE, copiar o elemento inteiro e não apenas a chave de ordenação.

Você pode também ter notado que os procedimentos que eu dei são um pouco ineficientes na questão de como usam vetores. Podemos combinar os vetores *equal*, *less* e *next* em um único vetor, mas deixo para você essa tarefa.

Insisto em mencionar que o tempo de execução é $\Theta(n)$ se m é uma constante. Quando m será uma constante? Um exemplo seria a ordenação de notas de provas. As notas vão de 0 a 100, mas o número de alunos varia. Eu poderia usar ordenação por contagem para ordenar as provas de n alunos no tempo $\Theta(n)$, visto que $m = 101$ é uma constante (lembre-se de que a faixa que está sendo ordenada é 0 a $m - 1$).

Todavia, na prática, acontece que a ordenação por contagem é útil como parte de um outro algoritmo de ordenação, a ordenação digital. Além de executar em tempo linear quando m é uma constante, a ordenação por contagem tem outra propriedade importante: ela é *estável*. Em uma ordenação estável, os elementos que têm a mesma chave de ordenação aparecem no vetor de saída na mesma ordem que aparecem no vetor de entrada. Em outras palavras, uma ordenação estável rompe vínculos entre dois elementos que têm chaves de ordenação iguais, colocando em primeiro lugar no vetor

de saída qualquer elemento que aparecer em primeiro lugar no vetor de entrada. Você pode ver por que a ordenação por contagem é estável examinando o laço da etapa 3 de REARRANGE. Se dois elementos de B têm a mesma chave de ordenação, digamos, key , então o procedimento aumenta $next[key]n$ imediatamente depois de mover para B o elemento que ocorre mais cedo em A ; desse modo, quando ele mover o elemento que ocorre mais tarde em A , esse elemento aparecerá mais adiante em B .

ORDENAÇÃO DIGITAL

Suponha que você tivesse de ordenar cadeias de caracteres de algum comprimento fixo. Por exemplo, estou escrevendo este parágrafo dentro de um avião; quando fiz minha reserva, recebi o código de confirmação XI7FS6. A companhia aérea compõe todos os códigos de confirmação como cadeias de seis caracteres, nas quais cada caractere é uma letra ou um dígito. Cada caractere pode adotar 36 valores (26 letras mais 10 dígitos) e, portanto, há $36^6 = 2.176.782.336$ códigos de confirmação possíveis. Embora esse número seja uma constante, é uma constante bem grande e, portanto, a empresa aérea provavelmente não recorre apenas à ordenação por contagem para ordenar códigos de confirmação.

Para sermos concretos, vamos dizer que podemos traduzir cada um dos 36 caracteres para um código numérico que vai de 0 a 35. O código para um dígito é o próprio dígito (de modo que o código para o dígito 5 é 5), e os códigos para letras começam em 10 para A e vão até 35 para Z.

Agora, para as coisas ficarem mais simples, suponha que cada código de confirmação compreenda somente dois caracteres (não se preocupe: logo voltaremos a seis caracteres). Embora pudéssemos executar ordenação por contagem com $m = 36^2 = 1.296$, em vez disso a executaremos *duas vezes* com $m = 36$. Nós a executamos pela primeira vez usando o caractere da *extrema direita* como a chave de ordenação. Então tomamos o resultado da execução da ordenação por contagem na primeira vez e o executamos uma segunda vez, mas agora usando o caractere da *extrema esquerda* como a chave de ordenação. Escolhemos ordenação por contagem porque ela funciona bem quando m é relativamente pequeno e porque é estável.

Por exemplo, suponha que tenhamos os códigos de confirmação de dois caracteres $\langle F6, E5, R6, X6, X2, T5, F2, T3 \rangle$. Depois de executar a ordenação por contagem no caractere da extrema direita, obtemos a ordem ordenada $\langle X2, F2, T3, E5, T5, F6, R6, X6 \rangle$. Observe que, como a ordenação por contagem é estável e X2 vem antes de F2 na ordem original, X2 vem antes de F2 depois da ordenação exatamente no caractere da extrema direita. Agora ordenamos o resultado no caractere da extrema esquerda, novamente usando ordenação por contagem, obtendo o resultado desejado $\langle E5, F2, F6, R6, T3, T5, X2, X6 \rangle$.

O que teria acontecido se tivéssemos ordenado em relação ao caractere da extrema esquerda primeiro? Depois de executar a ordenação por contagem no caractere da extrema esquerda, teríamos $\langle E5, F6, F2, R6, T5, T3, X6, X2 \rangle$ e, então, depois de executar a ordenação por contagem no caractere da extrema direita do resultado, obteríamos $\langle F2, X2, T3, E5, T5, F6, R6, X6 \rangle$, o que é incorreto.

Por que trabalhar da direita para a esquerda dá um resultado correto? Usar um método de ordenação estável é importante; poderia ser ordenação por contagem ou

qualquer outro método de ordenação estável. Vamos supor que estejamos trabalhando na i -ésima posição de caractere e considerarmos que, se examinarmos as $i - 1$ posições de caracteres da extrema direita, o vetor está ordenado. Considere quaisquer duas chaves de ordenação. Se elas forem diferentes na i -ésima posição de caractere, não importa o que esteja nas $i - 1$ posições à direita: o algoritmo de ordenação estável que ordena em relação à i -ésima posição os colocará na ordem correta. Se, por outro lado, elas tiverem o mesmo caractere na i -ésima posição, a que vem primeiro nas $i - 1$ posições de caracteres na extrema direita deverá vir primeiro e, usando esse método de ordenação estável, garantimos que é isso exatamente o que acontece.

Portanto, vamos voltar aos códigos de confirmação de seis caracteres e veremos como ordenar códigos de confirmação que começam na ordem $\langle \text{XI7FS6}, \text{PL4ZQ2}, \text{JI8FR9}, \text{XL8FQ6}, \text{PY2ZR5}, \text{KV7WS9}, \text{JL2ZV3}, \text{KI4WR2} \rangle$. Vamos numerar os caracteres da direita para a esquerda de 1 a 6. Então aqui estão os resultados depois de executar uma ordenação estável em relação ao i -ésimo caractere, trabalhando da direita para a esquerda:

i Ordem resultante

1. $\langle \text{PL4ZQ2}, \text{KI4WR2}, \text{JL2ZV3}, \text{PY2ZR5}, \text{XI7FS6}, \text{XL8FQ6}, \text{JI8FR9}, \text{KV7WS9} \rangle$
2. $\langle \text{PL4ZQ2}, \text{XL8FQ6}, \text{KI4WR2}, \text{PY2ZR5}, \text{JI8FR9}, \text{XI7FS6}, \text{KV7WS9}, \text{JL2ZV3} \rangle$
3. $\langle \text{XL8FQ6}, \text{JI8FR9}, \text{XI7FS6}, \text{KI4WR2}, \text{KV7WS9}, \text{PL4ZQ2}, \text{PY2ZR5}, \text{JL2ZV3} \rangle$
4. $\langle \text{PY2ZR5}, \text{JL2ZV3}, \text{KI4WR2}, \text{PL4ZQ2}, \text{XI7FS6}, \text{KV7WS9}, \text{XL8FQ6}, \text{JI8FR9} \rangle$
5. $\langle \text{KI4WR2}, \text{XI7FS6}, \text{JI8FR9}, \text{JL2ZV3}, \text{PL4ZQ2}, \text{XL8FQ6}, \text{KV7WS9}, \text{PY2ZR5} \rangle$
6. $\langle \text{JI8FR9}, \text{JL2ZV3}, \text{KI4WR2}, \text{KV7WS9}, \text{PL4ZQ2}, \text{PY2ZR5}, \text{XI7FS6}, \text{XL8FQ6} \rangle$

Para generalizar, no algoritmo de *ordenação digital* pressupomos que podemos pensar que cada chave de ordenação é um número de d dígitos, no qual cada dígito está na faixa 0 a $m - 1$. Executamos uma ordenação estável em cada dígito, indo da direita para a esquerda. Se usarmos ordenação por contagem como a ordenação estável, o tempo para ordenar em relação a um dígito é $\Theta(m - n)$ e o tempo para ordenar todos os d dígitos é $\Theta(d(m - n))$. Se m é uma constante (como 36 no exemplo do código de confirmação), então o tempo para ordenação digital é $\Theta(dn)$. Se d também for uma constante (como 6 para os códigos de confirmação), o tempo para ordenação digital será apenas $\Theta(n)$.

Quando a ordenação digital usa ordenação por contagem para ordenar em relação a cada dígito, ela nunca compara duas chaves de ordenação uma em relação à outra. Usa os dígitos individuais para indexar vetores dentro da ordenação por contagem. É por isso que a ordenação digital, assim como a ordenação por contagem, bate o limite inferior de $\Omega(\lg n)$ para ordenação por comparação.

O QUE MAIS LER?

O Capítulo 8 de CLRS [CLRS09] expande todo o material deste capítulo.

Grafos acíclicos dirigidos

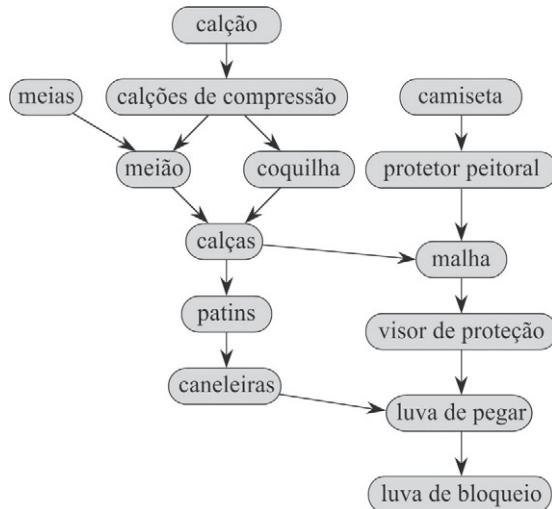
Lembre-se de que, na nota de rodapé na página 1, eu revelei que costumava jogar hóquei. Durante muitos anos, fui goleiro, mas a certa altura minha atuação se deteriorou até o ponto em que nem eu mesmo gostava de me ver jogar. Parecia que todo disco dava um jeito de bater no fundo da rede. Então, depois de um hiato de sete anos, voltei às traves (isto é, voltei a jogar no gol) e participei de alguns jogos.

Minha maior preocupação não era se seria um bom goleiro — eu sabia que ia ser horrível —, mas se eu lembraria como vestir todo o equipamento que um goleiro de hóquei tem de usar. Em hóquei no gelo, os goleiros usam muito equipamento (15-18 kg) e, ao me vestir para um jogo, eu tenho de vestir essa parafernália na ordem certa. Por exemplo, como sou destro, uso na mão esquerda uma luva enorme que serve para agarrar discos, denominada, é claro, luva de pegar. Depois de ter vestido a luva de pegar, minha mão esquerda não tem nenhuma destreza e eu não consigo fazer passar mais nada pela luva.

Quando estava me preparando para envergar todo o equipamento de goleiro, desenhei um diagrama que mostrava quais itens eu tinha de vestir antes de outros itens. O diagrama é ilustrado na página seguinte. Uma seta que vai do item A ao item B indica a seguinte restrição: A deve ser vestido antes de B. Por exemplo, eu tenho de vestir o protetor peitoral antes da malha. É claro que a expressão restritiva “tem de ser vestido antes de” é *transitiva*: se o item A deve ser vestido antes do item B, e o item B deve ser vestido antes do item C, então o item A deve ser vestido antes do item C. Portanto, eu tenho de vestir o protetor peitoral antes da malha, do visor de proteção, da luva de pegar e da luva de bloqueio.

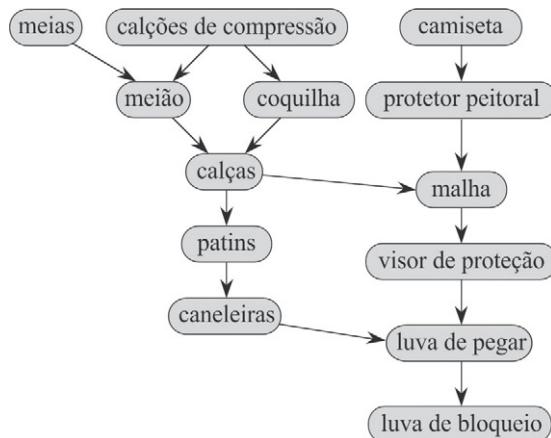
Todavia, para alguns itens, a ordem de vesti-los não importa. Posso vestir as meias antes ou depois do protetor peitoral, por exemplo.

Eu precisava determinar uma ordem para me vestir. Depois de ter desenhado o meu diagrama, tinha de organizar uma lista que contivesse todos os itens que eu tinha de vestir, em uma ordem simples e única que não violasse nenhuma das restrições “tem de ser vestido antes de”. Constatei que várias ordens funcionavam; abaixo do diagrama apresento três delas.

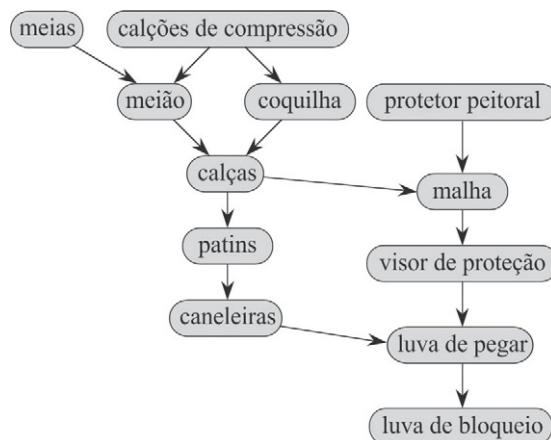


Ordem n. 1	Ordem n. 2	Ordem n. 3
calção	calção	meias
calcão de compressão	camiseta	camiseta
coquilha	coquilha	protetor peitoral
meião	protetor peitoral	calção de compressão
calças	meias	meião
patins	meião	coquilha
caneleiras	calças	calças
camiseta	malha	patins
protetor peitoral	visor de proteção	caneleiras
malha	patins	malha
visor de proteção	caneleiras	visor de proteção
luva de pegar	luva de pegar	luva de pegar
luva de bloqueio	luva de bloqueio	luva de bloqueio

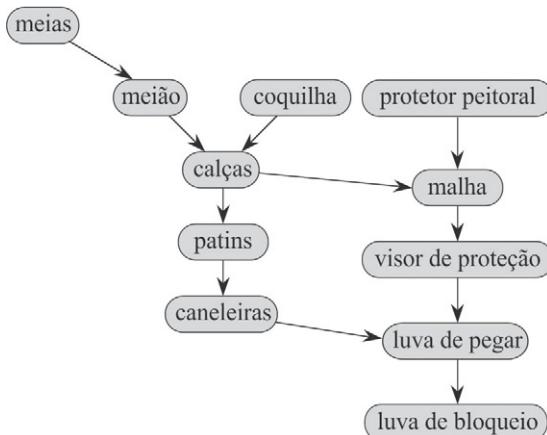
Como eu cheguei nessas ordens? Aqui eu explico como cheguei à ordem n. 2. Procurei um item para o qual não havia nenhuma seta dirigida, porque tal item não precisa ser vestido depois de nenhum item. Escolhi o calção como o primeiro item na ordem e, então, depois de ter vestido (conceitualmente) o calção eu o retirei do diagrama, o que resultou no diagrama na parte superior da página seguinte.



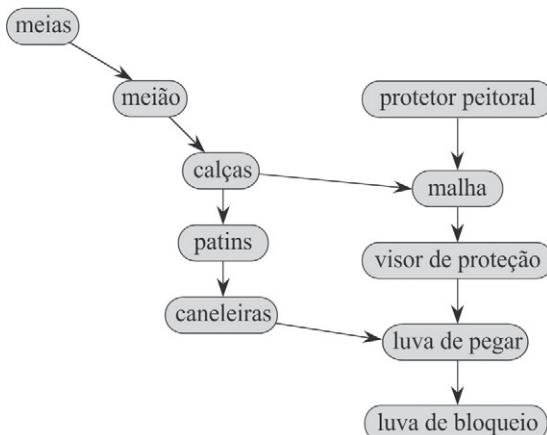
Então, novamente escolhi um item que não tinha nenhuma seta dirigindo-se a ele, dessa vez a camiseta. Adicionei a camiseta ao final da ordem e a retirei do diagrama, o que resultou no seguinte diagrama:



Mais uma vez, escolhi um item que não tinha nenhuma seta apontando para ele — calção de compressão —, adicionei-o ao final da ordem e o retirei do diagrama, o que resultou no diagrama da parte superior da página seguinte.



Em seguida, escolhi a coquilha:



Continuei fazendo isso — escolhia um item sem nenhuma seta apontando para ele, adicionava-o ao final da ordem e o retirava do diagrama — até não restar nenhum item. As três ordens mostradas na página 62 resultam de várias escolhas para o item que não tem nenhuma seta apontando para ele na página 62.

GRAFOS ACÍCLICOS DIRIGIDOS

Esses diagramas são exemplos específicos de *grafos dirigidos*, formados por *vértices*, que correspondem aos itens do equipamento de goleiro, e *arestas dirigidas*, representadas por setas. Cada aresta dirigida é um par ordenado da forma (u, v) , onde u e v são vértices. Por exemplo, a aresta da extrema esquerda no grafo dirigido na página 62 é $(\text{meias}, \text{meião})$. Quando um grafo dirigido contém uma aresta dirigida (u, v) , dizemos que v é *adjacente* a u e que (u, v) *sai de* u e *entra em* v , de modo que o vértice denominado *meião* é adjacente ao vértice denominado *meias*, e a aresta $(\text{meias}, \text{meião})$ sai do vértice denominado *meias* e entra no vértice denominado *meião*.

Os grafos dirigidos que vimos têm outra propriedade: não há nenhum meio de partir de um vértice e voltar a ele mesmo seguindo uma sequência de uma ou mais arestas. Denominamos tal grafo dirigido **grafo acíclico dirigido** (**directed acyclic graph** ou **dag**). Ele é acíclico porque não há nenhum meio de fazer um “ciclo” partindo de um vértice e voltando a ele mesmo (veremos uma definição mais formal de ciclo adiante neste capítulo).

Dags são ótimos para modelar dependências quando uma tarefa deve ocorrer antes de outra. Outro uso para dags surge no planejamento de projetos, como a construção de uma casa: por exemplo, a estrutura deve estar no lugar antes da colocação do teto. Ou na culinária, em que certas etapas devem ocorrer em certa ordem, mas para algumas etapas a ordem de ocorrência não importa; veremos um exemplo de dag para cozinhar mais adiante neste capítulo.

ORDENAÇÃO TOPOLOGÍCA

Quando precisei determinar uma ordem única, linear, para vestir o equipamento de goleiro, tive de executar uma “ordenação topológica”. Mais exatamente, uma **ordenação topológica** de um dag produz uma ordenação linear tal que, se (u, v) é uma aresta no dag, então u aparece antes de v na ordenação linear. A ordenação topológica é diferente da ordenação no sentido que usamos nos Capítulos 3 e 4.

A ordem linear produzida por uma ordenação topológica não é necessariamente única. Mas você já sabe disso, visto que cada uma das três ordens para vestir o equipamento de goleiro na página 62 poderia ser produzida por uma ordenação topológica.

Outro uso para ordenação topológica ocorreu em um trabalho de programação que executei há muito tempo. Estávamos criando sistemas de projeto auxiliados por computador e nossos sistemas podiam manter uma biblioteca de partes. As partes podiam conter outras partes, mas nenhuma dependência circular era permitida: uma parte nunca poderia conter ela mesma. Precisávamos passar as partes do projeto para uma fita (Eu disse que isso aconteceu há muito tempo), de forma que cada parte precedia quaisquer outras partes que a contivessem. Se cada parte é um vértice, e uma aresta (u, v) indica que a parte v contém a parte u , então precisávamos escrever as partes de acordo com uma ordem linear ordenada topologicamente.

Qual vértice seria um bom candidato a ser o primeiro na ordem linear? Qualquer vértice que não tivesse nenhuma aresta de entrada serviria. O número de arestas que entram em um vértice é o **grau de entrada** do vértice; portanto, poderíamos começar com qualquer vértice cujo grau de entrada fosse 0. Felizmente, todo dag deve ter no mínimo um vértice com grau de entrada 0 e no mínimo um vértice com **grau de saída 0** (nenhuma aresta sai do vértice), caso contrário haveria um ciclo.

Portanto, suponha que escolhemos qualquer vértice com grau de entrada 0 — vamos denominá-lo vértice u — e o colocamos no início da ordem linear. Como cuidamos do vértice u em primeiro lugar, todos os outros vértices serão colocados depois de u na ordem linear. Em particular, qualquer vértice v adjacente a u deve aparecer em algum lugar depois de u na ordem linear. Portanto, podemos remover u e todas as arestas que saem de u do dag com segurança, porque sabemos que já cuidamos das dependências que essas arestas definem. Quando removemos um vértice e as arestas que saem desse vértice de um dag, com o que ficamos? Outro dag! Afinal, não podemos criar um

ciclo removendo um vértice e arestas. Assim podemos repetir o processo com o dag que sobrou, procurando algum vértice com grau de entrada 0, colocando-o depois do vértice u na ordem linear, removendo arestas, e assim por diante.

O procedimento ao final desta página para ordenação topológica usa essa ideia; porém, em vez de realmente remover vértices e arestas do dag, ele apenas rastreia o grau de entrada de cada vértice, decrementando o grau de entrada para cada aresta de entrada que conceitualmente removemos. Visto que os índices de vetores são números inteiros, vamos presumir que identificamos cada vértice por um único inteiro na faixa 1 a n . Como o procedimento precisa identificar rapidamente algum vértice com grau de entrada 0, ele mantém o grau de entrada de cada vértice em um vetor de *graus de entrada* (*in-degree*) indexado pelos vértices e uma lista *seguinte* (*next*) de todos os vértices com grau de entrada 0. As etapas 1–3 inicializam o vetor *grau de entrada* (*in-degree*), a etapa 4 inicializa *next*, e a etapa 5 atualiza *grau de entrada* (*in-degree*) e *next* à medida que vértices e arestas são conceitualmente removidos. O procedimento pode escolher qualquer vértice em *next* como o próximo e colocá-lo na ordem linear.

Vamos ver como as primeiras iterações da etapa 5 funcionam no dag para vestir o equipamento de goleiro. Para executar o procedimento TOPOLOGICAL-SORT nesse dag, precisamos numerar os vértices, como mostrado na página 67. Somente os vértices 1, 2 e 9 têm grau de entrada 0 e, portanto, quando entramos no laço de etapa 5 a lista *next* contém somente esses três vértices. Para conseguir a ordem n . 1 na página 62, a ordem dos vértices em *next* seria 1, 2, 9. Então, na primeira iteração do laço da etapa 5, escolhemos o vértice 1 (calção) como vértice u , o eliminamos de *next*, adicionamos esse vértice ao final da ordem linear inicialmente vazia e então decrementamos *in-degree* [3] (calção de compressão).

Procedimento TOPOLOGICAL-SORT (G)

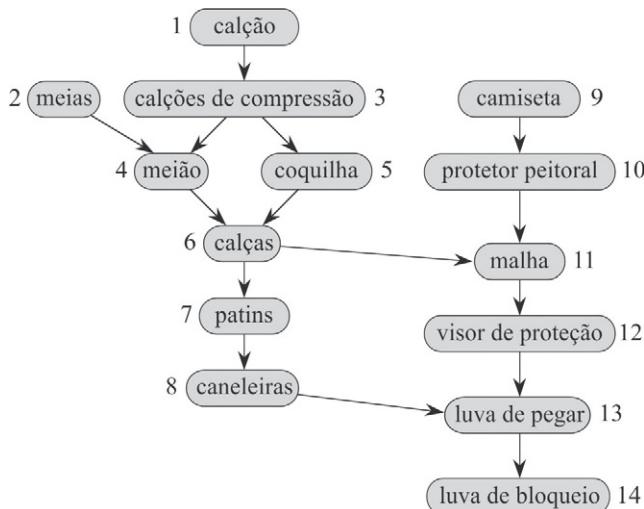
Entrada: G : um grafo dirigido acíclico com vértices numerados de 1 a n .

Saída: Uma ordem linear dos vértices tal que u aparece antes de v na ordem linear se (u, v) for uma aresta no grafo.

1. Seja *in-degree* [1.. n] um novo vetor, crie uma ordem linear de vértices vazia.
 2. Iguale todos os valores em *in-degree* a 0.
 3. Para cada vértice u :
 - a. Para cada vértice v adjacente a u :
 - i. Incremente *in-degree* [v].
 4. Faça uma lista *next* consistindo em todos os vértices u tais que *in-degree* *in-degree* [u] = 0.
 5. Enquanto *next* for não vazio, faça o seguinte:
 - a. Elimine um vértice de *next* e denomine-o vértice u .
 - b. Adicione u ao final da ordem linear.
 - c. Para cada vértice v adjacente a u :
 - i. Decremente *in-degree* [v].
 - ii. Se *in-degree* [v] = 0, insira v na lista *next*.
 6. Retorne a ordem linear.
-

Como a operação leva *in-degree* [3] até 0, inserimos o vértice 3 em *next*. Vamos pressupor que, quando inserirmos um vértice em *next*, nós o inserirmos como o primeiro vértice na lista. Tal lista, na qual sempre inserimos e eliminamos na mesma extremidade,

é conhecida como **pilha** porque é como uma pilha de pratos, na qual sempre pegamos um prato que está em cima da pilha e colocamos um novo prato também em cima da pilha (denominamos essa ordem **último a entrar, primeiro a sair** ou **LIFO** — “*last in, first out*”). Sob essa premissa, *next* torna-se 3, 2, 9 e, na próxima iteração do laço, escolhemos o vértice 3 como vértice *u*. Eliminamos esse vértice de *next*, o adicionamos no final da ordem linear, de modo que a ordem linear agora é “calção, calção de compressão” e decrementamos *in-degree* [4] (de 2 para 1) e *in-degree* [5] (de 1 para 0). Inserimos o vértice 5 (coquilha) em *next* e com isso *next* torna-se 5, 2, 9. Na próxima iteração, escolhemos o vértice 5 como vértice *u*, o eliminamos de *next*, o adicionamos ao final da ordem linear (agora “calção, calção de compressão, coquilha”) e decrementamos *in-degree* [6], levando-o de 2 para 1. Nenhum vértice é adicionado a *next* dessa vez e, portanto, na próxima iteração escolhemos o vértice 2 como vértice *u*, e assim por diante.



Para analisar o procedimento TOPOLOGICAL-SORT, em primeiro lugar temos de entender como representar um grafo dirigido e uma lista como *next*. Quando representamos um grafo, ele não precisa ser acíclico, porque a ausência ou a presença de ciclos não tem nenhum efeito sobre a representação de um grafo.

COMO REPRESENTAR UM GRAFO DIRIGIDO

Em um computador, podemos representar um grafo dirigido de alguns modos. Nossa convenção será que um grafo tem *n* vértices e *m* arestas. Continuamos a pressupor que cada vértice tem seu próprio número de 1 a *n*, de modo que podemos usar um vértice como índice para um vetor ou até mesmo como o número de linha ou coluna de uma matriz.

Por enquanto, apenas queremos saber quais vértices e arestas estão presentes (mais tarde, também associaremos um valor numérico a cada aresta). Poderíamos usar uma **matriz de adjacências** $n \times n$ na qual cada linha e cada coluna corresponde a um vértice, e a entrada na linha para o vértice *u* e a coluna para o vértice *v* é 1 se a aresta (u, v)

estiver presente ou 0 se o grafo não contiver a aresta (u, v) . Visto que uma matriz de adjacência tem n^2 entradas, deve ser verdade que $m \leq n^2$. Alternativamente, poderíamos apenas manter uma lista de todas as m arestas no grafo em nenhuma ordem particular. Como um híbrido entre uma matriz de adjacência e uma lista não ordenada, temos a **representação de lista de adjacências**, com um vetor de n elementos indexado pelos vértices no qual a entrada do vetor para cada vértice u é uma lista de todos os vértices adjacentes a u . No total, as listas têm m vértices, visto que há um único item de lista para cada uma das m arestas. Damos a seguir a matriz de adjacências e representações de lista de adjacências para o grafo dirigido da página 67:

Matriz de adjacência

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Listas de adjacência

1	3
2	4
3	4; 5
4	6
5	6
6	7; 11
7	8
8	13
9	10
10	11

11	12
12	13
13	14
14	(nenhuma)

A lista de arestas não ordenadas e a representação de lista de adjacências suscitam a questão de como representar uma lista. O melhor modo de representar uma lista depende dos tipos de operações que precisamos executar na lista. Para listas de arestas não ordenadas e listas de adjacências, sabemos de antemão quantas arestas estarão em cada lista, e o conteúdo das listas não mudará; portanto, podemos armazenar cada lista em um vetor. Podemos também usar um vetor para armazenar uma lista mesmo que o conteúdo da lista mude ao longo do tempo, desde que saibamos o número máximo de itens que estarão na lista a qualquer tempo. Se não precisarmos inserir um item no meio da lista ou eliminar um item do meio da lista, representar uma lista por um vetor será tão eficiente quanto qualquer outro modo.

Se precisarmos inserir no meio da lista, poderemos usar uma *lista ligada*, na qual cada item da lista inclui a localização de seu item sucessor na lista, o que simplifica o encaixe de um novo item depois de um item dado. Se também precisarmos eliminar do meio da lista, cada item na lista ligada também deve incluir a localização de seu item predecessor, de modo que possamos desencaixar rapidamente um item. Daqui em diante, presumiremos que podemos inserir ou eliminar de uma lista ligada em tempo constante. Uma lista ligada que tem somente ligações com sucessores é uma *lista simplesmente ligada*. Adicionar ligações com predecessores a torna uma *lista duplamente ligada*.

TEMPO DE EXECUÇÃO DE ORDENAÇÃO TOPOLOGÍCA

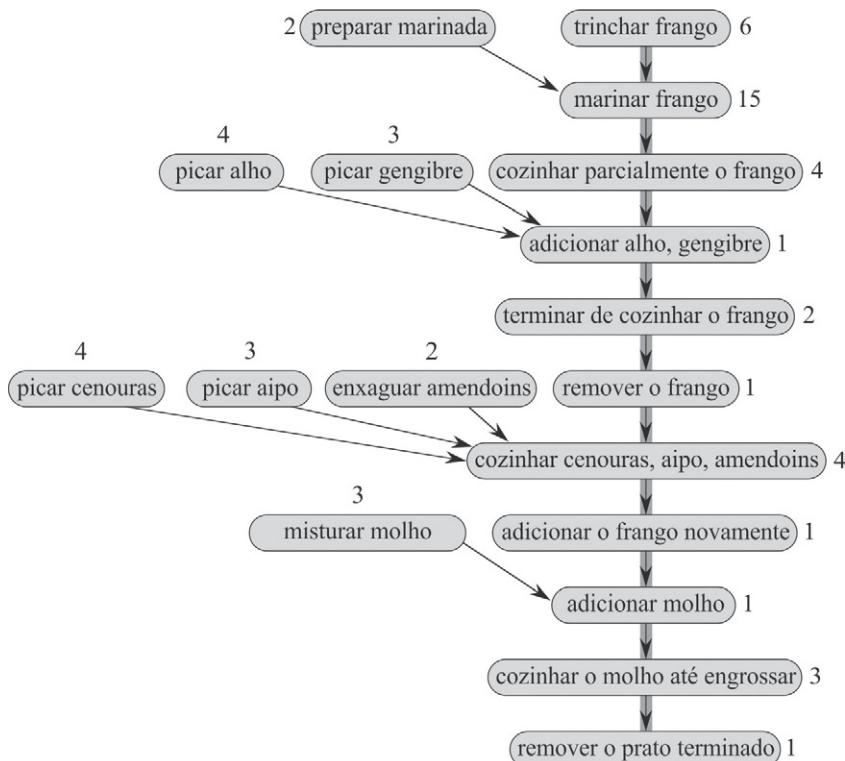
Se considerarmos que o dag usa a representação de lista de adjacências e que a lista *next* é uma lista ligada, poderemos mostrar que o procedimento TOPOLOGICAL-SORT leva tempo $\Theta(n + m)$. Visto que *next* é uma lista ligada, podemos inserir nela ou eliminar dela em tempo constante. A etapa 1 leva tempo constante e, como o vetor *in-degree* tem n elementos, a etapa 2 inicializa o vetor para todos os 0s em tempo $\Theta(n)$. A etapa 3 leva tempo $\Theta(n + m)$. O termo $\Theta(n)$ na etapa 3 surge porque o laço externo examina cada um dos n vértices, e o termo $\Theta(m)$ é porque o laço interno da etapa 3A visita cada uma das m arestas exatamente uma vez em todas as iterações do laço externo. A etapa 4 leva tempo $O(n)$, visto que a lista *next* começa com, no máximo, n vértices. A maior parte do trabalho ocorre na etapa 5. Como cada vértice é inserido em *next* exatamente uma vez, o laço principal itera n vezes. As etapas 5A e 5B levam tempo constante em cada iteração. Assim como na etapa 3A, o laço na etapa 5C itera m vezes no total, uma vez por aresta. As etapas 5Ci e 5Cii levam tempo constante por iteração, de modo que todas as iterações juntas da etapa 5C levam tempo $\Theta(m)$ e, portanto, o laço da etapa 5 leva tempo $\Theta(n + m)$. É claro que a etapa 6 leva tempo constante e, assim, quando somamos o tempo para todas as etapas, obtemos $\Theta(n + m)$.

CAMINHO CRÍTICO EM UM DIAGRAMA PERT

Gosto de relaxar cozinhando depois de um dia de trabalho e sempre gostei de cozinhar e comer frango *kung pao*. Eu tenho de preparar o frango, picar vegetais, preparar uma marinada, preparar um molho e cozinhar o prato. Exatamente como no caso de vestir o equipamento de goleiro, algumas etapas devem ocorrer antes de outras e, portanto, posso usar um dag para modelar o procedimento para preparar frango *kung pao*. Apresentamos o dag a seguir, nesta página.

Ao lado de cada vértice no dag aparece um número, que indica quantos minutos eu preciso para executar a tarefa correspondente ao vértice. Por exemplo, levo quatro minutos para picar o alho (porque descasco cada dente antes e uso *muito* alho). Se você somar os tempos para todas as tarefas, poderá ver que, se eu fosse executá-las em sequência, levaria uma hora para preparar o frango *kung pao*.

Todavia, se eu tivesse ajuda, poderíamos executar várias das tarefas simultaneamente. Por exemplo, uma pessoa pode preparar a marinada, enquanto outra trincha o frango. Com um número suficiente de pessoas ajudando e suficiente espaço, facas, tábua de cortar e tigelas, poderíamos executar muitas das tarefas simultaneamente. Se você examinar quaisquer duas tarefas no dag e constatar que não há nenhum modo de seguir setas para passar de uma para outra, eu poderia designar cada uma das tarefas a uma pessoa diferente e executá-las simultaneamente.



Dados recursos ilimitados (pessoas, espaço, equipamento culinário) para executar tarefas simultaneamente, em quanto tempo poderíamos preparar frango *kung pao*? O dag é um exemplo de **diagrama PERT**, um acrônimo para *program evaluation and review technique* (técnica de avaliação e revisão de programa). O tempo para concluir o serviço inteiro, mesmo com o máximo possível de tarefas executadas simultaneamente, é denominado “caminho crítico” no diagrama PERT. Para entender o que é um caminho crítico, em primeiro lugar temos de entender o que é um caminho e depois poderemos definir um caminho crítico.

Um **caminho** em um grafo é uma sequência de vértices e arestas que permitem que você vá de um vértice a outro (ou volte para ele mesmo); dizemos que o caminho contém os vértices, bem como as arestas percorridas. Por exemplo, um caminho no dag para preparar frango *kung pao* tem, na ordem, os vértices denominados “picar alho”, “adicionar alho, gengibre”, “terminar de cozinhar o frango” e “remover o frango” juntamente com as arestas que conectam esses vértices. O caminho de um vértice de volta a si mesmo é um **ciclo**, mas é claro que os dags não têm ciclos.

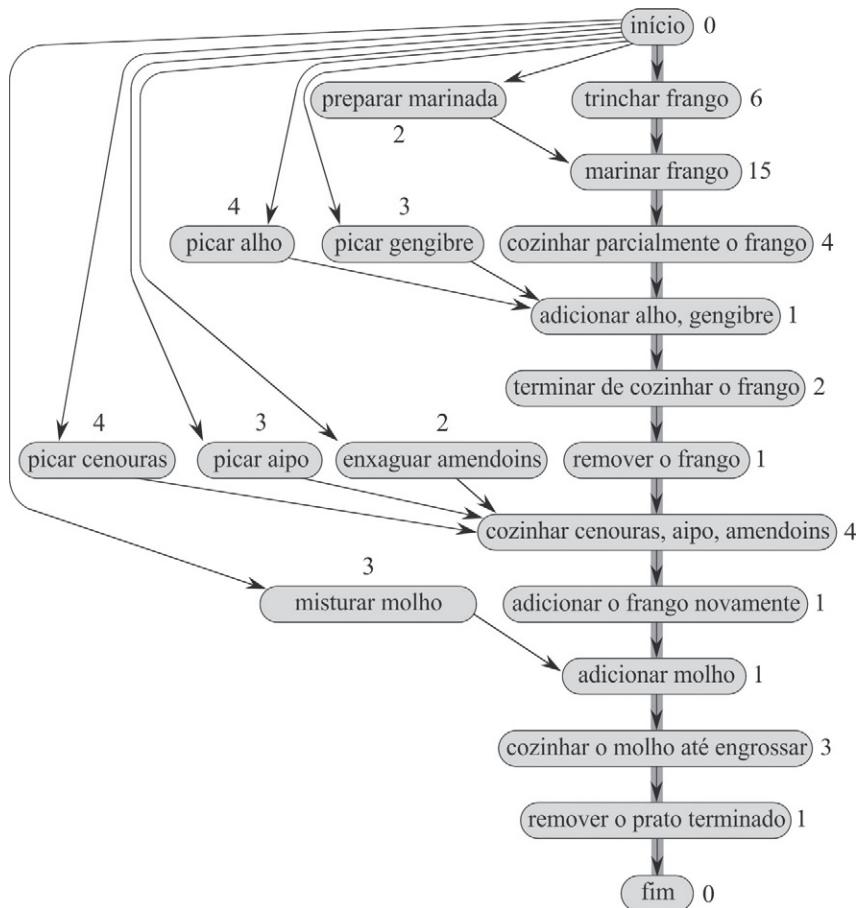
Um **caminho crítico** em um diagrama PERT é um caminho para o qual a soma dos tempos de tarefas é máxima de todos os caminhos. A soma dos tempos de tarefas ao longo do caminho crítico dá o tempo mínimo possível para o trabalho inteiro, não importando quantas tarefas são executadas simultaneamente. Eu sombrei o caminho crítico no diagrama PERT para preparar frango *kung pao*. Se você somar os tempos de tarefas ao longo do caminho crítico, verá que, não importando quantas pessoas ajudarem, levará no mínimo 39 minutos para preparar frango *kung pao*.¹

Considerando que todos os tempos de tarefas são positivos, um caminho crítico em um diagrama PERT deve começar em algum vértice com grau de entrada 0 e terminar em algum vértice com grau de saída 0. Em vez de verificar caminhos entre todos os pares de vértices nos quais um tem grau de entrada 0 e um tem grau de saída 0, basta adicionar dois vértices “fictícios”, “início” e “fim”, como mostra a figura na página seguinte. Como são vértices fictícios, nós lhes damos tempos de tarefa 0. Adicionamos uma aresta que parte do início e vai para cada vértice com grau de entrada 0 no diagrama PERT, e adicionamos uma aresta que parte de cada vértice com grau de saída 0 ao fim. Agora o único vértice com grau de entrada 0 é o início, e o único vértice com grau de saída 0 é o fim. Um caminho do início ao fim com a máxima soma de tempos de tarefas em seus vértices (sombreado) dá um caminho crítico no diagrama PERT — menos os vértices fictícios início e fim, é claro.

Uma vez adicionados os vértices fictícios, determinamos um caminho crítico marcando o caminho mais curto de todos do início ao fim, com base nos tempos de tarefas. Nesse ponto, você poderia pensar que eu cometi um erro na sentença anterior, já que o caminho crítico deve corresponder a um caminho mais longo, e não a um mais curto. De fato, é isso mesmo. Porém, como o diagrama PERT não tem nenhum ciclo, podemos alterar os tempos de tarefa de modo que o caminho mais curto de todos nos dá um

¹ Se você estiver imaginando por que os restaurantes chineses podem entregar um pedido de frango *kung pao* em muito menos tempo, é porque eles preparam muitos dos ingredientes com antecedência e seus fogões industriais podem cozinhar mais rapidamente do que o fogão que eu tenho em casa.

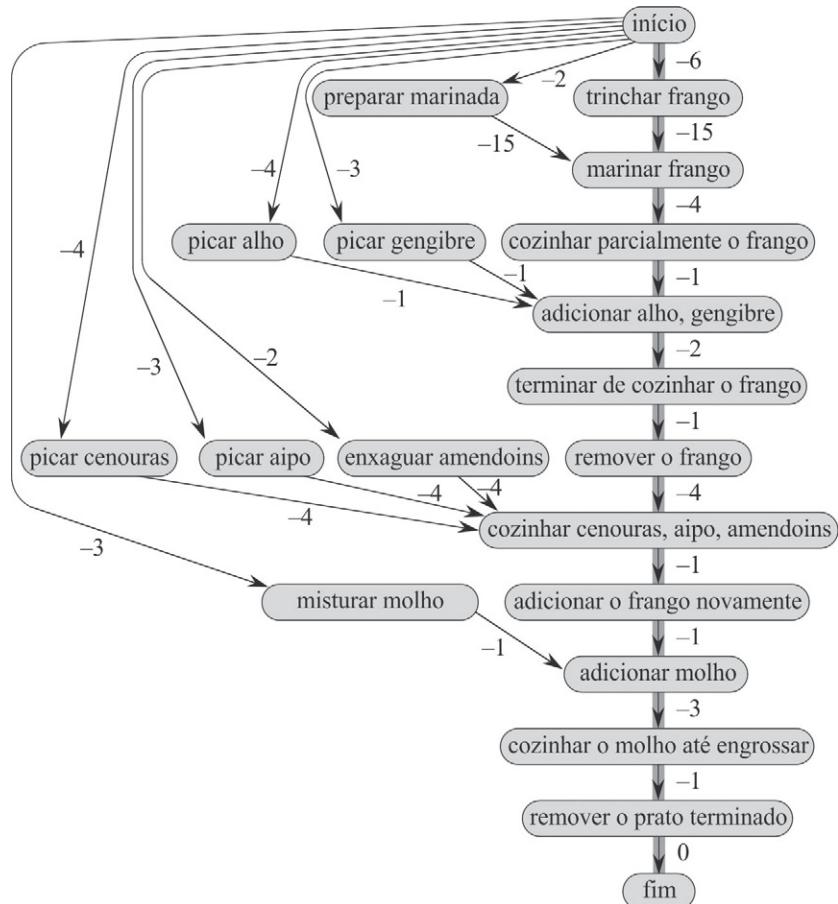
caminho crítico. Em particular, negativamos cada tempo de tarefa e determinamos um caminho do início ao fim com a soma *mínima* de tempos de tarefas.



Por que negativar tempos de tarefas para determinar um caminho com a soma mínima de tempos de tarefas? Porque resolver esse problema é um caso especial de determinar caminhos mais curtos (caminhos mínimos) e temos vários algoritmos para determinar caminhos mínimos. Todavia, quando falamos de caminhos mínimos, os valores que determinam comprimentos de caminhos estão associados a arestas, e não a vértices. Denominamos o valor que associamos a cada aresta seu **peso**. Um grafo dirigido com pesos de arestas é um **grafo dirigido ponderado**. “Peso” é um termo genérico para valores associados a arestas. Se um grafo dirigido ponderado representar uma rede rodoviária, cada aresta representará uma direção de uma estrada entre duas interseções, e o peso de uma aresta poderá representar o comprimento da estrada, o tempo requerido para percorrer a estrada ou o pedágio que um veículo paga para usar a estrada. O **peso de um caminho** é a soma dos pesos das arestas no caminho, de

modo que, se os pesos de arestas representarem distâncias de estrada, o peso de um caminho poderá indicar a distância total percorrida ao longo da estrada no caminho. Um **caminho mínimo** do vértice u ao vértice v é um caminho cuja soma de pesos de arestas é a mínima entre todos os caminhos de u a v . Caminhos mínimos não são necessariamente únicos, visto que um grafo dirigido de u a v poderia conter vários caminhos cujos pesos atingem o mínimo.

Para converter um diagrama PERT com tempos de tarefas negativados em um grafo dirigido ponderado, empurramos o tempo de tarefa de cada vértice para cada uma de suas arestas de entrada. Isto é, se o vértice v tiver um tempo de tarefa (não negativado) de t , estabelecemos que o peso de cada aresta (u, v) , entrando em v será $-t$. Damos a seguir o dag que obtivemos. Os pesos de arestas aparecem próximos de suas arestas:



Agora só resta determinar um caminho mínimo (sombreado) do início ao fim desse dag, com base nesses pesos de arestas. Um caminho crítico no diagrama PERT original corresponderá aos vértices no caminho mínimo que determinamos, menos início e fim. Portanto, agora vamos determinar um caminho mínimo em um dag.

CAMINHO MÍNIMO EM UM GRAFO DIRIGIDO ACÍCLICO

Há outra vantagem em aprender como determinar um caminho mínimo em um dag: lançaremos as fundações para determinar caminhos mínimos em grafos dirigidos arbitrários que podem ter ciclos. Examinaremos esse problema mais geral no Capítulo 6. Como fizemos para ordenar topologicamente um dag, presumiremos que o dag é armazenado com a representação de lista de adjacências e que, com cada aresta (u, v) , também armazenamos seu peso como *weight* (u, v) .

Em um dag que derivamos de um diagrama PERT, queremos um caminho mínimo do *vértice-fonte*, que denominaremos “*início*”, até um *vértice-alvo* específico denominado “*fim*”. Aqui, resolveremos o problema mais geral de determinar **caminhos mínimos de fonte única**, no qual determinaremos caminhos mínimos de um vértice-fonte até *todos* os outros vértices. Por convenção, damos ao vértice-fonte o símbolo s e queremos computar duas coisas para cada vértice v . A primeira é o peso de um caminho mínimo de s a v , que denotamos $sp(s, v)$. A segunda é o *predecessor* de v em um caminho mínimo de s a v : um vértice u tal que um caminho mínimo de s a v é um caminho de s a u e então uma única aresta (u, v) . Numeraremos os n vértices de 1 a n , de modo que nossos algoritmos para caminhos mínimos aqui e no Capítulo 6 possam armazenar esses resultados em vetores $shortest[1..n]$ e $pred[1..n]$, respectivamente. À medida que os algoritmos se desenrolam, os valores em $shortest[v]$ e $pred[v]$ podem não ser seus valores corretos finais, mas serão quando os algoritmos terminarem.

Precisamos tratar um par de casos que podem surgir. O primeiro é: e se não houver nenhum caminho de s a v ? Então definimos $sp(s, v) = \infty$, de modo que $shortest[v]$ deve ser ∞ . Visto que v não teria nenhum predecessor em um caminho mínimo que vem de s , também podemos dizer que $pred[v]$ deve ser o valor especial `NULL`. Além disso, todos os caminhos mínimos que saem de s começam com s e também não têm nenhum predecessor; assim dizemos que $pred[s]$ também deve ser `NULL`. O outro caso surge somente em grafos que têm ciclos e pesos de arestas negativos: e se o peso de um ciclo for negativo? Então poderíamos percorrer o ciclo para sempre, diminuindo o peso do caminho em cada rodada. Se pudermos ir de s a um ciclo de peso negativo e depois a v , então $sp(s, v)$ é indefinido. Todavia, por enquanto estamos preocupados somente com grafos acíclicos e, portanto, não há nenhum ciclo, muito menos ciclos de peso negativo com os quais nos preocuparmos.

Para computar caminhos mínimos que partem de um vértice-fonte s , começamos com $shortest[s] = 0$ (visto que não temos de ir a lugar nenhum para ir de um vértice a ele mesmo), $shortest[v] = \infty$ para todos os outros vértices v (visto que não sabemos de antemão quais vértices podem ser alcançados a partir de s) e $pred[v] = \text{NULL}$ para todos os vértices v . Então aplicamos uma série de **etapas de relaxação** às arestas do grafo:

Procedimento RELAX (u, v)

Entradas: u, v : vértices para os quais há uma aresta (u, v) .

Resultado: O valor de $shortest[v]$ pode decrescer e, se decrescer, $pred[v]$ torna-se u .

1. Se $shortest[u] + weight(u, v) < shortest[v]$, iguale $shortest[v]$ a $shortest[u] + weight(u, v)$ e iguale $pred[v]$ a u .

Quando chamamos $\text{RELAX}(u, v)$, estamos determinando se podemos melhorar o caminho mínimo atual de s a v tomando (u, v) como a última aresta. Comparamos o peso do caminho mínimo atual até u mais o peso de aresta (u, v) com o peso do caminho mínimo atual até v . Se for melhor pegar a aresta (u, v) , atualizamos $\text{shortest}[v]$ para esse novo peso e estabelecemos que o predecessor de v em um caminho mínimo é u .

Se relaxarmos arestas ao longo de um caminho mínimo, em ordem, obteremos os resultados corretos. Você pode estar pensando como podemos ter certeza de que relaxamos as arestas em ordem ao longo de um caminho mínimo quando nem mesmo sabemos qual é esse caminho — afinal, é isso que estamos tentando determinar —, mas você verá que é fácil para um dag. Vamos relaxar todas as arestas no dag, e as arestas de cada caminho mínimo serão entremeadas em ordem, à medida que percorremos todas as arestas e relaxamos cada uma delas.

Damos a seguir uma definição mais precisa de como relaxar arestas ao longo de um caminho mínimo, e ela se aplica a qualquer grafo dirigido, com ou sem ciclos:

Inicie com $\text{shortest}[u] = \infty$ e $\text{pred}[u] = \text{NULL}$ para todos os vértices, exceto $\text{shortest}[s] = 0$ para o vértice-fonte s .

Então relaxe as arestas ao longo de um caminho mínimo de s a qualquer vértice v , *em ordem, começando na aresta que sai de s e terminando com a aresta que entra em v* . Relaxações de outras arestas podem ser entremeadas livremente com as relaxações ao longo desse caminho mínimo, mas somente relaxações podem mudar os valores de shortest ou pred .

Depois de relaxadas as arestas, os valores shortest e pred de v estão corretos: $\text{shortest}[v] = sp(s, v)$ e $\text{pred}[v]$ é o vértice que precede v em algum caminho mínimo que vem de s .

É bem fácil ver por que relaxar as arestas ao longo de um caminho mínimo, em ordem, funciona. Suponha que um caminho mínimo de s a v visite os vértices $s, v_1, v_2, v_3 \dots v_k, v$, nessa ordem. Depois de a aresta (s, v_1) ter sido relaxada, $\text{shortest}[v_1]$ deve ter o peso de caminho mínimo correto para v_1 , e $\text{pred}[v_1]$ deve ser s . Depois de (v_1, v_2) ter sido relaxado, $\text{shortest}[v_2]$ e $\text{pred}[v_2]$ devem estar corretos. E assim por diante até relaxar (v_k, v) . Depois disso $\text{shortest}[v]$ e $\text{pred}[v] = \text{têm seus valores corretos}$.

Essa é uma ótima notícia. Em um dag, é realmente fácil relaxar cada aresta exatamente uma vez e, no entanto, relaxar as arestas ao longo de *todo* caminho mínimo, em ordem. Como? Em primeiro lugar ordene o dag topologicamente. Depois considere cada vértice, tomado na ordem linear ordenada topologicamente, e relaxe todas as arestas que saem do vértice. Visto que toda aresta deve sair de um vértice mais cedo na ordem linear e entrar em um vértice mais tarde na ordem, todo caminho no dag deve visitar vértices em uma ordem consistente com a ordem linear.

Procedimento DAG-SHORTEST-PATHS (G, s)

Entradas:

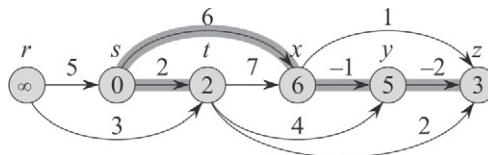
- G : um grafo acíclico dirigido ponderado que contém um conjunto V de n vértices e um conjunto E de m arestas dirigidas.
- s : um vértice-fonte em V .

Resultado: Para cada vértice v em V (exceto o vértice-fonte), $\text{shortest}[v]$ é o peso $sp(s, v)$ de um caminho mínimo de s a v , e $\text{pred}[v]$ é o vértice que precede v em algum caminho mínimo. Para

o vértice-fonte s , $\text{shortest}[s] = 0$ e $\text{pred}[s] = \text{NULL}$. Se não houver nenhum caminho de s a v , $\text{shortest}[v] = \infty$ e $\text{pred}[v] = \text{NULL}$.

1. Chame $\text{TOPOLOGICAL-SORT}(G)$ e estabeleça l como a ordem linear de vértices retornados pela chamada.
 2. Igual $\text{shortest}[v]$ a ∞ para cada vértice v exceto s , iguale $\text{shortest}[s]$ a 0 e $\text{pred}[v]$ a NULL para cada vértice v .
 3. Para cada vértice u , tomado na ordem dada de l :
 - a. Para cada vértice v adjacente a s :
 - i. Chame $\text{RELAX}(u, v)$.
-

A seguir, vemos um dag com pesos próximos das arestas. Os valores shortest obtidos do tempo de execução DAG-SHORTEST-PATHS da fonte ao vértice s aparecem dentro dos vértices, e as arestas sombreadas indicam os valores pred . Os vértices estão dispostos da esquerda para a direita na ordem linear retornada da ordenação topológica, de modo que todas as arestas vão da esquerda para a direita.



Se uma aresta (u, v) estiver sombreada, $\text{pred}[v] = u$ e $\text{shortest}[v] = \text{shortest}[u] + \text{weight}(u, v)$; por exemplo, visto que (x, y) está sombreada, $\text{pred}[y] = x$ e $\text{shortest}[y]$ (que é 5) é igual a $\text{shortest}[x]$ (que é 6) + $\text{weight}(x, y)$ (que é -1). Não há nenhum caminho de s a r e, portanto, $\text{shortest}[r] = \infty$ e $\text{pred}[r] = \text{NULL}$ (nenhuma aresta sombreada entra em r).

A primeira iteração do laço da etapa 3 relaxa as arestas (r, s) e (r, t) que saem de r ; porém, como $\text{shortest}[r] = \infty$, essas relaxações nada mudam. A próxima iteração do laço relaxa as arestas (s, t) e (s, x) que saem de s , fazendo com que $\text{shortest}[t]$ seja igualada a 2, $\text{shortest}[x]$ seja igualada a 6 e $\text{pred}[t]$ e $\text{pred}[x]$ sejam igualados a s . A iteração seguinte relaxa as arestas (t, x) , (t, y) e (t, z) , que saem de t . O valor de $\text{shortest}[x]$ não muda, visto que $\text{shortest}[t] + \text{weight}(t, x)$, que é $2 + 7 = 9$, é maior que $\text{shortest}[x]$, que é 6; porém $\text{shortest}[y]$ torna-se 6, $\text{shortest}[z]$ torna-se 4 e $\text{pred}[y]$ e $\text{pred}[z]$ são igualados a t . A próxima iteração relaxa as arestas (x, y) e (x, z) , que saem de x , fazendo com que $\text{shortest}[y]$ torne-se 5 e $\text{pred}[y]$ seja igualado a x ; $\text{shortest}[z]$ e $\text{pred}[z]$ permanecem sem mudar. A iteração final relaxa a aresta (y, z) que sai de y , fazendo com que $\text{shortest}[z]$ torne-se 3 e $\text{pred}[z]$ seja igualado a y .

É fácil ver como DAG-SHORTEST-PATHS executa em tempo $\Theta(n + m)$. Como vimos, a etapa 1 leva tempo $\Theta(n + m)$ e é claro que a etapa 2 inicializa dois valores para cada vértice e, portanto, leva tempo $\Theta(n)$. Como vimos antes, o laço externo da etapa 3 examina cada vértice exatamente uma vez, e o laço interno da etapa 3A examina cada aresta exatamente uma vez em todas as iterações. Como cada chamada de RELAX na etapa 3Ai leva tempo constante, a etapa 3 leva tempo $\Theta(n + m)$. Somando os tempos de execução para as etapas temos o tempo $\Theta(n + m)$ para o procedimento.

Voltando aos diagramas PERT, agora é fácil ver que determinar um caminho crítico leva tempo $\Theta(n + m)$, onde o diagrama PERT tem n vértices e m arestas. Somamos os dois vértices, início e fim, e somamos no máximo m arestas que saem de início e no máximo m arestas que entram em fim, teremos um total de no máximo $3m$ arestas no dag. Negativar os pesos e empurrá-los dos vértices para as arestas leva tempo $\Theta(m)$ e determinar um caminho mínimo no dag resultante leva tempo $\Theta(n + m)$.

O QUE MAIS LER?

O Capítulo 22 de CLRS [CLRS09] apresenta um algoritmo para ordenar topologicamente um dag diferente do apresentado neste capítulo, que aparece no volume 1 de *The Art of Computer Programming* de Knuth [Knu97]. O método em CLRS é um pouco mais simples à primeira vista, porém é menos intuitivo que a abordagem neste capítulo e recorre à técnica de visitar vértices em um grafo conhecida como “busca em profundidade”. O algoritmo para determinar caminhos mínimos de fonte única em um dag aparece no Capítulo 24 de CLRS.

Você pode ler mais sobre diagramas PERT, que estão em uso desde os anos 1950, em qualquer dos muitos livros que tratam de gerenciamento de projetos.

Caminhos mínimos

No Capítulo 5, vimos como encontrar caminhos mínimos de fonte única em um grafo acíclico dirigido. O algoritmo para tal dependia de o grafo ser acíclico — sem ciclos —, de modo que podíamos primeiro ordenar topologicamente os vértices do grafo.

Todavia, a maioria dos grafos que modelam situações da vida real tem ciclos. Por exemplo, em um grafo que modela uma rede rodoviária, cada vértice representa uma interseção, e cada aresta dirigida representa uma estrada que se pode percorrer em uma direção entre as interseções (as estradas de mão dupla seriam representadas por duas arestas distintas, em sentidos opostos). Tais grafos devem ter ciclos, senão, assim que você saísse de uma interseção, não poderia voltar a ela. Portanto, quando o seu GPS está calculando a rota mais curta ou mais rápida até um destino, o grafo com o qual ele trabalha tem ciclos, e muitos.

Quando o seu GPS determina a rota mais rápida entre a sua localização atual e um destino especificado, está resolvendo o **problema do caminho mínimo**. Para isso, o aparelho provavelmente usa um algoritmo que encontra todos os caminhos mínimos que partem de uma fonte única, mas o GPS só dá atenção ao caminho mínimo que ele encontra até o destino específico.

O seu GPS trabalha com um grafo dirigido ponderado, no qual os pesos das arestas representam distância ou tempo de viagem. Como não se pode dirigir por uma distância negativa nem chegar antes de partir, todos os pesos de arestas no seu grafo GPS são positivos. Suponho que alguns deles poderiam ser 0 por alguma razão insólita, portanto vamos dizer que os pesos das arestas são não negativos. Quando todos os pesos de arestas são não negativos, não temos de nos preocupar com ciclos de peso negativo, e todos os caminhos mínimos são bem definidos.

Como outro exemplo de caminhos mínimos de fonte única, considere o jogo dos “seis graus de Kevin Bacon”, no qual os jogadores tentam ligar atores de cinema a Kevin Bacon. Em um grafo, cada vértice representa um ator, e o grafo contém arestas (u, v) e (v, u) se os atores representados pelos vértices u e v já apareceram em um mesmo filme. Dado algum ator, um jogador tenta determinar o caminho mínimo do vértice para aquele ator até o vértice para Kevin Bacon. O número de arestas no caminho mínimo (em outras palavras, o peso do caminho mínimo quando cada peso de aresta é 1) é o “número de Kevin Bacon” do ator. Como exemplo, Renée Adorée atuou em um filme com Bessie Love, que apareceu em um filme com Eli Wallach, que fez um filme com Kevin Bacon e, portanto, o número de Kevin Bacon de Renée Adorée é 3.

Os matemáticos têm um conceito semelhante no número de Erdős, que dá o caminho mínimo entre o grande Paul Erdős e qualquer outro matemático por uma cadeia de relacionamentos entre coautores.¹

E o que dizer sobre grafos com arestas de peso negativo? Como estão relacionados com o mundo real? Veremos que podemos expressar o problema de determinar se existe uma oportunidade de arbitragem no câmbio de moedas determinando se um grafo que pode ter arestas de peso negativo tem um ciclo de peso negativo.

Em termos de algoritmos, em primeiro lugar exploraremos o algoritmo de Dijkstra para encontrar caminhos mínimos que partem de um único vértice-fonte e vão a todos os outros vértices. O algoritmo de Dijkstra funciona em grafos que têm duas importantes diferenças em relação aos grafos que vimos no Capítulo 5: todos os pesos de arestas devem ser não negativos, e o grafo pode conter ciclos. Ele está no cerne do modo de determinação de rotas do GPS. Examinaremos também algumas escolhas que podemos fazer quando implementamos o algoritmo de Dijkstra. Então, veremos o algoritmo de Bellman-Ford, um método extraordinariamente simples de determinar caminhos mínimos de fonte única, mesmo quando arestas de peso negativo estão presentes. Podemos usar o resultado do algoritmo de Bellman-Ford para determinar se o grafo contém um ciclo de peso negativo e, se contiver, identificar os vértices e arestas no ciclo. O algoritmo de Dijkstra e o algoritmo de Bellman-Ford datam do final de década de 1950, portanto saíram-se muito bem no teste do tempo. Encerraremos o capítulo com o algoritmo de Floyd-Warshall para o problema de todos os pares, no qual queremos encontrar um caminho mínimo entre todo par de vértices.

Exatamente como fizemos no Capítulo 5 para determinar caminhos mínimos em um dag, vamos pressupor que temos um vértice-fonte s e o peso $weight(u, v)$ de cada aresta (u, v) e que queremos calcular, para cada vértice v , o peso do caminho mínimo $sp(s, v)$ de s a v e o vértice que precede v em algum caminho mínimo que parte de s . Armazenaremos o resultado em $shortest[v]$ e $pred[v]$, respectivamente.

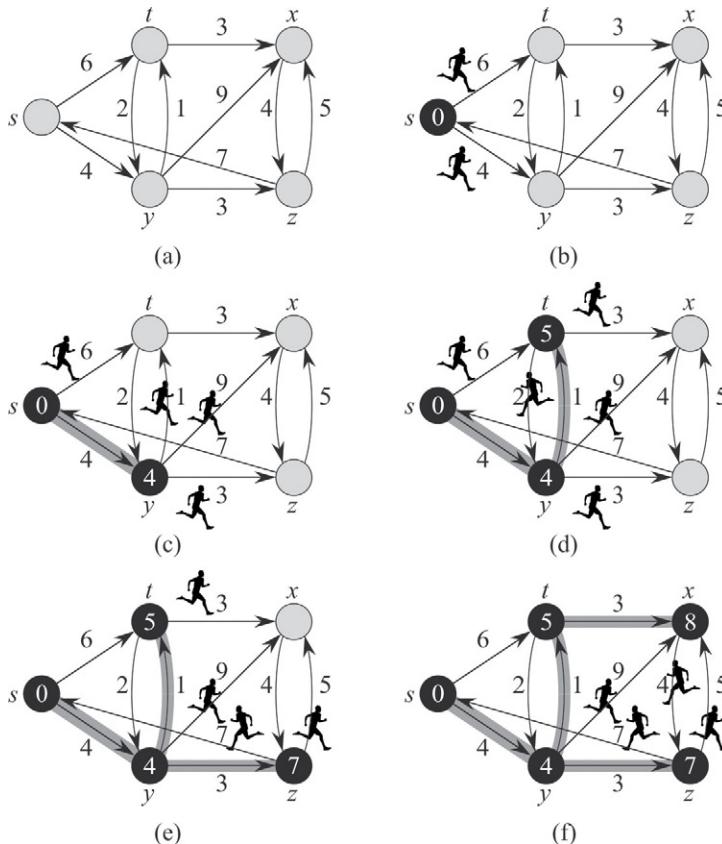
ALGORITMO DE DIJKSTRA

Eu gosto de pensar no algoritmo de Dijkstra² como uma simulação do envio de vários corredores pelo grafo.

Idealmente, a simulação funciona como descrevemos a seguir, se bem que veremos que o algoritmo de Dijkstra funciona de um modo ligeiramente diferente. Ele começa enviando corredores do vértice-fonte a todos os vértices adjacentes. Na primeira vez que um corredor chega a qualquer vértice, os corredores saem imediatamente daquele vértice e dirigem-se a todos os seus vértices adjacentes. Veja a parte (a) desta figura:

¹ Acredite ou não, existe até mesmo algo denominado número de Erdős-Bacon, que é a soma dos números de Erdős e de Bacon, e várias pessoas têm números de Erdős-Bacon finitos, inclusive o próprio Paul Erdős!

² O nome deve-se a Edsger Dijkstra, que propôs o algoritmo em 1959.



Elá mostra um grafo dirigido com vértice-fonte s e pesos ao lado das arestas. Pense no peso de uma aresta como o número de minutos que um corredor demoraria para percorrer a aresta.

A parte (b) ilustra o início da simulação, no tempo 0. Nesse tempo, mostrado dentro do vértice s , os corredores saem de s e se dirigem aos seus dois vértices adjacentes, t e y . O vértice s em preto indica que sabemos que $\text{shortest}[s] = 0$.

Quatro minutos mais tarde, no tempo 4, chega o corredor que vai ao vértice y , o que é mostrado na parte (c). Como esse corredor é o primeiro a chegar a y , sabemos que $\text{shortest}[y] = 4$ e, portanto, y está em preto na figura. A aresta sombreada (s, y) indica que o primeiro corredor a chegar ao vértice y veio do vértice s , portanto $\text{pred}[y] = s$. No tempo 4, o corredor que vem do vértice s e se dirige ao vértice t ainda está em trânsito, e os corredores saem do vértice y no tempo 4, dirigindo-se aos vértices y , y e y .

O próximo evento, representado na parte (d), ocorre um minuto depois, no tempo 5, quando o corredor que vem do vértice y chega ao vértice t . O corredor que vem de s a t ainda tem de chegar. Como o primeiro corredor a chegar ao vértice t veio do vértice y no tempo 5, igualamos $\text{shortest}[t]$ a 5 e $\text{pred}[t]$ a y (indicado pela aresta sombreada (y, t)). Os corredores partem do vértice t dirigindo-se aos vértices x e y nesse instante.

O corredor que vem do vértice s finalmente chega ao vértice t no tempo 6, porém o corredor que veio do vértice y já chegou lá um minuto antes, e portanto o esforço do corredor que veio de s a t de nada valeu.

No tempo 7, representado na parte (e), dois corredores chegam ao seu destino. O corredor que vem do vértice t ao vértice y chega, mas o corredor que veio de s a y já chegou no tempo 4, e portanto a simulação esquece do corredor que foi de t a y . Ao mesmo tempo, o corredor que vem de y chega ao vértice z . Igualamos $\text{shortest}[z]$ a 7 e $\text{pred}[z]$ a y , e os corredores saem do vértice z , a caminho dos vértices s e x .

O próximo evento ocorre no tempo 8, como mostrado na parte (f), quando o corredor que vem do vértice t chega ao vértice x . Igualamos $\text{shortest}[x]$ a 8 e $\text{pred}[x]$ a t , e um corredor sai do vértice x dirigindo-se ao vértice z .

Nesse ponto, um corredor já chegou a todos os vértices, e a simulação pode parar. Todos os corredores que ainda estão em trânsito chegarão a seus vértices de destino depois de algum outro corredor já ter chegado. Assim que um corredor chegou a todos os vértices, o valor shortest para cada vértice é igual ao peso do caminho mínimo que vem do vértice s , e o valor pred para cada vértice é o predecessor em um caminho mínimo que vem de s .

É assim que a simulação ocorre idealmente. Ela confiou que o tempo para um corredor percorrer uma aresta é igual ao peso da aresta. O algoritmo de Dijkstra funciona de modo ligeiramente diferente. Ele trata todas as arestas do mesmo jeito, de modo que, quando considera as arestas que saem de um vértice, ele processa os vértices adjacentes junto, sem nenhuma ordem particular. Por exemplo, quando o algoritmo de Dijkstra processa as arestas que saem do vértice s na figura da página 81, ele declara que $\text{shortest}[y]=4$, $\text{shortest}[t]=6$ e $\text{pred}[y]$ e $\text{pred}[t]$ são s até aqui. Quando o algoritmo de Dijkstra considerar, mais adiante, a aresta (y, t) , ele diminuirá o peso do caminho mínimo até o vértice t que encontrou até então, de modo que $\text{shortest}[t]$ vai de 6 para 5 e $\text{pred}[t]$ troca de s para y .

O algoritmo de Dijkstra funciona chamando o procedimento RELAX, da página 86, uma vez por aresta. Relaxar uma aresta (u, v) corresponde a um corredor que vem do vértice u chegar ao vértice v . O algoritmo mantém um conjunto Q de vértices cujos valores finais de shortest e pred ainda não são conhecidos; todos os vértices que não estão em Q têm seus valores finais shortest e pred . Depois de inicializar $\text{shortest}[s]$ para 0 para o vértice-fonte s , $\text{shortest}[v]$ para ∞ para todos os outros vértices e $\text{pred}[v]$ para NULL para todos os vértices, ele acha repetidamente o vértice u no conjunto Q que tem o valor shortest mais baixo, remove esse vértice de Q e relaxa todas as arestas que saem de u . Eis o procedimento:

Procedimento DIJKSTRA (G, s)

Entradas:

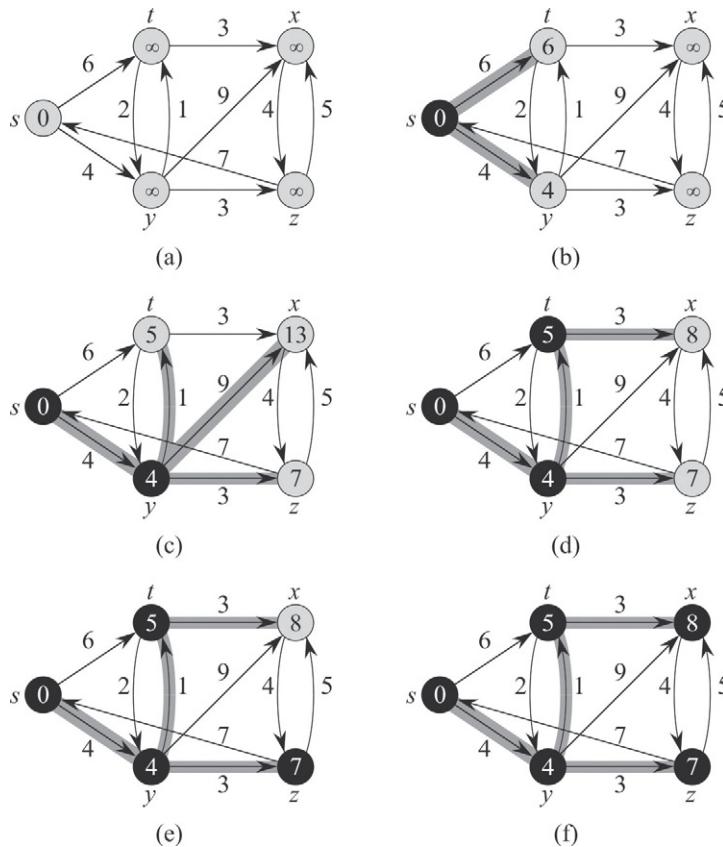
- G : um grafo dirigido que contém um conjunto V de n vértices e um conjunto E de m arestas dirigidas com pesos não negativos.
- s : um vértice-fonte em V .

Resultado: Para cada vértice v em V , exceto o vértice-fonte, $\text{shortest}[v]$ é o peso $sp(s, v)$ de um caminho mínimo de s a v e $\text{pred}[v]$ é o vértice que precede v em algum caminho mínimo. Para o vértice-fonte s , $\text{shortest}[s]=0$ e $\text{pred}[s]=\text{NULL}$. Se não houver nenhum caminho

de s a v , $\text{shortest}[v] = \infty$ e $\text{pred}[s] = \text{NULL}$ (o mesmo que DAG-SHORTEST-PATHS na página 75).

1. Igual $\text{shortest}[v]$ a ∞ para cada vértice v exceto s , iguale $\text{shortest}[s]$ a 0 e iguale $\text{pred}[v]$ a NULL para cada vértice v .
2. Ajuste Q para conter todos os vértices.
3. Enquanto Q não estiver vazio, faça o seguinte:
 - a. Ache o vértice u no conjunto Q que tem o valor shortest mais baixo e remova-o de Q .
 - b. Para cada vértice v adjacente a u :
 - i. Chame RELAX(u, v).

Na figura a seguir, cada parte mostra o valor shortest (que aparece dentro de cada vértice), o valor pred (indicado pelas arestas sombreadas) e o conjunto Q (os vértices que estão sombreados, não negros) um pouco antes de cada iteração do laço na etapa 3.



O vértice que acabou de ser pintado de negro em cada parte da figura é o vértice escolhido como vértice u na etapa 3A. Na simulação com corredores, tão logo um vértice recebe valores shortest e pred , eles não podem mais mudar, mas aqui um vértice poderia receber valores shortest e pred por ter relaxado uma aresta, e uma relaxação

mais adiante de alguma outra aresta poderia mudar esses valores. Por exemplo, depois de a aresta (y, x) ser relaxada na parte (c) da figura, o valor de $\text{shortest}[x]$ diminui de ∞ para 13 e $\text{pred}[x]$ torna-se y . A próxima iteração do laço na etapa 3 (parte (d)) relaxa a aresta (t, x) , $\text{shortest}[x]$ decresce ainda mais, até 8, e $\text{pred}[x]$ torna-se t . Na próxima iteração (parte (e)), a aresta (z, x) é relaxada, mas dessa vez $\text{shortest}[x]$ não muda porque seu valor, 8, é menor que $\text{shortest}[z] + \text{weight}(z, x)$, que é igual a 12.

O algoritmo de Dijkstra mantém a seguinte invariante de laço:

No início de cada iteração do laço na etapa 3, $\text{shortest}[v] = sp(s, v)$ para cada vértice v que não está no conjunto Q . Isto é, para cada vértice v que não está em Q , o valor de $\text{shortest}[v]$ é o peso de um caminho mínimo de s a v .

Damos aqui uma versão simplificada do raciocínio que fundamenta essa invariante de laço (uma prova formal é um pouco mais complicada). Inicialmente, todos os vértices estão no conjunto Q ; portanto, a invariante de laço não se aplica a nenhum vértice quando entra na primeira iteração do laço na etapa 3. Suponha que, ao entrarmos em uma iteração desse laço, os pesos dos caminhos mínimos de todos os vértices que não estão no conjunto Q estejam corretos e seus valores são shortest . Então, toda aresta que sai desses vértices foi relaxada em alguma execução da etapa 3Bi. Considere o vértice u em Q , que tem o menor valor shortest . Seu valor shortest nunca mais poderá ser diminuído. Por que não? Porque as únicas arestas que ainda têm de ser relaxadas são arestas que saem dos vértices em Q , e todo vértice em Q tem um valor shortest no mínimo tão grande quanto $\text{shortest}[u]$. Visto que todos os pesos de arestas são não negativos, devemos ter $\text{shortest}[u] \leq \text{shortest}[v] + \text{weight}(v, u)$ para todo vértice v em Q e, assim, nenhuma etapa de relaxação futura diminuirá $\text{shortest}[u]$. Portanto, $\text{shortest}[u]$ é tão baixo quanto pode ser, e podemos remover o vértice u do conjunto Q e relaxar todas as arestas que saem de u . Quando o laço da etapa 3 terminar, o conjunto Q estará vazio; portanto, os valores de todos os pesos de caminhos mínimos corretos são shortest .

Podemos começar a analisar o tempo de execução do procedimento DIJKSTRA; porém, antes de analisá-lo totalmente, teremos de decidir alguns detalhes de implementação. Lembre-se de que no Capítulo 5 denotamos o número de vértices por n e o número de arestas por m , e $m \leq n^2$. Sabemos que a etapa 1 leva tempo $\Theta(n)$. Sabemos também que o laço da etapa 3 itera exatamente n vezes porque o conjunto Q inicialmente contém todos os n vértices, cada iteração do laço remove um vértice de Q , e os vértices nunca voltam a ser adicionados a Q . O laço da etapa 3A processa cada vértice e cada aresta exatamente uma vez no curso do algoritmo (vimos a mesma ideia nos procedimentos TOPOLOGICAL-SORT e DAG-SHORTEST-PATHS no Capítulo 5).

O que ainda falta a analisar? Precisamos entender quanto tempo leva para colocar todos os n vértices no conjunto Q (etapa 2), quanto tempo leva para determinar qual vértice em Q tem o valor shortest mais baixo e remover esse vértice de Q (etapa 3A) e quais ajustes de contabilidade precisamos fazer (se houver algum) quando os valores shortest e pred de um vértice mudam por terem chamado RELAX. Vamos nomear essas operações:

- $\text{INSERT}(Q, v)$ insere o vértice v no conjunto Q (o algoritmo de Dijkstra chama $\text{INSERT } n$ vezes).

- EXTRACT-MIN (Q) remove o vértice em Q que tem o mínimo valor de shortest e retorna esse vértice ao seu chamador (o algoritmo de Dijkstra chama EXTRACT-MIN n vezes).
- DECREASE-KEY (Q, v) executa qualquer contabilidade necessária em Q para registrar que $\text{shortest}[v]$ foi diminuído de uma chamada a RELAX (o algoritmo de Dijkstra chama DECREASE-KEY até m vezes).

Essas três operações, tomadas em conjunto, definem uma **fila de prioridade**.

As descrições das operações de fila de prioridade dizem apenas *o que* as operações fazem, não *como* elas fazem. Em projeto de software, separar *o que* as operações fazem de *como* fazem é conhecido como **abstração**. Denominamos o conjunto de operações, especificadas por *o que* elas fazem, mas não *como* elas fazem, um **tipo de dado abstrato**, ou **TDA**, de modo que uma fila de prioridade é um **TDA**.

Podemos implementar as operações de fila de prioridade — a parte do *como* — por uma de várias estruturas de dados. Uma **estrutura de dados** é um modo específico de armazenar e acessar dados em um computador — por exemplo, um vetor. No caso de filas de prioridade, veremos três estruturas de dados diferentes que podem implementar as operações. Projetistas de software devem ser capazes de inserir qualquer estrutura de dados que implemente as operações de um TDA. Mas isso não é tão simples quando se trata de algoritmos porque, para diferentes estruturas de dados, o modo como elas implementam as operações podem levar quantidades de tempo diferentes. De fato, as três estruturas de dados diferentes que veremos para implementar a fila de prioridade TDA dão tempos de execução diferentes para o algoritmo de Dijkstra.

Uma versão reescrita do procedimento DIJKSTRA, que chama explicitamente as operações de fila de prioridade, aparece logo a seguir. Vamos examinar as três estruturas de dados para implementar operações de fila de prioridade e ver como elas afetam o tempo de execução do algoritmo de Dijkstra.

Implementação de vetor simples

O modo mais simples de implementar as operações de fila de prioridade é armazenar os vértices em um vetor com n posições. Se a fila de prioridade contém k vértices no momento em questão, então eles estão nas k primeiras posições do vetor, em nenhuma ordem particular. Juntamente com o vetor, precisamos manter uma contagem do número de vértices que o vetor contém no momento em questão. A operação INSERT é fácil: basta adicionar o vértice à próxima posição não usada no vetor e incrementar a contagem. DECREASE-KEY é ainda mais fácil: nada a fazer!

Procedimento DIJKSTRA (G, s)

Entradas e Resultado: Os mesmos de antes.

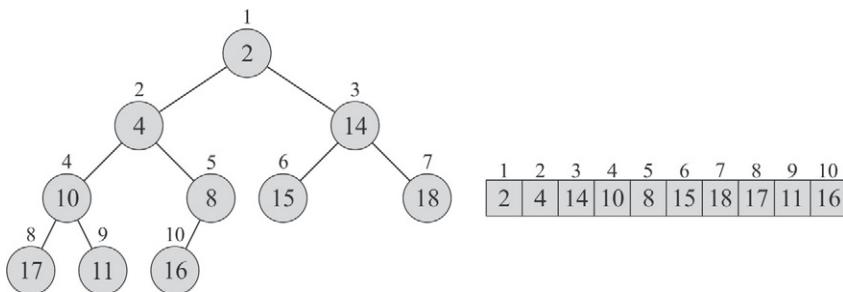
1. Igual $\text{shortest}[v]$ a ∞ para cada vértice v exceto s , iguale $\text{shortest}[s]$ a 0 e iguale $\text{pred}[v]$ a NULL para cada vértice v .
2. Faça (Q, v) uma fila de prioridade vazia.
3. Para cada vértice v :
 - a. Chame INSERT (Q, v).
4. Enquanto Q não estiver vazio, faça o seguinte:
 - a. Chame EXTRACT-MIN (Q) e determine que u contenha o vértice retornado.

- b. Para cada vértice v adjacente a u :
- Chame $\text{RELAX}(u, v)$.
 - Se a chamada a $\text{RELAX}(u, v)$ diminuir o valor de $\text{shortest}[v]$, chame $\text{DECREASE-KEY}(Q, v)$.

As duas operações levam tempo constante. Todavia, a operação EXTRACT-MIN leva tempo $O(n)$, visto que temos de examinar todos os vértices que estão no vetor no momento em questão para achar o que tem o valor *shortest* mais baixo. Uma vez identificado esse vértice, eliminá-lo é fácil: basta mover o vértice que está na última posição para a posição do vértice eliminado e decrementar a contagem. As n chamadas EXTRACT-MIN levam tempo $O(n^2)$. Embora as chamadas a RELAX levem tempo $O(m)$, lembre-se de que $m \leq n^2$. Portanto, com essa implementação da fila de prioridade, o algoritmo de Dijkstra leva tempo $O(n^2)$, sendo que o tempo é dominado pelo tempo gasto em EXTRACT-MIN.

Implementação de heap binário

Um heap binário organiza dados como uma árvore binária armazenada em um vetor. Uma **árvore binária** é um tipo de grafo, porém nós nos referimos a seus vértices como **nós**, as arestas são não dirigidas e cada nó tem 0, 1 ou 2 nós abaixo dele, que são denominados seus **filhos**. No lado esquerdo da figura a seguir damos um exemplo de árvore binária, com os nós numerados. Nós que não têm nenhum filho, como os nós 6 a 10, são *folhas*.³



Um **heap binário** é uma árvore binária com três propriedades adicionais. A primeira é que a árvore é completamente preenchida em todos os níveis, exceto possivelmente o mais baixo, que é preenchido da esquerda até um ponto. A segunda é que cada nó contém uma chave, mostrada dentro de cada nó na figura. A terceira é que as chaves obedecem à **propriedade do heap**: a chave de cada nó é menor ou igual às chaves de seus filhos. A árvore binária na figura é também um heap binário.

³ Cientistas da computação acham mais fácil desenhar árvores com a raiz no topo e os ramos estendendo-se para baixo, em vez de desenhá-las como árvores reais, com a raiz na parte inferior e os ramos dirigindo-se para cima.

Podemos armazenar um heap binário em um vetor, como mostrado à direita na figura. Em razão da propriedade do heap, o nó que tem a chave mínima deve sempre estar na posição 1. Os filhos do nó na posição i estão nas posições $2i$ e $2i + 1$, e o nó acima do nó na posição i — seu **pai** — está na posição $i/2$. É fácil percorrer um heap binário para cima e para baixo quando o armazenamos em um vetor.

Um heap binário tem outra característica importante: se ele consistir em n nós, sua **altura** — o número de arestas da raiz até a folha mais longínqua — é apenas $\lfloor \lg n \rfloor$. Portanto, podemos percorrer um caminho da raiz até uma folha ou de uma folha até a raiz no tempo de apenas $O(\lg n)$.

Como os heaps binários têm altura $\lfloor \lg n \rfloor$, podemos executar as três operações de fila de prioridade no tempo $O(\lg n)$ cada. Para INSERT, adicione uma nova folha na primeira posição disponível. Então, desde que a chave no nó seja maior que a chave em seu pai, troque o conteúdo⁴ do nó pelo conteúdo de seu pai e suba um nível em direção à raiz. Em outras palavras, faça a “bolha” do conteúdo subir em direção à raiz até a propriedade de heap valer. Visto que o caminho até a raiz tem no máximo $\lfloor \lg n \rfloor$ arestas, ocorrem no máximo $\lfloor \lg n \rfloor - 1$ trocas e, portanto, INSERT leva o tempo $O(\lg n)$. Para executar DECREASE-KEY, use a mesma ideia: decremente a chave e faça a bolha de conteúdo subir em direção à raiz até a propriedade do heap se manter, novamente levando tempo $O(\lg n)$. Para executar EXTRACT-MIN, salve o conteúdo da raiz para retornar ao chamador. Em seguida, tome a última folha (o nó de número mais alto) e coloque seu conteúdo na posição da raiz. Então faça a bolha do conteúdo “descer” o conteúdo da raiz, trocando o conteúdo do nó e do filho cuja chave é a menor até a propriedade do heap valer. Finalmente, retorne o conteúdo salvado da raiz. Novamente, como o caminho descendente da raiz até uma folha tem no máximo $\lfloor \lg n \rfloor$ arestas, ocorrem no máximo $\lfloor \lg n \rfloor - 1$ trocas e, portanto, EXTRACT-MIN leva o tempo $O(\lg n)$.

Quando o algoritmo de Dijkstra usa implementação de heap binário de uma fila de prioridade, gasta o tempo $O(n \lg n)$ inserindo vértices, o tempo $O(n \lg n)$ em operações EXTRACT-MIN e $O(m \lg n)$ em operações DECREASE-KEY. (Na verdade, inserir os n vértices leva apenas o tempo $\Theta(n)$, visto que inicialmente apenas o vértice-fonte s tem valor *shortest* de 0 e todos os outros vértices têm valores *shortest* de ∞ .) Quando o grafo é **esparsa** — o número m de arestas é muito menor que n^2 —, implementar a fila de prioridade com um heap binário é mais eficiente que usar um vetor simples. Grafos que modelam redes rodoviárias são esparsos, visto que de uma interseção média partem somente quatro estradas e, portanto, m seria aproximadamente $4n$. Por outro lado, quando o grafo é **denso** — m é próximo de n^2 , de modo que o grafo contém muitas arestas —, o tempo $O(m \lg n)$ que o algoritmo de Dijkstra gasta em chamadas DECREASE-KEY pode torná-lo mais lento que usar um vetor simples para a fila de prioridade.

Outra coisa sobre heaps binários: podemos usá-los para ordenar no tempo $O(n \lg n)$:

⁴ O conteúdo de um nó inclui a chave e qualquer outra informação associada à chave, por exemplo, qual vértice está associado a esse nó.

Procedimento HEAPSORT (A, n)

Entradas:

- A : um vetor.
- n : o número de elementos em A a ordenar.

Saída: Um vetor B contendo os elementos de A , ordenados.

1. Construa um heap binário Q com os elementos de A .
 2. Seja $b[1..n]$ um novo vetor.
 3. Para $i = 1$ a n :
 - a. Chame EXTRACT-MIN(Q) e iguale $B[i]$ ao valor retornado.
 4. Retorne o vetor B .
-

A etapa 1 converte o vetor de entrada em um heap binário, o que podemos fazer de um de dois modos. Um modo é iniciar com um heap binário vazio e inserir cada elemento do vetor, levando o tempo $O(n \lg n)$. O outro modo constrói o heap binário diretamente dentro do vetor, trabalhando de baixo para cima, levando somente o tempo $O(n)$. Também é possível ordenar usando um heap no lugar, de modo que não precisamos do vetor B extra.

Implementação de heap de Fibonacci

Podemos também implementar uma fila de prioridade por uma estrutura de dados complicada denominada “heap de Fibonacci” ou “heap F”. Com um heap F, as n chamadas INSERT e EXTRACT-MIN levam um tempo total $O(n \lg n)$, as m chamadas DECREASE-KEY levam um tempo total $\Theta(m)$ e, portanto, o algoritmo de Dijkstra leva somente o tempo $O(n \lg n + m)$. Na prática, as pessoas não usam frequentemente heaps F, por um par de razões. Uma é que uma operação individual poderia levar muito mais tempo que a média, embora no total as operações levem os tempos que acabamos de citar. A segunda razão é que heaps F são um pouco complicados; portanto, os fatores constantes ocultos na notação assintótica não são tão bons quanto para heaps binários.

O ALGORITMO DE BELLMAN-FORD

Se alguns pesos de arestas forem negativos, o algoritmo de Dijkstra poderá retornar resultados incorretos. O algoritmo de Bellman-Ford⁵ pode manipular pesos de arestas negativos, e podemos usar sua saída para detectar e ajudar a identificar um ciclo de peso negativo.

O algoritmo de Bellman-Ford é extraordinariamente simples. Depois de inicializar os valores $shortest$ e $pred$, ele apenas relaxa todas as m arestas $n - 1$ vezes. O procedimento aparece na próxima página, e a figura abaixo dele demonstra como o algoritmo funciona em um grafo pequeno. O vértice-fonte é s , os valores $shortest$ aparecem dentro dos vértices, e as arestas sombreadas indicam valores $pred$: se a aresta (u, v) é sombreada, então $pred[v] = u$. Nesse exemplo, consideramos que cada passagem sobre todas as arestas as relaxa na ordem fixa $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. A parte (a) mostra a situação um pouco antes da primeira passagem, e

⁵ Baseado nos algoritmos separados de Richard Bellman, de 1958, e de Lester Ford, de 1962.

as partes (b) a (e) mostram a situação depois de cada passagem sucessiva. Os valores *shortest* e *pred* na parte (e) são os valores finais.

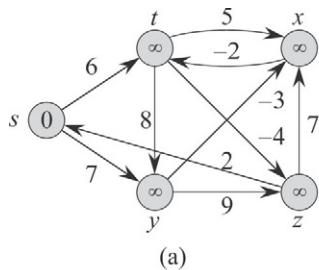
Procedimento BELLMAN-FORD (G, s)

Entradas:

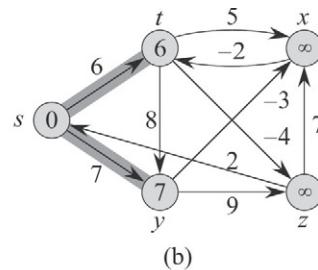
- G : um grafo dirigido contendo um conjunto V de n vértices e um conjunto E de m arestas dirigidas com pesos arbitrários.
- s : um vértice-fonte em V .

Resultado: O mesmo de DIJKSTRA (página 85).

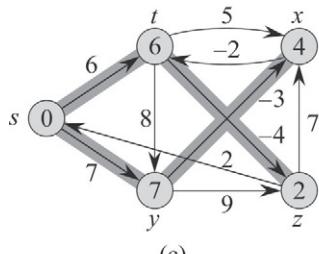
1. Igual $\text{shortest}[v]$ a ∞ para cada vértice v exceto s , iguale $\text{shortest}[s]$ a 0 e $\text{pred}[v]$ a NULL para cada vértice s .
2. Para $i = 1$ a $n - 1$:
 - a. Para cada aresta (u, v) em E :
 - i. Chame RELAX(u, v).



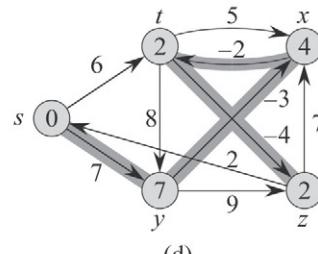
(a)



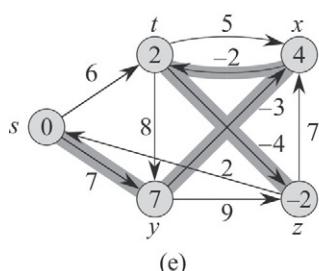
(b)



(c)



(d)



(e)

Como um algoritmo tão simples pode produzir a resposta correta? Considere um caminho mínimo da fonte s até qualquer vértice v . Lembre-se de que dissemos, na página 75, que, se relaxarmos as arestas, em ordem, ao longo de um caminho mínimo

de s a v , $\text{shortest}[v]$ e $\text{pred}[v]$ estarão corretos. Agora, se não forem permitidos ciclos de peso negativo, há sempre um caminho mínimo de s a v que não contém um ciclo. Por quê? Suponha que um caminho mínimo de s a v contenha um ciclo. Como o ciclo deve ter peso não negativo, poderíamos extirpar o ciclo e acabar tendo em mãos um caminho de s a v cujo peso não é mais alto que o do caminho que contém o ciclo. Todo caminho acíclico deve conter no máximo $n - 1$ arestas porque, se um caminho contiver n arestas, ele deve visitar alguma vértice duas vezes, o que constituirá um ciclo. Assim, se houver um caminho mínimo de s a v , haverá um que contém no máximo $n - 1$ arestas. Na primeira vez que a etapa 2A relaxa todas as arestas, ela deve relaxar a primeira aresta nesse caminho mínimo. A segunda vez que a etapa 2A relaxa todas as arestas, ela deve relaxar a segunda aresta no caminho mínimo, e assim por diante. Depois do tempo $(n - 1)st$, todas as arestas no caminho mínimo foram relaxadas, em ordem, e portanto $\text{shortest}[v]$ e $\text{pred}[v]$ estão corretos. Bem esperto!

Agora suponha que o grafo contenha um ciclo de peso negativo e que já executamos o procedimento BELLMAN-FORD nele. Você pode repetir várias vezes um ciclo de peso negativo obtendo um caminho de peso mais baixo a cada vez. Isso significa que há, no mínimo, uma aresta (u, v) no ciclo para a qual $\text{shortest}[v]$ decrescerá se você a relaxar novamente — mesmo que essa aresta já tenha sido relaxada $n - 1$ vezes.

Portanto, mostramos agora como encontrar um ciclo de peso negativo, se existir algum, depois de executar BELLMAN-FORD. Percorra as arestas mais uma vez. Se encontrarmos uma aresta (u, v) para a qual $\text{shortest}[u] + \text{weight}(u, v) < \text{shortest}[v]$, então sabemos que o vértice v está em um ciclo de peso negativo ou pode ser alcançado de um ciclo de peso negativo. Podemos encontrar um vértice no ciclo de peso negativo percorrendo de trás para diante os valores pred de v , mantendo um registro dos vértices que já visitamos até alcançarmos um vértice x que tínhamos visitado antes. Então podemos percorrer os valores pred de x de trás para a frente, até voltarmos a x , e todos os vértices entre eles, juntamente com x , constituirão um ciclo de peso negativo. O procedimento FIND-NEGATIVE-WEIGHT CYCLE mostra como determinar se um grafo tem um ciclo de peso negativo e como construir nesse caso.

Procedimento FIND-NEGATIVE-WEIGHT-CYCLE (G)

Entrada: (G): um grafo dirigido contendo um conjunto V de n vértices e um conjunto E de m arestas dirigidas com pesos arbitrários nas quais o procedimento BELLMAN-FORD já foi executado.

Resultado: Uma lista de vértices em um ciclo de peso negativo, em ordem, ou uma lista vazia se o grafo não tiver nenhum ciclo de peso negativo.

1. Percorra todas as arestas para encontrar qualquer aresta (u, v) tal que $\text{shortest}[u] + \text{weight}(u, v) < \text{shortest}[v]$
2. Se tal aresta existir, retorne uma lista vazia.
3. Caso contrário (há alguma (u, v) para a qual $\text{shortest}[u] + \text{weight}(u, v) < \text{shortest}[v]$), faça o seguinte:
 - a. Seja visited um novo vetor com um elemento para cada vértice.
 - Iguale todos os elementos de visited a FALSE.
 - b. Iguale x a v .
 - c. Enquanto $\text{visited}[x]$ é FALSE, faça o seguinte:
 - i. Iguale $\text{visited}[x]$ a TRUE.
 - ii. Iguale x a $\text{pred}[x]$

- d. Nesse ponto, sabemos que x é um vértice em um ciclo de peso negativo. Igual v a $\text{pred}[x]$.
 - e. Crie uma lista *cycle* de vértices contendo inicialmente apenas x .
 - f. Enquanto v não é x , faça o seguinte:
 - i. Insira o vértice v no início de *cycle*.
 - ii. Igual v a $\text{pred}[v]$.
 - g. Retorne *cycle*.
-

É fácil analisar o tempo de execução do algoritmo de Bellman-Ford. O laço da etapa 2 itera $n - 1$ vezes e, a cada vez que ele executa, o laço da etapa 2A itera m vezes, uma vez por aresta. Portanto, o tempo total de execução, é $\Theta(nm)$. Para descobrir se um ciclo de peso negativo existe, relaxe cada aresta mais uma vez até que o relaxamento altere um valor *shortest* ou até todas as arestas terem sido relaxadas, o que leva tempo $O(m)$. Se houver um ciclo de peso negativo, ele pode conter no máximo n arestas e, portanto, o tempo para rastreá-lo é $\Theta(n)$.

No início deste capítulo, prometi mostrar como os ciclos de peso negativo estão relacionados com a arbitragem de oportunidades de câmbio de moedas. As taxas de câmbio para moedas flutuam rapidamente. Imagine que, em algum momento, as seguintes taxas de câmbio estejam em vigor:

- 1 dólar americano compra 0,7292 euro
- 1 euro compra 105,374 ienes japoneses
- 1 iene japonês compra 0,3931 rublo russo
- 1 rublo russo compra 0,0341 dólar americano.

Então, você poderia pegar um dólar americano, comprar 0,7292 euro com ele, pegar o 0,7292 euro e comprar 76,8387 ienes (porque $0,7292 \cdot 105,374 = 76,8387$, com quatro casas decimais), tomar os 76,8387 ienes e comprar 30,2053 rublos (porque $76,8387 \cdot 0,3931 = 30,2053$, com quatro casas decimais) e, finalmente, pegar os 30,2053 rublos e comprar 1,03 dólar (porque $30,2053 \cdot 0,0341 = 1,0300$, com quatro casas decimais). Se você pudesse executar todas as quatro transações antes de a taxa de câmbio mudar, poderia obter 3% de retorno para o seu investimento de um dólar. Se começasse com um milhão de dólares, poderia ter um lucro de 30 mil dólares sem fazer nada!

Tal cenário é uma *oportunidade de arbitragem*. Eis como descobrir uma oportunidade de arbitragem encontrando um ciclo de peso negativo. Suponha que você esteja examinando n moedas $c_1, c_2, c_3, \dots, c_n$ e que tenha todas as taxas de câmbio entre pares de moedas. Suponha que com uma unidade da moeda c_i você possa comprar r_{ij} unidades da moeda c_j , de modo que r_{ij} é a taxa de câmbio entre as moedas c_i e c_j . Aqui, tanto i quanto j variam de 1 a n (presumimos que $r_{ii} = 1$ para cada moeda c_i).

Uma oportunidade de arbitragem corresponderia a uma sequência de k moedas $\langle c_{j_1}, c_{j_2}, c_{j_3}, \dots, c_{j_k} \rangle$ tal que, quando você multiplicar pelas taxas de câmbio, obterá um produto estritamente maior que 1:

$$r_{j_1, j_2} \cdot r_{j_2, j_3} \cdots r_{j_{k-1}, j_k} \cdot r_{j_k, j_{k+1}} > 1.$$

Agora tome logaritmos de ambos os lados. Não importa qual base usamos, vamos fazer o que os cientistas da computação fazem e usar 2. Como o logaritmo de um

produto é a soma dos logaritmos individuais — isto é, $\lg(x \cdot y) = \lg x + \lg y$ —, temos uma situação na qual

$$\lg r_{j_1, j_2} + \lg r_{j_2, j_3} + \cdots + \lg r_{j_{k-1}, j_k} + \lg r_{j_k, j_{k+1}} > 0.$$

Negativando ambos os lados dessa desigualdade, temos

$$(-\lg r_{j_1, j_2}) + (-\lg r_{j_2, j_3}) + \cdots + (-\lg r_{j_{k-1}, j_k}) + (-\lg r_{j_k, j_{k+1}}) > 0$$

que corresponde a um ciclo com pesos de arestas que são os negativos dos logaritmos das taxas de câmbio.

Para encontrar uma oportunidade de arbitragem, se existir alguma, construa um grafo dirigido com um vértice v_i para cada moeda c_i . Para cada par de moedas c_i e c_j , crie arestas dirigidas (v_i, v_j) e (v_j, v_i) com pesos $-\lg r_{ij}$ e $-\lg r_{ji}$, respectivamente. Adicione um novo vértice s com uma aresta de peso 0 (s, v_i) a cada um dos vértices v_1 a v_n . Execute o algoritmo de Bellman-Ford nesse grafo tendo s como o vértice-fonte e use o resultado para determinar se ele contém um ciclo de peso negativo. Se contiver, os vértices nesse ciclo corresponderão às moedas em uma oportunidade de arbitragem. O número total de arestas m é $n + n(n - 1) = n^2$, e portanto o algoritmo de Bellman-Ford leva tempo $O(n^3)$, mais outro $O(n^2)$ para descobrir se há um ciclo de peso negativo, e mais outro $O(n)$ para rastreá-lo, caso ele exista. Embora o tempo $O(n^3)$ pareça lento, na prática não é tão ruim porque os fatores constantes nos tempos de execução dos laços são baixos. Codifiquei o programa de arbitragem no meu MacBook Pro de 2,4 GHz e o executei para 182 moedas, que é o total de moedas que há no mundo inteiro. Uma vez carregadas as taxas de câmbio (escolhi valores aleatórios para as taxas de câmbio), o programa levou aproximadamente 0,02 segundo para executar.

O ALGORITMO DE FLOYD-WARSHALL

Agora suponha que você queira descobrir um caminho mínimo de cada vértice a cada outro vértice. Esse é o problema do *caminho mínimo para todos os pares*.

O exemplo clássico do caminho mínimo para todos os pares — ao qual já vi vários autores se referirem — é a tabela que você vê em atlas e guias rodoviários que dão as distâncias entre várias cidades. Você acha a linha para uma cidade e a coluna para a outra cidade, e a distância entre elas está na interseção da linha com a coluna.

Há um problema com esse exemplo: *não é para todos os pares*. Se fosse para todos os pares, a tabela teria de ter uma linha e uma coluna para toda interseção, e não apenas para toda cidade. Então, o número de linhas e colunas só para os Estados Unidos estaria na casa dos milhões. Não, o modo de fazer uma tabela que você vê em um atlas é encontrar caminhos mínimos de fonte única partindo de cada cidade e colocar um subconjunto dos resultados — caminhos mínimos para apenas as outras cidades, e não para todas as interseções — na tabela.

Qual seria uma aplicação legítima de caminhos mínimos para todos os pares? Determinar o *diâmetro* de uma rede: o mais longo de todos os caminhos mínimos.

Por exemplo, suponha que um grafo dirigido represente uma rede de comunicação e que o peso de uma aresta dá o tempo que leva para uma mensagem percorrer um link de comunicação. Então, o diâmetro dá o tempo de trânsito mais longo possível para uma mensagem na rede.

É claro que podemos computar caminhos mínimos para todos os pares computando caminhos mínimos de fonte única de cada vértice por vez. Se todos os pesos de arestas forem não negativos, poderemos executar o algoritmo de Dijkstra para cada um dos n vértices, sendo que cada chamada levará o tempo $O(m \lg n)$ se usarmos um heap binário ou o tempo $O(n \lg n + m)$ se usarmos um heap de Fibonacci, para um tempo de execução total de $O(nm \lg n)$ ou $O(n^2 \lg n + nm)$. Se o grafo for esparsa, essa abordagem funcionará bem. Mas, se o grafo for denso, de modo que m esteja próximo de n^2 , $O(nm \lg n)$ será $O(n^3 \lg n)$. Mesmo com um grafo denso e um heap de Fibonacci, $O(n^2 \lg n + mn)$ é $O(n^3)$, e o fator constante induzido do heap de Fibonacci pode ser significativo. É claro que, se o grafo puder conter arestas de peso negativo, não poderemos usar o algoritmo de Dijkstra, e o tempo de execução do algoritmo de Bellman-Ford a partir de cada um de n vértices em um grafo denso dará um tempo de execução $\Theta(n^2m)$, que é $\Theta(n^4)$.

Em vez disso, usando o algoritmo de Floyd-Warshall,⁶ podemos resolver o problema de todos os pares no tempo $\Theta(n^3)$ — independentemente de o grafo ser esparsa, denso ou entre esses dois, e até permitir que o grafo tenha arestas de peso negativo, porém nenhum ciclo de peso negativo — e o fator constante oculto na notação Θ é pequeno. Além disso, o algoritmo de Floyd-Warshall ilustra uma técnica algorítmica engenhosa denominada “programação dinâmica”.

O algoritmo de Floyd-Warshall recorre a uma propriedade óbvia de caminhos mínimos. Suponha que você esteja indo de carro da cidade de Nova York a Seattle ao longo de uma rota mais curta e que essa rota mais curta de Nova York a Seattle passe por Chicago e depois por Spokane antes de chegar a Seattle. Então, a porção da rota mais curta de Nova York a Seattle que passa por Chicago deve ser em si a rota mais curta de Chicago a Spokane. Por quê? Porque, se houvesse uma rota mais curta de Chicago a Spokane, nós a teríamos usado na rota mais curta de Nova York a Seattle! Como eu disse, óbvio. Para aplicar esse princípio em grafos dirigidos:

Se um caminho mínimo, denominado p , do vértice u ao vértice v for do vértice u ao vértice x , ao vértice y , ao vértice v , então a porção de p que está entre x e y é em si um caminho mínimo de x a y . Isto é, qualquer subcaminho de um caminho mínimo é em si caminho mínimo.

O algoritmo de Floyd-Warshall mantém registro dos pesos de caminhos e predecessores de vértices em vetores indexados não somente em uma dimensão, mas em três. Você pode imaginar um vetor unidimensional como uma tabela, exatamente como a que vimos na página 10. Um vetor bidimensional seria como uma matriz, tal como a matriz de adjacência na página 68; você precisa de dois índices (linha e coluna) para identificar uma entrada. Você também pode pensar em um vetor bidimensional como um vetor unidimensional no qual cada entrada é em si um vetor unidimensional. Um

⁶ O nome deve-se a Robert Floyd e Stephen Warshall.

vetor tridimensional seria como um vetor unidimensional de dois vetores bidimensionais; você precisa de um índice em cada uma das três dimensões para identificar uma entrada. Usaremos vírgulas para separar as dimensões quando indexarmos um vetor multidimensional.

No algoritmo de Floyd-Warshall, consideramos que os vértices são numerados de 1 a n . Números de vértices tornam-se importantes porque o algoritmo de Floyd-Warshall usa a seguinte definição:

shortest[u, v, x] é o peso de um caminho mínimo do vértice u ao vértice v no qual cada vértice intermediário — um vértice no caminho que não seja u e v — é numerado de 1 a x .

(Portanto, imagine u , v e x como inteiros na faixa 1 a n que representam vértices.) Essa definição não requer que os vértices intermediários consistam em *todos* os x vértices numerados de 1 a x ; exige apenas que cada vértice intermediário — não importando quantos forem — sejam numerados x ou menos que x . Visto que todos os vértices são numerados até no máximo n , deve ser o caso que $\text{shortest}[u, v, n]$ é igual a $\text{sp}(u, v)$, o peso de um caminho mínimo de u a v .

Vamos considerar dois vértices, u e v , e escolher um número x na faixa de 1 a n . Considere todos os caminhos de u a v nos quais todos os vértices intermediários sejam numerados até, no máximo, x . De todos esses caminhos, seja o caminho p um caminho de peso mínimo. O caminho p contém ou não contém o vértice x e sabemos que, fora possivelmente u ou v , ele não contém qualquer vértice com número mais alto que x . Há duas possibilidades:

- Primeira possibilidade: x não é um vértice intermediário no caminho p . Então, todos os vértices intermediários do caminho p são numerados, no máximo, $x - 1$. O que isso significa? Significa que o peso de um caminho mínimo de u a v com todos os vértices intermediários numerados no máximo x é o mesmo que o peso de um caminho mínimo de u a v com todos os vértices intermediários numerados até no máximo $x - 1$. Em outras palavras, $\text{shortest}[u, v, x]$ é igual a $\text{shortest}[u, v, x - 1]$.
- Segunda possibilidade: x aparece como um vértice intermediário no caminho p . Como qualquer subcaminho de um caminho mínimo é em si um caminho mínimo, a porção do caminho p que vai de u a x é um caminho mínimo de u a x . Do mesmo modo, a porção de p que vai de x a v é um caminho mínimo de x a v . Como o vértice x é um ponto final de cada um desses subcaminhos, ele não é um vértice intermediário em nenhum deles e, assim, os vértices intermediários em cada um desses subcaminhos são todos numerados, no máximo, $x - 1$. Portanto, o peso de um caminho mínimo de p a x com todos os vértices intermediários numerados no máximo x é a soma dos pesos de dois caminhos mínimos: um de u a x com todos os vértices intermediários numerados no máximo $x - 1$ e um de x a v , também com todos os vértices intermediários numerados no máximo $x - 1$. Em outras palavras, $\text{shortest}[u, v, x]$ é igual a $\text{shortest}[u, x, x - 1] + \text{shortest}[x, v, x - 1]$.

Como x é ou não é um vértice intermediário em um caminho mínimo de u a v , podemos concluir que $\text{shortest}[u, v, x]$ é o menor de $\text{shortest}[u, x, x - 1] + \text{shortest}[x, v, x - 1]$ e $\text{shortest}[u, v, x - 1]$.

O melhor modo de representar o grafo no algoritmo de Floyd-Warshall é por uma variante da representação da matriz de adjacência na página 68. Em vez de cada elemento da matriz estar restrito a 0 ou 1, a entrada para a aresta (u, v) contém o peso da aresta, sendo que um peso de ∞ indica que a aresta está ausente. Visto que $\text{shortest}[u, v, 0]$ denota o peso de um caminho mínimo de u a v com todos os vértices intermediários numerados no máximo 0, tal caminho não tem nenhum vértice intermediário. Isto é, ele consiste em apenas uma única aresta e, portanto, essa matriz é exatamente o que queremos para $\text{shortest}[u, v, 0]$.

Dados os valores $\text{shortest}[u, v, 0]$ (que são os pesos de arestas), o algoritmo de Floyd-Warshall computa valores $\text{shortest}[u, v, x]$ primeiro para todos os pares de vértices u e v com x igual a 1. Então o algoritmo computa valores $\text{shortest}[u, v, x]$ para todos os pares de vértices u e v com x igual a 2. Em seguida, para x igual a 3, e assim por diante, até n .

E quanto a manter o registro de predecessores? Vamos definir $\text{pred}[u, v, x]$ de modo análogo à definição de $\text{shortest}[u, v, x]$, como o predecessor do vértice v sobre um caminho mínimo que parte do vértice u no qual todos os vértices intermediários são numerados no máximo x . Podemos atualizar os valores $\text{pred}[u, v, x]$ à medida que computamos os valores $\text{shortest}[u, v, x]$, da maneira descrita a seguir. Se $\text{shortest}[u, v, x]$ é o mesmo que $\text{shortest}[u, v, x - 1]$, então o caminho mínimo que encontramos de u a v com todos os vértices intermediários numerados no máximo x é o mesmo que um com todos os vértices intermediários numerados no máximo $x - 1$. O predecessor do vértice v deve ser o mesmo em ambos os caminhos; portanto, podemos estabelecer $\text{pred}[u, v, x]$ como o mesmo que $\text{pred}[u, v, x - 1]$. E quando $\text{shortest}[u, v, x]$ é menor que $\text{shortest}[u, v, x - 1]$? Isso acontece quando encontramos um caminho u a v que tem vértice x como um vértice intermediário e tem peso mais baixo que o caminho mínimo de u a v com todos os vértices intermediários numerados no máximo $x - 1$. Como x deve ser um vértice intermediário nesse caminho mínimo recém-encontrado, o predecessor de v no caminho que parte de u deve ser o mesmo do predecessor de v no caminho que parte de x . Nesse caso, fazemos $\text{pred}[u, v, x]$ ser o mesmo que $\text{pred}[x, v, x - 1]$.

Agora temos todas as peças que precisamos para montar o algoritmo de Floyd-Warshall. Eis o procedimento:

Procedimento FLOYD-WARSHALL (G)

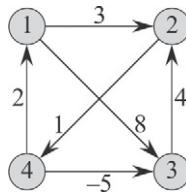
Entrada: G : um grafo representado por uma matriz de adjacência ponderada W com n linhas e n colunas (uma linha e uma coluna por vértice). A entrada na linha u e coluna v , denotada w_{uv} , é o peso da aresta (u, v) , se essa aresta estiver presente em G , e é ∞ caso contrário.

Saída: Para cada par de vértices u e v , o valor de $\text{shortest}[u, v, n]$ contém o peso de um caminho mínimo de u a v , e $\text{pred}[u, v, n]$ é o vértice predecessor de v em um caminho mínimo que vem de u .

1. Sejam shortest e pred novos vetores $n \times n \times (n+1)$.
2. Para cada u e v de 1 a n :
 - a. Iguale $\text{shortest}[u, v, 0]$ a w_{uv} .
 - b. Se (u, v) for uma aresta em G , então iguale $\text{pred}[u, v, 0]$ a u . Caso contrário, iguale $\text{pred}[u, v, 0]$ a NULL.
3. Para $x = 1$ a n :

- a. Para $u = 1$ a n :
- Para $v = 1$ a n :
 - Se $\text{shortest}[u, v, x] < \text{shortest}[u, x, x-1] + \text{shortest}[x, v, x-1]$, iguale $\text{shortest}[u, v, x]$ a $\text{shortest}[u, x, x-1] + \text{shortest}[x, v, x-1]$ e $\text{pred}[u, v, x]$ a $\text{pred}[x, v, x-1]$.
 - Caso contrário, iguale $\text{shortest}[u, v, x]$ a $\text{shortest}[u, v, x-1]$ e $\text{pred}[u, v, x]$ a $\text{pred}[u, v, x-1]$.
 - Retorne os vetores shortest e pred .
-

Para esse grafo



a matriz de adjacência W , contendo os pesos de arestas, é

$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & \infty & -5 & 0 \end{pmatrix}$$

que também dá os valores $\text{shortest}[u, v, 0]^7$ (os pesos de caminhos com, no máximo, uma aresta). Por exemplo, $\text{shortest}[2, 4, 0]$ é 1, porque podemos ir do vértice 2 ao vértice 4 diretamente, sem nenhum vértice intermediário, tomando a aresta (2, 4) com peso 1. De modo semelhante, $\text{shortest}[4, 3, 0]$ é -5. Eis a matriz que dá os valores de $\text{pred}[u, v, 0]$:

$$\begin{pmatrix} \text{NULL} & 1 & 1 & \text{NULL} \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & \text{NULL} \\ 4 & \text{NULL} & 4 & \text{NULL} \end{pmatrix}$$

Por exemplo, $\text{pred}[2, 4, 0]$ é 2 porque o predecessor do vértice 4 é o vértice 2, usando a aresta (2, 4), com peso 1, e $\text{pred}[2, 3, 0]$ é NULL porque não há nenhuma aresta (2, 3)

Depois de executar o laço da etapa 3 para $x = 1$ (para examinar caminhos que podem incluir o vértice 1 como vértice intermediário), os valores de $\text{shortest}[u, v, 1]$ e $\text{pred}[u, v, 1]$ são

$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & 5 & -5 & 0 \end{pmatrix}$$

⁷ Como um vetor tridimensional é um vetor unidimensional de dois vetores bidimensionais, para um valor fixo de x podemos pensar em $\text{shortest}[u, v, x]$ como um vetor bidimensional

e

$$\left(\begin{array}{cccc} \text{NULL} & 1 & 1 & \text{NULL} \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & \text{NULL} \\ 4 & 1 & 4 & \text{NULL} \end{array} \right)$$

Depois que o laço executa para $x = 2$, os valores $\text{shortest}[u, v, 2]$ e $\text{pred}[u, v, 2]$ são

$$\left(\begin{array}{cccc} 0 & 3 & 8 & 4 \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & 5 \\ 2 & 5 & -5 & 0 \end{array} \right) \text{ e } \left(\begin{array}{cccc} \text{NULL} & 1 & 1 & 2 \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & 2 \\ 4 & 1 & 4 & \text{NULL} \end{array} \right).$$

Depois $x = 3$:

$$\left(\begin{array}{cccc} 0 & 3 & 8 & 4 \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & 5 \\ 2 & -1 & -5 & 0 \end{array} \right) \text{ e } \left(\begin{array}{cccc} \text{NULL} & 1 & 1 & 2 \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & 2 \\ 4 & 3 & 4 & \text{NULL} \end{array} \right)$$

E nossos valores finais $\text{shortest}[u, v, 4]$ e $\text{pred}[u, v, 4]$, depois da execução do laço para $x = 4$, são

$$\left(\begin{array}{cccc} 0 & 3 & -1 & 4 \\ 3 & 0 & -4 & 1 \\ 7 & 4 & 0 & 5 \\ 2 & -1 & -5 & 0 \end{array} \right) \text{ e } \left(\begin{array}{cccc} \text{NULL} & 1 & 4 & 2 \\ 4 & \text{NULL} & 4 & 2 \\ 4 & 3 & \text{NULL} & 2 \\ 4 & 3 & 4 & \text{NULL} \end{array} \right)$$

Podemos ver, por exemplo, que o caminho mínimo do vértice 1 ao vértice 3 tem peso -1. Esse caminho vai do vértice 1 ao vértice 2, ao vértice 4, ao vértice 3, o que podemos ver refazendo o caminho de trás para diante: $\text{pred}[1, 3, 4]$ é 4, $\text{pred}[1, 4, 4]$ é 2 e $\text{pred}[1, 2, 4]$ é 1.

Eu afirmei que o algoritmo de Floyd-Warshall executa no tempo $\Theta(n^3)$, e é fácil ver por quê. Temos laços aninhados com profundidade três, e cada um itera n vezes. Em cada iteração do laço da etapa 3, o laço da etapa 3A itera todas as n vezes. De modo semelhante, em cada iteração do laço da etapa 3A, o laço da etapa 3Ai itera todas as n vezes. Visto que o laço externo da etapa 3 também itera n vezes, o laço mais interno (etapa 3Ai) itera n^3 vezes no total. Cada iteração do laço mais interno leva tempo constante e, portanto, o algoritmo leva tempo $\Theta(n^3)$.

Parece que esse algoritmo também ocupa espaço $\Theta(n^3)$ na memória. Afinal, ele cria dois vetores $n \times n \times (n+1)$. Visto que cada entrada de vetor usa uma quantidade constante de memória, esses vetores ocupam espaço $\Theta(n^3)$ na memória. Todavia, acontece que podemos nos safar com apenas $\Theta(n^3)$ de espaço de memória. Como? Basta criar shortest e pred como vetores $n \times n$ e esquecer o terceiro índice para

shortest e *pred* em todos os lugares. Embora as etapas 3Aia e 3Aib continuem atualizando os mesmos valores $\text{shortest}[u, v]_-$ e $\text{pred}[u, v]$, no fim esses vetores acabam tendo os valores corretos!

Anteriormente, mencionei que o algoritmo de Floyd-Warshall ilustra uma técnica denominada **programação dinâmica**. Essa técnica aplica-se somente quando:

1. estamos tentando encontrar uma solução ótima para um problema,
2. podemos desmembrar uma instância do problema em instâncias de um ou mais subproblemas,
3. usamos soluções para o(s) subproblema(s) para resolver o problema original e
4. se usarmos uma solução para um subproblema dentro de uma solução ótima para o problema original, a solução para o subproblema que usarmos deve ser a solução ótima para o subproblema.

Podemos resumir essas condições sob o nome coletivo **subestrutura ótima** e, mais sucintamente, dizemos que uma solução ótima para um problema contém dentro de si soluções ótimas para subproblemas. Em programação dinâmica, temos alguma noção do “tamanho” de um subproblema e, frequentemente, resolvemos os subproblemas em ordem crescente de tamanho, portanto resolvemos os subproblemas menores em primeiro lugar e, depois, tendo em mãos soluções ótimas para os subproblemas menores, podemos tentar resolver subproblemas maiores otimamente usando soluções ótimas para os subproblemas menores.

Essa descrição de programação dinâmica parece um tanto abstrata, portanto vamos ver como o algoritmo de Floyd-Warshall a usa. Enunciamos um subproblema como

Compute $\text{shortest}[u, v, x]_-$, que é o peso de um caminho mínimo do vértice u ao vértice v no qual cada vértice intermediário é numerado de 1 a x .

Aqui, o “tamanho” de um subproblema é o vértice de numeração mais alta que permitimos ser um vértice intermediário de um caminho mínimo: em outras palavras, o valor de x . A subestrutura ótima entra no jogo em razão da seguinte propriedade:

Considere um caminho mínimo p do vértice u ao vértice v e seja x o vértice intermediário de numeração mais alto nesse caminho. Então, a porção de p que vai de u a x é um caminho mínimo de u a x com todos os vértices intermediários numerados de 1 a $x - 1$, e a porção de p que vai de x a v é um caminho mínimo de x a v com todos os vértices intermediários numerados de 1 a $x - 1$.

Resolvemos o problema de computar $\text{shortest}[u, v, x]_-$ computando em primeiro lugar $\text{shortest}[u, v, x - 1]_-$, $\text{shortest}[u, x, x - 1]_-$ e $\text{shortest}[x, v, x - 1]_-$, e então usando o menor dos $\text{shortest}[u, v, x - 1]$ e $\text{shortest}[u, x, x - 1] + \text{shortest}[x, v, x - 1]$. Como computamos todos os valores shortest nos quais o terceiro índice é $x - 1$ antes de tentarmos computar qualquer dos valores shortest nos quais o terceiro índice é x , temos todas as informações que precisamos quando computamos $\text{shortest}[u, v, x]$.

Uma prática comum em programação dinâmica é armazenar soluções ótimas para subproblemas ($\text{shortest}[u, v, x - 1]$, $\text{shortest}[u, x, x - 1]$ e $\text{shortest}[x, v, x - 1]_-$) em uma tabela e então consultá-las à medida que computamos uma solução ótima para o problema original ($\text{shortest}[u, v, x]_-$). Denominamos tal abordagem “de baixo para cima” (“bottom up”), visto que ela funciona de subproblemas menores para subproblemas maiores. Outra abordagem é resolver subproblemas “de cima para baixo”

(“*top down*”), trabalhando de subproblemas maiores para os menores, novamente armazenando o resultado de cada subproblema em uma tabela.

A programação dinâmica aplica-se a uma ampla gama de problemas de otimização, e apenas alguns deles têm a ver com grafos. Nós a veremos novamente no Capítulo 7, quando determinarmos a subsequência comum mais longa de duas cadeias de caracteres.

O QUE MAIS LER?

O Capítulo 24 de CLRS [CLRS09] trata do algoritmo de Dijkstra e do algoritmo de Bellman-Ford. O Capítulo 25 de CLRS trata dos algoritmos de caminho mínimo para todos os pares, incluindo Floyd-Warshall, um algoritmo de caminho mínimo para todos os pares baseado em multiplicação de matriz, com tempo de execução $\Theta(n^3 \lg n)$ e um inteligente algoritmo de Donald Johnson, projetado para determinar os caminhos mínimos para todos os pares em grafos esparsos no tempo $O(n^2 \lg n + nm)$.

Quando pesos de arestas são inteiros pequenos não negativos não maiores que uma quantidade conhecida C , uma implementação mais complexa da fila de prioridade no algoritmo de Dijkstra dá tempos de execução assintótica melhores que um heap de Fibonacci. Por exemplo, Ahuja, Mehlhorn, Orlin e Tarjan [AMOT90] incorporaram um “heap redistributivo” ao algoritmo de Dijkstra, resultando em tempo de execução $O(m + n\sqrt{\lg C})$.

Algoritmos em cadeias

Uma **cadeia** (“string”) é apenas uma sequência de caracteres proveniente de algum conjunto de caracteres subjacente. Por exemplo, este livro contém caracteres provenientes do conjunto de letras, dígitos, símbolos de pontuação e símbolos matemáticos, que são uma parte bastante grande, porém finita, de um conjunto de caracteres. Os biólogos codificam sequências de DNA como cadeias usando apenas quatro caracteres — **A**, **C**, **G**, **T** — que representam as moléculas básicas adenina, citosina, guanina e timina.

Podemos fazer todo tipo de perguntas sobre cadeias, mas neste capítulo focalizaremos algoritmos para três problemas que usam cadeias como entradas:

1. Determinar uma subsequência comum mais longa de duas cadeias.
2. Dado um conjunto de operações que pode transformar uma cadeia em outra, e o custo de cada operação, determinar um modo de menor custo para transformar uma cadeia em outra.
3. Determinar todas as ocorrências de uma cadeia-padrão dentro de uma cadeia de texto.

Os dois primeiros desses problemas têm aplicações na biologia computacional. Quanto mais longa for uma subsequência comum que pudermos encontrar entre duas sequências de DNA, mais semelhantes elas serão. Um modo de alinhar cadeias de DNA é transformar uma cadeia em outra; quanto mais baixo o custo dessa transformação, mais semelhantes são as cadeias. O último problema, o de encontrar ocorrências de um padrão dentro de um texto, é também conhecido como **correspondência de cadeias**. Aparece em todos os tipos de programas, por exemplo, toda vez que você usa um comando “Find” (“localizar”). Também aparece em biologia computacional, porque podemos examinar uma cadeia de DNA dentro de outra.

SUBSEQUÊNCIA COMUM MAIS LONGA

Vamos começar definindo o que significa “sequência” e “subsequência”. Uma **sequência** é uma lista de itens na qual a ordem dos itens tem importância. Um dado item pode aparecer em uma sequência várias vezes. As sequências com as quais trabalharemos neste capítulo são cadeias de caracteres, e usaremos o termo “cadeia” em vez de “sequência”. De modo semelhante, presumiremos que os itens que compõem uma sequência são caracteres. Por exemplo, a cadeia **GACA** contém o mesmo caractere (**A**) várias vezes e é diferente da cadeia **CAAG**, que tem a mesma população de caracteres, mas em uma ordem diferente. Uma **subsequência Z** de uma cadeia **X** é **X**, possivelmente com itens eliminados. Por exemplo, se **X** é a cadeia **GAC**, então ela tem

oito subsequências: GAC (nenhum caractere eliminado), GA (C eliminado), GC (A eliminado), AC (G eliminado), G (A e C eliminados), A (G e C eliminados), C (G e A eliminados) e a cadeia vazia (todos os caracteres eliminados). Se X e Y são cadeias, então Z é uma **subsequência comum** de X e Y se for uma subsequência de ambas. Por exemplo, se X é a cadeia CATCGA e Y é a cadeia GTACCGTCA, então CCA é uma subsequência comum de X e Y que consiste em três caracteres. Todavia, não é uma **subsequência comum mais longa (longest common subsequence — LCS)**, visto que a subsequência comum CTCA tem quatro caracteres. Na verdade, CTCA é uma subsequência comum mais longa, mas não é a única, já que TCGA é outra subsequência com quatro caracteres. As noções de subsequência e subcadeia são diferentes: uma **subcadeia** é uma subsequência de uma cadeia na qual os caracteres devem ser extraídos de posições contíguas na cadeia. Para a cadeia CATCGA, a subsequência ATCG é uma subcadeia, mas a subsequência CTCA não é.

Nossa meta é, dadas duas cadeias X e Y , determinar a subsequência comum mais longa Z de X e Y . Usaremos a técnica de programação dinâmica, que vimos no Capítulo 6, para resolver esse problema.

Você pode determinar uma subsequência comum mais longa sem recorrer à programação dinâmica, mas eu não recomendo. Você poderia tentar cada subsequência de X e verificar se ela é uma subsequência de Y trabalhando das mais longas para as mais curtas (ou menores) subsequências de X , verificando cada uma em relação a Y e parando assim que encontrar uma subsequência de X e Y (você sabe que a certa altura encontrará uma, visto que a cadeia vazia é uma subsequência de todas as cadeias). Se X tem comprimento m , então tem 2^m subsequências e, portanto, mesmo que ignoremos o tempo para verificar cada subsequência em relação a X , o tempo para encontrar uma LCS seria no mínimo exponencial em relação ao comprimento de X no pior caso.

Lembre-se de que no Capítulo 6 dissemos que, para aplicar programação dinâmica, precisamos de uma estrutura ótima: uma solução ótima para um problema contém soluções ótimas para seus subproblemas. Para encontrar uma LCS de duas cadeias via programação dinâmica, em primeiro lugar precisamos decidir o que constitui um subproblema. Os prefixos funcionam. Se X é uma cadeia $x_1x_2x_3 \cdots x_m$, então o **i -ésimo prefixo** de X é a cadeia $x_1x_2x_3 \cdots x_i$, e a denotamos X_i . Aqui, exigimos que i esteja na faixa 0 a m , e X_0 seja a cadeia vazia. Por exemplo, se X é CATCGA, então X_4 é CATC.

Podemos ver que uma LCS de duas cadeias contém em seu interior uma LCS dos prefixos das duas cadeias. Vamos considerar duas cadeias $X = x_1x_2x_3 \cdots x_m$ e $Y = y_1y_2y_3 \cdots y_n$. Elas têm alguma LCS, digamos Z , onde $Z = z_1z_2z_3 \cdots z_k$ para algum comprimento k , que poderia estar em qualquer lugar de 0 ao menor de m e n . O que podemos deduzir sobre Z ? Vamos examinar os últimos caracteres em X e Y : x_m e y_n . Eles são iguais ou não são.

- Se forem iguais, o último caractere z_k de Z deve ser o mesmo que aquele caractere. O que sabemos sobre o resto de Z , que é $Z_{k-1} = z_1z_2z_3 \cdots z_{k-1}$? Sabemos que Z_{k-1} deve ser uma LCS do resto de X e Y , a saber: $X_{m-1} = x_1x_2x_3 \cdots x_{m-1}$

e $Y_{n-1} = y_1y_2y_3 \cdots y_{n-1}$. Pelo nosso exemplo de antes — onde $X = \text{CATCGA}$ e $Y = \text{GTACCGTCA}$ e uma LCS é $Z = \text{CTCA}$ —, o último caractere, A, de X e Y é o último caractere de Z , e vemos que $Z_3 = \text{CTC}$ deve ser uma LCS de $X_5 = \text{CATCG}$ e $Y_8 = \text{GTACCGTCA}$.

- Se não forem iguais, Z_k pode ser igual ao último caractere x_m de X ou o último caractere y_n de Y , mas não ambos. Ou poderia não ser o mesmo que o último caractere de X ou Y . Se z_k não é o mesmo que x_m , ignore o último caractere de X : Z deve ser uma LCS de X_{m-1} e Y . De modo semelhante, se z_k não é o mesmo que y_n , ignore o último caractere de Y : Z deve ser uma LCS de X e Y_{n-1} . Continuando o exemplo dado, seja $X = \text{CATCG}$, $Y = \text{GTACCGTC}$ e $Z = \text{CTC}$. Aqui, z_3 é o mesmo que y_8 (C), mas não é o mesmo que x_5 (G) e, assim, Z é uma LCS de $X_4 = \text{CATC}$ e Y .

Portanto, esse problema tem subestrutura ótima: uma LCS de duas cadeias contém em seu interior uma LCS dos prefixos das duas cadeias.

Como continuar? Precisamos resolver um ou dois subproblemas, dependendo de os últimos caracteres de X e Y serem os mesmos. Se forem, resolvemos apenas um subproblema — encontrar uma LCS de X_{m-1} e Y_{n-1} — e anexar aquele último caractere para obter uma LCS de X e Y . Se os últimos caracteres de X e Y não forem os mesmos, temos de resolver dois subproblemas — encontrar uma LCS de X_{m-1} e Y , e uma LCS de X e Y_{n-1} — e usar a mais longa dessas duas subsequências comuns mais longas como uma LCS de X e Y . Se essas duas subsequências comuns mais longas tiverem o mesmo comprimento, usamos qualquer delas — não importa qual.

Abordaremos o problema de encontrar uma LCS de X e Y em duas etapas. Na primeira determinaremos o comprimento de uma LCS de X e Y , bem como os comprimentos das subsequências comuns mais longas de todos os prefixos de X e Y . Você poderia ficar surpreso por podermos determinar o comprimento da LCS sem saber qual é a LCS. Depois de computar os comprimentos da LCS, faremos a “engenharia reversa” para explicar como computamos esses comprimentos para encontrar uma LCS propriamente dita de X e Y .

Para que as coisas fiquem um pouco mais precisas, vamos denotar o comprimento de uma LCS dos prefixos X_i e Y_j de $l[i, j]$. O comprimento de uma LCS de X e Y é dado por $l[m, n]$. Podemos iniciar os índices i e j em 0, visto que, se qualquer um dos prefixos tiver comprimento 0, sabemos qual é sua LCS: é uma cadeia vazia. Em outras palavras, $l[0, j]_+$ e $l[i, 0]_+$ é igual a 0 para todos os valores de i e j . Quando i e j são positivos, determinamos $l[i, j]_-$ examinando os menores valores de i e/ou j :

- Se i e j são positivos e x_i é o mesmo que y_j , então $l[i, j]_-$ é igual a $l[i - 1, j - 1] + 1$.
- Se i e j são positivos e x_i é diferente de y_j , então $l[i, j]_-$ é igual ao maior de $l[i, j - 1]$ e $l[i - 1, j]$.

Imagine que os valores de $l[i, j]_-$ estejam armazenados em uma tabela. Precisamos computar esses valores em ordem crescente dos índices i e j . Damos a seguir a tabela $l[i, j]_-$ para nossas cadeias de exemplo (veremos o que as partes sombreadas significam um pouco mais adiante):

		j	0	1	2	3	4	5	6	7	8	9
		y_j	G	T	A	C	C	G	T	C	A	
i	x_i	$l[i,j]$										
0			0	0	0	0	0	0	0	0	0	0
1	C		0	0	0	0	1	1	1	1	1	1
2	A		0	0	0	1	1	1	1	1	1	2
3	T		0	0	1	1	1	1	1	2	2	2
4	C		0	0	1	1	2	2	2	2	3	3
5	G		0	1	1	1	2	2	3	3	3	3
6	A		0	1	1	2	2	2	3	3	3	4

Por exemplo, $l[5, 8]_-$ é 3, o que significa que uma LCS de $X_5 = \text{CATC}$ e $Y_8 = \text{GTACCGTC}$ tem comprimento 3, como vimos na página 102-103.

Para computar valores de tabela em ordem crescente dos índices, antes de computarmos uma entrada particular $l[i, j]$, onde i e j são positivos, precisamos computar as entradas $l[i, j - 1]$ (imediatamente à esquerda de $l[i, j]$), $l[i - 1, j]$ (imediatamente acima de $l[i, j]$) e $l[i - 1, j - 1]_-$ (acima e à esquerda de $l[i, j]_-$).¹ É fácil computar as entradas de tabela desse modo: podemos computá-las linha por linha, da esquerda para a direita dentro de cada linha, ou coluna por coluna, de cima para baixo dentro de cada coluna.

O procedimento dado a seguir trata a tabela como um arranjo bidimensional $l[0..m, 0..n]_-$. Depois de preencher a coluna da extrema esquerda e a linha de cima com 0s, ele preenche o restante do arranjo linha por linha.

Procedimento COMPUTE-LCS-TABLE (X, Y)

Entradas: X e Y : duas cadeias de comprimento m e n e, respectivamente.

Saída: O arranjo $l[0..m, 0..n]_-$. O valor de $l[m, n]_-$ é o comprimento de uma subsequência comum mais longa de X e Y .

1. Seja $l[0..m, 0..n]_-$ um novo arranjo.
 2. Para $i = 0$ a m :
 - a. Iguale $l[i, 0]_-$ a 0.
 3. Para $j = 0$ a n :
 - a. Iguale $l[0, j]$ a 0.
 4. Para $i = 1$ a m :
 - a. Para $j = 1$ a n :
 - i. Se x_i é o mesmo de y_j , então iguale $l[i, j]_-$ a $l[i - 1, j - 1]_-$.
 - ii. Caso contrário (x_i é diferente de y_j), iguale $l[i, j]_-$ ao maior de $l[i, j - 1]_-$ e $l[i - 1, j]_-$.
 5. Retorne o arranjo l .
-

Visto que leva tempo constante para preencher cada entrada da tabela, e a tabela contém $(m+1) \cdot (n+1)$ entradas, o tempo de execução de COMPUTE-LCS-TABLE é $\Theta(mn)$.

¹Até mesmo mencionar $l[i - 1, j - 1]_-$ é redundante, visto que precisamos tê-lo computado antes de computar $l[i, j - 1]$ e $l[i - 1, j]_-$.

A boa notícia é que, uma vez computada a tabela $l[i, j]$, sua entrada inferior direita, $l[m, n]$, nos dá o comprimento de uma LCS de X e Y . A má notícia é que nenhuma entrada isolada na tabela nos diz quais caracteres estão em uma LCS. Podemos usar a tabela, juntamente com as cadeias X e Y , para construir uma LCS usando o tempo adicional $\Theta(m+n)$. Determinamos como chegamos ao valor em $l[i, j]$ executando engenharia reversa nessa computação, tomando como base $l[i, j]$ e os valores dos quais ele depende: $x_i, y_j, l[i - 1, j - 1], l[i, j - 1]$ e $l[i - 1, j]$.

Eu gosto de escrever esse procedimento recursivamente, quando montamos uma LCS de trás para a frente. O procedimento executa recursões e, quando encontra caracteres em X e Y que são os mesmos, anexa o caractere ao final da LCS que constrói. A chamada inicial é ASSEMBLE-LCS. $(X, Y, l, i, j) /$.

Procedimento ASSEMBLE-LCS (X, Y, l, i, j)

Entradas:

- X e Y : duas cadeias.
- l : o arranjo preenchido pelo procedimento COMPUTE-LCS-TABLE.
- i e j : índices para X e Y , respectivamente, bem como para l .

Saída: Uma LCS de x_i e y_j .

1. Se $l[i, j]$ é igual a 0, retorne a cadeia vazia.
2. Caso contrário (porque $l[i, j]$ é positivo, i e j são positivos), se x_i é o mesmo que y_j , o retorne a cadeia formada, em primeiro lugar chamando recursivamente ASSEMBLE-LCS. $(X, Y, l, i - 1, j - 1) /$, então anexando x_i (ou y_j) ao final da cadeia retornada pela chamada recursiva.
3. Caso contrário (x_i é diferente de y_j), se $l[i, j - 1]$ é maior que $l[i - 1, j]$, retorne a cadeia retornada chamando recursivamente ASSEMBLE-LCS $(X, Y, l, i, j - 1)$.
4. Caso contrário (x_i é diferente de y_j e $l[i, j - 1]$ é menor ou igual a $l[i - 1, j]$), retorne a cadeia retornada chamando recursivamente ASSEMBLE-LCS $(X, Y, l, i - 1, j) /$.

Na tabela da página 104, as entradas $l[i, j]$ sombreadas são as que a recursão visita com a chamada inicial ASSEMBLE-LCS $(X, Y, l, 6, 9) /$, e os caracteres x_i sombreados são os anexados à LCS que está sendo construída. Para ter uma ideia de como ASSEMBLE-LCS funciona, comece em $i = 6$ e $i = 9$. Aqui, constatamos que x_6 e y_9 são o caractere A. Portanto, A será o último caractere da LCS de x_6 e y_9 , e executamos recursão na etapa 2. A chamada recursiva tem $i = 5$ e $j = 8$. Dessa vez, constatamos que x_5 e y_8 são caracteres diferentes e também constatamos que $l[5, 7]$ é igual a $l[4, 8]$, e portanto executamos recursão na etapa 4. Agora a chamada recursiva tem $i = 4$ e $j = 8$. E assim por diante. Se você ler os x_i caracteres sombreados de cima para baixo, obterá a cadeia CTCA, que é uma LCS. Se tivéssemos rompido vínculos entre $l[i, j - 1]$ e $l[i - 1, j]$ em favor de ir para a esquerda (etapa 3), em vez de ir para cima (etapa 4), a LCS produzida seria TCGA.

Como é que o procedimento ASSEMBLE-LCS leva tempo $O(m + n)$? Observe que, em cada chamada recursiva, i decresce ou j decresce, ou ambos decrescem. Portanto, depois de $m + n$ chamadas recursivas, é garantido que um ou outro desses índices chega a 0 e a recursão termina na etapa 1.

TRANSFORMANDO UMA CADEIA EM OUTRA

Agora vamos ver como transformar uma cadeia X em outra cadeia Y . Iniciaremos com X e a converteremos em Y , caractere por caractere. Consideraremos que X e Y consistem em m e n caracteres, respectivamente. Como antes, denotaremos o i -ésimo caractere de cada cadeia usando o nome da cadeia em letra minúscula, com subscrito i , de modo que o i -ésimo caractere de X é x_i e o j -ésimo caractere de Y é y_j .

Para converter X em Y , construiremos uma cadeia, que denominaremos Z , de modo que, quando terminarmos, Z e Y serão as mesmas. Mantemos um índice i em X e um índice j em Z . Executaremos uma sequência de operações específicas de transformação, que podem alterar Z e esses índices. Começamos com i e j em 1, e devemos examinar todo caractere em X durante o processo, o que significa que pararemos somente quando i alcançar $m + 1$.

Eis as operações que consideramos:

- **Copiar** um caractere x_i de X para Z igualando z_i a x_i e incrementando i e j .
- **Substituir** um caractere x_i de X por outro caractere a igualando z_i a a e incrementando i e j .
- **Eliminar** um caractere x_i de X incrementando i mas sem mexer em j .
- **Inserir** um caractere a em Z igualando z_j a a e incrementando j mas sem mexer em i .

Outras operações são possíveis — tal como intercambiar dois caracteres adjacentes ou eliminar caracteres x_i até x_m em uma única operação —, mas aqui consideraremos apenas as operações *copiar*, *substituir*, *eliminar* e *inserir* (*copy*, *replace*, *delete* e *insert*).

Como exemplo, damos uma sequência de operações que transforma a cadeia ATGATCGGCAT na cadeia CAATGTGAATC, onde os caracteres sombreados são x_i e z_j depois de cada operação:

Operação	X	Z
initial strings	ATGATCGGCAT	
delete A	ATGATCGGCAT	
replace T by C	ATGATCGGCAT	C
replace G by A	ATGATCGGCAT	CA
copy A	ATGATCGGCAT	CAA
copy T	ATGATCGGCAT	CAAT
replace C by G	ATGATCGGCAT	CAATG
replace G by T	ATGATCGGCAT	CAATGT
copy G	ATGATCGGCAT	CAATGTG
replace C by A	ATGATCGGCAT	CAATGTGA
copy A	ATGATCGGCAT	CAATGTGAA
copy T	ATGATCGGCAT	CAATGTGAAT
insert C	ATGATCGGCAT	CAATGTGAATC

Outras sequências de operações também funcionariam. Por exemplo, poderíamos apenas eliminar cada caractere de X por vez e inserir cada caractere de Y em Z .

Cada uma das operações de transformação possui um custo, que é uma constante e depende somente do tipo da operação e não dos caracteres envolvidos. Nossa meta é encontrar uma sequência de operações que transforme X em Y e tenha um custo total mínimo. Vamos denotar o custo da operação *copy* por c_c , o custo da operação *replace* por c_R , o custo da operação *delete* por c_D e o custo da operação *insert* por c_I . Para a sequência de operações no exemplo anterior, o custo total seria $5c_c + 5c_R + c_D + c_I$. Fica subentendido que cada c_c e c_R é menor que $c_D + c_I$ porque, caso contrário, em vez de custar c_C para copiar um caractere ou custar c_R para substituir um caractere, custaria apenas $c_D + c_I$ para eliminar o caractere e inserir ou o mesmo caractere (em vez de copiá-lo) ou um caractere diferente (em vez de substituí-lo).

Por que você iria querer transformar uma cadeia em outra? A biologia computacional nos dá uma aplicação. Os biólogos computacionais frequentemente alinham duas sequências de DNA para medir o grau de semelhança entre elas. Em um modo de alinhar duas sequências X e Y , alinhamos caracteres idênticos o máximo possível inserindo espaços nas duas sequências (inclusive em qualquer das extremidades), de modo que as sequências resultantes, digamos X' e Y' , têm o mesmo comprimento, mas não têm um espaço na mesma posição. Isto é, não podemos ter x'_i e y'_i como um espaço. Depois de alinhar, designamos uma nota a cada posição:

- -1 se x_i e y_i são os mesmos e não um espaço.
- +1 se x_i for diferente de y_i e nenhum for um espaço.
- +2 se x_i ou y_i for um espaço.

A nota dada a um alinhamento é a soma das notas dadas para as posições individuais. Quanto mais baixa a nota, mais próximo será o alinhamento das duas cadeias. Para as cadeias no exemplo anterior, podemos alinhá-las da seguinte maneira, onde \sqcup indica um espaço:

$$\begin{array}{ccccccccc} X': & \text{ATGATCG} & \text{GCAT} \\ & \sqcup & \sqcup \\ Y': & \text{CAAT} & \text{GTGAATC} \\ & \sqcup & \sqcup \\ & *+---*-*-+--* & \end{array}$$

Um - sob uma posição indica uma nota -1 para essa posição, um + indica uma nota +1 e um * indica +2. Esse alinhamento particular tem nota total de $(6 \cdot -1) + (3 \cdot 1) + (4 \cdot 2)$ ou 5.

Há muitos modos possíveis de inserir espaços e alinhar duas sequências. Para descobrir o modo que produz a melhor correspondência — a que tem a menor nota usamos transformação de cadeia com custos $c_c = -1$, $c_R = +1$ e $c_D = c_I = +2$. Quanto maior o número de caracteres idênticos correspondentes, melhor o alinhamento, e o custo negativo da operação *copy* dá um incentivo para emparelhar caracteres idênticos. Um espaço em Y' corresponde a um caractere eliminado, de modo que no exemplo anterior, o primeiro espaço em Y' corresponde a eliminar o primeiro caractere (A) de X . Um espaço em X' corresponde a um caractere inserido, de modo que, no exemplo anterior, o primeiro espaço em X' corresponde a inserir o caractere T.

Portanto, vamos ver como transformar uma cadeia X em uma cadeia Y . Usaremos programação dinâmica, com subproblemas da forma “converter o prefixo da cadeia x_i no prefixo da cadeia Y_j ”, onde i vai de 0 a m e j vai de 0 a n . Denominaremos esse subproblema “problema $X_i \rightarrow Y_j$ ”, e o problema com o qual iniciamos é o problema $X_m \rightarrow Y_n$. Vamos denotar o custo de uma solução ótima para o problema $X_i \rightarrow Y_j$ por $\text{cost}[i, j]$. Como exemplo, tome $X = \text{ACAAGC}$ e $Y = \text{CCGT}$, de modo que queremos resolver o problema $X_6 \rightarrow Y_4$, e usaremos os custos da operação para alinhar sequências de DNA: $c_c = -1$, $c_R = +1$, e $c_D = c_1 = +2$. Resolveremos subproblemas da forma $X_i \rightarrow Y_j$, onde i vai de 0 a 6 e j vai de 0 a 4. Por exemplo, o problema $X_3 \rightarrow Y_2$ é transformar a cadeia de prefixo $X_3 = \text{ACA}$ na cadeia de prefixo $Y_2 = \text{CC}$.

É fácil determinar $\text{cost}[i, j]$ quando i ou j é 0, porque X_0 e Y_0 são cadeia vazias. Converta uma cadeia vazia em Y_j por meio de j operações *insert*, de modo que $\text{cost}[0, j]$ é igual a $j c_I$. De modo semelhante, converta X_i em uma cadeia vazia por meio de i operações *delete*, de modo que $\text{cost}[i, 0]$ é igual a $i c_D$. Quando i e j são 0, estamos convertendo a cadeia vazia nela mesmo, e portanto $\text{cost}[0, 0]$ é obviamente 0.

Quando i e j são positivos, precisamos examinar como a subestrutura ótima aplica-se à transformação de uma cadeia em outra. Vamos supor — por enquanto — que sabemos qual foi a última operação usada para converter X_i em Y_j . Foi uma das quatro operações *copy*, *replace*, *delete* ou *insert*.

- Se a última operação foi *copy*, então x_i e y_j devem ter sido o mesmo caractere. O subproblema que resta é converter x_{i-1} em y_{j-1} , e uma solução ótima para o problema $X_i \rightarrow Y_j$ deve incluir uma solução ótima para o problema $X_{i-1} \rightarrow Y_{j-1}$. Por quê? Porque, se tivéssemos usado uma solução para o problema $X_{i-1} \rightarrow Y_{j-1}$ que não tivesse o custo mínimo, poderíamos usar a solução de custo mínimo em vez de obter uma solução melhor para o problema $X_i \rightarrow Y_j$ do que a que obtivemos. Portanto, supondo que a última operação foi *copy*, sabemos que $\text{cost}[i, j]$ é igual a $\text{cost}[i-1, j-1] + c_C$.

Em nosso exemplo, vamos examinar o problema $X_5 \rightarrow Y_3$. Ambos, x_5 e y_3 , são o caractere G e, assim, se a última operação foi *copy* G, como $c_c = -1$, devemos ter $\text{cost}[5, 3] = \text{cost}[4, 2] - 1$. Se $\text{cost}[4, 2] = 4$, então $\text{cost}[5, 3] = 3$. Se pudéssemos ter encontrado uma solução para o problema $X_4 \rightarrow Y_2$ com um custo menor que 4, poderíamos usar tal solução para encontrar uma solução para o problema $X_5 \rightarrow Y_3$ com um custo menor que 3.

- Se a última operação foi *replace*, e sob a razoável premissa de que não podemos “substituir” um caractere por ele mesmo, então x_i e y_j devem ser diferentes. Usando o mesmo argumento da subestrutura ótima que usamos para a operação *copy*, vemos que, pressupondo que a última operação foi *replace*, $\text{cost}[i, j]$ é igual a $\text{cost}[i, j] + c_R$.

Em nosso exemplo, considere o problema $X_5 \rightarrow Y_4$. Dessa vez, x_5 e y_4 são caracteres diferentes (G e T, respectivamente) e, assim, a última operação foi *replace* G por T; então, como $c_R = +1$, devemos ter $\text{cost}[5, 4] = \text{cost}[4, 3] + 1$. Se $\text{cost}[4, 3] = 3$, então $\text{cost}[5, 4] = 4$.

Se a última operação foi *delete*, não temos nenhuma restrição a x_i ou y_j . Imagine que a operação *delete* salte o caractere x_i e não mexe com o prefixo y_j , de modo que

o subproblema que precisamos resolver é o problema $X_{i-1} \rightarrow Y_j$. Considerando que a última operação foi uma *delete*, sabemos que $\text{cost}[i, j] = \text{cost}[i-1, j] + c_D$.

Em nosso exemplo, considere o problema $X_6 \rightarrow Y_3$. Se a última operação foi uma *delete* (o caractere eliminado deve ser x_6 , que é C), então como $c_D = +2$, devemos ter $\text{cost}[6, 3] = \text{cost}[5, 3] + 2$. Se $\text{cost}[5, 3]$ é 3, então $\text{cost}[6, 3]$ deve ser 5.

- Por fim, se a última operação foi uma *insert*, isso não mexe com x_i , mas adiciona o caractere y_j , e o subproblema a resolver é $X_i \rightarrow Y_{j-1}$. Considerando que a última operação foi uma *insert*, sabemos que $\text{cost}[i, j] = \text{cost}[i, j-1] + c_I$.

Em nosso exemplo, considere o problema $X_2 \rightarrow Y_3$. Se a última operação foi uma *insert* (o caractere inserido deve ser y_3 , que é G), então como $c_I = +2$, devemos ter $\text{cost}[2, 3] = \text{cost}[2, 2] + 2$. Se $\text{cost}[2, 2]$ é 0, então $\text{cost}[2, 3]$ deve ser 2.

É claro que não sabemos de antemão qual das quatro operações foi a última usada. Queremos usar aquela que dá o valor mais baixo para $\text{cost}[i, j]$. Para uma combinação dada de i e j , três das quatro operações se aplicam. As operações *delete* e *insert* sempre aplicam-se quando i e j são positivos, e exatamente uma de *copy* e *replace*, se aplica, dependendo de x_i e y_j serem o mesmo caractere. Para computar $\text{cost}[i, j]$ a partir de outros valores de cost , determine qual das duas, *copy* e *replace*, se aplica e tome o valor mínimo de $\text{cost}[i, j]$ que as três possíveis operações dão. Isto é, $\text{cost}[i, j]$ é o menor dos seguintes quatro valores:

- $\text{cost}[i-1, j-1] + c_C$, mas somente se x_i e y_j são o mesmo caractere,
- $\text{cost}[i-1, j-1] + c_R$, mas somente se x_i e y_j forem diferentes,
- $\text{cost}[i-1, j] + c_D$,
- $\text{cost}[i, j-1] + c_I$.

Exatamente como fizemos para preencher a tabela l quando computamos uma LCS, podemos preencher a tabela cost linha por linha. Isso porque, exatamente como na tabela l , cada entrada $\text{cost}[i, j]$, onde x_i e y_j são positivos, depende de já ter computado as entradas imediatamente à esquerda, imediatamente acima e acima e à esquerda.

Além da tabela cost , preencheremos uma tabela op , onde $op[i, j]$ dá a última operação usada para converter x_i em y_j . Podemos preencher a entrada $op[i, j]$ quando preencheremos $\text{cost}[i, j]$. O procedimento COMPUTE-TRANSFORM-TABLES na página seguinte preenche as tabelas *custo* e *op*, linha por linha, tratando as tabelas *cost* e *op* como dois arranjos bidimensionais.

A página 111 tem as tabelas *cost* e *op* computadas por COMPUTE-TRANSFORM-TABLES para nosso exemplo de transformar $X = \text{ACAAGC}$ em $Y = \text{CCCT}$ com $c_c = -1$, $c_R = +1$ e $c_D = c_I = +2$. Na linha i e na coluna j aparecem os valores de $\text{cost}[i, j]$ e $op[i, j]$, com nomes de operação abreviados. Por exemplo, a última operação usada ao transformar $X_5 = \text{ACAAG}$ em $Y_2 = \text{CC}$ substitui G por C, e uma sequência de operações ótimas para transformar ACAAG em CC tem um custo total de 6.

O procedimento COMPUTE-TRANSFORM-TABLES preenche cada entrada das tabelas em tempo constante, exatamente como o procedimento COMPUTE-LCS-TABLE. Como cada uma das tabelas contém $(m+1) \cdot (n+1)$ entradas, COMPUTE-TRANSFORM-TABLES executa em tempo $\Theta(mn)$.

Para construir a sequência de operações que transforma X em Y , consultamos a tabela *op*, começando na última entrada, $op[m, n]$. Executamos a recursão, de um

modo muito parecido com o procedimento ASSEMBLE-LCS, anexando cada operação encontrada na tabela op ao final da sequência de operações. O procedimento ASSEMBLE-TRANSFORMATION aparece na página 111. A chamada inicial é ASSEMBLE-TRANSFORMATION (op, m, n) A sequência de operações para converter $X = ACAAG$ em uma cadeia Z que é a mesma que $Y = CCGT$ aparece abaixo das tabelas $cost$ e op na página 111.

Exatamente como em ASSEMBLE-LCS, cada chamada recursiva do procedimento ASSEMBLE-TRANSFORMATION decresce i ou j , ou ambos, e assim a recursão se exaure depois de no máximo $m + n$ chamadas recursivas. Visto que cada chamada recursiva leva tempo constante antes e depois da recursão, o procedimento ASSEMBLE-TRANSFORMATION executa em tempo $O(m + n)$.

Uma sutileza no procedimento ASSEMBLE-TRANSFORMATION merece exame mais detalhado. A recursão se exaure somente quando i e j chegam a 0. Suponha que i ou j , mas não ambos, seja igual a 0. Cada um dos três casos nas etapas 2A, 2B e 2C executa recursão com o valor de i ou j , ou ambos, decrescido de 1. Poderia haver uma chamada recursiva na qual i ou j tenha o valor -1? Felizmente, a resposta é não. Suponha que $j = 0$ e i seja positivo em uma chamada de ASSEMBLE-TRANSFORMATION. Pelo modo como a tabela op é construída, $op[1, 0]$ é uma operação *delete*, portanto a etapa 2B executa. A chamada recursiva na etapa 2B chama

Procedimento COMPUTE-TRANSFORM-TABLES (X, Y, c_c, c_R, c_D, c_I)

Entradas:

1. A e Y : duas cadeias de comprimento m ou n , respectivamente.

2. c_C, c_R, c_D, c_I : os custos das operações *copy*, *replace*, *delete* e *insert*, respectivamente.

Saída: Arranjos $cost[0..m, 0..n]$ e $op[0..m, 0..n]$. O valor em $cost[i, j]$ é o custo mínimo para transformar o prefixo X_i no prefixo Y_j , de modo que $cost[m, n]$ é o custo mínimo para transformar X em Y . A operação em $op[i, j]$ é a última operação executada na transformação de $X_i Y_j$.

1. Seja $cost[0..m, 0..n]$ e $op[0..m, 0..n]$ novos arranjos.

2. Iguale $cost[0, 0]$ a 0.

3. Para $i = 1$ a m :

- a. Iguale $cost[i, 0] \leftarrow c_D$, e iguale $op[0, j] \leftarrow \text{delete } x_i$.

4. Para $j = 1$ a n :

- a. Iguale $cost[0, j] \leftarrow j \cdot c_I$, e $op[i, 0] \leftarrow \text{insert } y_j$.

5. Para $i = 1$ a m :

- a. Para $j = 1$ a n :

(Determine qual, *copy* ou *replace*, se aplica, e iguale $cost[i, j]$ e $op[i, j]$ de acordo com qual das três operações aplicáveis minimiza $cost[i, j]$.)

- i. Estabeleça $cost[i, j]$ e $op[i, j]$ da seguinte maneira:

- a. Se x_i e y_j são o mesmo, iguale $cost[i, j]$ a $cost[i-1, j-1] + c_c$ e $op[i, j]$ a *copy* x_i .

- b. Caso contrário (x_i e y_j são diferentes), iguale $cost[i, j]$ a $cost[i-1, j-1] + c_R$ e $op[i, j]$ a *replace* x_i por y_j .

- ii. Se $cost[i-1, j] + c_D < cost[i, j]$, iguale $cost[i, j]$ a $cost[i-1, j] + c_D$ e $op[i, j]$ a *delete* x_i .

- iii. Se $cost[i, j-1] + c_I < cost[i, j]$, iguale $cost[i, j]$ a $cost[i, j-1] + c_I$ e $op[i, j]$ a *insert* y_j .

6. Retorne os arranjos $cost$ e op .

		j	0	1	2	3	4
		y_j		C	C	G	T
i	x_i						
0			0	2 <i>ins C</i>	4 <i>ins C</i>	6 <i>ins G</i>	8 <i>ins T</i>
1	A		2 <i>del A</i>	1 <i>rep A de C</i>	3 <i>rep A de C</i>	5 <i>rep A de G</i>	7 <i>rep A de T</i>
2	C		4 <i>del C</i>	1 <i>copy C</i>	0 <i>copy C</i>	2 <i>ins G</i>	4 <i>ins T</i>
3	A		6 <i>del A</i>	3 <i>del A</i>	2 <i>rep A de C</i>	1 <i>rep A de G</i>	3 <i>rep A de T</i>
4	A		8 <i>del A</i>	5 <i>del A</i>	4 <i>rep A de C</i>	3 <i>rep A de G</i>	2 <i>rep A de T</i>
5	G		10 <i>del G</i>	7 <i>del G</i>	6 <i>rep G de C</i>	3 c <i>opy G</i>	4 <i>rep G de T</i>
6	C		12 <i>del C</i>	9 <i>copy C</i>	6 <i>copy C</i>	5 <i>del C</i>	4 <i>rep C de T</i>

Operação	X	Z
cadeias iniciais	ACAAGC	
delete A	A CAAGC	
copy C	AC AAGC	C
delete A	AC AAGC	C
replace A de C	ACAAGC	CC
copy G	ACAAGC	CCG
replace C de T	ACAAGC	CCGT

ASSEMBLE-TRANSFORMATION ($op, i-1, j$) /; assim, o valor de j na chamada recursiva permanece em 0. De modo semelhante, se $i = 0$ e j é positivo $op[0, j]$ é uma operação *insert*, de modo que a etapa 2C executa, e na chamada recursiva a ASSEMBLE-TRANSFORMATION ($op, i, j-1$), o valor de i permanece em 0.

Procedimento ASSEMBLE-TRANSFORMATION (op, i, j)

Entradas:

- op : a tabela de operação preenchida por COMPUTE-TRANSFORM-TABLES.
- i e j : índices para a tabela op .

Saída: Uma sequência de operações que transforma a cadeia X na cadeia Y , onde X e Y são as cadeias fornecidas como entradas por COMPUTE-TRANSFORM-TABLES.

1. Se i e j são iguais a 0, retorne uma sequência vazia.
2. Caso contrário (no mínimo, i ou j é positivo), faça o seguinte:

- a. Se $op[i, j]$ é uma operação *copy* ou *replace*, retorne a sequência formada, em primeiro lugar chamando recursivamente ASSEMBLE-TRANSFORMATION $op[i - 1, j - 1]$ e anexando $op[i, j]$ à sequência retornada pela chamada recursiva.

- b. Caso contrário ($op[i, j]$ não é nem uma operação *copy* nem uma operação *replace*), se $op[i, j]$ é uma operação *delete*, retorne a sequência formada, em primeiro lugar chamando recursivamente ASSEMBLE-TRANSFORMATION $op[i - 1, j]/$ e anexando $op[i, j]$ à sequência retornada da chamada recursiva.
 - c. Caso contrário ($op[i, j]$ não é uma operação *copy*, *replace* ou *delete* e, portanto, deve ser uma operação *insert*), retorne a sequência formada, em primeiro lugar chamando recursivamente ASSEMBLE-TRANSFORMATION $op[i, j - 1]/$ e anexando $op[i, j]$ à sequência retornada da chamada recursiva.
-

CORRESPONDÊNCIA DE CADEIAS

No problema da correspondência de cadeias, temos duas cadeias: uma **cadeia de texto T** e uma **cadeia-padrão P** . Queremos determinar *todas* as ocorrências de P em T . Encurtaremos os nomes para “texto” e “padrão” e consideraremos que texto e padrão consistem em n e m caracteres, respectivamente, onde $m \leq n$ (visto que não tem sentido procurar um padrão que é mais longo que o texto). Denotaremos os caracteres em P e T por $p_1 p_2 p_3 \cdots p_m$ e $t_1 t_2 t_3 \cdots t_n$, respectivamente.

Como queremos encontrar todas as ocorrências do padrão P no texto T , uma solução será todas as quantidades às quais podemos deslocar P para achá-lo em T . Em outras palavras, dizemos que o padrão P **ocorre com deslocamento s** no texto T se a subcadeia de T que começa em t_{s+1} é a mesma do padrão P : $t_{s+1} = p_m, t_{s+2} = p_2$, e assim por diante, até $t_{s+m} = p_m$. O mínimo deslocamento possível seria 0 e, como o padrão não deve ultrapassar o final do texto, o máximo deslocamento possível seria $n - m$. Queremos conhecer todos os deslocamentos de P que ocorrem em T . Por exemplo, se o texto T é GTAA-CAGTAAACG e o padrão P é AAC, então P ocorre em T com deslocamentos 2 e 9.

Se estivermos verificando para ver se o padrão P ocorre no texto T com uma quantidade de deslocamento dada s , teremos de verificar todos os m caracteres em P em relação aos caracteres de T . Considerando que leva tempo constante para verificar um único caractere em P em relação a um único caractere em T , levaria o tempo $\Theta(m)$ para verificar todos os m caracteres no pior caso. É claro que, uma vez encontrada uma discordância entre os caracteres de P e T , não temos de verificar o resto dos caracteres. O pior caso ocorre a cada quantidade de deslocamento para a qual P não ocorre em T .

Seria bem fácil apenas verificar o padrão em relação ao texto para todo deslocamento possível, indo de 0 a $n - m$. Mostramos aqui como isso funcionaria para verificar o padrão AAC em relação ao texto GTAACAGTAAACG para cada deslocamento possível. As concordâncias de caracteres estão sombreadas:

Quantidade de deslocamento	Texto e padrão
0	GTAACAGTAAACG AAC
1	GTAACAGTAAACG AAC
2	GTAACAGTAAACG AAC
3	GTAACAGTAAACG AAC

Quantidade de deslocamento	Texto e padrão
4	GTAACAGTAAACG AAC
5	GTAACAGTAAACG AAC
6	GTAACAGTAAACG AAC
7	GTAACAGTAAACG AAC
8	GTAACAGTAAACG AAC
9	GTAACAGTAAACG AAC
10	GTAACAGTAAACG AAC

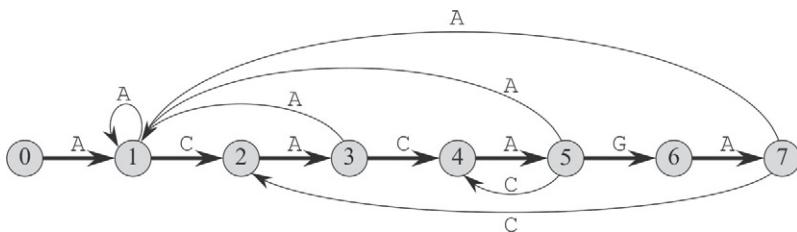
Todavia, essa abordagem simples é bastante ineficiente: com $n - m + 1$ deslocamentos possíveis, cada um levando tempo $O(m)$ para verificar, o tempo de execução seria $O((n - m)m)$. Examinaríamos quase todo caractere no texto m vezes.

Podemos fazer algo melhor, porque o método simples de verificar o padrão em relação ao texto para todo deslocamento possível joga fora informações valiosas. No exemplo que acabamos de dar, quando examinamos a quantidade de deslocamento $s = 2$, já vimos todos os caracteres na subcadeia $t_3t_4t_5 = AAC$. Porém, no próximo deslocamento, $s = 3$, examinamos t_4 e t_5 novamente. Seria mais eficiente evitar examinar esses caracteres novamente se fosse possível. Vamos examinar uma abordagem inteligente para correspondência de cadeias que evita o desperdício de tempo causado pelas repetidas varreduras do texto. Em vez de examinar caracteres de texto m vezes, ela examina cada caractere do texto exatamente uma vez.

Essa abordagem mais eficiente recorre a um **autômato finito**. Embora o nome pareça impressionante, o conceito é bem simples. Há inúmeras aplicações de autômatos finitos, mas aqui focalizaremos a utilização de autômatos finitos para correspondência de cadeias. Um autômato finito, ou **AF**, para abreviar, é apenas um conjunto de **estados** e um modo de ir de estado para estado tendo como base uma sequência de caracteres de entrada. O AF começa em um estado particular e consome caracteres de sua entrada, um caractere por vez. Tendo como base o estado em que está e o caractere que acabou de consumir, ele passa para um novo estado.

Em nossa aplicação de correspondência de cadeias, a sequência de entrada será os caracteres do texto T , e o AF terá $m + 1$ estados, um a mais que o número de caracteres no padrão P , numerados de 0 a m . (A parte “finito” do nome “autômato finito” deve-se ao fato de que o número de estados é finito.) O AF começa no estado 0. Quando ele está no estado k , os k caracteres de texto mais recentes que ele consumiu correspondem aos k primeiros caracteres do padrão. Portanto, sempre que o AF chega ao estado m , ele acabou de ver o padrão inteiro no texto.

Vamos examinar um exemplo usando apenas os caracteres A, C, G, e T. Suponha que o padrão seja ACACAGA, com $m = 7$ caracteres. Eis aqui o AF correspondente, com estados 0 até 7:

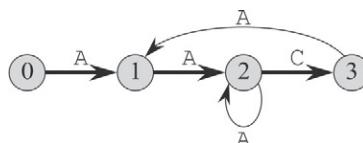


Círculos representam estados, e setas, rotuladas com caracteres, mostram como o AF transita de estado a estado de acordo com caracteres de entrada. Por exemplo, as setas que saem do estado 5 são denominadas A, C e G. A seta que aponta para o estado 1, rotulada com A, indica que, quando o AF está no estado 5 e consome o caractere de texto A, ele passa para o estado 1. De modo semelhante, a seta que aponta para o estado 4, rotulada com C, nos diz que, quando o AF está no estado 5 e consome o caractere de texto C, ele passa para o estado 4. Observe que eu desenhei a “espinha” horizontal do AF com setas reforçadas e que os rótulos nas setas da espinha, lidos da esquerda para a direita, dão o padrão ACACAGA. Sempre que o padrão ocorre no texto, o AF desloca-se para a direita um estado para cada caractere, até alcançar o último estado, onde declara que encontrou uma ocorrência do padrão no texto. Observe também que algumas setas estão faltando; por exemplo, qualquer seta denominada T. Se uma seta estiver faltando, a transição correspondente vai para o estado 0.

O AF armazena internamente uma tabela *next-state*, que é indexada por todos os estados e por todos os caracteres de entrada possíveis. O valor em $\text{next-state}[s, a]$ é o número do estado para o qual passar se o AF estiver no estado s no momento em questão e tenha acabado de consumir um caractere a do texto. Apresentamos a seguir a tabela *next-state* inteira para o padrão ACACAGA:

estado	caractere			
	A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

O AF move-se um estado para a direita para cada caractere que corresponde ao padrão, e para cada caractere que não corresponde ao padrão ele se move para a esquerda ou permanece no mesmo estado ($next-state[1, A]$). Veremos como construir uma tabela *next-state* mais adiante; porém, antes, vamos acompanhar o que o AF para o padrão AAC faz no texto de entrada GTAACAGTAAACG. Aqui está o AF:



Por esse desenho, você pode dizer que a tabela *next-state* é a seguinte:

estado	caractere			
	A	C	G	T
0	1	0	0	0
1	2	0	0	0
2	2	3	0	0
3	1	0	0	0

Aqui estão os estados para os quais AF passa e os caracteres de texto que ele consome para chegar lá.

estado 0 0 0 1 2 3 1 0 0 1 2 2 3 0

caractere G T A A C A G T A A A C G

Eu sombrei as duas vezes em que o AF alcança o estado 3, visto que sempre que ele chega ao estado 3 encontra uma ocorrência do padrão AAC.

Damos a seguir o procedimento FA-STRING-MATCHER para correspondência de cadeias. Ele pressupõe que a tabela *next-state* já foi construída.

Procedimento FA-STRING-MATCHER ($T, next-state, m, n$)

Entradas:

- T, n : uma cadeia de texto e seu comprimento.
- $next-state$: a tabela de transições de estado, formada de acordo com o padrão objeto da concordância.
- m : o comprimento do padrão. A tabela *next-state* tem linhas indexadas de 0 a m e colunas indexadas pelos caracteres que podem ocorrer no texto.

Saída: Imprime todas as quantidades de deslocamento para as quais o padrão ocorre no texto.

1. Iguale $state$ a 0.

2. Para $i = 1$ a n :

a. Iguale $state$ ao valor de $next-state[state, t_i]$

b. Se $state$ for igual a m , imprima “O padrão ocorre com deslocamento” $i - m$.

Se executarmos FA-STRING-MATCHER no exemplo anterior, no qual m é igual a 3, o AF alcança o estado 3 depois de consumir os caracteres t_5 e t_{12} . Portanto, o

procedimento imprimiria “Padrão ocorre com deslocamento 2” ($2 = 5 - 3$) e “Padrão ocorre com deslocamento 9” ($9 = 12 - 3$).

Visto que cada iteração do laço da etapa 2 leva tempo constante e esse laço executa exatamente n iterações, é simples ver que o tempo de execução de FA-STRING-MATCHER é $\Theta(n)$.

Essa é a parte fácil. A parte difícil é construir a tabela *next-state* do autômato finito para um padrão dado. Lembre-se da ideia:

Quando o autômato finito está no estado k , os k caracteres mais recentes que ele consumiu são os k primeiros caracteres do padrão.

Para tornar essa ideia concreta, vamos voltar ao AF na página 113 para o padrão ACACAGA e pensar por que *next-state* OE5; C_ é 4. Se o AF chegou ao estado 5, então os cinco caracteres mais recentes que ele consumiu do texto são ACACA, o que você pode ver examinando a espinha do AF. Se o próximo caractere consumido é C, então ele não corresponde ao padrão, e o AF não pode continuar até o estado 6. Porém, o AF tampouco tem de voltar atrás totalmente até o estado 0. Por que não? Porque agora os quatro caracteres mais recentemente consumidos são ACAC, que são os quatro primeiros caracteres do padrão ACACAGA. É por isso que, quando o AF está no estado 5 e consome um C, passa para o estado 4: ele viu mais recentemente os quatro primeiros caracteres do padrão.

Agora estamos quase prontos para dar a regra para construir a tabela *next-state*, mas antes precisamos de um par de definições. Lembre-se de que, para i na faixa 0 a m , o prefixo P_i do padrão P é a subcadeia que consiste nos i primeiros caracteres de P (quando i é 0, o prefixo é a cadeia vazia). Defina um *sufixo* do padrão, desse mesmo modo, como uma subcadeia de caracteres a partir do final de P . Por exemplo, AGA é um sufixo do padrão ACACAGA. E defina a *concatenação* de uma cadeia X e um caractere a como uma nova cadeia formada pela anexação de a ao final de X , e a denote Xa . Por exemplo, a concatenação da cadeia CA com o caractere T é a cadeia CAT.

Agora estamos finalmente prontos para construir $next-state[k, a]$, onde k é um número de estado que vai de 0 a m e a é qualquer caractere que poderia aparecer no texto. No estado k , acabamos de ver o prefixo P_k no texto. Isto é, os k caracteres de texto mais recentemente vistos são os mesmos que os primeiros k caracteres do padrão. Quando vemos o próximo caractere, digamos a , vimos $P_k a$ (a concatenação de P_k com a) no texto. Nesse ponto, qual é o comprimento de um prefixo de P que acabamos de ver? Outro modo de fazer essa pergunta é: qual é o comprimento de um prefixo de P que aparece no final de $P_k a$? Esse comprimento será o número do próximo estado.

Mais sucintamente:

Tome o prefixo P_k (os k primeiros caracteres de P) e concatene-o com o caractere a . Denote a cadeia resultante $P_k a$. Encontre o prefixo mais longo de P , que é também um sufixo de $P_k a$. Então, $next-state[k, a]$ é o comprimento desse prefixo mais longo.

Sim, há alguns prefixes e sufixos envolvidos, portanto vamos ver como determinar que $next-state[5, C]$ é 4 para o padrão $P = ACACAGA ACACAGA$. Visto que k é 5 nesse exemplo, tomamos o prefixo P_5 , que é ACACA, e o concatenamos com o caractere C, o que dá ACACAC. Queremos encontrar o prefixo mais longo de ACACAGA, que é também um sufixo de ACACAC. Visto que a cadeia ACACAC

tem comprimento 6, e um sufixo não pode ser mais longo do que a cadeia da qual ele é um sufixo, podemos iniciar examinando P_6 e continuar para baixo até prefixos cada vez mais curtos. Aqui, P_6 é ACACAG, e não é um sufixo de ACACAC. Portanto, agora consideramos P_5 , que é ACACA e também não é um sufixo de ACACAC. Em seguida consideramos P_4 , que é ACAC. Mas agora esse prefixo é um sufixo de ACA-CAC e, portanto, paramos e determinamos que $\text{next-state}[5, \text{C}]$ deve ser igual a 4.

Você bem poderia se perguntar se podemos sempre encontrar um prefixo de P que é também um sufixo de $P_k a$. A resposta é sim, porque a cadeia vazia é um prefixo, e um sufixo de toda cadeia. Quando acontece que o prefixo mais longo de P , que é também um sufixo de $P_k a$, é uma cadeia vazia, igualamos $\text{next-state}[k, a]$ a 0. Ainda usando o padrão $P = \text{ACACAGA}$, vamos ver como determinar $\text{next-state}[3, \text{G}]$. Concatenar P_3 com G dá a cadeia ACAG. Percorremos os prefixos de P , começando com P_4 (visto que o comprimento de ACAG é 4) e continuamos para baixo. Nenhum dos prefixos ACAC, ACA, AC ou A é um sufixo de ACAG, portanto nos contentamos com a cadeia vazia como o prefixo mais longo que funciona. Visto que a cadeia vazia tem comprimento 0, igualamos $\text{next-state}[3, \text{G}]$ a 0.

Quanto tempo leva para preencher a tabela *next-state* inteira? Sabemos que ela tem uma linha para cada estado no AF e, portanto, tem $m + 1$ linhas, numeradas de 0 a m . O número de colunas depende do número de caracteres que podem ocorrer no texto; vamos denominar esse número q , de modo que a tabela *next-state* tenha $q(m + 1)$ entradas. Para preencher uma entrada $\text{next-state}[k, a]$, fazemos o seguinte:

1. Forme a cadeia $P_k a$.
2. Iguale i ao menor de $k + 1$ (o comprimento de $P_k a$) e m (o comprimento de P).
3. Enquanto P_i não é um sufixo de $P_k a$, faça o seguinte:
 - a. Iguale i à $i - 1$.

Não sabemos de antemão quantas vezes o laço da etapa 3 executará, mas sabemos que ele faz, no máximo, $m + 1$ iterações. Também não sabemos de antemão quantos caracteres de P_i e $P_k a$ devem ser verificados no teste da etapa 3, mas sabemos que é sempre, no máximo, i , que é, no máximo, m . Visto que o laço itera, no máximo, $m + 1$ vezes e cada iteração verifica no máximo m caracteres, leva o tempo $O(m^2)$ para preencher $\text{next-state}[k, a]$. Como a tabela *next-state* contém $q(m + 1)$ entradas, o tempo total para preenchê-la é $O(m^3 q)$.

Na prática, o tempo para preencher a tabela *next-state* não é tão ruim. Eu codifiquei o algoritmo de correspondência de cadeias em C++ no meu Mac-Book Pro de 2,4-GHz e o compilei com nível de otimização -O3. Dei a ele o padrão **a man, a plan, a canal, panama** com o conjunto ASCII de 128 caracteres como alfabeto. O programa construiu uma tabela *next-state* com 31 linhas e 127 colunas (omiti a coluna para os caracteres nulos) em aproximadamente 1,35 milissegundo. Com um padrão mais curto, é claro que o programa é mais rápido: ele levou aproximadamente 0,07 milissegundo para construir a tabela quando o padrão era apenas **panama**.

Não obstante, algumas aplicações executam correspondência de cadeias frequentemente e, nessas aplicações, o tempo $O(m^3 q)$ para construir a tabela *next-state* poderia ser um problema. Não entrarei em detalhes, mas há um modo de reduzir o tempo $\Theta(mq)$. Na verdade, podemos fazer algo ainda melhor. O algoritmo “KMP”

(desenvolvido por Knuth, Morris e Pratt) usa um autômato finito, mas evita totalmente criar e preencher a tabela *next-state*. Em vez disso, ele usa um arranjo *move-to* com apenas m números de estado que permitem ao AF emular uma tabela *next-state*, e leva apenas o tempo $\Theta(m)$ para preencher o arranjo *move-to*. Novamente, é um pouco complicado demais para entrarmos em detalhes, mas eu executei o algoritmo KMP em meu MacBook Pro e, para o padrão **a man, a plan, a canal, panama**, ele levou aproximadamente um microsegundo para construir o arranjo *move-to*. Para o padrão mais curto *panama*, levou aproximadamente 600 nanosegundos (0,0000006 segundo). Nada mal! Do mesmo modo que o procedimento FA-STRING-MATCHER, o algoritmo KMP leva o tempo $\Theta(n)$ para achar a correspondência entre o padrão e o texto, uma vez que tenha construído o arranjo *move-to*.

O QUE MAIS LER?

O Capítulo 15 de CLRS [CLRS09] aborda programação dinâmica em detalhe, incluindo como encontrar uma subsequência comum mais longa. O algoritmo neste capítulo para transformar uma cadeia em outra dá parte da solução para um problema no Capítulo 15 de CLRS. (O problema no CLRS inclui as duas operações, intercambiando caracteres adjacentes e eliminando um sufixo de X , que eu não considerei neste capítulo. Você não pensou que eu iria irritar os meus coautores entregando de mão beijada a solução inteira, não é?)

Algoritmos de correspondência de cadeias aparecem no Capítulo 32 de CLRS. Esse capítulo dá o algoritmo baseado em autômatos finitos e também um tratamento completo do algoritmo KMP. A primeira edição de *Introduction to Algorithms* [CLR90] incluía o algoritmo de Boyer-Moore, que é particularmente eficiente quando o padrão é longo e o número de caracteres no alfabeto é grande.

Fundamentos de criptografia

Quando você compra alguma coisa pela Internet, provavelmente tem de fornecer o número do seu cartão de crédito a um servidor no site do vendedor ou a um servidor no site de algum serviço de pagamento terceirizado. Para que o número do seu cartão de crédito chegue a um servidor, você o envia pela Internet. A Internet é uma rede pública, e qualquer um pode discernir os bits que passam por ela. Portanto, se o número do seu cartão de crédito passar pela Internet sem ser disfarçado de alguma maneira, qualquer pessoa poderá saber qual é esse número e começar a comprar mercadorias e serviços na sua conta.

Agora, é improvável que alguém esteja sentado lá só esperando que *você* envie alguma coisa parecida com um número de cartão de crédito pela Internet. É mais provável que se esteja esperando que *alguém* faça isso e que *você* talvez seja uma infeliz vítima. Seria muito mais seguro para você disfarçar o número do seu cartão de crédito sempre que enviá-lo pela Internet. Na verdade, você provavelmente faz isso. Se usa um site seguro — um site cujo URL começa com “https:” em vez do usual “http:” —, o seu navegador disfarça a informação que envia por um processo denominado **criptografia**. (O protocolo https também provê “autenticação” de modo que você sabe que está se conectando ao site com o qual acha que está se conectando.) Neste capítulo, examinaremos criptografia, bem como o processo oposto, **decifração**, que faz a informação criptografada voltar à sua forma original. Juntos, os processos de criptografia e decifração formam o fundamento da área de criptografia.

Embora eu considere que o número do meu cartão de crédito seja uma informação importante a salvaguardar, também reconheço que não é assim tão importante no esquema geral das coisas. Se alguém roubar o número do meu cartão de crédito, a segurança nacional não estará em risco. Porém, se alguém puder bisbilhotar instruções dadas pelo Departamento de Estado a um diplomata ou se conseguir esquadrinhar informações militares, a segurança nacional poderá, de fato, estar em risco. Portanto, não somente precisamos de modos para criptografar e decifrar informações, mas esses modos precisarão ser dificílimos de derrotar.

Neste capítulo, examinaremos algumas das ideias básicas subjacentes à criptografia e decifração. A criptografia moderna vai longe, muito mais longe do que o que *eu* estou apresentando aqui. Não tente desenvolver um sistema seguro baseado unicamente no material deste capítulo; você precisaria entender criptografia moderna com muito mais

detalhes para criar um sistema que seja seguro tanto na teoria quanto na prática. Por exemplo, você precisaria seguir padrões estabelecidos como os publicados pelo National Institute of Standards and Technology. Como Ron Rivest (um dos inventores do criptossistema RSA, que veremos mais adiante neste capítulo) me escreveu: “Em geral, cripto é como uma disputa de artes marciais e, para usá-la na prática, você precisa entender as últimas manobras dos adversários.” Mas este capítulo lhe dará uma ideia de alguns algoritmos que foram motivados pela necessidade de saber como criptografar e decifrar informações.

Em criptografia, denominamos a informação original **texto comum**, e a versão criptografada, **texto cifrado**. Portanto, a criptografia converte texto comum em texto cifrado, e a decifração converte texto cifrado em texto comum original. A informação necessária para tal conversão é conhecida como **chave** criptográfica.

CIFRAS DE SUBSTITUIÇÃO SIMPLES

Em uma **cifra de substituição simples**, você criptografa um texto apenas substituindo uma letra por outra e decifra um texto criptografado invertendo a substituição. Júlio César teria se comunicado com seus generais usando uma **cifra de deslocamento**, pela qual o remetente substituía cada letra em uma mensagem pela letra que aparece três lugares adiante no alfabeto, voltando ao início ao chegar ao final do alfabeto. Em nosso alfabeto de 26 letras, por exemplo, A seria substituído por D e Y seria substituído por B (depois de Y vem Z e, então, A e B). Na cifra de deslocamento de César, se um general precisasse de mais tropas, poderia criptografar o texto comum *Send me a hundred more soldiers* (Envie mais 100 soldados) como o texto cifrado *Vhqg ph d kxqguhg pruh vroglhuv*. Ao receber esse texto cifrado, César substituiria cada letra pela letra que aparece três lugares antes no alfabeto, voltando até o início do alfabeto para recuperar o texto comum original *Send me a hundred mais soldiers* (na época de César, é claro, a mensagem teria sido escrita em latim usando o alfabeto latino existente na época).

Se você interceptar uma mensagem e souber que ela foi criptografada com uma cifra de deslocamento, é ridicamente fácil decifrá-la, ainda que não saiba de antemão qual é a quantidade a deslocar (a chave); basta tentar todos os deslocamentos possíveis até o texto decifrado fazer sentido como texto comum. Para um alfabeto de 26 caracteres, você precisaria tentar apenas 25 deslocamentos.

A sua cifra poderia ser um pouco mais segura se convertesse cada caractere em algum outro caractere único, mas não necessariamente aquele que aparece um número fixo de lugares mais adiante no alfabeto. Isto é, você cria uma permutação dos caracteres e a usa como a sua chave. Ainda é uma cifra de substituição simples, mas é melhor do que uma cifra de deslocamento. Se você tiver n caracteres em seu conjunto de caracteres, um bisbilhoteiro que interceptasse uma mensagem teria de discernir qual das $n!$ (fatorial n) permutações você usou. A função fatorial cresce muito rapidamente em n ; na verdade, cresce mais rapidamente que a função exponencial.

Então, por que não somente converter unicamente cada caractere em algum outro caractere? Se você já tentou resolver o quebra-cabeça “criptoquote” (“criptocitação”) que aparece em muitos jornais, sabe que pode usar frequências de letras e combinações de letras para reduzir suas escolhas. Suponha que o texto comum *Send me a hundred more soldiers* fosse convertido para o texto cifrado *Krcz sr h deczxrz sfxr kfjzgrxk*. No texto cifrado, a

letra *r* aparece com maior frequência, e você poderia adivinhar — corretamente — que seu caractere correspondente no texto comum é *e*, a letra que ocorre mais comumente nos textos em inglês. Então você poderia ver a palavra de duas letras *sr* no texto cifrado e adivinhar que o caractere de texto comum correspondente ao texto cifrado *s* deve ser *b* ou *h* ou *m* ou *w*, visto que as únicas palavras de duas letras em inglês que terminam em *e* são *be*, *he*, *me* e *we*. Você poderia também determinar que o texto comum *a* corresponde ao texto cifrado *h* porque a única palavra de uma única letra minúscula em inglês é *a*.

É claro que, se você estiver criptografando números de cartões de crédito, não terá de se preocupar muito com frequências de letras ou combinação de letras. Mas os 10 dígitos dão somente 10! modos únicos de converter um dígito em outro, ou 3.628.800. Para um computador, isso não é muito, especialmente quando comparado com os 10^{16} possíveis números de cartões de crédito (16 dígitos decimais), e um bisbilhoteiro poderia automatizar tentativas para viabilizar compras por cada um dos 10! modos — e possivelmente seria bem-sucedido com alguns números de cartão de crédito que não o seu.

Você poderia ter notado um outro problema com a utilização de uma cifra de substituição simples: ambos, remetente e destinatário, têm de concordar com a chave. Além disso, se você estiver enviando mensagens diferentes a participantes diferentes e não quiser que cada participante consiga decifrar mensagens dirigidas a outras pessoas, precisará estabelecer uma chave separada para cada participante.

CRIPTOGRAFIA DE CHAVE SIMÉTRICA

Quando o remetente e o destinatário usam a mesma chave, estão praticando *criptografia de chave simétrica*. Eles devem combinar com antecedência qual chave estão usando.

Cifras de chave única

Considerando, por enquanto, que você concorda em usar criptografia de chave simétrica, mas que uma cifra de substituição simples não é suficientemente segura, outra opção é a cifra de chave única. Cifras de chave única funcionam com bits. Como é claro que você sabe, *bit* é uma abreviação de “dígito binário”, e um bit pode adotar apenas dois valores: 0 e 1. Os computadores digitais armazenam informações em sequências de bits. Algumas sequências de bits representam números, outras representam caracteres (usando os conjuntos de caracteres padrões ASCII ou Unicode) e ainda outras até representam instruções que o computador executa.

Cifras de chave única aplicam a operação *exclusive-or* ou *XOR* aos bits. Usamos \oplus para denotar essa operação:

$$\begin{aligned} 0 \oplus 0 &= 0 \\ 0 \oplus 1 &= 1 \\ 1 \oplus 0 &= 1 \\ 1 \oplus 1 &= 0 \end{aligned}$$

O modo mais simples de imaginar a operação XOR é que, se *x* é um bit, então $x \oplus 0 = x$ e $x \oplus 1 = \text{o oposto de } x$. Se *x* e *y* são bits, então se $(x \oplus y) \oplus y = x$: aplicar uma operação XOR a *x* com o mesmo valor duas vezes dará *x*.

Suponha que eu queira lhe enviar uma mensagem de um bit. Eu poderia lhe enviar um 0 ou um 1 como o texto cifrado, e nós teríamos de combinar se eu estava enviando o valor de bit que queria enviar ou o oposto desse valor de bit. Pelas lentes da operação XOR, teríamos de combinar se eu estava aplicando uma operação XOR àquele bit com 0 ou com 1. Então, se você fosse aplicar a operação XOR entre o bit do texto cifrado que recebeu e o bit ao qual eu tinha aplicado a operação XOR — a chave —, você recuperará o texto comum original.

Agora suponha que eu queira lhe enviar uma mensagem de dois bits. Eu poderia deixar ambos os bits em paz, trocar a ordem de ambos os bits, trocar a ordem do primeiro bit mas não a do segundo ou trocar a ordem do segundo bit, mas não a do primeiro. Novamente, nós teríamos de combinar de quais bits eu estaria trocando a ordem (se eu trocassem a ordem de algum). Em termos da operação XOR em dois bits, teríamos de combinar qual das duas sequências de dois bits 00, 01, 10 ou 11 era a chave com a qual eu estaria aplicando a operação XOR aos bits do texto comum para formar texto cifrado. Novamente, você poderia aplicar a operação XOR aos dois bits do texto cifrado com a mesma chave de dois bits com a qual eu tinha aplicado a operação XOR ao texto comum para recuperar o texto comum original.

Se o texto comum exigisse b bits — talvez ele compreenda caracteres ASCII ou Unicode que totalizam b bits —, eu poderia gerar uma sequência aleatória de b bits como chave, passar a você essa chave de b bits e aplicar a operação XOR, bit por bit, ao texto comum com a chave para formar o texto cifrado. Assim que você recebesse o texto cifrado de b bits, poderia aplicar a operação XOR a ele, bit por bit, com a chave para recuperar o texto comum de b bits. Esse sistema é denominado *cifra de chave única ou ‘one-pad’*,¹ e a chave é denominada *pad*.

Contanto que os bits da chave sejam escolhidos aleatoriamente — e examinaremos essa questão mais adiante —, é quase impossível um bisbilhoteiro decifrar o texto cifrado adivinhando a chave. Mesmo que o bisbilhoteiro soubesse alguma coisa sobre o texto comum — por exemplo, que ele está em inglês —, para qualquer texto cifrado e qualquer texto comum *potencial*, existe uma chave que converte o texto comum potencial em um texto cifrado,² e essa chave é a operação XOR bit por bit entre o texto comum potencial e o texto cifrado. (Isso porque, se o texto comum potencial é t , o

¹ O nome vem da realização pré-computador da ideia pela qual cada participante tinha um bloquinho (*pad*) de papel no qual uma chave estava escrita em cada folha, e os participantes tinham sequências de chaves idênticas. Uma chave podia ser usada uma única vez, e a folha onde ela estava escrita era arrancada do bloquinho, expondo a próxima chave. Esse sistema baseado em papel usava uma cifra de deslocamento, mas na base de letra por letra, na qual cada letra correspondente da chave dava a quantidade de deslocamento, de 0 a 25 para z . Por exemplo, visto que z significa deslocar 25, m significa deslocar 12 e n significa deslocar 13, a chave zmn converte o texto comum *dog* no texto cifrado *cat*. Todavia, diferentemente do sistema baseado na operação XOR, deslocar as letras no texto cifrado na mesma direção com a mesma chave não recupera o texto comum; nesse caso, ele daria “*bmg*”. Em vez disso, você tem de deslocar as letras do texto cifrado na direção oposta.

² Para o esquema letra por letra citado na nota de pé de página anterior, a chave zmn converte o texto comum *dog* no texto cifrado *cat*, mas podemos chegar a esse texto cifrado com um texto comum diferente, *elk*, e uma chave diferente, *ypy*.

texto cifrado é c , e a chave é k ; então, não somente $t \oplus k = c$, mas também $t \oplus c = k$; a operação \oplus aplica-se bit por bit a t , k e c , de modo que a operação XOR entre o i -ésimo bit de t e o i -ésimo bit de k é igual ao i -ésimo bit de c .) Assim, criptografar com uma cifra de chave única impede o bisbilhoteiro de conseguir qualquer informação adicional sobre o texto comum.

Cifras de chave única dão boa segurança, mas as chaves exigem tantos bits quanto o texto comum; esses bits devem ser escolhidos aleatoriamente e as chaves precisam ser compartilhadas entre as partes com antecedência. Como o nome subentende, você deve usar uma cifra de chave única apenas uma vez. Se usar a mesma chave k para os textos comuns t_1 e t_2 , então $(t_1 \oplus k) \oplus (t_2 \oplus k) = t_1 \oplus t_2$, o que pode revelar onde os dois textos comuns têm os mesmos bits.

Cifras de bloco e encadeamento

Quando o texto comum é longo, o ‘bloquinho’ em uma cifra de chave única tem de ser igualmente longo, o que pode ser bastante inflexível. Em vez disso, alguns sistemas de chave simétrica combinam duas técnicas adicionais: usam uma chave mais curta e desmembram o texto comum em vários blocos, aplicando a chave a cada bloco por vez. Isto é, eles consideram que o texto comum é 1 blocos $t_1, t_2, t_3, \dots, t_l$ e criptografam esses blocos de texto comum em 1 blocos $c_1, c_2, c_3, \dots, c_l$ de texto cifrado. Tal sistema é conhecido como *cifra de bloco*.

Na prática, cifras de bloco criptografam usando um sistema bem mais complicado do que simplesmente aplicando operação XOR, como na cifra de chave única. Um criptossistema de chave simétrica frequentemente usado, o AES (Advanced Encryption Standard), incorpora uma cifra de bloco. Não entrarei em detalhes sobre o AES; direi apenas que ele usa métodos elaborados para fatiar e picar um bloco de texto comum e produzir texto cifrado. O AES usa um tamanho de chave de 128, 192 ou 256 bits e um tamanho de bloco de 128 bits.

Todavia, há ainda um problema com cifras de bloco. Se o mesmo bloco aparecer duas vezes no texto comum, o mesmo bloco criptografado aparecerá duas vezes no texto cifrado. Um modo de resolver esse problema usa a técnica de *encadeamento de cifra de bloco*. Suponha que você queira me enviar uma mensagem criptografada. Você retalha o texto comum t em 1 blocos $t_1, t_2, t_3, \dots, t_l$ e cria os 1 blocos $c_1, c_2, c_3, \dots, c_l$ de texto cifrado da maneira descrita a seguir. Vamos dizer que você criptografará um bloco aplicando alguma função E a ele e que eu decifrarei um bloco de texto cifrado aplicando alguma função D . Você cria o primeiro bloco de texto cifrado, c_1 , como seria de esperar: $c_1 = E(t_1)$. Mas, antes de criptografar o segundo bloco, você aplica uma operação XOR, bit por bit, entre ele e c_1 , de modo que $c_2 = E(c_1 \oplus t_2)$. Para o terceiro bloco, você aplica uma operação XOR entre ele e c_2 : $c_3 = E(c_2 \oplus t_3)$. E assim por diante, de modo que, em geral, você computa o i -ésimo bloco de texto cifrado com base no $(i - 1)$ º bloco de texto cifrado e o i -ésimo bloco de texto comum: $c_i = E(c_{i-1} \oplus t_i)$. Essa fórmula funciona até mesmo para computar c_1 a partir de t_1 se você iniciar com c_0 sendo todo 0s (porque $(0 \oplus x)$ dá x). Para decifrar, eu primeiro computo $t_1 = D(c_1)$. A partir de c_1 e c_2 , eu posso computar t_1 primeiro computando $D(c_2)$, que é igual a $c_1 \oplus t_2$, e então aplicando uma operação XOR entre o resultado e c_1 . Em geral, decifro

c_i para determinar t_i computando $t_i = D(c_2) \oplus c_{i-1}$; como ocorre com a criptografia, esse esquema funciona até mesmo para computar t_1 se eu iniciar com c_0 sendo todos 0s.

Ainda não passamos da fase crítica. Mesmo com encadeamento de cifra de bloco, se você me enviar a mesma mensagem duas vezes enviará a mesma sequência de blocos de texto cifrado a cada vez. Um bisbilhoteiro saberia que você está me enviando a mesma mensagem duas vezes, o que poderia ser uma informação valiosa para ele. Uma solução é não iniciar com c_0 sendo todos 0s. Em vez disso, você gera aleatoriamente c_0 , usa isso quando criptografar o primeiro bloco de texto comum e eu o uso quando decifro o primeiro bloco de texto cifrado; denominamos esse c_0 gerado aleatoriamente *vetor de inicialização*.

Concordância com informação comum

Para a criptografia de chave simétrica funcionar, o remetente e o destinatário precisam concordar com a chave. Além disso, se estiverem usando uma cifra de bloco com encadeamento de cifra de bloco, eles talvez também precisam concordar com o vetor de inicialização. Como você pode imaginar, raramente é prático concordar com esses valores antecipadamente. Então, como o remetente e o destinatário concordam com a chave e com o vetor de inicialização? Veremos mais adiante neste capítulo (página 132) como um criptossistema híbrido pode transmiti-los com segurança.

CRIPTOGRAFIA DE CHAVE PÚBLICA

É óbvio que, para o destinatário de uma mensagem conseguir decifrá-la, o remetente e o destinatário devem conhecer a chave usada para criptografar. Certo?

Errado.

Em *criptografia de chave pública*, cada participante tem duas chaves: uma **chave pública** e uma **chave secreta**. Descreverei a criptografia de chave pública com dois participantes, você e eu, e denotarei minha chave pública P e minha chave secreta S . Você tem as suas próprias chaves pública e secreta. Outros participantes têm suas próprias chaves pública e secreta.

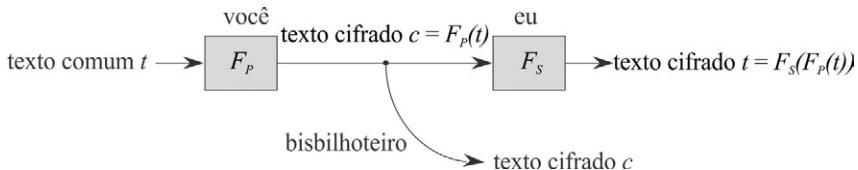
Chaves secretas são secretas, mas as chaves públicas todos podem conhecer. Elas podem até mesmo aparecer em um diretório centralizado que permite que todos conheçam a chave pública de todos os outros. Sob as condições certas, você e eu podemos usar qualquer dessas chaves para criptografar e decifrar. Por “condições certas” eu quero dizer que existem funções que usam as chaves pública e secreta para criptografar texto comum e obter texto cifrado ou decifrar texto cifrado e obter texto comum. Vamos denotar a função que eu uso como a minha chave pública F_P e a função que eu uso com a minha chave secreta F_S . As chaves pública e secreta têm uma relação especial:

$$t = F_s(F_p(t)),$$

de modo que, se você usar a minha chave pública para criptografar texto comum e obter texto cifrado, e eu usar a minha chave secreta para decifrar o texto cifrado, eu recupero o texto comum original. Algumas outras aplicações de criptografia de chave pública exigem que $t = F_s(F_p(t))$, de modo que, se eu criptografar texto comum com minha chave secreta, qualquer um pode decifrar o texto cifrado.

Qualquer pessoa deve ser capaz de computar a minha função chave pública F_P eficientemente, mas somente eu devo ser capaz de computar a minha função chave secreta F_S em qualquer quantidade de tempo razoável. O tempo exigido para conseguir adivinhar a minha F_S sem saber qual é a minha chave secreta deve ser proibitivamente grande para qualquer outra pessoa. (Sim, estou sendo vago aqui, mas logo veremos uma implementação real de criptografia de chave pública). O mesmo vale para as chaves pública e secreta de qualquer outra pessoa: a função chave pública F_P é eficientemente computável, mas somente o portador da chave secreta pode computar a função chave secreta F_P razoavelmente.

Eis como você pode me enviar uma mensagem usando criptografia de chave pública:



Você inicia com o texto comum t . Você acha a minha chave pública P ; talvez a obtenha diretamente de mim ou talvez a encontre em um diretório. Assim que tiver P , você criptografa o texto comum para produzir o texto cifrado $c = F_p(t)$, o que você pode fazer eficientemente. Você me envia o texto cifrado, de modo que qualquer bisbilhoteiro que intercepte o que você me enviou verá somente o texto cifrado. Eu pego o texto cifrado c e o decifro usando a minha chave secreta, reproduzindo o texto comum $c = F_s(c)$. Você ou qualquer outra pessoa pode criptografar para produzir o texto cifrado com razoável rapidez, mas somente eu posso decifrar o texto cifrado para reproduzir o texto comum em qualquer quantidade de tempo razoável.

Na prática, precisamos assegurar que as funções F_P e F_S funcionem juntas corretamente. Queremos que F_P produza um texto cifrado diferente para cada texto comum possível. Em vez disso, suponha que F_P deu o mesmo resultado para dois textos comuns diferentes, t_1 e t_2 , isto é, $F_p(t_1) = F_p(t_2)$. Então, quando eu receber um texto cifrado $F_p(t_1)$ e tentar decifrá-lo aplicando a função F_S , não sei se voltarei a t_1 ou a t_2 . Por outro lado, tudo bem — na verdade é até preferível — que a criptografia incorpore um elemento de aleatorização, de modo que o mesmo texto comum é criptografado em textos cifrados diferentes cada vez que passar por F_P . (O criptossistema RSA, que veremos logo adiante, é muito mais seguro quando o texto comum é somente uma pequena porção daquilo que é criptografado, e o grosso das informações criptografadas é um “recheio” aleatório.). É claro que a função decifração F_S precisaria ser projetada para compensar, de modo que pudesse converter vários textos cifrados para o mesmo texto comum.³

³ O beisebol usa um sistema semelhante. Dirigentes e treinadores informam aos jogadores quais jogadas usar mediante um elaborado sistema de gestos denominado “sinais”. Por exemplo, tocar o ombro direito pode significar executar uma jogada bate e corre, e tocar a coxa esquerda pode significar um *bunt* (acertar a bola de leve). Um dirigente ou treinador faz uma longa série de sinais, mas somente alguns deles são significativos; o resto são engodos. Quem faz os sinais e quem os recebe têm um sistema pelo qual concordam de antemão quais sinais são significativos, algumas vezes com base na ordenação da sequência de sinais e outras vezes com base em um sinal “indicador”. O dirigente ou treinador pode dar uma série arbitrariamente longa de sinais para indicar qualquer jogada particular, na qual a maioria dos sinais na série nada significa.

Todavia, surge um problema. O texto comum t poderia adotar um número arbitrário de valores possíveis — na verdade, ele poderia ser arbitrariamente longo —, e o número de valores de texto cifrado ao qual F_P poderia converter t tem de ser, no mínimo, igual ao número de valores que t poderia adotar. Como podemos construir as funções F_P e F_S sob essas restrições adicionais: F_P tem de ser fácil de computar para todo mundo e F_S tem de ser fácil somente para mim? É difícil, mas é factível se pudermos limitar o número de textos comuns possíveis, isto é, usamos uma cifra de bloco.

O CRIPTOSSISTEMA RSA

Criptografia de chave pública é um lindo conceito, mas depende de ser capaz de determinar funções F_P e F_S que funcionem corretamente juntas. F_P é fácil para qualquer um computar e F_S é fácil somente para o portador da chave secreta computar. Denominamos um esquema que cumpre esses critérios *criptossistema de chave pública*, e o *criptossistema RSA*, ou apenas **RSA**.⁴ é um desses esquemas.

O RSA depende de diversas facetas da teoria dos números, muitas das quais estão relacionadas à *aritmética modular*. Em aritmética modular, escolhemos um inteiro positivo, digamos n , e sempre que chegamos a n imediatamente voltamos a 0. É como aritmética comum com inteiros, mas sempre dividimos por n e tomamos o resto. Por exemplo, se estivermos trabalhando com módulo 5, os únicos valores possíveis são 0, 1, 2, 3, 4 e $3 + 4 = 2$, visto que 7 dividido por 5 dá resto 2. Vamos definir um operador, mod, para calcular restos, de modo que possamos dizer $7 \text{ mod } 5 = 2$. Aritmética modular é como aritmética de relógio, porém substituindo 12 por 0 no mostrador do relógio. Se você foi dormir às 11 horas e dormir oito horas, acordará às 7: $(11 + 8) \text{ mod } 12 = 7$.

O que é particularmente interessante na aritmética modular é que podemos adotar operações mod no meio de expressões e não mudar o resultado:⁵

$$\begin{aligned} (a + b) \text{ mod } n &= ((a \text{ mod } n) + (b \text{ mod } n)) \text{ mod } n; \\ ab \text{ mod } n &= ((a \text{ mod } n)(b \text{ mod } n)) \text{ mod } n; \\ a^b \text{ mod } n &= (a \text{ mod } n)^b \text{ mod } n. \end{aligned}$$

Para qualquer inteiro x , temos que $xn \text{ mod } n = 0$.

Além disso, para que o RSA cumpra os critérios para um criptossistema de chave pública, duas propriedades da teoria dos números relacionadas a números primos devem valer. Como você deve saber, um *número primo* é um inteiro maior que 1 que tem somente dois fatores inteiros: 1 e ele mesmo. Por exemplo, 7 é primo, mas 6 não é, já que pode ser fatorado como 2·3. A primeira propriedade à qual RSA recorre é que, se você tiver um número que é o produto de dois números primos grandes secretos, ninguém mais pode

⁴ O nome deve-se a seus inventores, Ronald Rivest, Adi Shamir e Leonard Adelman.

⁵ Como exemplo, para ver que $ab \text{ mod } n = ((a \text{ mod } n)(b \text{ mod } n)) \text{ mod } n$, suponha que $a \text{ mod } n = x$ e $b \text{ mod } n = y$. Então existem inteiros i e j tais que $a = ni + x$ e $b = nj + y$ e, portanto,

$$\begin{aligned} ab \text{ mod } n &= (ni + x)(nj + y) \text{ mod } n \\ &= (n^2ij + xnj + yni + xy) \text{ mod } n \\ &= ((n^2ij \text{ mod } n) + (xnj \text{ mod } n) + (yni \text{ mod } n) + (xy \text{ mod } n)) \text{ mod } n \\ &= xy \text{ mod } n \\ &= ((a \text{ mod } n)(b \text{ mod } n)) \text{ mod } n \end{aligned}$$

determinar esses fatores em qualquer quantidade de tempo razoável. Lembre-se de que dissemos no Capítulo 1 que alguém poderia testar todos os divisores ímpares possíveis até a raiz quadrada do número, mas se o número for grande — centenas de milhares de dígitos —, sua raiz quadrada tem metade desse número de dígitos, o que ainda pode ser muito grande. Embora *teoricamente* alguém possa encontrar um desses fatores, os recursos exigidos (tempo e/ou poder de computação) tornariam impraticável achar um fator.⁶

A segunda propriedade é que, embora fatorar um número primo grande seja difícil, não é difícil determinar se um número grande é primo. Você pode achar que é impossível determinar que um número não é primo — isto é, o número é **composto** — sem determinar, no mínimo, um fator não trivial (um fator que não é 1 nem é o próprio número). Na verdade, é possível fazer isso. Um modo é o teste de primalidade AKS,⁷ o primeiro algoritmo a determinar se um número de n bits é primo no tempo $O(n^c)$ para alguma constante c . Embora o teste de primalidade AKS seja considerado teoricamente eficiente, ainda não é prático para números grandes. Em vez disso, podemos usar o teste de primalidade Miller-Rabin. A desvantagem do teste Miller-Rabin é que ele pode cometer erros e declarar que um número é primo quando na verdade ele é composto (todavia, se o teste declarar que um número é composto, o número é definitivamente composto). A boa notícia é que a taxa de erro é 1 em 2^s , onde podemos escolher o valor positivo de s que quisermos. Portanto, se estivermos dispostos a conviver com um erro em, digamos, cada 2^{50} testes, podemos determinar com certeza *quase* perfeita se um número é primo. Lembre-se de que dissemos no Capítulo 1 que 2^{50} é aproximadamente um milhão de bilhões, ou aproximadamente 1.000.000.000.000.000. Se você ainda se sentir desconfortável com um erro em 2^{50} testes, com um pouco mais de esforço pode transformá-lo em um erro em 2^{60} testes; 2^{60} é aproximadamente mil vezes maior que 2^{50} . Isso porque o tempo para executar o teste Miller-Rabin aumenta apenas linearmente com o parâmetro s e, portanto, aumentar s de 10, isto é, de 50 para 60, aumenta o tempo de execução somente em 20%, mas diminui a taxa de erro por um fator de 2^{10} , que é igual a 1.024.

Eis como eu me prepararia para usar o criptossistema RSA. Depois de vermos como o RSA funciona, temos de abordar vários detalhes.

1. Escolha aleatoriamente dois números primos muito grandes, p e q , que não são iguais um ao outro. Quão grande é muito grande? No mínimo, 1.024 bits cada ou, no mínimo, 309 dígitos decimais. Maior é melhor ainda.
2. Compute $n = pq$. Esse é um número que tem, no mínimo, 2.048 bits ou, no mínimo, 618 dígitos decimais.
3. Compute $r = (p - 1)(q - 1)$, que é quase tão grande quanto n .
4. Selecione um inteiro ímpar pequeno e que seja *relativamente primo* de r : o único divisor comum de e e r deve ser 1. Qualquer tal inteiro pequeno serve.
5. Compute d como o *inverso multiplicativo* de e , módulo r . Isto é, $ed \bmod r$ deve ser igual a 1.

⁶ Por exemplo, se o número tiver 1.000 bits, sua raiz quadrada terá 500 bits, e pode ser tão grande quanto 2^{500} . Mesmo que alguém pudesse testar um trilhão de trilhões possíveis divisores por segundo, o Sol já teria se apagado há muito tempo, antes de esse alguém alcançar 2^{500} .

⁷ O nome deve-se a seus inventores, Manindra Agrawal, Neeraj Kayal e Nitin Saxena.

6. Declare que minha **chave pública RSA** é o par $P = (e, n)$.
7. Mantenha o par $S = (d, n)$ como minha **chave secreta RSA** e não a revele a ninguém.
8. Defina as funções F_P e F_S por

$$F_p(x) = x^e \bmod n;$$

$$F_s(x) = x^d \bmod n :$$

Essas funções podem operar sobre um bloco de texto comum ou sobre um bloco de texto cifrado, cujos bits interpretamos como representativos de inteiros grandes.

Vamos examinar um exemplo, mas usando números pequenos para que possamos entender o que está acontecendo.

1. Escolha os números primos $p = 17$ e $q = 29$.
2. Compute $n = pq = 493$.
3. Compute $r = (p - 1)(q - 1) = 448$.
4. Selecione $e = 5$, que é relativamente primo de 448 (ou seja, são primos entre si).
5. Compute $d = 269$. Para verificar: $ed = 5 \cdot 269 = 1345$ e, portanto, $ed \bmod r = 1345 \bmod 448 = (3 \cdot 448 + 1) \bmod 448 = 1$.
6. Declare que minha chave pública RSA é $P = (5, 493)$.
7. Mantenha $S = (269, 493)$ como minha chave secreta RSA.
8. Como exemplo, vamos computar $F_P(327)$:

$$F_p(327) = 327^5 \bmod 493 = 3.738.856.210.407 \bmod 493 = 259$$

Se computarmos $F_S(259) = 259^{269} \bmod 493$, devemos obter 327 de volta. Obtemos, mas na verdade você não quer ver todos os dígitos na expressão 259^{269} . Você pode procurar na Internet uma calculadora de precisão arbitrária e testar esse número com ela (eu testei). Então, novamente, como estamos trabalhando com aritmética modular, não precisamos computar o valor real de 259^{269} ; podemos expressar todos os resultados intermediários módulo 493; portanto, se você quisesse, poderia começar com o produto 1 e fazer o seguinte 269 vezes: multiplique o que você tem por 259 e tome o produto módulo 493. Você obterá um resultado de 327 (eu fiz isso, ou melhor, um programa de computador que eu escrevi fez).

- Damos a seguir os detalhes a abordar para montar e usar o RSA:
- Como se trabalha com números que têm centenas de dígitos?
 - Embora não seja um obstáculo testar se um número é primo, como eu sei que posso encontrar números primos grandes em uma quantidade de tempo razoável?
 - Como eu acho e de modo que e e r são relativamente primos?
 - Como eu computo d de modo que seja o inverso multiplicativo de e , módulo r ?
 - Se d é grande, como eu computo $x^d \bmod n$ em uma quantidade de tempo razoável?
 - Como eu sei que as funções F_P e F_S são inversas uma da outra?

Como fazer aritmética com números grandes

É claro que números tão grandes como os exigidos pelo RSA não caberão nos registradores encontrados na maioria dos computadores, que contêm no máximo 64 bits. Felizmente, diversos pacotes de software e até mesmo algumas linguagens de programação — Python,

por exemplo — permitem que você trabalhe com inteiros que não têm limites fixos para seu tamanho.

Além do mais, toda a aritmética utilizada em RSA é aritmética modular, o que nos permite limitar os tamanhos dos inteiros que estão sendo calculados. Por exemplo, como estamos calculando $x^d \bmod n$, estaremos calculando resultados intermediários que são x elevado a várias potências, mas todas módulo n , o que significa que todos os resultados intermediários calculados estarão na faixa de 0 a $n - 1$. Se você fixar os tamanhos máximos de p e q , terá fixado o tamanho máximo de n , o que por sua vez significa que é factível implementar RSA em hardware especializado.

Como encontrar um número primo grande

Eu posso encontrar um número primo grande gerando repetida e aleatoriamente um número ímpar grande e usando o teste de primalidade Miller-Rabin para determinar se esse número é primo, parando assim que encontrar um número primo. Esse esquema pressupõe que não levará muito tempo para eu encontrar um número primo. E se os números primos forem extremamente raros à medida que os números ficam grandes? Eu poderia gastar imensa quantidade de tempo procurando um primo “agulha” em um palheiro de compostos.

Mas eu não preciso me preocupar. O *teorema do número primo* nos diz que, à medida que m se aproxima do infinito, a quantidade de números primos menores ou iguais a m aproxima-se de $m/\ln m$, onde $\ln m$ é o logaritmo natural de m . Se eu apenas selecionar aleatoriamente um inteiro m , há aproximadamente *uma* em $\ln m$ chances de ele ser primo. A teoria da probabilidade nos diz que, na média, eu preciso experimentar apenas aproximadamente $\ln m$ números próximos de m antes de encontrar um que é primo. Se eu estiver procurando números primos p e q com 1.024 bits, então m é 2^{1024} e $\ln m$ é aproximadamente 710. Um computador pode executar rapidamente o teste de primalidade Miller-Rabin em 710 números.

Na prática, eu poderia executar um teste de primalidade mais simples que o Miller-Rabin. O *pequeno teorema de Fermat* afirma que, se m é um número primo, então $x^{m-1} \bmod m$ é igual a 1 para qualquer número x na faixa de 1 a $m - 1$. O inverso — se $x^{m-1} \bmod m$ é igual a 1 para qualquer número m na faixa de 1 a $m - 1$, então m é primo — não é necessariamente verdade, mas exceções são muito raras para números grandes. Na verdade, é quase sempre suficiente apenas experimentar inteiros ímpares m e declarar que $m - 1$ é primo se $2^{m-1} \bmod m$ for igual a 1. Veremos na página 131 como computar $2^{m-1} \bmod m$ com apenas $\Theta(\lg m)$ multiplicações.

Como encontrar um número que é relativamente primo de outro

Eu preciso encontrar um inteiro pequeno ímpar e que seja relativamente primo de r . Dois números são relativamente primos (ou primos entre si) se seu máximo divisor comum for 1. Usarei um algoritmo para computar o máximo divisor comum de dois inteiros que remonta a Euclides, matemático grego da Antiguidade. Há um teorema na teoria dos números que diz que, se a e b são inteiros e ambos não são zero, então seu máximo divisor comum g é igual a $ai + bj$ para alguns inteiros i e j (além do mais, g é

o menor número que pode ser formado desse modo, mas esse fato não nos importa). Um dos coeficientes i e j pode ser negativo; por exemplo, o máximo divisor comum de 30 e 18 é 6 e $6 = 30i + 18j$ quando $i = -2$ e $j = 2$.

O algoritmo de Euclides é apresentado sob uma forma que dá o máximo divisor comum g de a e b , juntamente com os coeficientes i e j . Esses coeficientes virão a calhar um pouco mais adiante, quando eu precisar encontrar o inverso multiplicativo de e módulo r . Se eu tiver um valor candidato para e , chamarei EUCLID(r,e). Se o primeiro elemento da tripla retornada pela chamada for 1, o valor candidato para e é relativamente primo de r . Se o primeiro elemento for qualquer outro número, r e o valor candidato para e têm um divisor maior que 1 em comum e eles não são relativamente primos.

Procedimento EUCLID (a,b)

Entradas: a e b : Dois inteiros.

Saída: Uma tripla (g,i,j) tal que g é o máximo divisor comum de i e j e $g = ai + bj$.

1. Se b for igual a 0, retorne a tripla $(a,1,0)$.
2. Caso contrário (b não é 0), faça o seguinte:
 - a. Chame recursivamente EUCLID($b, a \bmod b$) e designe o resultado retornado à tripla (g,i',j') . Isto é, iguale g ao primeiro elemento da tripla retornada, iguale i' ao segundo elemento da tripla retornada e j' ao terceiro elemento da tripla retornada.
 - b. Iguala i a $i' - a/b \cdot j'$.
 - c. Iguala j a $i' - a/b \cdot j'$.
 - d. Retorne a tripla (g,i,j) .

Não vou comentar por que esse procedimento funciona⁸ nem vou analisar seu tempo de execução, mas só vou lhe dizer que, se eu chamar EUCLID(r,e), o número de chamada recursivas será $O(\lg r)$. Portanto, posso verificar rapidamente se 1 é o máximo divisor comum de r e um valor candidato para e (lembre-se de que e é pequeno). Se não, posso tentar um valor candidato diferente para e , e assim por diante, até encontrar um que seja relativamente primo de r . Quantos candidatos eu espero ter de experimentar? Não muitos. Se eu restringir minhas escolhas para e a números primos ímpares menores que r (o que é fácil de verificar pelo teste Miller-Rabin ou pelo teste baseado no pequeno teorema de Fermat), qualquer escolha terá grande probabilidade de ser relativamente primo de r . Isso porque, pelo teorema do número primo, aproximadamente $r/\ln r$ números primos são menores que r , mas um outro teorema mostra que r não pode ter mais do que $\ln r$ fatores primos. Portanto, é improvável que eu encontre um fator primo de r .

⁸ A chamada EUCLID(0,0) retorna a tripla $(0,1,0)$, portanto considera que 0 é o máximo divisor comum de 0 e 0. Isso poderia lhe parecer peculiar (Eu ia dizer “odd” (estranho), mas é o contexto errado para esse outro significado de “odd” [ímpar]). Porém, como r é positivo, o parâmetro a na primeira chamada a EUCLID será positivo e, em qualquer chamada recursiva, a deve ser positivo. Portanto, não nos importa o que EUCLID(0,0) retorna.

Como computar inversos multiplicativos em aritmética modular

Tão logo eu tenha r e e , preciso computar d como o inverso de e , módulo r , de modo que $ed \bmod r$ seja igual a 1. Já sabemos que a chamada $\text{EUCLID}(r, e)$ retornou uma tripla da forma $(1, i, j)$ e que 1 é o máximo divisor comum de r e e (porque eles são relativamente primos) e que $1 = ri + ej$. Agora posso igualar d a $j \bmod r$.⁹ Isso porque estamos trabalhando com módulo r e, portanto, podemos tomar ambos os lados com módulo r :

$$\begin{aligned} 1 \bmod r &= (ri + ej) \bmod r \\ &= ri \bmod r + ej \bmod r \\ &= 0 + ej \bmod r \\ &= ej \bmod r \\ &= (e \bmod r) \cdot (j \bmod r) \bmod r \\ &= e(j \bmod r) \bmod r \end{aligned}$$

(A última linha decorre porque $e < r$, o que implica que $e \bmod r = e$.) Portanto, temos que $1 = e(j \bmod r) \bmod r$, o que significa que eu posso igualar d ao valor j na tripla retornada da chamada a $\text{EUCLID}(r, e)$, tomando módulo r . Uso $j \bmod r$ em vez de apenas j caso j não esteja na faixa de 0 a $r - 1$.

Como elevar um número a uma potência inteira rapidamente

Embora e seja pequeno, d poderia ser grande, e eu preciso computar $x^d \bmod n$ para computar a função F_S . Embora eu possa trabalhar em módulo n , o que significa que todos os valores com os quais eu trabalho estarão na faixa 0 a $n - 1$, não quero ter de multiplicar números d vezes. Felizmente, não tenho de fazer isso. Posso multiplicar números apenas $\Theta(\lg d)$ vezes usando uma técnica conhecida como *elevação ao quadrado repetida*. Posso usar essa mesma técnica para o teste de primalidade baseado no pequeno teorema de Fermat.

Eis a ideia. Sabemos que d é não negativo. Suponha em primeiro lugar que d seja par. Então x^d é igual a $(x^{d/2})^2$. Agora suponha que d seja ímpar. Então x^d é igual a $(x^{(d-1)/2})^2 \cdot x$. Essas observações nos dão um modo recursivo interessante para computar x^d , no qual o caso-base ocorre quando d é 0: x^d é igual a 1.

O seguinte procedimento incorpora essa abordagem executando tudo com aritmética módulo n :

Procedimento MODULAR-EXPONENTIATION (x, d, n)

Entradas: x, d, n : três inteiros, com x e d não negativos e n positivo.

Saída: Retorna o valor de $x^d \bmod n$.

1. Se d é igual a 0, retorne 1.
2. Caso contrário (d é positivo), se d é par, chame recursivamente MODULAR-EXPONENTIATION ($x, d/2, n$), iguale z ao resultado dessa chamada recursiva e retorne $(z^2 \cdot x) \bmod n$.
3. Caso contrário (d é positivo e ímpar), chame recursivamente MODULAR-EXPONENTIATION ($x, (d-1)/2, n$), iguale z ao resultado dessa chamada recursiva e retorne $(z^2 \cdot x) \bmod n$.

⁹Lembre-se de que j poderia ser negativo. Um modo de pensar em $jj \bmod r$ quando j é negativo e r é positivo é iniciar com j e continuar adicionando r até que o número que você obtiver seja não negativo. Esse número é igual a $j \bmod r$. Por exemplo, para determinar $-27 \bmod 10$, você obtém os números $-27, -17, -7$ e 3 . Tão logo chegue a 3 , pode parar e dizer que $-27 \bmod 10$ é igual a 3 .

O parâmetro d se reduz, no máximo, à metade em cada chamada recursiva. Depois de, no máximo, $\lfloor \lg d \rfloor + 1$ chamadas, d vai a zero e a recursão termina. Portanto, esse procedimento multiplica números $\Theta(\lg d)$ / vezes.

Mostrando que as funções F_P e F_S são inversas uma da outra

Advertência: Grande quantidade de teoria dos números e aritmética modular nos espera. Se você se contentar em aceitar sem prova que as funções F_P e F_S são inversas uma da outra, salte os próximos cinco parágrafos e volte à leitura em “Criptossistemas híbridos”.

Para o RSA ser um criptossistema de chave pública, as funções F_P e F_S devem ser inversas uma da outra. Se tomarmos um bloco t de texto comum, tratarmos esse bloco como um inteiro menor que n e o alimentarmos a F_P , obteremos $t^e \bmod n$, e se alimentarmos esse resultado em F_S , obteremos $(t^d)^e \bmod n$, que é igual a $t^{ed} \bmod n$. Se invertermos a ordem, primeiro F_S e depois F_P , obteremos $t^{ed} \bmod n$, que novamente é igual a $t^{ed} \bmod n$. Precisamos mostrar que, para qualquer bloco de texto comum t interpretado como um inteiro menor que n , temos $t^{ed} \bmod n$ igual a t .

Damos a seguir um esboço da nossa abordagem. Lembre-se de que $n = pq$. Mostraremos que $t^{ed} \bmod p = t \bmod p$ e que $t^{ed} \bmod q = t \bmod q$. Então, usando outro fato da teoria dos números, concluiremos que $t^{ed} \bmod pq = t \bmod pq$ — em outras palavras, que $t^{ed} \bmod n = t \bmod n$, o que é apenas t porque t é menor que n .

Precisamos usar novamente o pequeno teorema de Fermat, e ele ajuda a explicar por que determinamos que r é o produto $(p - 1)(q - 1)$ (você já não estava imaginando de onde veio isso?). Visto que p é primo, se $t \bmod p$ é não zero, então. $(t \bmod p)^{p-1} \bmod p = 1$

Lembre-se de que definimos e e d de modo que sejam inversos multiplicativos, módulo r : $ed \bmod r = 1$. Em outras palavras, $ed = 1 + h(p - 1)(q - 1)$ para algum inteiro h . Se $t \bmod p$ não é 0, então temos o seguinte:

$$\begin{aligned} t^{ed} \bmod p &= (t \bmod p)^{ed} \bmod p \\ &= (t \bmod p)^{1+h(p-1)(q-1)} \bmod p \\ &= ((t \bmod p) \cdot ((t \bmod p)^{p-1} \bmod p)^{h(q-1)}) \bmod p \\ &= (t \bmod p) \cdot (1^{h(q-1)} \bmod p) \\ &= t \bmod p \end{aligned}$$

É claro que, se $t \bmod p$ é 0, então $t^{ed} \bmod p$ é igual a 0.

Um argumento semelhante mostra que, se $t \bmod q$ não é 0, $t^{ed} \bmod q$ é igual a $t \bmod q$ e, se $t \bmod p$ é 0, $t^{ed} \bmod p$ é igual a 0.

Precisamos de mais um fato da teoria dos números para arrematar: como p e q são relativamente primos (cada um deles é primo), se $x \bmod p = y \bmod p$ e $x \bmod q = y \bmod q$, então $x \bmod pq = y \bmod pq$ (Esse fato vem do “teorema do restaurante chinês”). Anexando t^{ed} a x e t a y , e lembrando que $n = pq$ e que t é menor que n , temos $t^{ed} \bmod n = t \bmod n = t$, que é exatamente o que precisávamos mostrar. Até que enfim!

CRIPTOSSISTEMAS HÍBRIDOS

Embora possamos executar aritmética com números grandes, na prática pagamos um preço em velocidade. Criptografar e decifrar uma mensagem longa, que contenha centenas ou

milhares de blocos de texto comum, pode causar um atraso significativo. O RSA é frequentemente usado em um sistema híbrido, parte de chave pública e parte de chave simétrica.

Mostramos agora como você pode me enviar uma mensagem criptografada em um sistema híbrido. Em primeiro lugar, combinamos qual sistema de chave pública e qual sistema de chave simétrica estamos usando; digamos, RSA e AES. Você seleciona uma chave k para AES e a criptografa com a minha chave pública RSA, produzindo $F_p(k)$. Usando a chave k , você criptografa a sequência de blocos de texto comum com AES para produzir uma sequência de blocos de texto cifrado. Você me envia $F_p(k)$ e a sequência de blocos de texto cifrado. Eu decifro $F_p(k)$ computando $F_s(F_p(k))$, o que me dá a chave AES k , e uso k para decifrar os blocos de texto cifrado AES, recuperando assim o bloco de texto comum. Se estivermos usando encadeamento de cifra de bloco e precisarmos de um vetor de inicialização, você poderá criptografá-lo com RSA ou com AES.

COMPUTANDO NÚMEROS ALEATÓRIOS

Como já vimos, alguns criptossistemas exigem a geração de números aleatórios — inteiros positivos aleatórios, para sermos exatos. Como representamos um inteiro por uma sequência de bits, o que realmente queremos é um modo de gerar bits aleatórios, que então podemos interpretar como um inteiro.

Bits aleatórios só podem vir de processos aleatórios. Como um programa executado em um computador pode ser um processo aleatório? Em muitos casos não pode, porque um programa de computador que é construído a partir de instruções bem definidas, determinísticas, sempre produzirá o mesmo resultado, desde que os dados iniciais sejam os mesmos. Para suportar software criptográfico, alguns processadores modernos proveem uma instrução que gera bits aleatórios com base em um processo aleatório, como ruído térmico dentro de circuitos. Os projetistas desses processadores enfrentam um desafio triplo: gerar os bits a uma taxa suficientemente rápida para aplicações que demandam números aleatórios, assegurar que os bits gerados estejam de acordo com testes estatísticos de aleatoriedade e consumir uma quantidade razoável de potência de computação enquanto geram e testam os bits aleatórios.

Programas criptográficos usualmente obtêm bits de um *gerador de números pseudoaleatórios* (pseudorandom number generator — PRNG). Um PRNG é um programa determinístico que produz uma sequência de valores, baseados em um valor inicial, ou *semente*, e em uma regra determinística incorporada ao programa que diz como gerar o próximo valor na sequência a partir do valor corrente. Se você iniciar um PRNG com a mesma semente a cada vez, obterá a mesma sequência de valores a cada vez. Esse comportamento repetível é bom para depuração, mas ruim para criptografia. Padrões recentes para geradores de números aleatórios para criptossistemas exigem a implementação de PRNGs específicos.

Se você estiver usando um PRNG para gerar bits que parecem aleatórios, é bom iniciá-lo com uma semente diferente a cada vez, e essa semente deve ser aleatória. Em particular, a semente deve ser baseada em bits sem nenhum viés (não favorecem nem 0 nem 1), independente (não importando o que você sabe sobre o bit gerado anteriormente, qualquer outra pessoa terá apenas 50% de chance de adivinhar corretamente o próximo bit) e imprevisível para um adversário que estiver tentando decifrar o seu

criptossistema. Se o seu processador tiver uma instrução que gera bits aleatórios, esse é um bom modo de criar a semente do PRNG.

O QUE MAIS LER?

Criptografia é apenas uma componente da segurança em sistemas de computadores. O livro de Smith e Marchesini [SM08] abrange segurança de computadores de um ponto de vista amplo, incluindo criptografia e modos de atacar criptossistemas.

Se quiser se aprofundar na criptografia, recomendo os livros de Katz e Lindell [KL08] e de Menezes, van Oorschot e Vanstone [MvOV96]. O Capítulo 31 de CLRS [CLRS09] dá um histórico rápido sobre a teoria dos números que leva à criptografia, bem como descrições de RSA e do teste de primalidade Miller-Rabin. Diffie e Hellman [DH76] propuseram criptografia de chave pública em 1976, e o artigo original que descreve o RSA de Rivest, Shamir e Adelman [RSA78] apareceu dois anos depois.

Se quiser mais detalhes sobre PRNGs aprovados, consulte o Annex C de Federal Information Processing Standards Publication 140-2 [FIP11]. Você pode ler sobre uma implementação de hardware de um gerador de números aleatórios baseado em ruído térmico no artigo de Taylor e Cox [TC11].

Compressão de dados

No capítulo anterior, examinamos como transformar informação para protegê-la contra um adversário. Todavia, proteger informação não é a única razão para transformá-la. Às vezes você quer realçá-la; por exemplo, você pode querer modificar uma imagem usando uma ferramenta de software como o Adobe Photoshop para eliminar o efeito dos olhos vermelhos ou mudar os tons da pele. Às vezes você quer adicionar redundância de modo que, se alguns bits forem incorretos, os erros podem ser detectados e corrigidos.

Neste capítulo, investigaremos um outro modo de transformar informação: comprimindo-a. Antes de passarmos para alguns dos métodos usados para comprimir e descomprimir informação, devemos responder a três perguntas:

1. Por que quereríamos comprimir informação?

Normalmente comprimimos informações por uma das razões: poupar tempo e/ou poupar espaço.

Tempo: quando transmitimos informações por uma rede, quanto menor o número de bits transmitidos, mais rápida será a transmissão. Portanto, o remetente frequentemente comprime os dados antes de enviá-los, envia os dados comprimidos e então o destinatário descomprime os dados que recebe.

Espaço: quando a quantidade de armazenagem disponível pode limitar a quantidade de informações possíveis de armazenar, você pode armazenar mais informações se elas estiverem comprimidas. Por exemplo, os formatos MP3 e JPEG comprimem som e imagens de um modo tal que a maioria das pessoas percebe pouca diferença (se é que percebem alguma) entre os materiais original e comprimido.

2. Qual é a qualidade das informações comprimidas?

Métodos de compressão podem ser sem perdas ou com perdas. Com **compressão sem perdas**, quando as informações comprimidas são descomprimidas elas são idênticas às informações originais. Com **compressão com perdas**, as informações descomprimidas são diferentes das originais, porém idealmente de uma maneira insignificante. Compressões MP3 e JPEG são com perdas, mas o método de compressão usado pelo programa *zip* é sem perdas.

De modo geral, quando comprimimos texto, queremos compressão sem perdas. Até mesmo uma diferença de um bit pode ser significativa.

As seguintes sentenças têm uma diferença de apenas um bit nos códigos ASCII de suas letras:¹

¹ Os códigos ASCII para p e t são, respectivamente, 01110000 e 01110100.

Don't forget the pop.
Don't forget the pot.

Essas sentenças podem ser interpretadas como um pedido para não esquecer, respectivamente, os refrigerantes (ao menos no centro-oeste dos Estados Unidos) ou a maconha — um bit faz uma grande diferença!

3. Por que é possível comprimir informações?

Essa pergunta é fácil de responder para compressão com perdas: você apenas tolera a diminuição da precisão. E a compressão sem perdas? Informações digitais contêm bits redundantes ou inúteis. Em ASCII, por exemplo, cada caractere ocupa um byte de oito bits e todos os caracteres comumente usados (sem incluir letras com acentos) têm um 0 no bit mais significativo (o da extrema esquerda). Isto é, os códigos de caracteres em ASCII vão de 0 a 255, mas todos os caracteres comumente usados caem na faixa de 0 a 127. Portanto, em muitos casos, um oitavo dos bits em textos ASCII são inúteis e seria fácil comprimir a maioria dos textos ASCII em 12,5%.

Para um exemplo mais drástico de como explorar redundância em compressão sem perdas, considere a transmissão de uma imagem em preto e branco, como fazem as máquinas de fax. As máquinas de fax transmitem uma imagem como uma série de *pels*:² pontos pretos ou brancos que juntos formam a imagem. Muitas máquinas de fax transmitem os pels de cima para baixo, linha por linha. Quando a imagem compreende a maioria do texto, grande parte da imagem é branca, e portanto cada linha provavelmente contém muitos pels brancos consecutivos. Se uma linha contém parte de uma linha horizontal preta, ela pode ter muitos pels pretos consecutivos. Em vez de indicar individualmente cada pel em uma carreira de cor igual, as máquinas de fax comprimem a informação para indicar o comprimento de cada carreira e a cor dos pels na carreira. Por exemplo, em um fax-padrão, uma carreira de 140 pels brancos é comprimida nos 11 bits 10010001000.

Compressão de dados é uma área bem estudada e, portanto, aqui eu só posso abordar pequena parte dela. Focalizarei a compressão sem perdas, mas você pode encontrar um par de boas referências que abordam compressão com perdas na seção “O que mais ler?”.

Neste capítulo, diferentemente dos capítulos anteriores, não focalizaremos tempos de execução. Eu os mencionarei quando adequado, porém estamos muito mais interessados no tamanho das informações comprimidas do que no tempo que leva para comprimi-las e descomprimi-las.

CÓDIGOS DE HUFFMAN

Vamos retornar às cadeias que representam DNA, por enquanto. Lembre-se de que dissemos no Capítulo 7 que os biólogos representam o DNA como cadeias usando quatro caracteres A, C, G e T. Suponha que tivéssemos uma cadeia de DNA

² Pels são como pixels em uma tela. “Pel” e “pixel” são abreviaturas de “picture element” (elemento de figura ou de quadro).

representada por n caracteres, na qual 45% dos caracteres são A, 5% são C, 5% são G e 45% são T, mas os caracteres aparecem na cadeia sem nenhuma ordem particular. Se usássemos o conjunto de caracteres ASCII para representar a cadeia, com cada caractere ocupando oito bits, precisaríamos de $8n$ bits para representar a cadeia inteira. É claro que podemos nos sair melhor do que isso. Visto que representamos cadeias de DNA por apenas quatro caracteres, na realidade precisamos de apenas dois bits para representar cada caractere (00, 01, 10, 11) e, portanto, reduzimos o espaço a $2n$ bits.

Porém, podemos nos sair ainda melhor se tirarmos proveito das frequências relativas dos caracteres. Vamos codificar os caracteres com as seguintes sequências de bits: A = 0, C = 100, G = 101, T = 11. Os caracteres mais frequentes ganham as sequências de bits mais curtas. Codificariam a cadeia de 20 caracteres TAATTAGAAATTC-TATTATA pela sequência de 33 bits 11001111010100011110011011110110 (veremos logo adiante por que eu escolhi essa codificação particular e quais propriedades ela tem). Dadas as frequências dos quatro caracteres, para codificar a cadeia de n caracteres precisamos apenas de $0,45 \cdot n \cdot 1 + 0,05 \cdot n \cdot 3 + 0,45 \cdot n \cdot 2 = 1,65$ bits (observe que, para a cadeia de amostra anterior, $33 = 1,65 \cdot 20$). Lançando mão das frequências relativas dos caracteres, podemos nos sair ainda melhor que $2n$ bits!

Na codificação que usamos, não somente os caracteres mais frequentes obtêm as sequências de bits mais curtas, porém há ainda mais uma coisa interessante sobre as codificações: nenhum código é um prefixo de qualquer outro código. O código para A é 0, e nenhum outro código começa com 0, o código para T é 11, e nenhum outro código começa com 11, e assim por diante. Denominamos tal código *código livre de prefixo*.³

A principal vantagem de códigos livres de prefixo surge quando descomprimimos. Como nenhum código é um prefixo de qualquer outro código, podemos concatenar sem ambiguidade os bits comprimidos com seus caracteres originais à medida que descomprimimos em ordem. Na sequência comprimida 11001111010100011110011011110110, por exemplo, nenhum caractere tem o código de um bit 1 e somente o código para T começa com 11; portanto, sabemos que o primeiro caractere do texto descomprimido deve ser T. Após extirparamos o 11 ficamos com 01111010100011110011011110110. Somente o código para A começa com 0; portanto, o primeiro caractere do que resta deve ser A. Depois de extirparamos o 0 e os bits 011110 correspondentes aos caracteres descomprimidos ATTA, os bits remanescentes serão 10100011110011011110110. Como somente o código para G começa com 101, o próximo caractere descomprimido deve ser G. E assim por diante.

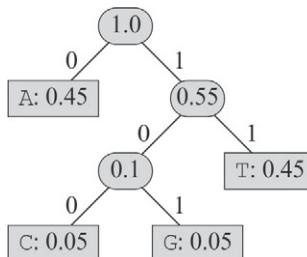
Se medirmos a eficiência de métodos de compressão de acordo com o comprimento médio das informações comprimidas, entre os códigos livres de prefixo, os de Huffman⁴ serão os melhores. Uma desvantagem da codificação Huffman tradicional é que

³ Em CLRS, nós os denominamos “códigos de prefixo”. Agora eu prefiro o nome mais adequado “livre de prefixo”.

⁴ O nome deve-se a seu inventor, David Huffman.

ela requer que as frequências de todos os caracteres sejam conhecidas de antemão; portanto, muitas vezes a compressão exige duas passagens sobre o texto descomprimido: uma para determinar frequências de caracteres e outra para mapear cada caractere para seu código. Veremos mais adiante como evitar a primeira passagem, à custa de computação extra.

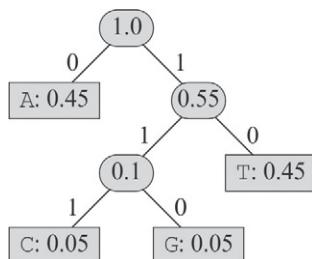
Tão logo saibamos quais são as frequências dos caracteres, o método de Huffman constrói uma árvore binária (se você esqueceu o que são árvores binárias, consulte a página 86). Essa árvore nos diz como formar os códigos, e é também conveniente tê-la quando descomprimimos. Eis como é a árvore para o nosso exemplo de codificação de DNA:



As folhas da árvore, desenhadas como retângulos, representam os caracteres, e a frequência de cada um aparece perto do caractere. As não folhas, ou **nós internos**, são desenhadas com cantos arredondados, sendo que cada nó interno contém a soma das frequências nas folhas abaixo dele. Logo veremos por que vale a pena armazenar frequências nos nós internos.

Perto de cada aresta na árvore aparece um 0 ou um 1. Para determinar o código para um caractere, siga o caminho que vai da raiz até a folha do caractere e concatene os bits ao longo do caminho. Por exemplo, para determinar o código para G, comece na raiz e, em primeiro lugar, siga a aresta, denominada 1, até seu filho da direita; então siga a aresta denominada 0 até o filho da esquerda (o nó interno com frequência 0,1) e, finalmente, siga a aresta denominada 1 até o filho da direita (a folha que contém G). A concatenação desses bits dá o código 101 para G.

Embora eu sempre tenha usado 0 para identificar as arestas até filhos da esquerda e 1 para identificar as arestas até filhos da direita, os rótulos em si não têm muita importância. Eu poderia muito bem ter identificado as arestas desse modo:



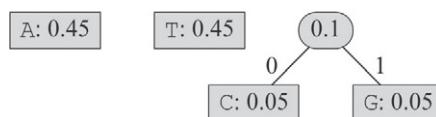
Com essa árvore, os códigos seriam A = 0, C = 111, G = 110, T = 10. Ainda seriam livres de prefixo, e o número de bits em cada código seria o mesmo de antes. Isso porque o número de bits no código para um caractere é igual à **profundidade** da folha do caractere: o número de arestas no caminho desde a raiz até uma folha. Todavia, a vida é mais simples se sempre usarmos 0 para identificar as arestas até os filhos à esquerda e 1 para identificar as arestas até filhos à direita.

Tão logo saibamos quais são as frequências dos caracteres, construímos a árvore binária de baixo para cima. Começamos com cada um dos n nós folha, fazendo a correspondência com os caracteres descomprimidos, como sua própria árvore individual, de modo que inicialmente cada folha é também uma raiz. Então encontramos repetidamente os dois nós de raiz com as frequências mais baixas, criamos uma nova raiz com esses nós como seus filhos e damos a essa nova raiz a soma das frequências de seus filhos. O processo continua até que todas as folhas estejam sob uma raiz. À medida que progredimos, identificamos cada aresta até o filho da esquerda com 0 e cada aresta até o filho da direita com 1, embora mais uma vez selecionamos as duas raízes que têm as frequências mais baixas, não importando qual dos filhos seja denominado o filho da esquerda e qual dos filhos seja denominado o filho da direita da nova raiz.

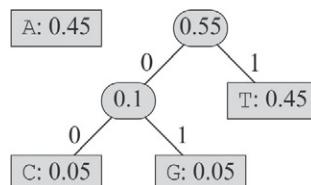
Eis como o processo se desenrola para o nosso exemplo do DNA. Começamos com quatro nós, cada folha representando um caractere:

A: 0.45	C: 0.05	G: 0.05	T: 0.45
---------	---------	---------	---------

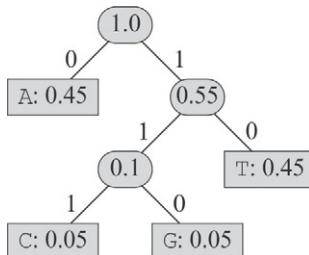
Os nós para C e G têm as frequências mais baixas, portanto criamos um novo nó, transformamos os nós C e G em filhos desse nó e lhe damos as frequências combinadas daqueles dois filhos:



Das três raízes remanescentes, a que acabamos de criar tem a frequência mais baixa, 0,1, e as duas outras têm frequências de 0,45. Podemos selecionar qualquer das duas como a segunda raiz; selecionamos a raiz para T e transformamos essa raiz e a raiz com frequência 0,1 em filhos de um novo nó cuja frequência é a soma das frequências dessas raízes, 0,55:



Restam somente duas raízes. Criamos um novo nó, transformamos essas raízes em filhos desse nó, e sua frequência (que não precisamos, visto que terminaremos) é a soma daquelas raízes, 1,0:



Agora que todas as folhas estão sob essa nova raiz, terminamos de construir a árvore binária.

Para sermos um pouco mais exatos, vamos definir um procedimento para construir a árvore binária. O procedimento, **BUILD-HUFFMAN-TREE**, adota como entrada dois arranjos de n elementos, $char freq$, onde $char[i]$ contém o i -ésimo caractere descomprimido e $freq[i]$ dá a frequência desse caractere. Ele também adota o valor de n . Para encontrar as duas raízes que têm as frequências mais baixas, o procedimento chama os procedimentos **INSERT** e **EXTRACT-MIN** para uma fila de prioridade (veja as páginas 84-85).

Procedimento BUILD-HUFFMAN-TREE (*char,freq,n*)

Entradas:

- *char*: um arranjo de n caracteres não comprimidos.
- *freq*: um arranjo de n frequências de caracteres.
- *n*: os tamanhos dos arranjos *char* e *freq*.

Saída: A raiz da árvore binária construída para códigos de Huffman.

1. Seja Q uma fila de prioridade vazia.
2. Para $i = 1$ a n :
 - a. Construa um novo nó z contendo $char[i]$ e cuja frequência é $freq[i]$.
 - b. Chame **INSERT**(Q,z)
3. Para $i = 1$ a $n - 1$:
 - a. Chame **EXTRACT-MIN**(Q) e iguale x ao nó extraído.
 - b. Chame **EXTRACT-MIN**(Q) e iguale y ao nó extraído.
 - c. Construa um novo nó z cuja frequência é a soma da frequência de x e da frequência de y .
 - d. Igualo o filho da esquerda de z a x e o filho da direita de z a y .
 - e. Chame **INSERT**(Q)/.
4. Chame **EXTRACT-MIN**(Q) e retorne o nó extraído.

Assim que o procedimento chega à etapa 4, resta somente um nó na fila de prioridade, e esse nó é a raiz da árvore binária inteira.

Você pode rastrear como esse procedimento funciona com as árvores binárias na página anterior. As raízes na fila de prioridade no início de cada iteração do laço na etapa 3 aparecem no topo de cada figura.

Vamos analisar rapidamente o tempo de execução de **BUILD-HUFFMAN-TREE**. Pressupondo que a fila de prioridade seja implementada a partir de um heap binário, cada operação **INSERT** e **EXTRACT-MIN** leva o tempo $O(\lg n)$. O procedimento chama cada uma dessas operações $2n - 1$ vezes, para um total de tempo $O(\lg n)$. Todo outro trabalho leva um total de tempo $O(n)$; portanto, **BUILD-HUFFMAN-TREE** executa em tempo $O(n \lg n)$.

Mencionei anteriormente que, ao descomprimir, é conveniente ter a árvore binária que BUILD-HUFFMAN-TREE constrói. Começando na raiz da árvore binária, percorra a árvore para baixo de acordo com os bits das informações comprimidas. Extirpe cada bit, indo para a esquerda se ele for um 0 e para a direita se for um 1. Ao chegar a uma folha pare, emita um caractere e retome a busca na raiz. Voltando ao nosso exemplo do DNA, quando descomprimimos a sequência de bits 110011110101000111110011011 11011, extirpamos o primeiro 1 e vamos para a direita a partir da raiz, extirpamos outro 1 e vamos para a direita novamente, chegando à folha para T. Emitimos T e retomamos a busca na raiz. Extirpamos o próximo bit, 0, e vamos para a esquerda a partir da raiz, chegando à folha A, que emitimos, e então voltamos à raiz. A descompressão continua desse modo até que todos os bits das informações comprimidas tenham sido processados.

Se tivermos a árvore binária já construída antes da descompressão, leva tempo constante para processar cada bit. E como o processo de descompressão consegue acesso à árvore binária? Uma possibilidade é incluir uma representação da árvore binária com as informações comprimidas. Uma outra possibilidade é incluir uma tabela de decodificação com as informações processadas. Cada entrada da tabela incluiria o caractere, o número de bits em seu código e o código em si. A partir dessa tabela, é possível construir a árvore binária em tempo linear no número total de bits em todos os códigos.

O procedimento BUILD-HUFFMAN-TREE serve como exemplo de um *algoritmo guloso*, no qual tomamos a decisão que nos parece melhor no momento. Como queremos que os caracteres que aparecem com menos frequência fiquem longe da raiz da árvore binária, a abordagem gulosa sempre seleciona as duas raízes que têm as frequências mais baixas para colocar sob um novo nó, que mais tarde pode se tornar filho de algum outro nó. O algoritmo de Dijkstra (página 80 e ss.) é outro algoritmo guloso porque sempre relaxa arestas que partem do vértice que tem o menor valor de todos os remanescentes em sua fila de prioridade.

Implementei a codificação de Huffman e a executei na versão on-line de *Moby Dick*. O texto original usou 1.193.826 bytes, mas a versão comprimida usou somente 673.579 bytes, ou 56,42% do tamanho do original, sem incluir a própria codificação. Em outras palavras, na média, cada caractere exigiu somente 4,51 bits para codificar. Não é muito surpreendente que o caractere mais frequente foi um espaço (15,96%), seguido por e (9,56%). Os caracteres menos frequentes, que apareceram somente duas vezes cada, foram \$, &, [e].

Códigos de Huffman adaptativos

Os praticantes frequentemente acham que executar duas passagens na entrada, uma para computar frequências de caracteres e outra para codificar caracteres, é uma operação demasiadamente lenta. Em vez disso, os programas de compressão e descompressão funcionam adaptativamente, atualizando frequências de caracteres e a árvore binária à medida que comprimem ou descomprimem em apenas uma passagem.

O programa de compressão começa com uma árvore binária vazia. Cada caractere que ele lê da entrada é novo ou já está na árvore binária. Se o caractere já estiver na árvore binária, o programa de compressão emite o código do caractere de acordo com a árvore binária vigente, aumenta a frequência do caractere e, se necessário, atualiza a árvore binária para refletir a nova frequência. Se o caractere ainda não estiver na árvore binária, o programa de compressão emite o caractere *não* codificado (tal como ele é), adiciona-o à árvore binária e atualiza a árvore binária de acordo.

O programa de descompressão espelha o que o programa de compressão faz. Ele também mantém uma árvore binária à medida que processa as informações comprimidas. Quando vê bits para um caractere na árvore binária, o programa desce pela árvore para determinar qual caractere os bits codificam, emite esse caractere, aumenta a frequência do caractere e atualiza a árvore binária. Quando vê um caractere que ainda não está na árvore, o programa de descompressão emite o caractere, adiciona-o à árvore binária e atualiza a árvore binária.

Todavia, há alguma coisa estranha aqui. Bits são bits, quer representem caracteres ASCII quer representem bits em um código de Huffman. Como o programa de descompressão pode determinar se os bits que está examinando representam um caractere codificado ou não codificado? A sequência de bits 101 representa o caractere codificado como 101 no momento em questão ou é o início de um caractere de oito bits não codificado? A resposta é preceder cada caractere não codificado com um *código de escape*: um código especial que indica que o próximo conjunto de bits representa um caractere não codificado. Se o texto original contiver k caracteres diferentes, então somente k códigos de escape aparecerão nas informações comprimidas, cada um precedendo a primeira ocorrência de um caractere. Códigos de escape não costumam aparecer com muita frequência e, portanto, não queremos lhes designar sequências curtas de bits à custa de um caractere que ocorre com mais frequência. Um bom modo de assegurar que códigos de escape não sejam curtos é incluir um caractere de código de escape na árvore binária, mas restringir sua frequência sempre a 0. À medida que uma árvore binária é atualizada, a sequência de bits do código de escape mudará tanto no programa de compressão quanto no programa de descompressão, mas sua folha será sempre a que estiver mais longe da raiz.

MÁQUINAS DE FAX

Anteriormente, mencionei que máquinas de fax comprimem informações para indicar as cores e os comprimentos das carreiras de pels idênticos nas linhas da imagem que está sendo transmitida. Esse esquema é conhecido como *codificação de comprimento de carreira*. Máquinas de fax combinam codificação de comprimento de carreira com códigos de Huffman. No padrão para máquinas de fax que usam linhas telefônicas comuns, 104 códigos indicam carreiras de comprimentos diferentes de pels brancos e 104 códigos indicam carreiras de comprimentos diferentes de pels pretos. Os códigos para carreiras de pels brancos são livres de prefixo, assim como os códigos para carreiras de pels pretos, embora alguns dos códigos para carreiras de pels brancos sejam prefixos de códigos para carreiras de pels pretos e vice-versa.

Para determinar quais códigos usar para cada carreira, um comitê de padronização tomou um conjunto de oito documentos representativos e contou quantas vezes cada carreira aparecia. Então os participantes do comitê construíram códigos de Huffman para essas carreiras. As carreiras mais frequentes, e por consequência os códigos mais curtos, foram para carreiras de dois, três e quatro pels pretos, com códigos 11, 10 e 011, respectivamente. Outras carreiras comuns foram um pel preto (010), cinco e seis pels pretos (0011 e 0010), dois a sete pels brancos (todos com códigos de quatro bits) e outras carreiras relativamente curtas. Uma carreira razoavelmente frequente consistia em 1.664 pels brancos e representava uma linha inteira de pels brancos. Outros

códigos curtos foram designados para carreiras de pels brancos cujos comprimentos são potências de 2 ou somas de duas potências de 2 (como 192, que é igual a $2^7 + 2^6$). Carreiras podem ser codificadas por concatenação de codificações de carreiras mais curtas. Anteriormente, dei um exemplo de código para uma carreira de 140 pels brancos, 10010001000. Esse código é na verdade a concatenação dos códigos para uma carreira de 128 pels brancos (10010) e uma carreira de 12 pels brancos (001000).

Além de comprimir informações somente dentro de cada linha da imagem, algumas máquinas de fax comprimem em ambas as dimensões da imagem. Carreiras de pels da mesma cor podem ocorrer na direção vertical, bem como na horizontal e, portanto, em vez de tratar cada linha como se ela fosse encontrada isoladamente, uma linha é codificada de acordo com o lugar em que é diferente da linha precedente. Para a maioria das linhas, a diferença em relação à linha anterior é apenas de alguns pels. Esse esquema acarreta necessariamente o risco de propagação de erros: um erro de codificação ou de transmissão faz com que várias linhas consecutivas sejam incorretas. Por essa razão, máquinas de fax que usam esse esquema e transmitem por linhas telefônicas limitam o número de linhas consecutivas que podem usá-lo, de modo que depois de certo número de linhas elas transmitem uma linha inteira de imagem usando o esquema da codificação de Huffman, em vez de transmitir apenas as diferenças em relação à linha anterior.

COMPRESSÃO LZW

Outra abordagem para compressão sem perdas, especialmente para texto, tira proveito das informações recorrentes no texto, embora não necessariamente em localizações consecutivas. Considere, por exemplo, uma famosa citação do discurso de posse do presidente John F. Kennedy:

Ask not what your country can do for you — ask what you can do for your country.

(Não pergunte o que o seu país pode fazer por você; pergunte o que você pode fazer pelo seu país.)

Exceto a palavra *not*, cada palavra na citação aparece duas vezes. Suponha que fizemos uma tabela com essas palavras:

índice	palavra
1	ask
2	not
3	what
4	your
5	country
6	can
7	do
8	for
9	you

Então poderíamos codificar a citação (ignorando letras maiúsculas e pontuação) por
1 2 3 4 5 6 7 8 9 1 3 9 6 7 8 4 5

Como essa citação consiste em poucas palavras, e um byte pode conter inteiros de 0 a 255, podemos armazenar cada índice em um único byte. Assim, podemos armazenar essa citação em apenas 17 bytes, um byte por palavra, mais qualquer espaço que precisarmos para armazenar a tabela. A um caractere por byte, a citação original, sem pontuação mas com espaços entre palavras, requer 77 bytes.

É claro que o espaço para armazenar a tabela importa, senão poderíamos apenas enumerar toda palavra possível e comprimir um arquivo armazenando somente índices de palavras. Para algumas palavras, esse esquema expande, em vez de comprimir. Por quê? Sejamos ambiciosos e consideremos que há menos que 2^{32} palavras, de modo que podemos armazenar cada índice em uma palavra de 32 bits. Representaríamos cada palavra por quatro bytes e, assim, esse esquema perde para palavras que têm três letras ou menos, que exigem somente um byte por letra, não comprimidas.

Todavia, o real obstáculo à enumeração de toda palavra possível é que o texto real inclui “palavras” que não são palavras, ou melhor, não são palavras na língua inglesa. Como exemplo extremo, considere a quadrinha de abertura de “Jabberwocky” de Lewis Carroll:

*Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.*

Considere também programas de computador, que frequentemente usam nomes de variáveis que não são palavras inglesas. Acrescente letras maiúsculas, pontuação e nome de lugares *realmente* longos,⁵ e você poderá ver que, se tentar comprimir texto enumerando toda palavra possível, terá de usar *muitos* índices. Certamente mais de 2^{32} e, como qualquer combinação de caracteres, *poderia* aparecer em texto, na realidade, uma quantidade sem limite.

Porém, nem tudo está perdido, já que ainda podemos tirar proveito de informações recorrentes. Basta que não fiquemos tão apegados a *palavras* recorrentes. Qualquer sequência de caracteres recorrente poderia ajudar. Vários esquemas de compressão recorrem a sequências de caracteres recorrentes. A que vamos examinar é conhecida como **LZW**⁶ e é a base para muitos programas de compressão usados na prática.

O LZW faz uma única passagem em sua entrada para compressão e para descompressão. Em ambas, constrói um dicionário de sequências de caracteres que viu e usa índices para esse dicionário para representar sequências de caracteres. Imagine o dicionário como um arranjo de cadeias de caracteres. Podemos indexar para esse arranjo, portanto podemos falar de sua *i*-ésima entrada. No início da entrada, as sequências tendem a ser curtas, e representar as sequências por índices poderia resultar

⁵ Como Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch, uma aldeia galesa.

⁶ Como você provavelmente adivinhou, o nome honra seus inventores. Terry Welch criou o LZW modificando o esquema de compressão LZ78, que foi proposto por Abraham Lempel e Jacob Ziv.

em expansão, em vez de compressão. Porém, à medida que o LZW progride em sua entrada, as sequências no dicionário tornam-se mais longas, e representá-las por um índice pode poupar bastante espaço. Por exemplo, eu passei o texto de *Moby Dick* por um compressor LZW e ele produziu em sua saída um índice que representa a sequência de 10 caracteres " from " the " 20 vezes (cada " indica um caractere de espaço). Também produziu um índice que representa a sequência de oito caracteres " of " the " 33 vezes.

O compressor e o descompressor semeiam o dicionário com uma sequência de um caractere para cada caractere no conjunto de caracteres. Usando o conjunto de caracteres ASCII completo, o dicionário começa com 256 sequências de um único caractere; a i -ésima entrada no dicionário contém o caractere cujo código ASCII é i .

Antes de passarmos para uma descrição geral do funcionamento do compressor, vamos examinar um par de situações de que ele trata. O compressor constrói cadeias inserindo-as no dicionário e produzindo como saída índices para o dicionário. Vamos supor que o compressor comece construindo uma cadeia com o caractere T, que leu de sua entrada. Como o dicionário tem todas as sequências de um caractere, o compressor encontra T no dicionário. Sempre que o compressor encontrar a cadeia que está construindo no dicionário, ele pega o próximo caractere da entrada e anexa esse caractere à cadeia que está construindo. Portanto, agora vamos supor que o próximo caractere da entrada seja A. O compressor anexa A à cadeia que está construindo, obtendo TA. Vamos supor que TA também esteja no dicionário. Então o compressor lê o próximo caractere da entrada, digamos, G. Ele anexa G à cadeia que está construindo, resultando em TAG, e dessa vez vamos supor que TAG *não* está no dicionário. O compressor faz três coisas: (1) produz o índice de dicionário da cadeia TA; (2) insere a cadeia TAG no dicionário; e (3) comece a construir uma nova cadeia, que contém inicialmente apenas o caractere (G) que fez com que a cadeia TAG não estivesse no dicionário.

Veja como o compressor trabalha em geral. Produz uma sequência de índices para o dicionário. Concatenando as cadeias com esses índices, ele dá o texto original. O compressor constrói cadeias no dicionário um caractere por vez, de modo que sempre que ele insere uma cadeia no dicionário essa cadeia é igual a alguma cadeia que já está no dicionário, porém com mais um caractere. O compressor gerencia uma cadeia s de caracteres consecutivos provenientes da entrada, mantendo a invariante do dicionário que sempre contém s em alguma entrada. Mesmo que s seja um único caractere, ela aparece no dicionário, porque o dicionário é semeado com uma sequência de um único caractere para cada caractere no conjunto de caracteres.

Inicialmente, s é apenas o primeiro caractere da entrada. Ao ler um novo caractere c , o compressor verifica para ver se a cadeia $s\ c$, formada pela anexação de c ao final de s , está no dicionário no momento em questão. Se estiver, ele anexa c ao final de s e denomina o resultado s ; em outras palavras, iguala s a $s\ c$. O compressor está construindo uma cadeia mais longa, que a certa altura inserirá no dicionário. Caso contrário, s está no dicionário mas $s\ c$ não está. Nesse caso, o compressor produz o índice de s no dicionário, insere $s\ c$ na próxima entrada de dicionário disponível e iguala s a apenas o caractere de entrada c . Por inserir $s\ c c$ no dicionário, o compressor adicionou uma cadeia que aumenta s de um caractere e, por igualar s a c , reinicia o processo de construir uma cadeia para ser consultada no dicionário. Como c é uma cadeia de um único caractere no dicionário, o compressor mantém a invariante s que aparece em

algum lugar no dicionário. Uma vez exaurida a entrada, o compressor produz o índice de qualquer cadeia s que restar.

O procedimento LZW-COMPRESSOR é apresentado ao final desta página. Vamos acompanhar um exemplo, comprimindo o texto TATAGATCTTAATATA (a sequência TAG que vimos na página anterior aparecerá). A tabela dada a seguir mostra o que acontece a cada iteração do laço na etapa 3. Os valores mostrados para a cadeia s estão no início da iteração.

Iteração	s	c	Saída	Nova cadeia no dicionário
1	T	A	84(T)	256: TA
2	A	T	65(A)	257: AT
3	T	A		
4	TA	G	256(TA)	258: TAG
5	G	A	71(G)	259: GA
6	A	T		
7	AT	C	257(AT)	260: ATC
8	C	T	67(C)	261: CT
9	T	T	84(T)	262: TT
10	T	A		
11	TA	A	256(TA)	263: TAA
12	A	T		
13	AT	A	257(AT)	264: ATA
14	A	T		
15	AT	A		
etapa 4	ATA		264 (ATA)	

Depois da etapa 1, o dicionário tem cadeias de um caractere para cada um dos 256 caracteres ASCII em entradas 0 até 255. A etapa 2 determina que a cadeia s contém apenas o primeiro caractere de entrada, T.

Procedimento LZW-COMPRESSOR(*text*)

Entrada: *text*: Uma sequência de caracteres no conjunto de caracteres ASCII.

Saída: Uma sequência de índices para um dicionário.

1. Para cada caractere c no conjunto de caracteres ASCII:
 - a. Insira c no dicionário, no índice igual ao código numérico de c em ASCII.
 2. Iguele s ao primeiro caractere advindo de *text*.
 3. Enquanto *text* não for exaurido, faça o seguinte:
 - a. Tome o próximo caractere de *text* e designe-o a c .
 - b. Se $s c$ está no dicionário, iguale s a $s c$.
 - c. Caso contrário ($s c$ ainda não estiver no dicionário), faça o seguinte:
 - i. Produza o índice de s no dicionário.
 - ii. Insira $s c$ na próxima entrada disponível no dicionário.
 - iii. Iguala s à cadeia de um único caractere c .
 4. Produza o índice de s no dicionário.
-

Na primeira iteração do laço principal de etapa 3, c é o próximo caractere de entrada, A. A concatenação $s c$ é a cadeia TA, que ainda não está no dicionário, e portanto a etapa 3C executa. Como a cadeia s contém apenas T, e o código ASCII de T é 84, a etapa 3Ci produz o índice 84. A etapa 3Cii insere a cadeia TA na próxima entrada disponível no dicionário, que é no índice 256, e a etapa 3Ciii reinicia construindo s , igualando-a apenas ao caractere A. Na segunda iteração do laço da etapa 3, c é o próximo caractere de entrada, T. A cadeia $s c = AT$ não está no dicionário e, portanto, a etapa 3C produz o índice 65 (o código ASCII para A), insere a cadeia AT na entrada 257 e faz com que s contenha T.

Vemos o benefício do dicionário nas duas iterações seguintes do laço da etapa 3. Na terceira iteração, c torna-se o próximo caractere de entrada, A. Agora a cadeia $s c = TA$ está presente no dicionário e, assim, o procedimento nada produz. Em vez disso, a etapa 3B anexa o caractere de entrada ao final de s , igualando s a TA. Na quarta iteração, c torna-se G. A cadeia $s c = TAG$ não está no dicionário e, assim, a etapa 3Ci produz o índice de dicionário 256 de s . Um número de saída dá não apenas um, mas dois caracteres: TA.

Nem todo índice de dicionário foi produzido no momento em que o LZW-COMPRESSOR termina, e alguns índices podem ser produzidos mais de uma vez. Se você concatenar todos os caracteres entre parênteses na coluna de saída, obterá o texto TATAGATCTTAATATA.

Esse exemplo é um pouco pequeno demais para mostrar o real benefício da compressão LZW. A entrada ocupa 16 bytes, e a saída consiste em 10 índices de dicionário. Cada índice requer mais de um byte. Mesmo que usemos dois bytes por índice na saída, ele ocupará 20 bytes. Se cada índice ocupar quatro bytes, um tamanho comum para valores de inteiros, a saída levará 40 bytes.

Textos mais longos tendem a produzir melhores resultados. A compressão LZW reduz o tamanho de *Moby Dick* de 1.193.826 bytes para 919.012 bytes. Aqui, o dicionário contém 230.007 entradas, e portanto os índices têm de ter no mínimo quatro bytes.⁷ A saída consiste em 229.753 índices ou 919.012 bytes. Não é tão comprimida como o resultado da codificação de Huffman (673.579 bytes), porém mais adiante veremos algumas ideias para melhorar a compressão.

A compressão LZW ajuda somente se pudermos descomprimir. Felizmente, o dicionário não tem de ser armazenado com as informações comprimidas (se tivesse, a menos que o texto original contivesse enorme quantidade de cadeias recorrentes, a saída da compressão LZW mais o dicionário constituiriam uma expansão, não uma compressão). Como mencionamos antes, a descompressão LZW reconstrói o dicionário diretamente das informações comprimidas.

Veja como a descompressão LZW funciona. Como o compressor, o descompressor semeia o dicionário com as 256 sequências de caracteres correspondentes ao conjunto de caracteres ASCII. Ele lê uma sequência de índices para o dicionário como sua en-

⁷ Estou considerando que representamos inteiros usando as representações-padrão de computador para inteiros, que ocupam um, dois, quatro ou oito bytes. Em teoria, poderíamos representar índices até 230.007 usando apenas três bytes, portanto a saída ocuparia 689.259 bytes.

trada e espelha o que o compressor fez para construir o dicionário. Sempre que produz saída, ela vem de uma cadeia que ele adicionou ao dicionário.

Na maioria das vezes, o próximo índice de dicionário na entrada é para uma entrada que já está no dicionário (logo veremos o que acontece no restante das vezes) e, portanto, o descompressor LZW encontra a cadeia no índice no dicionário e a produz como saída. Mas como ele pode construir o dicionário? Vamos pensar um pouco em como o compressor opera. Quando produz um índice na etapa 3C, ele constatou que, embora a cadeia s esteja no dicionário, a cadeia $s c$ não está. Ele produz o índice de s no dicionário, insere $s c$ no dicionário e começa a construir uma nova cadeia para armazenar, começando com c . O descompressor tem de se comportar do mesmo modo. Para cada índice que toma de sua entrada, ele produz a cadeia s no índice no dicionário. Mas ele também sabe que, no momento em que o compressor produziu o índice para s , o compressor não tinha a cadeia $s c$ no dicionário, onde c é o caractere imediatamente após s . O descompressor sabe que o compressor inseriu a cadeia $s c$ no dicionário, portanto é isso que o descompressor precisa fazer — a certa altura. Ele não pode inserir $s c$ ainda porque não viu o caractere c . Esse caractere virá como o primeiro caractere da próxima cadeia que o descompressor produzirá. Porém, o descompressor ainda não tem a próxima cadeia. Portanto, o descompressor precisa rastrear as duas cadeias consecutivas que produz. Se o descompressor produzir as cadeias X e Y , nessa ordem, ele concatenará o primeiro caractere de Y com X e inserirá a cadeia resultante no dicionário.

Vamos examinar um exemplo, que se refere à tabela na página 146, que mostra como o compressor opera sobre TATAGATCTTAATATA. Na iteração 11, o compressor produz o índice 256 para a cadeia TA e insere a cadeia TAA no dicionário. Isso porque, naquele momento, o compressor já tinha $s = TA$ no dicionário mas não $s c = TAA$. Aquele último A inicia a próxima produção de cadeia pelo compressor, AT (índice 257), na iteração 13. Portanto, quando o descompressor vê os índices 256 e 257, deve produzir TA e também deve lembrar dessa cadeia de modo que, quando produzir AT, poderá concatenar o A de AT com TA e inserir a cadeia resultante, TAA, no dicionário.

Em raras ocasiões, o próximo índice de dicionário na entrada do descompressor é para uma entrada que ainda não está no dicionário. Essa situação surge com uma frequência tão pequena que, quando descomprimimos *Moby Dick*, ela ocorreu para somente 15 dos 229.753 índices. Ela acontece quando a produção do índice pelo compressor é para a cadeia mais recentemente inserida no dicionário. Essa situação ocorre somente quando a cadeia nesse índice começa e termina com o mesmo caractere. Por quê? Lembre-se de que o compressor produz o índice para uma cadeia s somente quando encontra s no dicionário, mas $s c$ não está, e então ele insere $s c$ no dicionário, digamos, no índice i e inicia uma nova cadeia s que começa com c . Se a próxima produção de índice pelo compressor for igual a i , a cadeia no índice i no dicionário deve iniciar com $s c$, mas acabamos de ver que essa cadeia é $s c$. Portanto, se o próximo índice de dicionário na entrada do descompressor for para uma entrada que ainda não está no dicionário, o descompressor pode produzir a cadeia que mais recentemente inseriu no dicionário, concatenada com o primeiro caractere dessa cadeia, e inserir essa nova cadeia no dicionário.

Como essas situações são raras, temos de forçar um pouco para dar um exemplo. A cadeia TATATAT faz com que isso ocorra. O compressor faz o seguinte: produz

índice 84 (T) e insere TA no índice 256; produz índice 65 (A) e insere AT no índice 257; produz índice 256 (TA) e insere TAT no índice 258; finalmente, produz índice 258 (TAT — a cadeia que acabou de inserir). O descompressor, ao ler o índice 258, toma a cadeia que produziu mais recentemente, TA, concatena o primeiro caracteres dessa cadeia, T, produz a cadeia resultante TAT e insere essa cadeia no dicionário.

Embora essa rara situação ocorra somente quando a cadeia começa e termina com o mesmo caractere, ela não ocorre toda vez que a cadeia começa e termina com o mesmo caractere. Por exemplo, ao comprimir *Moby Dick*, a cadeia cujo índice foi produzido tinha o mesmo caractere no início e no final 11.376 vezes (um pouquinho menos que 5% das vezes), sem ser a cadeia mais recentemente inserida no dicionário.

O procedimento LZW-DECOMPRESSOR, torna todas essas ações precisas. A tabela a seguir mostra o que acontece em cada iteração do laço na etapa 4 quando são dados como entrada os índices na coluna de saída na tabela da página 146. As cadeias indexadas no dicionário como *anterior* e *atual* são produzidas em iterações consecutivas, e os valores mostrados para *anterior* e *atual* em cada iteração estão depois da etapa 4B.

Iteração	anterior	atual	Saída(s)	Nova cadeia no dicionário
Etapas 2,3		84	T	
1	84	65	A	256: TA
2	65	256	TA	257: AT
3	256	71	G	258: TAG
4	71	257	AT	259: GA
5	257	67	C	260: ATC
6	67	84	T	261: CT
7	84	256	TA	262: TT
8	256	257	AT	263: TAA
9	257	264	ATA	264: ATA

Exceto para a última iteração, o índice da entrada já está no dicionário, de modo que a etapa 4D executa somente na última iteração. Observe que o dicionário construído por LZW-DECOMPRESSOR corresponde ao construído por LZW-COMPRESSOR.

Procedimento LZW-DECOMPRESSOR (indices)

Entrada: *índices*: uma sequência de índices para um dicionário, criada por LZW-COMPRESSOR.

Saída: O texto que LZW-COMPRESSOR tomou como entrada.

1. Para cada caractere *c* no conjunto de caracteres ASCII:

- a.** Insira *c* no dicionário, no índice igual ao código numérico de *c* em ASCII.
- 2.** Iguale *atual* ao primeiro índice em *índices*.
- 3.** Produza a cadeia no dicionário no índice *atual*.

4. Enquanto *índices* não for exaurido, faça o seguinte:
 - a. Iguale anterior a atual.
 - b. Tome o próximo número de índices e designe-o atual.
 - c. Se o dicionário contiver a entrada indexada por atual, faça o seguinte:
 - i. Torne *s* a cadeia na entrada de dicionário indexada por *atual*.
 - ii. Produza a cadeia *s*.
 - iii. Insira, na próxima entrada disponível no dicionário, a cadeia na entrada do dicionário indexada por *anterior*, concatenada com o primeiro caractere de *s*.
 - d. Caso contrário (o dicionário ainda não contém uma entrada indexada por *atual*), faça o seguinte:
 - i. Iguale *s* à cadeia na entrada do dicionário indexada por *anterior*, concatenada com o primeiro caractere dessa entrada de dicionário.
 - ii. Produza a cadeia *s*.
 - iii. Insira, na próxima entrada de dicionário disponível, a cadeia *s*.
-

Ainda não abordei como consultar informações no dicionário nos procedimentos LZW-COMPRESSOR e LZW-DECOMPRESSOR. O último é fácil: basta rastrear o último índice de dicionário usado e, se o índice em *atual* for menor ou igual ao último índice usado, a cadeia estará no dicionário. O procedimento LZW-COMPRESSOR tem uma tarefa mais difícil: dada uma cadeia, determinar se ela está no dicionário e, se estiver, qual é seu índice. É claro que poderíamos apenas executar uma busca linear no dicionário, mas se o dicionário contiver n itens, cada busca linear levará o tempo $O(n)$. Podemos nos sair melhor do que isso usando qualquer uma de um par de estruturas de dados. Todavia, não vou entrar em mais detalhes aqui. Uma é denominada *trie*, e é como a árvore binária que construímos para a codificação de Huffman, exceto que cada nó pode ter muitos filhos, e não apenas dois, e cada aresta é identificada por um caractere ASCII. A outra estrutura de dados é uma *tabela hash*, que nos dá um modo simples de encontrar cadeias no diretório, que é mais rápido, em média.

Melhorias para LZW

Como mencionei, não fiquei lá muito impressionado com o desempenho do método LZW na compressão do texto de *Moby Dick*. Parte do problema é gerado pelo grande dicionário. Com 230.007 entradas, cada índice requer no mínimo quatro bytes e, portanto, com uma saída de 229.753 índices, a versão comprimida requer quatro vezes isso ou 919.012 bytes. Então, novamente, podemos observar um par de propriedades dos índices que o compressor LZW produz. A primeira é que muitos desses são números pequenos, o que significa que têm muitos zeros na extremidade esquerda em suas representações de 32 bits. A segunda é que alguns dos índices ocorrerão com frequência muito maior que outros.

Quando ambas as propriedades são válidas, a codificação de Huffman provavelmente dará bons resultados. Eu modifiquei o programa da codificação de Huffman para trabalhar com inteiros de quatro bytes em vez de caracteres, e o executei com a saída do compressor LZW para *Moby Dick*. O arquivo resultante ocupa somente 460.971 bytes, ou 38,61% do tamanho original (1.193.826 bytes), que bate a codificação de Huffman sozinha. Todavia, observe que não estou incluindo o tamanho da codificação de Huffman nesse número. Exatamente como a compressão acarretava duas etapas —

comprimir o texto com LZW e depois comprimir os índices resultantes com codificação de Huffman —, a descompressão seria um processo de duas etapas: primeiro descomprimir com codificação de Huffman, depois descomprimir com LZW.

Outras abordagens para a compressão LZW focalizam a redução do número de bits necessários para conter os índices que o compressor produz. Como muitos dos índices são números pequenos, uma abordagem é usar um número menor de bits para os números menores, porém reservar, digamos, os dois primeiros bits para indicar quantos bits o número requer. Damos aqui um esquema como este:

- Se os dois primeiros bits são 00, o índice está na faixa 0 a 63 ($2^6 - 1$), exigindo outros seis bits e, por consequência, um byte no total.
- Se os dois primeiros bits são 01, o índice está na faixa 64 (2^6) a 16.383 ($2^{14} - 1$), exigindo outros 14 bits e, por consequência, dois bytes no total.
- Se os dois primeiros bits são 10, o índice está na faixa 16.384 (2^{14}) a 4.194.303 ($2^{22} - 1$), exigindo outros 22 bits e, por consequência, três bytes no total.
- Finalmente, se os dois primeiros bits são 11, o índice está na faixa 4.194.304 (2^{22}) a 1.073.741.823 ($2^6 - 1$), exigindo outros 30 bits e, por consequência, quatro bytes no total.

Em duas outras abordagens, os índices produzidos pelo compressor são todos do mesmo tamanho porque o compressor limita o tamanho do dicionário. Em uma abordagem, tão logo o dicionário alcance o tamanho máximo, nenhuma outra entrada poderá ser inserida. Em outra abordagem, tão logo o dicionário alcance o tamanho máximo, ele é limpado (exceto as primeiras 256 entradas), e o processo de preencher o dicionário reinicia do ponto no texto onde o dicionário está preenchido. Em todas essas abordagens, o descompressor deve espelhar a ação do compressor.

O QUE MAIS LER?

O livro de Salomon [Sal08] é particularmente claro e conciso, e no entanto abrange ampla gama de técnicas de compressão. O livro de Storer [Sto88], publicado 20 anos antes do livro de Salomon, é um texto clássico na área. A seção 16.3 de CLRS [CLRS09] se aprofunda em códigos de Huffman com algum detalhe, embora não prove que eles são os melhores códigos livres de prefixo que existem.

Difícil? Problemas

Quando compro produtos materiais pela Internet, o vendedor tem de entregá-los em minha casa. Na maioria das vezes, ele usa uma empresa especializada em entrega de pacotes. Não vou dizer qual dessas empresas é a mais frequentemente usada para os produtos que eu compro, mas direi que de vez em quando vejo algum caminhão marrom parado à minha porta.

CAMINHÕES MARRONS

A empresa de entrega de encomendas possui mais de 91.000 desses caminhões marrons nos Estados Unidos, bem como em muitos outros países no mundo inteiro. No mínimo, cinco dias por semana, cada caminhão começa e termina sua jornada em um depósito específico e entrega pacotes em numerosas localizações residenciais e comerciais. A empresa entregadora de encomendas tem grande interesse em minimizar o custo incorrido por cada caminhão à medida que faz muitas paradas por dia. Por exemplo, uma fonte on-line que consultei declarou que, desde que a empresa mapeou as rotas para seus motoristas de modo a reduzir o número de curvas para a esquerda, conseguiu reduzir em 747 km a distância total percorrida por seus veículos em um período de 18 meses, poupando mais de 193.055 litros de combustível, com o benefício adicional de reduzir as emissões de dióxido de carbono em 506 toneladas métricas.

Como a empresa pode minimizar o custo de enviar cada caminhão a cada dia? Suponha que determinado caminhão deva entregar pacotes a n localizações em um dia particular. Contando o depósito, há $n + 1$ localizações que o caminhão deve visitar. Para cada uma dessas $n + 1$ localizações, a empresa pode calcular os custos de enviar o caminhão dali a cada uma das outras n localizações, de modo que a empresa tem um tabela de custos $(n + 1) \times (n + 1)$ de localização a localização, na qual as entradas na diagonal nada significam, visto que a i -ésima linha e a i -ésima coluna correspondem à mesma localização. A empresa quer determinar a rota que começa e termina no depósito e visita todas as outras n localizações exatamente uma vez, tal que o custo total da rota inteira seja o mais baixo possível.

É possível escrever um programa de computador que resolverá esse problema. Afinal de contas, se considerarmos uma rota particular e soubermos qual é a ordem das paradas na rota, bastará consultar na tabela os custos de ir de localização a localização e somá-los. Então, basta enumerar todas as rotas possíveis e determinar qual delas tem o custo total mais baixo. O número de rotas possível é finito e, portanto, o programa terminará em algum ponto e dará a resposta. Esse programa parece não ser muito difícil de escrever, não é?

De fato, o programa não é difícil de escrever.

É difícil de executar.

O problema é que o número de rotas possíveis que visitam n localizações é enorme: $n!$ (n fatorial). Por quê? O caminhão parte do depósito. Dali, qualquer uma das outras n localizações pode ser a primeira parada. Da primeira parada, qualquer uma das $n - 1$ localizações restantes pode ser a segunda parada e, portanto, há $n \cdot (n - 1)$ combinações possíveis para as duas primeiras paradas, em ordem. Uma vez acertadas as duas primeiras paradas, qualquer uma das $n - 2$ localizações poderia ser a terceira parada, o que dá $n \cdot (n - 1) \cdot (n - 2)$ ordens possíveis para as três primeiras paradas. Estendendo esse raciocínio às n localizações de entrega, o número de ordens possíveis é $n \cdot (n - 1) \cdot (n - 2) \dots 3 \cdot 2 \cdot 1$, ou $n!$.

Lembre-se de que $n!$ cresce mais rapidamente que uma função exponencial; é uma função superexponencial. No Capítulo 8, salientei que $10!$ é igual a 3.628.800. Para um computador, esse não é um número tão grande. Mas os caminhões marrons entregam pacotes em muito mais do que apenas 10 localizações por dia. Suponha que um caminhão entregue encomendas em 20 endereços por dia. (Nos Estados Unidos, os caminhões da empresa carregam, em média, 170 pacotes; portanto, dando um desconto para vários pacotes entregues em uma única localização, 20 paradas por dia não parece ser uma estimativa exagerada.) Com 20 paradas, um programa de computador teria de enumerar $20!$ ordens possíveis e $20!$ é igual a 2.432.902.008.176.640.000. Se os computadores da empresa pudessem enumerar e avaliar um trilhão de ordens por segundo, precisariam de mais 28 dias para experimentar todas elas. E isso apenas para um dia de entregas para um de mais de 91.000 caminhões marrons.

Com essa abordagem, se a empresa fosse adquirir e operar a potência de computação necessária para determinar rotas de custo mais baixo para todos os caminhões marrons todos os dias, o custo de computação facilmente engoliria os ganhos obtidos das rotas mais eficientes. Não, essa ideia de enumerar todas as rotas possíveis e rastrear a melhor, embora matematicamente correta, simplesmente não é prática. Há um modo melhor de determinar a rota de menor custo para cada caminhão?

Ninguém sabe (ou, se alguém sabe, nunca contou para ninguém). Ninguém descobriu um modo melhor, no entanto ninguém provou que não pode existir um modo melhor. É ou não é uma grande frustração?

É uma frustração maior ainda do que você poderia imaginar. O problema de determinar as rotas de menor custo para caminhões marrons é mais conhecido como **problema do caixeiro-viajante**, assim denominado porque, em sua formulação original, um caixeiro-viajante¹ tem de visitar n cidades, começando e terminando na mesma cidade, e visitar todas as cidades seguindo a rota mais curta possível. Nenhum algoritmo que executa em tempo $O(n^c)$, para qualquer constante c , jamais foi encontrado para o problema do caixeiro-viajante. Não conhecemos um algoritmo que, dadas as distâncias entre cidades para n cidades, determine a melhor ordem possível para visitar as n cidades no tempo $O(n^{100})$, no tempo $O(n^{1.000})$ ou até mesmo no tempo $O(n^{1.000.000})$.

¹ Desculpem a linguagem discriminadora de gênero. O nome é histórico e, se o problema fosse enunciado hoje, imagino que seria conhecido como “problema da pessoa de vendas viajante”.

E fica ainda pior. Muitos problemas — *milhares* deles — compartilham essa característica: para uma entrada de tamanho n , não conhecemos nenhum algoritmo que execute em tempo $O(n^c)$ para qualquer constante c , porém ninguém provou que tal algoritmo não poderia existir. Esses problemas vêm de uma ampla variedade de domínios — lógica, grafos, aritmética e escalonamento entre eles.

Para aumentar ainda mais o nível de frustração, eis o fato mais assombroso: se houver um algoritmo que execute em tempo $O(n^c)$ para qualquer desses problemas, onde c é uma constante, haverá um algoritmo que executa em tempo $O(n^c)$ para todos esses problemas. Denominamos esses problemas **NP-completos**. Um algoritmo que executa no tempo $O(n^c)$ para uma entrada de tamanho n , onde c é uma constante, é um **algoritmo de tempo polinomial**, assim denominado porque n^c com algum coeficiente seria o termo mais significativo no tempo de execução. Não conhecemos nenhum algoritmo de tempo polinomial para nenhum problema NP-completo, mas ninguém provou que é impossível resolver algum problema NP-completo em tempo polinomial.

E a frustração é ainda maior: muitos problemas NP-completos são quase os mesmos problemas que sabemos como resolver em tempo polinomial. Uma lasquinha os separa. Por exemplo, lembre-se de que no Capítulo 6 dissemos que o algoritmo de Bellman-Ford determina caminhos mínimos que partem de uma única fonte em um grafo dirigido, mesmo que o grafo tenha arestas de peso negativo, no tempo $\Theta(nm)/,$ onde o grafo tem n vértices e m arestas. Se dermos listas de adjacência ao grafo, o tamanho da entrada será $\Theta(n + m)$. Vamos considerar que $m \geq n$; então, o tamanho da entrada é $\Theta(m)$ e $nm \leq m^2$, e portanto o tempo de execução do algoritmo de Bellman-Ford é polinomial no tamanho da entrada (você pode obter o mesmo resultado se $n > m$). Portanto, determinar os *caminhos mínimos* é fácil. Todavia, você poderia se surpreender ao saber que determinar um caminho acíclico *mais longo* (isto é, um caminho mais longo sem ciclos) entre dois vértices é NP-completo. Na verdade, apenas determinar se um grafo contém um caminho sem ciclos com, no mínimo, um número de arestas dado já é NP-completo.

Como outro exemplo de problemas relacionados, dos quais um é fácil e um é NP-completo, considere passeios de Euler e ciclos hamiltonianos. Ambos os problemas têm a ver com a determinação de caminhos em um grafo conectado, não dirigido. Em um **grafo não dirigido**, as arestas não têm nenhuma direção, de modo que (u,v) e (v,u) são a mesma aresta. Dizemos que a aresta (u,v) é **incidente** nos vértices u e v . Um **grafo conectado** tem um caminho entre todo par de vértices. Um **passeio de Euler**² começa e termina no mesmo vértice e visita cada aresta exatamente uma vez, embora possa visitar cada vértice mais de uma vez. Um **ciclo hamiltoniano**³ começa e termina no mesmo vértice e visita cada vértice exatamente uma vez (exceto, é claro, o vértice no qual ele

² O nome se deve ao matemático Leonhard Euler porque ele provou, em 1736, que não era possível fazer um volta inteira (um passeio) pela cidade de Königsberg, Prússia, atravessando cada uma de suas sete pontes exatamente uma vez e terminando no ponto inicial.

³ O nome honra W. R. Hamilton, que em 1856 descreveu um jogo matemático sobre um grafo conhecido como dodecaedro, no qual um jogador espeta cinco alfinetes em quaisquer cinco vértices consecutivos, e o outro jogador deve completar o caminho de modo a formar um ciclo que contenha todos os vértices.

começa e termina). Se perguntarmos se um grafo conectado, não dirigido, tem um passeio de Euler, o algoritmo é notavelmente fácil: determinar o *grau* de cada vértice, isto é, quantas arestas são incidentes nele. O grafo tem um passeio de Euler se e somente se o grau de todo vértice for par. Porém, se perguntarmos se um grafo conectado não dirigido tem um ciclo hamiltoniano, o problema é NP-completo. Observe que a pergunta não é “qual é a ordem dos vértices em um ciclo hamiltoniano nesse grafo?”, mas apenas a mais básica “sim ou não: é possível construir um ciclo hamiltoniano nesse grafo?”.

Surpreendentemente, problemas NP-completos aparecem com muita frequência e é por isso que incluímos material sobre eles neste livro. Se você estiver tentando encontrar um algoritmo de tempo polinomial para um problema que revela ser NP-completo, prepare-se para uma boa dose de desapontamento (mas veja a seção sobre perspectiva, nas páginas 177-180). O conceito de problema NP-completo apareceu no início da década de 1970 e já havia pessoas que tentavam resolver problemas que revelaram ser NP-completos (como o problema do caixeiro-viajante) bem antes disso. Até essa data, não sabemos se existe um algoritmo de tempo polinomial para qualquer problema NP-completo, nem sabemos se tal algoritmo pode existir. Muitos brilhantes cientistas da computação já gastaram anos nessa pergunta sem resolvê-la. Não estou dizendo que *você* não pode descobrir um algoritmo de tempo polinomial para um problema NP-completo, mas estaria enfrentando grandes probabilidades contrárias se tentasse.

AS CLASSES P E NP, E NP-COMPLETUDDE

Nos capítulos anteriores, eu me preocupei com as diferenças nos tempos de execução como $O(n^2)$ versus $O(n \lg n^2)$. Todavia, neste capítulo, ficaremos felizes se algum algoritmo executar em tempo polinomial, de modo que a diferença entre $O(n^2)$ versus $O(n \lg n)$ são insignificantes. Cientistas da computação geralmente consideram problemas resolvíveis por algoritmos de tempo polinomial como “tratáveis”, o que quer dizer “fáceis de lidar”. Se existir um algoritmo de tempo polinomial para um problema, então dizemos que esse problema está na *classe P*.

Nesse ponto, você bem que poderia estar imaginando como poderemos possivelmente considerar um problema que requer tempo $\Theta(n^{100})$ como tratável. Para uma entrada de tamanho $n = 10$, o número 10^{100} não é assustadoramente grande? Sim, é; na verdade, a quantidade 10^{100} é um *googol* (a origem do nome “Google”). Felizmente, não vemos algoritmos que levam tempo $\Theta(n^{100})$. Os problemas em P que encontramos na prática exigem muito menos tempo. Eu raramente vi algoritmos de tempo polinomial que levam tempo pior que, digamos, tempo $O(n^5)$. Além do mais, tão logo alguém descobre o primeiro algoritmo de tempo polinomial para um problema, frequentemente aparecem outros com algoritmos mais eficientes. Portanto, se alguém encontrasse o primeiro algoritmo de tempo polinomial para um problema, mas ele executasse em tempo $\Theta(n^{100})$, seria uma boa chance para outros virem logo atrás com algoritmos mais rápidos.

Agora suponha que lhe deram uma solução proposta para um problema e você quer verificar se a solução está correta. Por exemplo, no problema do ciclo hamiltoniano, uma solução proposta seria uma sequência de vértices. Para verificar se essa solução está correta, você precisaria verificar se todo vértice aparece na sequência exatamente uma vez, exceto que o primeiro e o último vértices devem ser o mesmo, e se a sequência

é $\langle v_1, v_2, v_3, \dots, v_n, v_1 \rangle$ o grafo deve conter arestas $(v_1, v_2), (v_2, v_3), (v_3, v_4), \dots, (v_{n-1}, v_n)$ e voltar a (v_n, v_1) . Você poderia facilmente verificar que essa solução é correta em tempo polinomial. Se for possível verificar uma solução proposta para um problema em tempo polinomial no tamanho da entrada para o problema, dizemos que esse problema está na **classe NP**.⁴ Denominamos a solução proposta **certificado** e, para que o problema esteja em NP, o tempo para verificar o certificado precisa ser polinomial no tamanho da entrada e no tamanho do certificado.

Se você puder resolver um problema em tempo polinomial, certamente poderia verificar um certificado para esse problema em tempo polinomial. Em outras palavras, todo problema em P está automaticamente em NP. O inverso — todo problema em NP está também em P? — é a pergunta que deixa os cientistas de computador perplexos todos esses anos. Frequentemente o denominamos “problema P = NP?”.

Os problemas NP-completos são os “mais difíceis” em NP. Informalmente, um problema é **NP-completo** se satisfizer duas condições: (1) está em NP; (2) se existir um algoritmo de tempo polinomial para o problema, haverá um modo de converter *todas* problemas em NP para esse problema de modo tal a resolvê-los todos em tempo polinomial. Se existir um algoritmo de tempo polinomial para *qualquer* problema NP -completo — isto é, se qualquer problema NP-completo estiver em P —, então P = NP. Como problemas NP-completos são os mais difíceis em NP, se acontecer de qualquer problema em NP não ser resolvível em tempo polinomial, nenhum dos problemas NP-completos o será. Um problema é **NP-difícil** se satisfizer a segunda condição para NP-completude, mas pode estar ou não em NP.

Damos a seguir uma conveniente lista de definições pertinentes:

- **P**: problemas resolvíveis em tempo polinomial, isto é, podemos resolver o problema em tempo polinomial no tamanho da entrada do problema.
- **Certificado**: uma solução proposta para um problema.
- **NP**: problemas verificáveis em tempo polinomial, isto é, dado um certificado, podemos verificar se o certificado é uma solução para o problema em tempo polinomial no tamanho da entrada para o problema e no tamanho do certificado.
- **NP-difícil**: um problema tal que, se houver um algoritmo de tempo polinomial para resolvê-lo, poderemos converter todo problema em NP para esse problema de modo tal a resolver todo problema em NP em tempo polinomial.
- **NP-completo**: um problema que é NP-difícil e também está em NP.

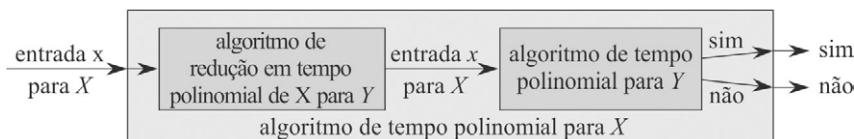
PROBLEMAS DE DECISÃO E REDUÇÕES

Quando falamos sobre as classes P e NP ou sobre o conceito de NP-completude, nos restringimos a **problemas de decisão**: sua saída é um único bit, que indica “sim” ou “não”. Eu expressei o problema do passeio de Euler e o problema do ciclo hamiltoniano desse modo: o grafo tem um passeio de Euler?; ele tem um ciclo hamiltoniano?

⁴ Você, provavelmente, supôs que o nome P vem de “tempo polinomial”. Se estiver imaginando de onde vem o nome NP, é de “tempo polinomial não determinístico”. É um modo equivalente, mas não tão intuitivo, de ver essa classe de problemas.

Todavia, alguns problemas são problemas de otimização, nos quais queremos encontrar as melhores soluções possíveis, em vez de problemas de decisão. Felizmente, muitas vezes podemos fechar parte dessa lacuna expressando um problema de otimização como um problema de decisão. Por exemplo, vamos considerar o problema do caminho mínimo. Nesse caso, usamos o algoritmo de Bellman-Ford para encontrar caminhos mínimos. Como podemos expressar o problema do caminho mínimo como um problema sim/não? Podemos perguntar: “O grafo contém um caminho entre dois vértices específicos cujo peso do caminho é, no máximo, um valor k dado?” Não estamos pedindo os vértices ou arestas no caminho, mas apenas se tal caminho existe. Considerando que pesos no caminho são inteiros, podemos determinar o peso real do caminho mínimo entre os dois vértices fazendo perguntas sim/não. Como? Faça a pergunta para $k = 1$. Se a resposta for não, tente com $k = 2$. Se a resposta for não, tente com $k = 4$. Continue dobrando o valor de k até a resposta ser sim. Se esse último valor de k for k' , a resposta está em algum lugar entre $k'/2$ e k' . Então determine a resposta verdadeira usando busca binária com um intervalo inicial de $k'/2$ a k . Essa abordagem não nos dirá quais vértices e arestas um caminho mínimo contém, porém ao menos nos dirá o peso de um caminho mínimo.

A segunda condição para um problema ser NP-completo requer que, se existir um algoritmo de tempo polinomial para o problema, haverá um modo de converter todo problema em NP para esse problema de modo tal a resolvê-los todos em tempo polinomial. Focalizando problemas de decisão, vamos ver a ideia geral que fundamenta a conversão de um problema de decisão X em outro problema de decisão, tal que, se houver um algoritmo de tempo polinomial para Y , haverá um algoritmo de tempo polinomial para X . Denominamos tal conversão **redução** porque estamos “reduzindo” a solução do problema X à solução do problema Y . Eis a ideia:



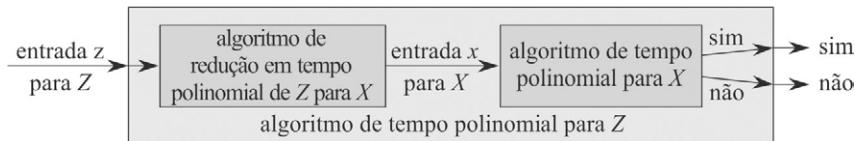
Temos alguma entrada x de tamanho n para o problema X . Transformamos essa entrada em uma entrada y para o problema Y , e fazemos isso em tempo polinomial em n , digamos $O(n^c)$, para alguma constante n . O modo como transformamos a entrada x na entrada y tem de obedecer a uma importante propriedade: se o algoritmo Y decidir “sim” na entrada y , o algoritmo X deve decidir “sim” na entrada x e, se Y decidir “não” em y , X deve decidir “não” em x . Denominamos essa transformação **algoritmo de redução em tempo polinomial**. Vamos ver quanto tempo leva o algoritmo para o problema X inteiro. O algoritmo de redução leva o tempo $O(n^c)$, e sua saída não pode levar mais tempo do que o tempo que ele levou; portanto, o tamanho da saída do algoritmo de redução é $O(n^c)$. Mas essa saída é a entrada y para o algoritmo para o problema Y . Visto que o algoritmo para Y é um algoritmo de tempo polinomial sobre uma entrada de tamanho m , ele executa no tempo $O(m^d)$ para alguma constante d .

Aqui, m é $O(n^c)$ e, assim, o algoritmo para Y leva o tempo $O((n^c)^d)$, ou $O(n^{cd})$. Como c e d são constantes, cd também é, e vemos que o algoritmo para Y é um algoritmo de tempo polinomial. O tempo total para o algoritmo para o problema X é $O(n^c + n^{cd})$, o que o faz, também, um algoritmo de tempo polinomial.

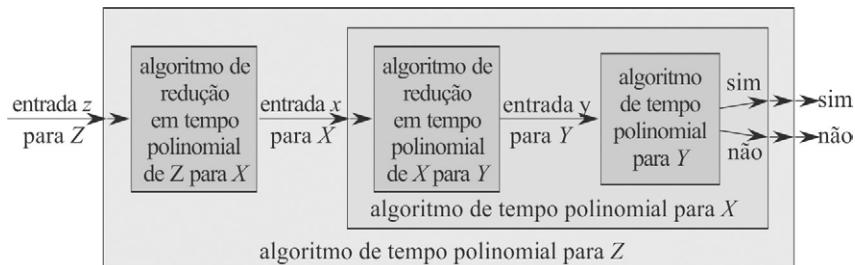
Essa abordagem mostra que, se o problema Y é “fácil” (resolvível em tempo polinomial), o problema X também é. Porém, usaremos reduções de tempo polinomial para mostrar não que os problemas são fáceis, mas que eles são difíceis:

Se o problema X é NP-difícil e podemos reduzi-lo ao problema Y em tempo polinomial, então o problema Y é NP-difícil também.

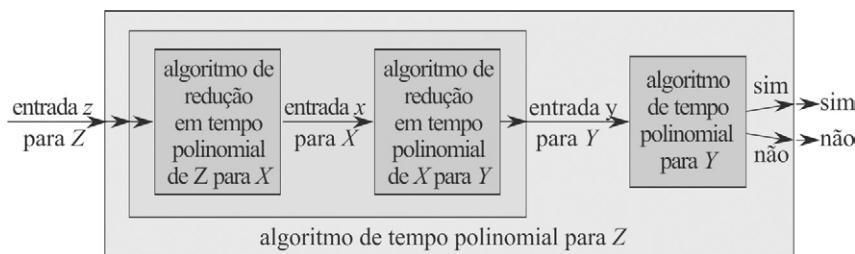
Por que essa afirmação valeria? Vamos supor que o problema X seja NP-difícil e que haja um algoritmo de redução em tempo polinomial para converter entradas para X em entradas para Y . Como X é NP-difícil, há um modo de converter qualquer problema, digamos Z tiver um algoritmo de tempo polinomial, Z também tem. Agora você sabe como ocorre aquela conversão, ou seja, uma redução em tempo polinomial:



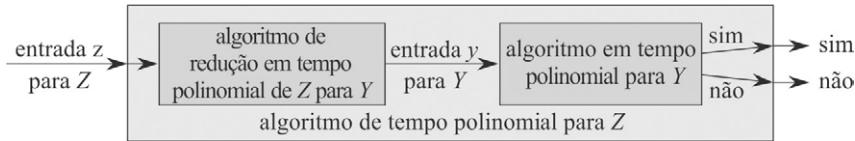
Como podemos converter entradas para X em entradas para Y com uma redução em tempo polinomial, podemos expandir X como fizemos antes:



Em vez de agrupar a redução em tempo polinomial de X para Y e o algoritmo para Y , vamos agrupar as duas reduções em tempo polinomial:



Agora percebemos que, se imediatamente após a redução em tempo polinomial de Z para X , fizermos a redução polinomial de X para Y , teremos uma redução em tempo polinomial de Z para Y :



Só para termos certeza de que as duas reduções em tempo polinomial em sequência constituem juntas uma única redução em tempo polinomial, usaremos uma análise semelhante à que fizemos antes. Suponha que a entrada z para o problema Z tenha tamanho n , que a redução de Z para X leve o tempo $O(n^c)$ e que a redução de X para Y em uma entrada de tamanho m leve o tempo $O(m^d)$, onde c e d são constantes. A saída da redução de Z para Z não pode levar mais tempo que o tempo que levou para ser produzida e, portanto, essa saída, que é também a entrada x para a redução de Z para Y , tem tamanho $O(n^c)$. Agora sabemos que o tamanho m da entrada para a redução de X para Y tem tamanho $m = O(n^c)$, e portanto o tempo que leva a redução de X para X é $O((n^c)^d)$, que é $O(n^{cd})$. Visto que c e d são constantes, essa segunda redução leva tempo polinomial em n .

O tempo gasto no último estágio, o algoritmo de tempo polinomial para Y , é também polinomial em n . Suponha que o algoritmo para Y em uma entrada de tamanho p leve o tempo $O(p^b)$, onde b é uma constante. Como antes, a saída de uma redução não pode exceder o tempo que leva para produzi-la, e portanto $p = O(n^{cd})$, o que significa que o algoritmo para Y leva o tempo $O((n^{cd})^b)$, ou $p = O(n^{bcd})$. Visto que b , c e d são constantes, o algoritmo para Y leva tempo polinomial no tamanho da entrada original n . No total, o algoritmo para Z leva o tempo $O(n^c + n^{cd} + n^{bcd})$, que é polinomial em n .

O que acabamos de ver? Mostramos que, se o problema X é NP-difícil e há um algoritmo de redução em tempo polinomial que transforma uma entrada x para X em uma entrada y para o problema Y , então Y é NP-difícil também. Como o fato de X ser NP-difícil significa que todo problema em NP se reduz a ele em tempo polinomial, escolhemos qualquer problema Z em NP que se reduza a X em tempo polinomial e mostramos que ele também se reduz a Y em tempo polinomial.

Nossa meta final é mostrar quais problemas são NP-completos. Portanto, agora tudo o que temos de fazer para mostrar que um problema Y é NP-completo é

- mostrar que ele está em NP, o que podemos fazer mostrando que há um modo de verificar um certificado para Y em tempo polinomial e
- tomar algum outro problema X que sabemos que é NP-difícil e dar uma redução em tempo polinomial de X para Y .

Há mais um pequeno detalhe que eu ignorei até aqui: o Problema Mãe. Precisamos iniciar com algum problema NP-completo M (o **Problema Mãe**) ao qual *todo* problema em NP se reduz em tempo polinomial. Então podemos reduzir M a algum

outro problema em tempo polinomial para mostrar que o outro problema é NP-difícil, reduzir o outro problema a ainda algum outro problema para mostrar que o último é NP-difícil, e assim por diante. Tenha em mente, também, que não há nenhum limite para a quantidade de outros problemas que podemos reduzir a um único problema, de modo que a árvore da família de problemas NP-completos começa com o Problema Mãe e então se ramifica.

PROBLEMA MÃE

Diferentes livros apresentam listas diferentes de Problemas Mãe. Tudo bem, visto que tão logo você reduza um Problema Mãe a algum outro problema, esse outro problema também poderia servir como Problema Mãe. Um Problema Mãe frequentemente visto é a satisfazibilidade da fórmula booleana. Farei uma descrição resumida desse problema, mas não provarei que todo problema em NP se reduz a ele em tempo polinomial. A prova é longa e — ouso dizer — tediosa.

Primeiro ponto: “booleana” é jargão matemático para lógica simples pela qual as variáveis podem adotar somente os valores 0 e 1 (denominados valores booleanos), e os operadores adotam um ou dois valores booleanos e produzem um valor booleano. Já vimos exclusive-or (XOR) no Capítulo 8. Operadores booleanos típicos são AND, OR, NOT, IMPLIES e IFF:

- $x \text{ AND } y$ é igual a 1 somente se x e y forem 1; caso contrário (um deles ou ambos são 0), $x \text{ AND } y$ é igual a 0.
- $x \text{ OR } y$ é igual a 0 somente se ambos forem 0; caso contrário (um deles ou ambos são 1), $x \text{ OR } y$ é igual a 1.
- NOT x é o oposto de x : é 0 se x é 1 e é 1 se x é 0.
- $x \text{ IMPLIES } y$ é 0 somente se x é 1 e y é 0; caso contrário (x é 0 ou x e y são 1) $x \text{ IMPLIES } y$ é 1.
- $x \text{ IFF } y$ significa “ x se e somente se y ” e é igual a 1 somente se x e y forem iguais (ambos 0 ou ambos 1); se x e y forem diferentes (um deles é 0 e o outro é 1), então $x \text{ IFF } y$ é igual a 0.

Há 16 operadores booleanos possíveis que tomam dois operandos, porém esses são os mais comuns.⁵ Uma **fórmula booleana** consiste em variáveis com valores booleanos, operadores booleanos e parênteses para agrupar.

No **problema de satisfazibilidade da fórmula booleana**, a entrada é uma fórmula booleana, e perguntamos se há algum modo de atribuir os valores 0 e 1 às variáveis na fórmula de modo que o resultado seja 1. Se tal modo existir, dizemos que a fórmula é **satisfazível**. Por exemplo, a fórmula booleana

$$((w \text{ IMPLIES } x) \text{ OR } \text{NOT}(((\text{NOT } w)\text{IFF } y)\text{OR } z)) \text{ AND } (\text{NOT } x)$$

é satisfazível: seja $w = 0$, $x = 0$, $y = 1$, e $z = 1$. Então o resultado é

⁵ Alguns desses 16 operadores booleanos de dois operandos não são terrivelmente interessantes, tal como o operador cujo resultado é 0, independentemente dos valores de seus operandos.

$$\begin{aligned}
 & ((0 \text{ IMPLIES } 0) \text{ OR NOT}(((\text{NOT } 0)\text{IFF } 1)\text{OR } 1)) \text{ AND } (\text{NOT } 0) \\
 & = (1 \text{OR NOT}((1 \text{IFF } 1)\text{OR } 1)) \text{ AND } 1 \\
 & = (1 \text{OR NOT}(1 \text{ OR } 1)) \text{ AND } 1 \\
 & = (1 \text{OR } 0) \text{ AND } 1 \\
 & = 1 \text{ AND } 1 \\
 & = 1.
 \end{aligned}$$

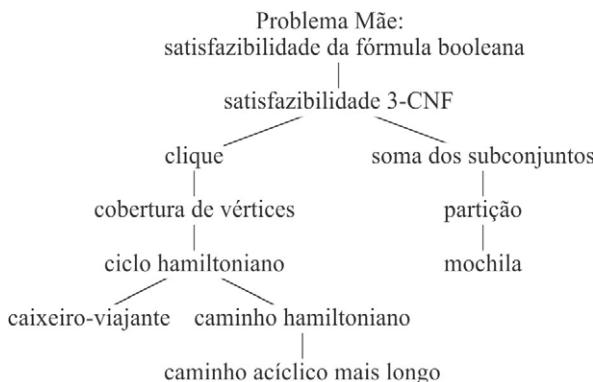
Por outro lado, eis uma fórmula simples que não é satisfazível:

$$x \text{ AND } (\text{NOT } x).$$

Se $x = 0$, o resultado dessa fórmula é $0 \text{ AND } 1$, que é 0; se, em vez disso, $x = 1$, o resultado dessa fórmula é $1 \text{ AND } 0$, que novamente é 0.

UM CATÁLOGO DE AMOSTRAS DE PROBLEMAS NP-COMPLETOS

Adotando a satisfazibilidade da fórmula booleana como nosso Problema Mãe, vamos ver alguns dos problemas que podemos mostrar que são NP-completos usando reduções em tempo polinomial. Damos a seguir a árvore da família das reduções que veremos:



Não mostrarei todas as reduções nessa árvore de família porque algumas delas são bastante longas e complicadas. Mas veremos duas que são interessantes porque mostram como reduzir um problema de um domínio para um domínio diferente, como lógica (satisfazibilidade 3-CNF) para grafos (o problema do clique).

Satisfazibilidade 3-CNF

Como as fórmulas booleanas podem conter quaisquer dos 16 operadores booleanos de dois operandos e como eles podem ser parentizados em qualquer número de modos, é difícil reduzir diretamente do problema de satisfazibilidade da fórmula booleana o

Problema Mãe. Em vez disso, definiremos um problema relacionado que é também sobre satisfazer fórmulas booleanas, mas que tem algumas restrições na estrutura da fórmula que é a entrada para o problema. Será muito mais fácil reduzir a partir desse problema restrinido. Vamos exigir que a fórmula seja ANDs de *cláusulas*, onde cada cláusula é uma OR de três termos e cada termo é um *literal*: ou uma variável ou a negação de uma variável (por exemplo, NOT x). A fórmula booleana nessa forma está na *forma normal conjuntiva de 3* ou **3-CNF**. Por exemplo, a fórmula booleana

$$(w \text{ OR } (\text{NOT } w) \text{ OR } (\text{NOT } x) \text{ AND } (y \text{ OR } x \text{ OR } z) \\ \text{AND } ((\text{NOT } w) \text{ OR } (\text{NOT } y) \text{ OR } (\text{NOT } z)))$$

está em 3-CNF. Sua primeira cláusula é $(w \text{ OR } (\text{NOT } w) \text{ OR } (\text{NOT } x))$.

Decidir se uma fórmula booleana em 3-CNF tem uma designação satisfatória às suas variáveis — o *problema da satisfazibilidade 3-CNF* — é NP-completo. Um certificado é uma designação proposta dos valores 0 e 1 às variáveis. Verificar um certificado é fácil: basta anexar os valores propostos às variáveis e verificar se o resultado da expressão é 1. Para mostrar que a satisfazibilidade 3-CNF é NP-difícil, reduzimos a partir da satisfazibilidade da fórmula booleana (não restrinida). Novamente, não entrarei nos detalhes (não são muito interessantes). Fica mais interessante quando reduzimos a partir de um problema em um domínio para um problema em um domínio diferente, que é o que estamos prestes a fazer.

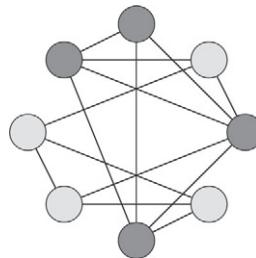
Eis um aspecto frustrante da satisfazibilidade 3-CNF: embora seja NP-completo, há um algoritmo de tempo polinomial para determinar se a fórmula 2-CNF é satisfazível. Uma fórmula 2-CNF é exatamente igual a uma fórmula 3-CNF exceto que ela tem dois literais, não três, em cada cláusula. Uma pequena mudança como essa passa um problema da categoria de tão difícil quanto os mais difíceis problemas em NP para a categoria de problema fácil!

Clique

Agora veremos uma redução interessante para problemas em domínios diferentes: da satisfazibilidade 3-CNF para um problema que tem a ver com grafos não dirigidos. Um *clique* em um grafo não dirigido G é um subconjunto S de vértices tal que o grafo tem uma aresta entre todo par de vértices em S . O *tamanho de um clique* é o número de vértices que ele contém.

Como você pode imaginar, cliques desempenham um papel na teoria da rede social. Modelando cada indivíduo como um vértice e os relacionamentos entre indivíduos como arestas não dirigidas, um clique representa um grupo de indivíduos no qual todos têm relacionamentos uns com os outros. Cliques também têm aplicações em bioinformática, engenharia e química.

O *problema do clique* toma duas entradas, um grafo G e um inteiro positivo k , e pergunta se G tem um clique de tamanho k . Por exemplo, o grafo na próxima página tem um clique de tamanho 4, mostrado com vértices bem escuro e nenhum outro clique de tamanho 4 ou maior.



Verificar um certificado é fácil. O certificado são os k vértices que formam um clique, e basta verificar se cada um dos k vértices tem uma aresta até os outros $k - 1$ vértices. Essa verificação é fácil de executar em tempo polinomial no tamanho do grafo. Agora sabemos que o problema do clique está em NP.

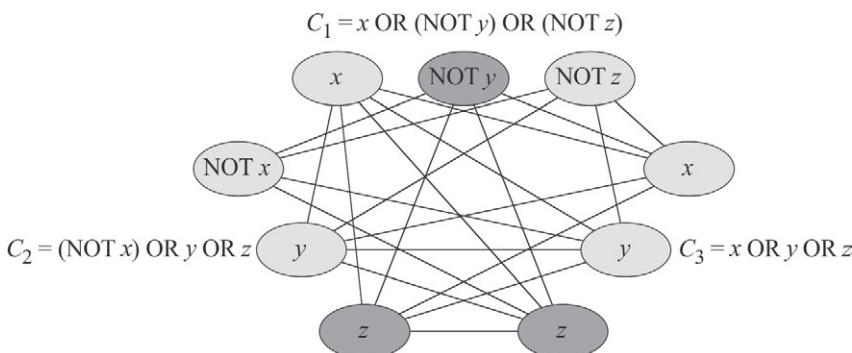
Como um programa de satisfazer fórmulas booleanas pode ser reduzido a um problema de grafo? Começamos com uma fórmula booleana em 3-CNF. Suponha que a fórmula seja $C_1 \text{ AND } C_2 \text{ AND } C_3 \text{ AND } \dots \text{ AND } C_k$, onde cada C_r é uma de k cláusulas. Com essa fórmula, construiremos um grafo em tempo polinomial, e esse grafo terá um clique k se e somente se a fórmula 3-CNF for satisfazível. Precisamos ver três coisas: a construção, um argumento que a construção executa em tempo polinomial no tamanho da fórmula 3-CNF e uma prova de que o grafo tem um clique de k se e somente se houver algum modo de designar as variáveis da fórmula 3-CNF de modo que o resultado seja 1.

Para construir um grafo a partir de uma fórmula 3-CNF, vamos focalizar a r -ésima cláusula, C_r . Ela tem três literais; vamos denominá-las l'_1, l'_2 e l'_3 , de modo que C_r é $l'_1 \text{ OR } l'_2 \text{ OR } l'_3$. Cada literal é uma variável ou a negação de uma variável. Criamos um vértice para cada literal, de modo que, para a cláusula C_r , criamos uma tripla de vértices, v_i^r, v_j^r e v_s^r . Adicionamos uma aresta entre os vértices v_i^r e v_j^s se duas condições forem válidas:

- v_i^r e v_j^s estão em triplas diferentes, isto é, r e s são números de cláusula diferentes, e
- seus literais correspondentes não são negações um do outro.

Por exemplo, o grafo corresponde à fórmula 3-CNF

$$(x \text{ OR } (\text{NOT } y) \text{ OR } (\text{NOT } z)) \text{ AND } ((\text{NOT } x) \text{ OR } y \text{ OR } z) \text{ AND } (x \text{ OR } y \text{ OR } z)$$



É bem fácil ver que essa redução pode ser executada em tempo polinomial. Se a fórmula 3-CNF tem k cláusulas, então tem $3k$ literais e, portanto, o grafo tem $3k$ vértices. No máximo, cada vértice tem uma aresta até todos os outros $3k - 1$ vértices, e assim o número de arestas é, no máximo, $3k(3k - 1)$, que é igual a $9k^2 - 3k$. O tamanho do grafo construído é polinomial no tamanho da entrada 3-CNF e é fácil determinar quais arestas entram no grafo.

Finalmente, precisamos mostrar que o grafo construído tem clique k se e somente se a fórmula 3-CNF for satisfazível. Começamos considerando que a fórmula é satisfazível, e mostraremos que o grafo tem clique k . Se existir uma designação satisfatória, cada cláusula C_i contém no mínimo um literal l_i^r cujo resultado é 1, e cada um de tais literais corresponde a um vértice v_i^r no grafo. Se selecionarmos um de tais literais de cada uma das k cláusulas, obteremos um conjunto S correspondente de k vértices. Eu alego que S é um clique k . Considere quaisquer dois vértices em S . Eles correspondem a literais em cláusulas diferentes cujo resultado é 1 na designação satisfatória. Esses literais não podem ser negações um do outro porque, se fossem, o resultado de um seria 1, mas o resultado do outro seria 0. Visto que esses literais não são negações um do outro, criamos uma aresta entre os dois vértices quando construímos o grafo. Como podemos escolher quaisquer dois vértices em S como esse par, vemos que há arestas entre todos os pares de vértices em S . Por consequência, S , um conjunto de k vértices, é um clique k .

Agora temos de mostrar a outra direção: se o grafo tem clique k , a fórmula 3-CNF é satisfazível. Nenhuma aresta no grafo conecta vértices na mesma tripla e, portanto, S contém exatamente um vértice por tripla. Para cada vértice v_i^r em S , designe 1 ao seu literal correspondente l_i^r na fórmula 3-CNF. Não temos de nos preocupar com designar um 1 a um literal e sua negação, visto que um clique k não pode conter vértices correspondentes a um literal e à sua negação. Visto que cada cláusula tem um literal cujo resultado é 1, cada cláusula é satisfeita e, assim, a fórmula 3-CNF inteira é satisfeita. Se quaisquer variáveis não corresponderem a vértices no clique, designe valores a elas arbitrariamente; elas não afetarão o fato de a fórmula ser satisfeita ou não.

No exemplo que acabamos de dar, uma designação satisfatória tem $y = 0$ e $z = 1$; não importa o que designamos a x . Um clique 3 correspondente consiste nos vértices em cor mais escura, que correspondem a NOT y da cláusula C_1 e a z das cláusulas C_2 e C_3 .

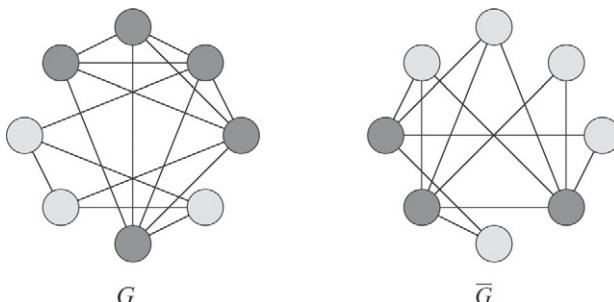
Assim, mostramos que existe uma redução em tempo polinomial do problema NP-completo da satisfazibilidade 3-CNF para o problema de encontrar um clique k . Se você tivesse uma fórmula booleana em 3-CNF com k cláusulas, e tivesse de encontrar uma designação satisfatória para a fórmula, poderia usar a construção que acabamos de ver para converter a fórmula em tempo polinomial em um grafo não dirigido e determinar se o grafo tinha um clique k . Se você pudesse determinar em tempo polinomial se o grafo tinha um clique k , teria determinado em tempo polinomial se a fórmula 3-CNF tinha uma designação satisfatória. Visto que a satisfazibilidade 3-CNF é NP-completa, determinar se um grafo contém clique k também é. Como bônus, você poderia determinar não somente se o grafo tinha clique k , mas quais vértices constituíam o clique k ; então poderia usar essa informação para determinar os valores a designar às variáveis da fórmula 3-CNF em uma designação satisfatória.

Cobertura de vértices

Uma **cobertura de vértices** em um grafo não dirigido G é um subconjunto S dos vértices tal que toda aresta em G é incidente no mínimo a um vértice em S . Dizemos que cada vértice em S “cobre” suas arestas incidentes. O **tamanho de uma cobertura de vértices** é o número de vértices que ela contém. Como no problema do clique, o **problema da cobertura de vértices** adota como entrada um grafo não dirigido G e um inteiro positivo m . Ele pergunta se G tem uma cobertura de vértices de tamanho m . Como o problema do clique, o problema da cobertura de vértices tem aplicações na bioinformática. Em outra aplicação, você tem um edifício com saguões e câmeras que podem ter um alcance de até 360 graus localizadas nas interseções dos saguões, e quer saber se m câmeras permitirão que você veja todos os saguões. Aqui, as arestas modelam saguões e os vértices modelam interseções. Em mais outra aplicação, determinar cobertura de vértices ajuda a engendrar estratégias para frustrar ataques de vermes em redes de computadores.

Um certificado para o problema da cobertura de vértices é, sem nenhuma surpresa, uma cobertura de vértices proposta. É fácil verificar em tempo polinomial no tamanho do grafo se a cobertura de vértices proposta tem tamanho m e realmente cobre todas as arestas e, assim, vermos que esse problema está em NP.

A árvore de família NP-completude na página 162 lhe diz que reduzimos o problema do clique ao problema da cobertura de vértices. Suponha que a entrada para o problema do clique seja um grafo não dirigido G com n vértices e um inteiro positivo k . Em tempo polinomial, produziremos um grafo de entrada \bar{G} para o problema da cobertura de vértices tal que G tem um clique de tamanho k se e somente se \bar{G} tiver uma cobertura de vértices de tamanho $n - k$. Essa redução é realmente fácil. O grafo \bar{G} tem os mesmos vértices que G e tem exatamente as arestas opostas a G . Em outras palavras, a aresta (u,v) está em \bar{G} se e somente se (u,v) não estiver em \bar{G} . Você bem que poderia ter adivinhado que a cobertura de vértices de tamanho $n - k$ em \bar{G} consiste nos vértices que *não* estão no clique de k vértices em G e você estaria correto! Damos a seguir exemplos de grafos G e \bar{G} , com oito vértices. Os cinco vértices que formam um clique em G e os três vértices restantes que formam uma cobertura de vértices em \bar{G} são apresentados em tom mais escuro:



Observe que toda aresta em \bar{G} é incidente no mínimo a um vértice em cor mais escura.

Precisamos mostrar que G tem u clique k se e somente se \bar{G} tiver uma cobertura de vértices de tamanho $n - k$. Começamos supondo que G tem um clique k C . Deixemos S consistir nos $n - k$ vértices que não estão em C . Eu alego que toda aresta em \bar{G} é incidente no mínimo em um vértice em S . Seja (u,v) qualquer aresta em \bar{G} . Ela está em \bar{G} porque não estava em G . Como (u,v) não está em G , no mínimo um dos vértices u e v não está no clique C de G , porque uma aresta conecta todo par de vértices em C . Visto que, no mínimo, um de u e v não está em C , no mínimo um de u e v está em S , o que significa que a aresta (u,v) é incidente no mínimo a um dos vértices em S . Visto que escolhemos (u,v) para ser qualquer aresta em \bar{G} , vemos que S é uma cobertura de vértices para \bar{G} .

Agora vamos pelo outro caminho. Suponha que \bar{G} tenha uma cobertura de vértices S que contém $n - k$ vértices, e façamos C consistir nos k vértices que não estão em S . Toda aresta em \bar{G} é incidente em algum vértice em S . Em outras palavras, se (u, v) é uma aresta em \bar{G} , então no mínimo um de u e v está em S . Se você se lembrar da definição de contrapositivo da página 19, poderá ver que o contrapositivo dessa implicação é que nem u nem v estão em S , então (u, v) não está em \bar{G} — portanto, (u, v) está em \bar{G} . Em outras palavras, se u e v estão em C , então a aresta (u, v) está presente em G . Visto que u e v são qualquer par de vértices em C , vemos que há uma aresta em G entre todos os pares de vértices em C . Isto é, C é um clique k .

Assim, mostramos que existe uma redução em tempo polinomial do problema NP-completo de determinar se um grafo não dirigido contém um clique k para o problema de determinar se um grafo não dirigido contém uma cobertura de vértices de tamanho $n - k$. Se você tivesse um grafo não dirigido G e quisesse saber se ele contém um clique k , poderia usar a construção que acabamos de ver para converter G em tempo polinomial em \bar{G} e determinar se \bar{G} continha uma cobertura de vértices com $n - k$ vértices. Se você pudesse determinar em tempo polinomial se \bar{G} tinha uma cobertura de vértices de tamanho $n - k$, teria determinado em tempo polinomial se G tinha um clique k . Visto que o problema do clique é NP-completo, o problema da cobertura de vértices também é. Como bônus, se você pudesse determinar não somente se \bar{G} tinha uma cobertura de vértices de $n - k$ vértices, mas também quais vértices constituíam a cobertura, poderia usar essa informação para determinar os vértices no clique k .

Ciclo hamiltoniano e caminho hamiltoniano

Já vimos o problema do ciclo hamiltoniano: um grafo conectado, não dirigido, contém um ciclo hamiltoniano (um caminho que começa e termina no mesmo vértice e visita todos os outros vértices exatamente uma vez)? As aplicações desse problema são um pouco misteriosas, mas pela árvore de família da NP-completude, na página 162, você pode ver que usamos esse problema para mostrar que o problema do caixeiro-viajante é NP-completo, e vimos como o problema do caixeiro-viajante surge na prática.

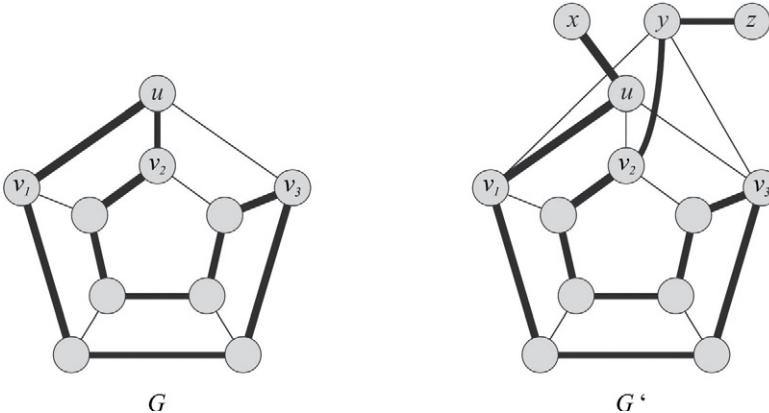
Um problema estreitamente relacionado é o *problema do caminho hamiltoniano*, que pergunta se o grafo contém um caminho que visita cada vértice exatamente uma vez, mas não exige que o caminho seja um ciclo fechado. Esse problema é, também,

NP-completo, e o usaremos na página 169 para mostrar que o problema do caminho acíclico mais longo é NP-completo.

Para ambos os problemas hamiltonianos, o certificado é óbvio: a ordem dos vértices no ciclo hamiltoniano ou no caminho hamiltoniano (para um ciclo hamiltoniano, não repita o primeiro vértice no final). Dado um certificado, basta verificar que cada vértice aparece exatamente uma vez na lista e que o grafo contém uma aresta entre cada par de vértices consecutivos na ordenação. Para o problema do ciclo hamiltoniano, também temos de verificar se existe uma aresta entre o primeiro e o último vértice.

Não detalharei a redução em tempo polinomial do problema da cobertura de vértices para o problema do ciclo hamiltoniano, que mostra que o último é NP-difícil. É bastante complicada e recorre a um *widget*, que é um pedaço de um grafo que impõe certas propriedades. O widget usado na redução tem a seguinte propriedade: qualquer ciclo hamiltoniano no grafo construído pela redução pode percorrer o widget apenas em um de três modos.

Para reduzir o problema do ciclo hamiltoniano ao problema do caminho hamiltoniano, começamos com um grafo G conectado, não dirigido, com n vértices, e a partir dele formaremos um novo grafo G' conectado, não dirigido, com $n + 3$ vértices. Escolhemos qualquer vértice u em G , e denominamos seus vértices adjacentes v_1, v_2, \dots, v_k . Para construir G' , adicionamos três novos vértices, x, y e z , e adicionamos as arestas (u, x) e (y, z) juntamente com as arestas $(v_1, y), (v_2, y), \dots, (v_k, y)$ entre y e todos os vértices adjacentes a u . Eis um exemplo:



Arestas grossas indicam um ciclo hamiltoniano em G e um caminho hamiltoniano correspondente em G' . Essa redução leva tempo polinomial, visto que G' contém apenas mais três vértices que G e no máximo $n + 1$ arestas adicionais.

Como sempre, precisamos mostrar que a redução funciona: que G tem um ciclo hamiltoniano se e somente se G' tem um caminho hamiltoniano. Suponha que G tenha um ciclo hamiltoniano. Ele deve conter uma aresta (u, v_i) para algum vértice v_i adjacente a u e, portanto, adjacente a y em G' . Para formar um caminho hamiltoniano em G' , indo de x para z , tome todas as arestas do ciclo hamiltoniano exceto (u, v_i) e adicione as arestas $(u, x), (v_i, y)$ e (y, z) . No exemplo anterior, v_i é o vértice v_2 , e

portanto o caminho hamiltoniano omite a aresta (v_2, y) e adiciona as arestas (u, x) , (v_2, y) e (y, z) .

Agora suponha que G' tenha um caminho hamiltoniano. Como cada um dos vértices x e z tem somente uma aresta incidente, o caminho hamiltoniano deve ir de x a z e conter uma aresta (v_i, y) para algum vértice adjacente a y e, portanto, adjacente a u . Para encontrar um ciclo hamiltoniano em G , elimine x, y, z e todas as suas arestas incidentes, e use todas as arestas no caminho hamiltoniano em G' , juntamente com (v_i, u) .

Um desenlace semelhante aos de nossas reduções anteriores se mantém aqui. Existe uma redução em tempo polinomial do problema NP-completo de determinar se um grafo conectado, não dirigido, contém um ciclo hamiltoniano para o problema de determinar se um grafo conectado, não dirigido, contém um caminho hamiltoniano. Visto que o primeiro é NP-completo, o último também é. Além do mais, conhecer as arestas no caminho hamiltoniano dá as arestas no ciclo hamiltoniano.

Caixeiro-viajante

Na versão de decisão do **problema do caixeiro-viajante**, temos um grafo não dirigido completo com um peso inteiro não negativo em cada aresta e um inteiro não negativo k . Um **grafo completo** tem uma aresta entre todo par de vértices, de modo que, se um grafo completo tiver n vértices, então tem $n(n - 1)$ arestas. Perguntamos se o grafo tem um ciclo que contém todos os vértices cujo peso total é, no máximo, k .

É bem fácil mostrar que esse problema está em NP. Um certificado consiste nos vértices do ciclo, em ordem. Podemos facilmente verificar em tempo polinomial se as arestas nesse ciclo visitam todos os vértices e têm peso total de k ou menos.

Para mostrar que o problema do caixeiro-viajante é NP-difícil, o reduzimos a partir do problema do ciclo hamiltoniano — outra redução simples. Dado um grafo G como entrada para o problema do ciclo hamiltoniano, construímos um grafo completo G' com os mesmos vértices de G . Igualamos o peso de aresta (u, v) em G' a 0 se (u, v) é uma aresta em G e o igualamos a 1 se não há nenhuma aresta (u, v) em G . Igualamos k a 0. Essa redução leva tempo polinomial no tamanho de G , visto que acrescenta, no máximo, $n(n - 1)$ arestas.

Para mostrar que a redução funciona, precisamos mostrar que G tem um ciclo hamiltoniano se e somente se G' tem um ciclo de peso 0 que inclui todos os vértices. Mais uma vez, o argumento é fácil. Suponha que G tenha um ciclo hamiltoniano. Então cada aresta no ciclo está em G e, assim, cada uma dessas arestas obtém um peso de 0 em G' . Desse modo, G' tem um ciclo que contém todos os vértices, e o peso total desse ciclo é 0. Ao contrário, agora suponha que G' tenha um ciclo que contém todos os vértices e cujo peso total é 0. Então cada aresta nesse ciclo deve também estar em G , e portanto G tem um ciclo hamiltoniano.

Nem preciso repetir o desenlace já familiar, não é?

Caminho acíclico mais longo

Na versão de decisão do **problema do caminho acíclico mais longo**, temos um grafo não dirigido G e um inteiro k , e perguntamos se G contém dois vértices que têm um caminho acíclico entre eles com, no mínimo, k arestas.

Novamente, um certificado para o problema do caminho acíclico mais longo é fácil de verificar. Consiste nos vértices no caminho proposto, em ordem. Podemos verificar em tempo polinomial que a lista contém no mínimo $k + 1$ vértices ($k + 1$ porque um caminho com k arestas contém $k + 1$ vértices) sem nenhum vértice repetido e que há uma aresta entre todo par de vértices consecutivos na lista.

Ainda outra redução simples mostra que esse problema é NP-difícil. Reduzimos a partir do problema do caminho hamiltoniano. Dado um grafo G com n vértices como entrada para o problema do caminho hamiltoniano, a entrada para o problema do caminho acíclico mais longo é o grafo G , sem nenhuma mudança, e o inteiro $k = n - 1$. Se essa não for uma redução em tempo polinomial, não sei qual será.

Mostramos que a redução funciona mostrando que G tem um caminho hamiltoniano se e somente se tiver um caminho que contenha no mínimo $n - 1$ arestas. Mas um caminho hamiltoniano é um caminho acíclico que contém $n - 1$ arestas, então terminamos!

Soma de subconjuntos

No **problema da soma de subconjuntos**, a entrada é um conjunto finito S de inteiros positivos, sem nenhuma ordem particular, e um número-alvo t , que é também um inteiro positivo. Perguntamos se existe um subconjunto S' de S cujos elementos somam exatamente t . Por exemplo, se S é o conjunto $\{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$ e $t = 138457$, então o subconjunto $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$ é uma solução. Um certificado, é claro, é um subconjunto de S , que podemos verificar apenas somando os números no subconjunto e verificando se sua soma é igual a t .

Como você pode ver na árvore de família da NP-completude, na página 162, mostramos que o problema da soma de subconjuntos é NP-difícil reduzindo a partir da satisfazibilidade 3-CNF. Damos aqui outra redução que cruza domínios de problemas, transformando um problema de lógica em um problema de aritmética. Você verá que a transformação é inteligente, porém, afinal de contas, bastante direta.

Começamos com uma fórmula booleana 3-CNF F que tem n variáveis e k cláusulas. Vamos nomear as variáveis $v_1, v_2, v_3, \dots, v_n$ e as cláusulas $C_1, C_2, C_3, \dots, C_k$. Cada cláusula contém exatamente três literais (lembre-se de que cada literal é v_i ou NOT v_i) unidos por ORs, e a fórmula inteira é C_1 AND C_2 AND C_3 AND... AND C_k . Em outras palavras, para dada designação de 0 ou 1, a cada variável, cada cláusula é satisfeita se de qualquer de seus literais resultar 1, e a fórmula completa F é satisfeita somente se todas as suas cláusulas são satisfeitas.

Antes de construirmos o conjunto S para o problema da soma de subconjuntos, vamos construir o número-alvo t a partir da fórmula 3-CNF F . Nós o construiremos como um inteiro decimal com $n + k$ dígitos. Os k dígitos menos significativos (os k dígitos da extrema direita) de t correspondem às k cláusulas de F , e cada um desses dígitos é um 4. Os n dígitos mais significativos de t correspondem às n variáveis de F , e cada um desses dígitos é um 1. Se a fórmula F tiver, digamos, três variáveis e quatro cláusulas, t dá 1114444. Como veremos, se houver um subconjunto de S cuja soma seja t , os dígitos de t que correspondem às variáveis (os 1) garantirão que designamos um valor a cada variável em F , e os dígitos de t que correspondem às cláusulas (os 4) garantirão que cada cláusula de F seja satisfeita.

O conjunto S consistirá em $2n + 2k$ inteiros. Contém inteiros x_i e x'_i para cada uma das n variáveis v_i , na fórmula 3-CNF F , e inteiros q_j e q'_j para cada uma das k cláusulas C_j em F . Construímos cada inteiro em S , dígito por dígito, em decimal. Veja um exemplo com $n = 3$ variáveis e $k = 4$ cláusulas, de modo que a fórmula 3-CNF é $F = C_1 \text{ AND } C_2 \text{ AND } C_3 \text{ AND } C_4$, e sejam as cláusulas

$$C_1 = v_1 \text{ OR } (\text{NOT } v_2) \text{ OR } (\text{NOT } v_3) \quad D \text{ v1OR .NOT v2/OR .NOT v3;}$$

$$C_2 = (\text{NOT } v_1) \text{ OR } (\text{NOT } v_2) \text{ OR } (\text{NOT } v_3);$$

$$C_3 = (\text{NOT } v_1) \text{ OR } (\text{NOT } v_2) \text{ OR } v_3;$$

$$C_4 = v_1 \text{ OR } v_2 \text{ OR } v_3$$

Eis o conjunto S e o alvo t correspondentes:

	v_1	v_2	v_3	C_1	C_2	C_3	C_4
x_1	=	1	0	0	1	0	0
x'_1	=	1	0	0	0	1	1
x_2	=	0	1	0	0	0	0
x'_2	=	0	1	0	1	1	0
x_3	=	0	0	1	0	0	1
x'_3	=	0	0	1	1	0	0
q_1	=	0	0	0	1	0	0
q'_1	=	0	0	0	2	0	0
q_2	=	0	0	0	0	1	0
q'_2	=	0	0	0	0	2	0
q_3	=	0	0	0	0	0	1
q'_3	=	0	0	0	0	0	2
q_4	=	0	0	0	0	0	0
q'_4	=	0	0	0	0	0	2
t	=	1	1	1	4	4	4

Observe que os elementos sombreados de S — 1000110, 101110, 10011, 1000, 2000, 200, 10, 1 e 2 — somam 1114444. Logo veremos a que correspondem esses elementos na fórmula 3-CNF F .

Construímos os inteiros em S de modo que, dígito por dígito, toda coluna no diagrama acima some ou 2 (as n colunas da extrema esquerda) ou 6 (as k colunas da extrema direita). Observe que, quando elementos em S são somados, não pode ocorrer nenhum “vai um” de posição de qualquer dígito e podemos trabalhar com os números dígito por dígito.

No diagrama, cada linha é identificada por um elemento em S . As primeiras $2n$ linhas correspondem às n variáveis da fórmula 3-CNF, e as últimas $2k$ linhas são “folgas” cuja finalidade veremos mais adiante. As linhas identificadas por elementos x_i e x'_i correspondem, respectivamente, a ocorrências dos literais v_i e $\text{NOT } v_i$ em F .

Diremos que essas linhas “são” os literais, entendendo que queremos dizer que elas correspondem aos literais. A meta é incluir no subconjunto S' exatamente n das $2n$ primeiras linhas — na verdade, exatamente uma de cada par x_i, x'_i — que corresponderão à designação satisfatória para a fórmula 3-CNF F . Como exigimos que as linhas que escolhemos a partir dos literais somem 1 em cada uma das n colunas da extrema esquerda, nos certificamos de que, para cada variável v_i na fórmula 3-CNF, incluímos em S' uma linha para x_i e x'_i , mas não para ambos. As k colunas da extrema direita garantem que as linhas que incluímos em S' são literais que satisfazem cada cláusula na fórmula 3-CNF.

Vamos focalizar, por enquanto, as n colunas da extrema esquerda, que identificamos pelas variáveis v_1, v_2, \dots, v_n . Para dada variável v_i , x_i e x'_i têm um 1 no dígito correspondente a v_i e 0 em todas as outras posições de dígitos correspondentes a outras variáveis. Por exemplo, os três dígitos da extrema esquerda de x_2 e x'_2 são 010. Os dígitos das últimas $2k$ linhas nas n colunas na extrema esquerda são 0. Como o alvo t tem um 1 em cada uma das posições variáveis, exatamente um de x_i e x'_i deve estar no subconjunto S' de modo a contribuir para a soma. Ter x_i em S' corresponde a igualar v_i a 1, e ter x'_i em S' corresponde a igualar v_i a 0.

Agora voltamos nossa atenção para as k colunas da extrema direita, que correspondem às cláusulas. Essas colunas garantem que cada cláusula seja satisfeita, como veremos a seguir. Se o literal v_i aparecer na cláusula C_j , então x_i tem um 1 na coluna para C_j ; se o literal NOT v_i aparecer na cláusula C_j , então x'_i tem um 1 na coluna C_j . Como cada cláusula na fórmula 3-CNF contém exatamente três literais distintos, a coluna para cada cláusula deve conter exatamente três 1s entre todas as linhas x_i e x'_i . Para dada cláusula C_j , as linhas entre as primeiras $2n$ que estão incluídas em S' correspondem a satisfazer 0, 1, 2 ou 3 dos literais em C_j , e portanto essas linhas contribuem 0, 1, 2 ou 3 para a soma para a coluna de C_j .

Porém, o dígito-alvo para cada cláusula é 4, e é aí que entram os elementos de “folga” q_j e q'_j , para $j = 1, 2, 3, \dots, k$. Eles garantem que, para cada cláusula, o subconjunto S' inclui algum literal na cláusula (algum x_i ou x'_i que tem um 1 na coluna para essa cláusula). A linha para q_j tem 1 na coluna para a cláusula C_j e 0 em todos os outros lugares, e a linha para q'_j é a mesma, exceto que tem um 2. Podemos incluir essas linhas para obter o dígito-alvo de 4, mas somente se o subconjunto S' incluir no mínimo um literal de C_j . Quais dessas linhas de folga precisam ser incluídas depende de quantos dos literais da cláusula C_j estão incluídos em S' . Se S' incluir apenas um literal, ambas as linhas de folga serão necessárias, porque a soma na coluna é 1 advinda do literal, mais 1 procedente de q_j , mais 2 que vem de q'_j . Se S' incluir dois literais, apenas q'_j é necessário, porque a soma da coluna é 2 advindo dos dois literais, mais 2 advindo de q'_j . Se S' incluir três literais, apenas q_j é necessário, porque a soma da coluna é 3 advindo dos três literais, mais 1 que vem de q_j . Porém, se nenhum literal advindo da cláusula C_j estiver incluído em S' , $q_j + q'_j = 3$ não é suficiente para atingir o dígito-alvo 4. Portanto, podemos conseguir o dígito-alvo 4 para cada cláusula somente se algum literal na cláusula for incluído no subconjunto S' .

Agora, que já vimos a redução, podemos ver que ela leva tempo polinomial. Estamos criando $2n + 2k + 1$ inteiros (incluindo o alvo t), cada um com $n + k$ dígitos. Você

pode ver pelo diagrama que, dos inteiros construídos, não há dois iguais, e portanto S é realmente um conjunto (a definição de conjunto não permite elementos repetidos).

Para mostrar que a redução funciona, precisamos mostrar que a fórmula 3-CNF F tem uma designação satisfazível se e somente se existir um subconjunto S' de S que some exatamente t . Nesse ponto, você já viu a ideia, mas vamos recapitular. Em primeiro lugar, suponha que F tenha uma designação satisfatória. Se essa designação igualar v_i a 1, inclua x_i em S' ; caso contrário, inclua x'_i . Como exatamente um de x_i e x'_i está em S , a coluna para v_i deve somar 1, correspondendo ao dígito adequado de t . Como a designação satisfaz cada cláusula C_j , as linhas x_i e x'_i devem contribuir com 1, 2 ou 3 (o número de literais em C_j , que são 1) para a soma na coluna de C_j . Incluir as linhas de folga necessárias q_j e/ou q'_j em S' atinge o dígito-alvo 4.

Ao contrário, suponha que S tenha um subconjunto S' que soma exatamente t . Para t ter um 1 nas n posições da extrema esquerda, S' deve incluir exatamente um de x_i e x'_i para cada variável v_i . Se incluir x_i , iguale v_i a 1; se incluir x'_i , iguale v_i a 0. Como as linhas de folga q_j e q'_j somadas não podem atingir o dígito-alvo 4 na coluna para a cláusula C_j , o subconjunto S' deve também incluir, no mínimo, uma linha x_i ou x'_i com um 1 na coluna de C_j . Se incluir x_i , o literal v_i aparece na cláusula C_j , e a cláusula é satisfeita. Se S' incluir x'_i , o literal NOT v_i aparece na cláusula C_j , e a cláusula é satisfeita. Assim, cada cláusula é satisfeita e existe uma designação satisfatória para a fórmula 3-CNF F .

Assim, vemos que, se pudermos resolver o problema da soma de subconjuntos em tempo polinomial, poderemos também determinar se uma fórmula 3-CNF é satisfazível em tempo polinomial. Visto que a satisfazibilidade 3-CNF é NP-completa, o problema da soma de subconjuntos também é. Além do mais, se soubermos quais inteiros no alvo construído S somam t , poderemos determinar como igualar as variáveis na fórmula 3-CNF de modo que seu valor seja 1.

Outra observação sobre a redução que eu usei: os dígitos não têm de ser dígitos decimais. O que importa é que não possa ocorrer nenhum “vai um” de um lugar para outro na soma dos inteiros. Visto que a soma de nenhuma coluna pode passar de 6, tudo bem se interpretarmos os números em qualquer base 7 ou maior. Na verdade, o exemplo que eu dei na página 170 vem dos números no diagrama, mas interpretados em base 7.

Partição

O **problema da partição** está intimamente relacionado ao problema da soma de subconjuntos. Na realidade, é um caso especial do problema da soma de subconjuntos: se z é igual à soma de todos os inteiros no conjunto S , então o alvo t é exatamente $z/2$. Em outras palavras, a meta é determinar se existe uma partição do conjunto S em dois conjuntos disjuntos S' e S'' tal que cada inteiro em S esteja em S' ou em S'' , mas não ambos (é isso que significa S' e S'' particionarem S) e a soma dos inteiros em S' é igual à soma dos inteiros em S'' . Como no problema da soma de subconjuntos, um certificado é um subconjunto de S .

Para mostrar que o problema da partição é NP-difícil, reduzimos a partir do problema da soma de subconjuntos (nenhuma grande surpresa). Dados um conjunto R de inteiros positivos e um inteiro-alvo positivo t como entradas para o problema da soma

de subconjuntos, construímos em tempo polinomial um conjunto S como entrada para o problema da partição. Em primeiro lugar, computamos' como a soma de todos os inteiros em R . Pressupomos que z não seja igual a $2t$ porque, se for, o problema já é um problema de partição. (Se $z = 2t$, então $t = z/2$ e estamos tentando encontrar um subconjunto de R que some o mesmo total que os inteiros que não estão no subconjunto.) Então escolha qualquer inteiro y que seja maior que $t + z$ e $2z$. Defina o conjunto S para conter todos os inteiros em R e dois inteiros adicionais: $y - t$ e $y - z + t$. Como y é maior que $t + z$ e $2z'$, sabemos que $y - t$ e $y - z + t$ são maiores que z (a soma dos inteiros em R) e, portanto, esses dois inteiros não podem estar em R . (Lembre-se de que, como S é um conjunto, todos os seus elementos devem ser únicos. Também sabemos que, como z não é igual a $2t$, devemos ter $y - t \neq y - z + t$ e, portanto, os dois novos inteiros são únicos.) Observe que a soma de todos os inteiros em S é igual a $z + (y - t) + (y - z + t)$, que é exatamente $2y$. Portanto, se S for particionado em dois subconjuntos disjuntos com somas iguais, cada subconjunto deve somar y .

Para mostrar que a redução funciona, precisamos mostrar que existe um subconjunto R' de R cujos inteiros somam t se e somente se existir uma partição de S em S' e S'' tal que os inteiros em S' e os inteiros em S'' têm a mesma soma. Em primeiro lugar, vamos supor que algum subconjunto R' de R tenha inteiros que somam t . Então os inteiros em R que não estão em R' devem somar $z - t$. Vamos definir o conjunto S' de modo que tenha todos os inteiros em R' juntamente com $y - t$ (de modo que S'' tenha $y - z + t$ juntamente com todos os inteiros em R que não estão em R'). Agora basta mostrar que os inteiros em S' somam y , mas isso é fácil: os inteiros em R' somam t , e inserir $y - t$ dá uma soma y .

Ao contrário, vamos supor que exista uma partição de S em S' e S'' e que ambas somem y . Eu alego que os dois inteiros que adicionamos a R quando formamos S ($y - t$ e $y - z + t$) não podem estar ambos em S' nem podem estar ambos em S'' . Por quê? Se eles estivessem no mesmo conjunto, esse conjunto somaria, no mínimo, $(y - t) + (y - z + t)$, que é igual a $2y - z$. Mas lembre-se de que y é maior que z (na verdade é maior que $2z$) e, portanto, $2y - z$ é maior que y . Por isso, se $y - t$ e $y - z + t$ estivessem no mesmo conjunto, a soma desse conjunto seria maior que y . Portanto, sabemos que ou $y - t$ e $y - z + t$ está em S' e o outro está em S'' . Não importa em qual conjunto dizemos que $y - t$ está; portanto, vamos dizer que ele está em S' . Agora sabemos que os inteiros em S' somam y , o que significa que os inteiros em S' , exceto $y - t$, devem somar $y - (y - t)$ ou t . Visto que $y - z + t$ não pode estar também em S' , sabemos que todos os outros inteiros em S' vêm de R . Por consequência, há um subconjunto de R que soma t .

Mochila

No **problema da mochila**, temos um conjunto de n itens, cada um com um peso e um valor, e perguntamos se existe um subconjunto de itens cujo peso total é, no máximo, um peso W dado, cujo valor total é no mínimo um valor V dado. Esse problema é a versão de decisão de um problema de otimização no qual queremos encher uma mochila com o subconjunto mais valioso de itens, desde que não excedamos um limite de peso. Esse problema de otimização tem aplicações óbvias, por exemplo, decidir

quais itens levar em uma mochila durante uma excursão ou quais objetos um ladrão deve surripiar.

O problema da partição é realmente apenas um caso especial do problema da mochila, no qual o valor de cada item é igual ao seu peso, e W e V são iguais à metade do peso total. Se pudéssemos resolver o problema da mochila em tempo polinomial, poderíamos resolver o problema da partição em tempo polinomial. Portanto, o problema da mochila é no mínimo tão difícil quanto o problema da partição, e nem precisamos passar por todo o processo de redução para mostrar que o problema da mochila é NP-completo.

ESTRATÉGIAS GERAIS

Como você já deve ter percebido, não há nenhum modo geral que sirva para reduzir um problema a um outro problema de modo a provar a dificuldade NP. Algumas reduções são bastante simples, como reduzir o problema do ciclo hamiltoniano ao problema do caixeiro-viajante, e outros são extremamente complicados. Damos a seguir algumas coisas que você deve lembrar e algumas estratégias que muitas vezes ajudam.

Vá do geral para o específico

Ao reduzir o problema X ao problema Y , você sempre tem de começar com uma entrada arbitrária para o problema X , porém pode restringir a entrada para o problema Y o quanto quiser. Por exemplo, ao reduzir da satisfazibilidade 3-CNF para o problema da soma de subconjuntos, a redução tinha de ser capaz de tratar *qualquer* fórmula 3-CNF como sua entrada, mas a entrada da soma de subconjuntos que produzia tinha uma estrutura particular: $2n + 2k$ inteiros no conjunto e cada inteiro era formado de um modo particular. A redução não conseguia produzir *toda* entrada possível para o problema da soma de subconjuntos, mas tudo bem. O ponto que queremos ressaltar é que podemos resolver um problema de satisfazibilidade 3-CNF transformando a entrada em uma entrada para o problema da soma de subconjuntos e depois usar a resposta para o problema da soma de subconjuntos como a resposta para o problema da satisfazibilidade 3-CNF.

Porém, observe que toda redução tem de ser desta forma: transformar *qualquer* entrada para o problema X em *alguma* entrada para um problema Y , mesmo quando encadeamos reduções. Se você quiser reduzir o problema X ao problema Y e também o problema Y ao problema Z , a primeira redução tem de transformar *qualquer* entrada para X em *alguma* entrada para Y , e a segunda redução tem de transformar *qualquer* entrada para Y em *alguma* entrada para Z . Não é suficiente que a segunda redução transforme somente os tipos de entradas para Y que são produzidos da redução a partir de X .

Aproveite as restrições no problema a partir do qual você está reduzindo

Em geral, ao reduzir do problema X para o problema Y , você pode decidir que o problema X imponha mais restrições à sua entrada. Por exemplo, é quase sempre muito mais fácil reduzir a partir da satisfazibilidade 3-CNF do que reduzir a partir do Problema Mãe de satisfazibilidade da fórmula booleana. Fórmulas booleanas podem

ser arbitrariamente complicadas, mas você viu como podemos explorar a estrutura de fórmulas 3-CNF na redução.

De modo semelhante, em geral é mais direto reduzir a partir do problema do ciclo hamiltoniano do que a partir do problema do caixeiro-viajante, ainda que eles sejam tão semelhantes. Isso porque, no problema do caixeiro-viajante, os pesos de arestas podem ser quaisquer inteiros positivos, e não apenas 0 ou 1 que exigimos ao reduzi-lo. O problema do ciclo hamiltoniano é mais restritivo porque cada aresta tem somente um de dois “valores”: presente ou ausente.

Procure casos especiais

Diversos problemas NP-completos são apenas casos especiais de outros problemas NP-completos, mais ou menos como o problema da partição é um caso especial do problema da mochila. Se você souber que o problema X é NP-completo e que é um caso especial do problema Y , o problema Y deve ser NP-completo também. Isso porque, como vimos no problema da mochila, uma solução em tempo polinomial para o problema Y daria automaticamente uma solução em tempo polinomial para o problema X . Mais intuitivamente, o problema Y , por ser mais geral que o problema X , é no mínimo tão difícil.

Selecione um problema adequado a partir do qual reduzir

Frequentemente, é uma boa estratégia reduzir a partir de um problema que está no mesmo domínio ou em um domínio no mínimo relacionado ao problema que você está tentando provar que é NP-completo. Por exemplo, mostramos que o problema da cobertura de vértices — um problema de grafo — era NP-completo reduzindo-o a partir do problema do clique — também um problema de grafo. Dali em diante, a árvore de família da NP-completude mostrou que o reduzimos aos problemas do ciclo hamiltoniano, caminho hamiltoniano, caixeiro-viajante e caminho acíclico mais longo, todos eles também problemas de grafo.

Todavia, às vezes é melhor saltar de um domínio para outro, por exemplo quando reduzimos da satisfazibilidade 3-CNF ao problema do clique ou ao problema da soma de subconjuntos. Frequentemente a satisfazibilidade 3-CNF demonstra ser uma boa escolha a partir da qual reduzir quando cruzamos domínios.

Dentro dos problemas de grafos, se você precisar selecionar uma porção do grafo sem se importar com a ordenação, o problema da cobertura de vértices é muitas vezes um bom lugar para começar. Se a ordenação importar, considere partir do problema do ciclo hamiltoniano ou do problema do caminho hamiltoniano.

Dê grandes recompensas e aplique grandes multas

Quando transformamos o grafo de entrada G do problema do ciclo hamiltoniano no grafo ponderado G' como entrada para o problema do caixeiro-viajante, na realidade queríamos incentivar a utilização de arestas presentes em G quando escolhemos arestas para a rota do caixeiro-viajante. Fizemos isso dando a essas arestas um peso muito baixo: 0. Em outras palavras, demos uma grande recompensa para a utilização dessas arestas.

Alternativamente, poderíamos ter dado às arestas em G um peso finito e às arestas que não estão em G um peso infinito, impondo assim pesada multa pela utilização de arestas que não estão em G . Se tivéssemos adotado essa abordagem e dado a cada aresta em G um peso W , teríamos de ter igualado o peso-alvo k da rota inteira do caixeiro-viajante a nW .

Projete widgets

Não entrei no projeto de widgets porque widgets podem ficar complicados. Os widgets podem ser úteis para impor certas propriedades. Os livros citados na seção “O que mais ler?” dão exemplos de como construir e usar widgets em reduções.

PERSPECTIVA

O quadro que pintei aqui é bastante sombrio, não é? Imagine um cenário no qual você tenta obter um algoritmo de tempo polinomial para resolver um problema e, não importando o quanto tente, não consegue obtê-lo, simplesmente não consegue fechar o negócio. Depois de algum tempo você ficará animadíssimo só por ter encontrado um algoritmo de tempo $O(n^5)$, ainda que saiba que n^5 cresce com tremenda rapidez. Talvez esse problema seja parecido com um que você sabe que é fácil de resolver em tempo polinomial (como satisfazibilidade 2-CNF versus 3-CNF ou passeio de Euler versus ciclo hamiltoniano) e fique frustradíssimo por não poder adaptar o algoritmo de tempo polinomial ao seu problema. A certa altura, você suspeita que talvez — mas só talvez — tenha ficado batendo a cabeça na parede para resolver um problema NP-completo. Veja só, conseguiu reduzir um problema NP-completo conhecido ao seu problema e agora sabe que ele é NP-difícil.

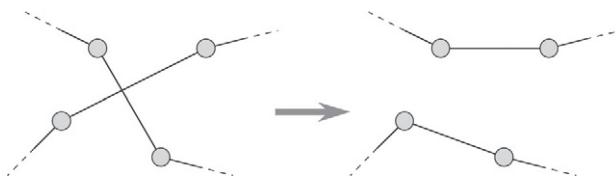
Esse é o fim da história? Não há nenhuma esperança de você conseguir resolver o problema em qualquer quantidade de tempo razoável?

Não é bem assim. Quando um problema é NP-completo, significa que *algumas* entradas são problemáticas, mas não necessariamente que *todas as* entradas são ruins. Por exemplo, determinar um caminho acíclico mais longo em um grafo dirigido é NP-completo, mas, se você sabe que o grafo é acíclico, pode encontrar um caminho acíclico mais longo não apenas em tempo polinomial, mas em tempo $O(n + m)$ (quando o grafo tem n vértices e m arestas). Lembre-se de que fizemos exatamente isso quando determinamos um caminho crítico no diagrama PERT no Capítulo 5. Como outro exemplo, se você estiver tentando resolver o problema da partição e os inteiros no conjunto somam um número ímpar, sabe que não há nenhum modo de particionar o conjunto de maneira que ambas as partes tenham somas iguais.

A boa notícia vai além de tais casos patológicos especiais. Daqui em diante vamos focalizar problemas de otimização cujas variantes de decisão são NP-completas, como o problema do caixeiro-viajante. Alguns métodos rápidos dão resultados bons e, frequentemente, muito bons. A técnica de *ramificar e podar* (“*branch and bound*”) organiza uma busca de uma solução ótima em uma estrutura semelhante à de árvore e corta pedaços da árvore, eliminando assim grandes porções do espaço de busca, tendo como base a simples ideia de que, se for possível determinar que todas as soluções que emanam de um nó da árvore de busca não podem ser melhores que a melhor solução

encontrada até esse ponto, então não se preocupe com verificar soluções dentro do espaço representado por aquele nó ou por qualquer coisa abaixo dele.

Outra técnica que frequentemente ajuda é a **busca na vizinhança**, que toma uma solução e aplica operações locais para tentar melhorar a solução até não ocorrer mais nenhuma melhoria. Considere o problema do caixeiro-viajante no qual todos os vértices são pontos no plano e o peso de cada aresta é a distância planar entre os pontos. Mesmo com essa restrição, o problema é NP-completo. Na técnica **2-opt**, sempre que duas arestas se cruzam, troque-as, o que resulta em um ciclo mais curto:



Além disso, grande quantidade de **algoritmos de aproximação** dão resultados em que é garantido que estão dentro de certo fator em relação ao valor ótimo. Por exemplo, se a entrada para um problema do caixeiro-viajante obedecer à **desigualdade do triângulo** — para todos os vértices u , v e x , o peso de aresta (u, v) é, no máximo, a soma dos pesos das arestas (u, x) e (x, v) —, há um algoritmo de aproximação simples que sempre determina uma rota para o caixeiro-viajante cujo peso total é no máximo duas vezes a de peso mais baixo, e esse algoritmo executa em tempo linear no tamanho da entrada. Há um algoritmo de aproximação de tempo polinomial ainda melhor para essa situação, que dá uma rota cujo peso total é no máximo $3/2$ vezes a de peso mais baixo.

Uma coisa estranha é que, se dois problemas NP-completos forem intimamente relacionados, a solução produzida por um bom algoritmo de aproximação para um poderia produzir uma solução ruim para o outro. Isto é, uma solução que é quase ótima para um dos problemas não mapeia necessariamente para uma solução que não é nem um pouco próxima para o outro problema.

Não obstante, em muitas situações do mundo real, uma solução quase ótima já é bom demais. Voltando à discussão sobre a empresa que entrega encomendas em caminhões marrons, o pessoal fica feliz se encontrar rotas quase ótimas para seus caminhões marrons, mesmo que elas não sejam as melhores possíveis. Todo dinheiro que elas puderem economizar com o planejamento de rotas eficientes ajuda a linha de resultados da contabilidade.

PROBLEMAS INDECÍSIVEIS

Então, novamente, se você ficou com a impressão de que problemas NP-completos são os mais difíceis no mundo dos algoritmos, terá uma pequena surpresa. Cientistas teóricos da computação definiram ampla hierarquia de classes de complexidade, com base na quantidade de tempo e na quantidade de recursos necessários para resolver um problema. Alguns problemas levam uma quantidade de tempo que é provavelmente exponencial no tamanho da entrada.

E fica ainda pior. Para alguns problemas, nenhum algoritmo é possível. Isto é, há problemas para os quais é provavelmente impossível criar um algoritmo que sempre dê uma resposta correta. Denominamos esses problemas *indecisíveis*, e o mais conhecido é o *problema da parada*, que o matemático Alan Turing provou ser indecisível em 1937. No problema da parada, a entrada é um programa de computador A e a entrada x para A. A meta é determinar se o programa A, executando na entrada x , acabará parando. Isto é, A com entrada x executa até o fim?

Você talvez esteja achando que poderia escrever um programa — vamos denominá-lo programa B — que lê o programa A, lê x e simula A executando com a entrada x . Tudo bem se A na entrada x realmente executar até o fim. E se não executar? Como o programa B saberia quando declarar que A nunca parará? B não poderia verificar A entrando em algum tipo de laço infinito? A resposta é que, embora você possa escrever B para verificar alguns casos nos quais A não para, é provavelmente impossível escrever um programa B de modo que *ele* sempre pare e lhe diga corretamente se A na entrada x para.

Como não é possível escrever um programa que determine se outro programa que está executando em uma entrada particular para, também não é possível escrever um programa que determine se outro programa está de acordo com sua especificação. Como um programa pode dizer se outro programa dá a resposta correta se ele não pode nem mesmo dizer se o programa para? E nada mais temos a dizer sobre os testes de software automatizados!

Para que você não comece a pensar que somente problemas indecisíveis têm a ver com propriedades de programas de computador, o *problema da correspondência de posto (Post's Correspondence Problem — PCP)* é sobre cadeias, como vimos no Capítulo 7. Suponha que temos, no mínimo, dois caracteres e duas listas de n cadeias, A e B sobre esses caracteres. Digamos que A consista nas cadeias $A_1, A_2, A_3, \dots, A_n$ e B consista nas cadeias $B_1, B_2, B_3, \dots, B_n$. O problema é determinar se existe uma sequência de índices $i_1, i_2, i_3, \dots, i_m$ tal que $A_{i_1}A_{i_2}A_{i_3}\dots A_{i_m}$ (isto é, as cadeias $A_{i_1}, A_{i_2}, A_{i_3}, \dots, A_{i_m}$ concatenadas) dá a mesma cadeia de $B_{i_1}B_{i_2}B_{i_3}\dots B_{i_m}$. Por exemplo, suponha que os caracteres sejam e, h, m, n, o, r e y, que $n = 5$, e que

$$\begin{array}{ll}
 A_1 = ey.; & B_1 = ym.; \\
 A_2 = er.; & B_2 = r.; \\
 A_3 = mo.; & B_3 = oon.; \\
 A_4 = on.; & B_4 = e.; \\
 A_5 = h.; & B_5 = hon..
 \end{array}$$

Então, uma solução é a sequência de índices $\langle 5, 4, 1, 3, 4, 2 \rangle$, visto que $A_5A_4A_1A_3A_4A_2$ e $B_5B_4B_1B_3B_4B_2$ formam a palavra *honeymooner*. É claro que, se há uma solução, há um número infinito de soluções, visto que você pode apenas ficar repetindo a sequência de índices de uma solução (o que dá *honeymoonerhoneymooner* etc.). Para PCP ser indecisível, temos de permitir que as cadeias em A e B sejam usadas mais de uma vez, já que, caso contrário, você só poderia dar uma lista de todas as possíveis combinações de cadeias.

Embora o problema da correspondência de posto possa não parecer particularmente interessante por si só, podemos reduzi-lo a outros problemas para mostrar que também eles são indecísiveis. É a mesma ideia básica que usamos para mostrar que um problema é NP-difícil: dada uma instância de PCP, transforme-a em uma instância de algum outro problema Q , tal que a resposta à instância de Q dê a resposta à instância de PCP. Se pudéssemos decidir Q , poderíamos decidir PCP; porém, visto que sabemos que não podemos decidir PCP, então Q deve ser indecísivel.

Entre os problemas indecísiveis aos quais podemos reduzir PCP estão vários que têm a ver com *gramáticas livres de contexto* (*context-free grammars* — *CFGs*), que descrevem a sintaxe da maioria das linguagens de programação. Uma CFG é um conjunto de regras para gerar uma *linguagem formal*, que é um modo extravagante de dizer “um conjunto de cadeias”. Reduzindo a partir de PCP, podemos provar que é indecísivel se duas CFGs geram a mesma linguagem formal, se duas CFGs geram quaisquer cadeias em comum ou se dada CFG é *ambígua*: há dois modos diferentes de gerar a mesma cadeia usando as regras da CFG?

FECHAMENTO

Vimos boa gama de algoritmos em bastante variedade de domínios, não é? Vimos um algoritmo que leva tempo sublinear — árvore de busca binária. Vimos algoritmos que levam tempo linear — busca linear, ordenação por contagem, ordenação digital, ordenação topológica e determinação de caminhos mínimos em um dag. Vimos algoritmos que levam tempo $O(n \lg n)$ — ordenação por intercalação e quicksort (caso médio). Vimos algoritmos que levam tempo $O(n^2)$ — ordenação por seleção, ordenação por inserção e quicksort (pior caso). Vimos algoritmos de grafo que levam tempo descrito por alguma combinação não linear do número n de vértices com o número m de arestas — algoritmo de Dijkstra e algoritmo de Bellman-Ford. Vimos um algoritmo de grafo que leva tempo $\Theta(n^3)$ — o algoritmo de Floyd-Warshall. Agora vimos que, para alguns problemas, não temos nenhuma ideia se um algoritmo de tempo polinomial é até mesmo possível. E ainda vimos que, para alguns problemas, nenhum algoritmo é possível, independentemente do tempo de execução.

Mesmo com essa introdução relativamente breve ao mundo dos algoritmos de computador,⁶ você pode ver que a área cobre muito chão. E este livro cobre apenas a mais diminuta fatia da área. Além disso, eu restringi nossas análises a um modelo computacional particular, no qual somente um processador executa operações e o tempo para executar cada operação é mais ou menos o mesmo, independentemente de onde os dados residem na memória do computador. Muitos modelos computacionais alternativos têm sido propostos com o passar dos anos, como modelos com vários processadores, modelos nos quais o tempo para executar uma operação depende de onde seus dados estão localizados, modelos nos quais os dados chegam em uma corrente não repetível e modelos nos quais o computador é um dispositivo quântico.

⁶ Compare o tamanho deste livro com o CLRS, que chegou a 1.292 páginas em sua terceira edição.

Assim, você pode ver que essa área de algoritmos de computador tem muitas perguntas ainda não respondidas, bem como perguntas que ainda serão feitas. Faça um curso de algoritmos — pode até ser on-line — e nos ajude!

O QUE MAIS LER?

O livro sobre NP-completude é o de Garey e Johnson [GJ79]. Se você estiver interessado em mergulhar nesse tópico, leia-o. CLRS [CLRS09] tem um capítulo sobre NP-completude que se aprofunda em mais detalhes técnicos do que eu abordei aqui e também tem um capítulo sobre algoritmos de aproximação. Se quiser saber mais sobre computabilidade e complexidade, e conhecer uma prova muito interessante, curta e entendível de que o problema da parada é indeciso, recomendo o livro de Sipser [Sip06].

Referências

- [AHU74] Alfred V. Aho, John E. Hopcroft e Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [AMOT90] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin e Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37 (2) : 213–223, 1990.
- [CLR90] Thomas H. Cormen, Charles E. Leiserson e Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Primeira edição, 1990.
- [CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms*. The MIT Press, Terceira edição, 2009.
- [DH76] Whitfield Diffie e Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22 (6) :644–654, 1976.
- [FIP11] Annex C: Approved random number generators for FIPS PUB 140-2, Security requirements for cryptographic modules. <http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexc.pdf>, julho de 2011. Rascunho.
- [GJ79] Michael R. Garey e David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.
- [Gri81] David Gries. *The Science of Programming*. Springer, 1981.
- [KL08] Jonathan Katz e Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.
- [Knu97] Donald E. Knuth. *The Art of Computer Programming*, Volume 1: Fundamental Algorithms. Addison-Wesley, Terceira edição, 1997.
- [Knu98a] Donald E. Knuth. *The Art of Computer Programming*, Volume 2: Seminumerical Algorithms. Addison-Wesley, Terceira edição, 1998.
- [Knu98b] Donald E. Knuth. *The Art of Computer Programming*, Volume 3: Sorting and Searching. Addison-Wesley, Segunda edição, 1998.
- [Knu11] Donald E. Knuth. *The Art of Computer Programming*, Volume 4A: Combinatorial Algorithms, Part I. Addison-Wesley, 2011.
- [Mac12] John MacCormick. *Nine Algorithms That Changed the Future: The Ingenious Ideas That Drive Today's Computers*. Princeton University Press, 2012.
- [Mit96] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.
- [MvOV96] Alfred Menezes, Paul van Oorschot e Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [RSA78] 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. Veja também U.S. Patent 4,405,829.
- [Sal08] David Salomon. *A Concise Introduction to Data Compression*. Springer, 2008.
- [Sip06] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Segunda edição, 2006.
- [SM08] Sean Smith e John Marchesini. *The Craft of System Security*. Addison-Wesley, 2008.
- [Sto88] James A. Storer. *Data Compression: Methods and Theory*. Computador Science Press, 1988.
- [TC11] Greg Taylor e George Cox. Digital randomness. *IEEE Spectrum*, 48 (9) : 32–58, 2011.

Índice remissivo

notação O , 16–17
notação Ω , 17
notação Θ , 15–16
! (fatorial), 19, 121, 154
(piso) $\lfloor \rfloor$, 24

A

abstração, 85
achar números primos grandes, 128–129

ADT, 85

Advanced Encryption Standard (AES), 123, 132

algoritmo

aproximação, 3, 178

correção de, 2–3

definição, 1–2

empo polinomial, 155

guloso, 141

origem da palavra, 7

para um computador, 1

redução de tempo polinomial, 159

uso de recurso de, 3–5

algoritmo de aproximação, 3, 178

algoritmo de Bellman-Ford, 88–92

para encontrar um ciclo de peso negativo, 90–91

para encontrar uma oportunidade de arbitragem, 91–92

tempo de execução de, 90–91

algoritmo de computador, 1

algoritmo de correspondência de cadeia

de Knuth-Morris-Pratt, 118

algoritmo de correspondência de cadeia KMP, 118

algoritmo de correspondência de cadeias de Boyer-Moore, 118

algoritmo de Dijkstra, 80–88, 141

invariante de laço para, 84

tempo de execução de, 84–88

algoritmo de Floyd-Warshall, 92–98

e programação dinâmica, 93–95, 97

tempo de execução de, 96

algoritmo de Johnson, 98

algoritmo de ordenação híbrido, 50

algoritmo de tempo polinomial, 155

redução, 159

algoritmo guloso, 141

alinhamento de sequências de DNA, 107–108

altura de um nó, 87

AND, 161

aresta

de entrada, 65

de saída, 65

dirigida, 64

incidente, 155

não dirigida, 155

relaxamento de, 75

aresta de entrada, 65

aresta de saída, 65

aresta dirigida, 64

relaxamento de, 75

aresta incidente, 155

aresta não dirigida, 155

aritmética com números grandes, 128

aritmética modular, 125

computar inversos multiplicativos em,

130–131

exponenciação, 131–132

aritmética, modular, 125

arranjo, 10

busca em, 11–19, 20, 23–27

fila de prioridade implementada por,

85–86

ordenação de, 27–50, 53–59

árvore

binária, 86, 137

trie, 150

árvore binária, 86, 137

ASCII, 118, 121, 136

ASSEMBLE-LCS, 105

ASSEMBLE-TRANSFORMATION, 112

autômato finito, 113–118

tempo de execução de, 117–118

B

BAD-FACTORIAL, 20

beisebol, 125

BELLMAN-FORD, 89

BETTER-LINEAR-SEARCH, 13

BINARY-SEARCH, 25

bit, 121

bits aleatórios, 4

bloquinho, 123

BUILD-HUFFMAN-TREE, 140

busca, 11–19, 20

busca binária, 23–27

busca linear, 12–19, 20

busca binária, 23–27

invariante de laço para, 25

tempo de execução de, 26–27

versão recursiva, 26

busca linear, 12–19, 20

invariante de laço para, 18

limite inferior para, 17

tempo de execução de, 14–18

versão recursiva, 20

busca na vizinhança, 178

comparação de algoritmos, 48

C

cache, 14

cadeia, 99

de texto, 112

ocorrência de, com um deslocamento, 112

padrão, 112

subsequência comum mais longa, 99–106

transformação de, 106–110

cadeia de texto, 112

cadeia-padrão, 112

caminho, 70

acíclico mais longo, 170

hamiltoniano, 167–169

mínimo, 73

peso de, 72

caminho crítico, 69–74

tempo de execução para determinar, 76

caminho hamiltoniano, 167–169

caminho mínimo, 73

algoritmo de Bellman-Ford para, 88–92

algoritmo de Dijkstra para, 80–88

algoritmo de Floyd-Warshall para, 92–98

algoritmo de Johnson para, 98

em um grafo acíclico dirigido, 74–76

fonte única, 74

por multiplicação de matriz, 98

um par, 79

caminho mínimo de fonte única, 74

algoritmo de Bellman-Ford para, 88–92

algoritmo de Dijkstra para, 80–88

em um grafo acíclico dirigido, 74–76

caminho mínimo para um par, 79

caminhos mínimos para todos os pares

algoritmo de Floyd-Warshall para, 92–98

algoritmo de Johnson para, 98

por multiplicação de matriz, 98

caso-base, 19

certificado, 157

CFG, 180

chamada de um procedimento, 10

chave

em busca, 22

em criptografia, 120

em ordenação, 22

pública, 124, 128

segreta, 124, 128

chave única, 121–123

ciclo, 71

hamiltoniano, 155, 167–169

ciclo de peso negativo

e oportunidades de arbitragem determinadas pelo algoritmo de,

91–92

encontradas pelo algoritmo de Bellman-Ford, 90–91

ciclo hamiltoniano, 155, 167–169

cifra

- cifra de deslocamento, 120
- substituição simples, 120–121

cifra de bloco, 123–124

- encadeamento, 123–124

cifra de deslocamento, 120

cifra de substituição

- cifra de deslocamento, 120
- simples, 120–121

cifra de substituição simples, 120–121

- cifra de deslocamento, 120

cláusula, 162

clique, 163

CLRS, 7, 20, 50, 59, 77, 98, 118, 134, 151, 181

codificação por comprimento da rodada, 141

código

- escape, 141

- Huffman, 136–141, 150

- livre de prefixo, 137

código de escape, 141

código de Huffman, 136–141, 150

- adaptativo, 141

- tempo de execução de, 140–141

código livre de prefixo, 137

coeficiente, 5, 15

colocar em cesta (binning), 23

compressão, 135–151

- com perdas, 135

- pelo código de Huffman adaptativo, 141 pelo código de Huffman,

- 136–141, 150

- por codificação do comprimento da carreira, 141

- por LZ78, 143

- por LZW, 142–151

- por máquinas de fax, 136, 141–142

- sem perdas, 135

compressão com perdas, 135

compressão de dados, *veja* compressão

compressão LZW, 142–151

compressão sem perdas, 135

COMPUTE-LCS-TABLE, 104

COMPUTE-TRANSFORM-TABLES, 110

comunicação, 4

concatenação, 116

contrapositivo, 19

corpo de um laço, 12, 14

corrção de um algoritmo, 2–3

correspondência de cadeia, 99, 112–118

- algoritmo de Boyer-Moore para, 118

- algoritmo KMP para, 118

- pelo método ingênuo, 118–119

- por autômatos finitos, 113–118

COUNT-KEYS-EQUAL, 54

COUNT-KEYS-LESS, 54

COUNTING-SORT, 57

criptografia, 1, 3, 119–134

- chave pública, 124–132

- chave simétrica, 121–124

- cifra de bloco, 123–124

- cifra de deslocamento, 120

- cifra de substituição simples, 120–121

criptossistema híbrido, 132–133

- de cifra única, 121–123

- encadeamento de cifra de bloco, 123–124

- por RSA, 125–132

criptografia de chave pública, 124–132

- por RSA, 125–132, 134

criptografia de chave simétrica, 121–124

- cifra de bloco, 123–124

- de cifra única, 121–123

- encadeamento de cifra de bloco, 123–124

criptossistema de chave pública, 125

criptossistema híbrido, 132–133

curto-circuito, 30

D

dados satélites, 22–23

DAG-SHORTEST-PATHS, 75

dag, *veja* grafo acíclico dirigido

decifração, 119

DECREASE-KEY, 85, 87, 88

decremento, 30

designar um valor a uma variável, 12

desigualdade em triângulo, 178

deslocamento de uma cadeia, 112

diagrama PERT, 69–74, 76

diâmetro de uma rede, 92

DIJKSTRA, 82, 86

dividir e conquistar, 33–34

DNA, 99, 136

alinhamento de sequências de, 107–108

2-opt, 178

E

elemento de um arranjo, 10

elevação ao quadrado repetida, 131

encadeamento de cifra de bloco, 123–124

entrada

- para um procedimento, 10

- tamanho de, 5

entrada de um arranjo, 10

equação de recorrência, 41

equipamento de goleiro, 61–64

erro de estouro de pilha, 20

espaço (slot), 23

estado, 113

estrutura de dados, 85

etapa de combinação, 33

- em ordenação por intercalação, 34

- em quicksort, 42

etapa de conquistar, 33

- em ordenação por intercalação, 34

- em quicksort, 42

etapa de dividir, 33

- em ordenação por intercalação, 34

- em quicksort, 41–42

etapa de relaxação, 75

EÚCLID, 129–130

exclusive-or, 121

exponenciação, modular, 131–132

EXTRACT-MIN, 85, 87–88

F

FA, *veja* autômato finito

FA-STRING-MATCHER, 115

FACTORIAL, 19

fila de prioridade, 85

- implementação de arranjo simples de, 85–86

- implementação de heap binário de, 86–87

- implementação de heap de Fibonacci de, 88

filho, 86

forma normal 3-conjuntiva, 163

fórmula booleana, 161

- problema da satisfazibilidade para, 161

- fórmula satisfazível, 161

função, 10

- linear, 15

função exponencial, 66

função fatorial (!), 19, 121, 154

função linear, 15

função piso ($\lfloor \cdot \rfloor$), 24

G

gerador de número pseudoaleatório, 133–134

googol, 156

grafo

- acíclico dirigido, 64–69

- caminho mínimo em, 74–76

- completo, 169

- conectado, 155

- dirigido, 64

- não dirigido, 155

- ordenação topológica de, 65–69

- ponderado, 72

- representação de, 67–69

grafo acíclico dirigido, 64–69

- caminho mínimo em, 74–76

- ordenação topológica de, 65–69

grafo completo, 169

grafo conectado, 155

grafo denso, 87

grafo dirigido ponderado, 72

grafo dirigido, 64

- acíclico, 64–69

- caminho mínimo em, 74–76

- ordenação topológica de, 65–69

- ponderado, 72

- representação de, 67–69

grafo esparsa, 87

grafo não dirigido, 155

gramática ambígua, 180

gramática livre de contexto, 180

gramática, 180



ELSEVIER

grau de entrada, 66
grau de saída, 66
grau de um vértice, 155

H
heap binário, 86–88
heap F, 88
heap redistributivo, 98
heap, 86–88
 redistributivo, 98
HEAPSORT, 87
heapsort, 87–88

I
IFF, 161
implementação de arranjo simples de uma fila de prioridade, 85–86
implementação de heap de Fibonacci de uma fila de prioridade, 88
IMPLIES, 161
incremento, 12
índice para um arranjo, 10
inicialização de uma invariante de laço, 18
INSERT, 84–85, 87, 88
INSERTION-SORT, 30
intercalação, 35–40
intercalação em tempo linear, 35–40
Introduction to Algorithms, veja CLRS
invariante de laço, 18–19
 para busca binária, 25
 para busca linear, 18
 para COUNT-KEYS-LESS, 54
 para o algoritmo Dijkstra, 84
 para ordenação por inserção, 30
 para ordenação por seleção, 28
 para particionamento, 45
inverso multiplicativo, 128, 130–131
inverso, multiplicativo, 128, 130–131
iteração de um laço, 12

J
Jabberwocky, 143
jogo dos seis graus de Kevin Bacon, 79–80
JPEG, 135

L
laço, 12, 14
 aninhado, 28
laços aninhados, 28
LCS, veja subsequência comum mais longa
lg, 66
LIFO, 67
limite, 15–17
limite inferior, 15, 17
 existencial, 53
 para busca linear, 17
 para ordenação por comparação, 52–53
 para ordenação por seleção, 29
 universal, 53
limite superior, 15–16
LINEAR-SEARCH, 12
linguagem formal, 180
lista
 aresta, 67
 de adjacência, 68
 ligada, 68
 literal, 163
lista duplamente ligada, 69
lista simplesmente ligada, 69
Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch,
 143
logaritmo, 66
logaritmo base, 2, 66
L^Z78, 143
LZW-COMPRESSOR, 146
LZW-DECOMPRESSOR, 149

M
manutenção de uma invariante de laço, 18
máquinas de fax, 136, 141–142
massacinzentaware, 9
matriz de adjacência, 67
melhor caso, 16
memória, 4
memória principal, 14
memória virtual, 14
MERGE, 40

MERGE-SORT, 34
método, 10
método ingênuo para concordância de cadeias, 112–113
 tempo de execução de, 113
método mestre, 41
Moby Dick, 141, 144, 147–148, 150
MODULAR-EXPONENTIATION, 131
MP3, 135
multiplicação de matriz, 98

N
nó, 86
 interno, 138
NOR, 161
notação “ó maiúsculo” (oh notation), 16–17
notação assintótica, 17
notação O, 16–17
notação omega, 17
notação “ómega maiúsculo” (big-omega notation), 17
notação teta, 15–16
NP, 157
NP-completo, 155, 157
NP-completude
 da cobertura de vértices, 166–167
 da mochila, 174
 da partição, 172–174
 da satisfazibilidade 3-CNF, 162–163
 das somas de subconjunto, 170–172
 de clique, 163–166
 do caixeiro-viajante, 169–170
 do caminho acíclico mais longo, 170
 do caminho hamiltoniano, 167–169
 do ciclo hamiltoniano, 167–169
 estratégias gerais para provar, 174–177
 perspectiva sobre, 177–180
NP-difícil, 157
número
 aleatório, 133
 composto, 127
Erdős, 80
Erdős-Bacon, 80
Kevin Bacon, 79–80
primo, 125
número aleatório, 133
número composto, 3, 127
número de Erdős, 80
número de Erdős, 80
número de Kevin Bacon, 79–80
número primo, 3, 125
 como encontrar, 128–129

O
ocorrência de uma cadeia-padrão com
 um deslocamento, 112
operação, 14
operação copiar para um caractere, 106
operação eliminar para um caractere, 106
operação inserir para um caractere, 106
operação substituir para um caractere, 106
operações de disco, 4
oportunidade de arbitragem, 91–92
ÓR, 161
ordem de crescimento de tempo de execução, 4
ordenação, 27–59
 algoritmo híbrido, 50
 aplicações de, 22
 comparação de algoritmos, 48–50
 estável, 58
 heapsort, 87–88
 limite inferior para, 52–53
 no lugar, 33
 ordenação digital, 58–59
 ordenação por contagem, 53–58
 ordenação por inserção, 29–33
 ordenação por intercalação, 33–41
 ordenação por seleção, 27–29
 quicksort, 41–48
 topológica, 65–69
ordenação digital, 58–59
 tempo de execução de, 59
ordenação estável, 58
ordenação lexicográfica, 21
ordenação no lugar, 33
ordenação por comparação, 52–53
ordenação por contagem, 53–58
 tempo de execução de, 57–58
ordenação por inserção, 29–33
 invariante de laço para, 30
 tempo de execução de, 32

ordenação por intercalação, 33–41
 tempo de execução de, 40–41
 ordenação por seleção, 27–29
 invariante de laço para, 28
 tempo de execução de, 28–29
 ordenação topológica, 65–69
 tempo de execução de, 69

P

P, 156
 pai, 87
 parâmetro, 10
 particionamento, 42, 44–46
 tempo de execução de, 46
PARTITION, 46
 passeio de Euler, 155
PCP, 179
 pegada, 4
 pel, 136
 pequeno teorema de Fermat, 129
 permutação, 27
 peso
 de um caminho, 72
 de uma aresta, 72
 pilha, 67
 pior caso, 16
 pivô, 42
 pixel, 136
 pôr no balde (bucketizing), 23
 predecessor, 74
 prefixo, 100
 primo, relativamente, 127, 129–130
PRNG, 133–134
 problema da clique, 163–166
 problema da cobertura de vértices, 166–167
 problema da correspondência do correio, 179
 problema da mochila, 174
 problema da parada, 179
 problema da partição, 172–174
 problema da satisfazibilidade 3-CNF, 162–163
 problema da soma de subconjunto, 170–172
 problema de decisão, 158
 problema do caixeteiro-viajante, 155, 169–170
 problema do caminho acíclico mais longo, 170
 problema indecidível, 178
Problema Mae, 161–162
 procedimento, 10
 chamada de, 10
 profundidade de uma folha, 138
 programação dinâmica, 96–98
 no algoritmo de Floyd-Warshall, 93–95, 97
 para a subsequência comum mais longa, 100–105
 para transformar cadeias, 108–110
 propriedade de heap, 86
 pseudocódigo, 9

Q

quicksort, 41–48
 aleatorizado, 47
 determinístico, 49
 mediana de três, 47
 tempo de execução de, 46–48
QUICKSORT, 43

R

ramificar e podar, 177
REALLY-SIMPLE-SORT, 52
REARRANGE, 55
 recursão, 19–20
RECURSIVE-BINARY-SEARCH, 26
RECURSIVE-LINEAR-SEARCH, 20
 redução, 158–159
 relativamente primo, 127, 129–130
RELAX, 75
 representação de um grafo dirigido, 67–69
 resultado (saída) de um procedimento, 10
 retornar valor de um procedimento, 10
RSA, 3, 125–132, 134

S

SELECTION-SORT, 27
 semente, 133
SENTINEL-LINEAR-SEARCH, 14
 sentinel, 13
 em intercalação, 39–40
 sequênciaria, 99
 série aritmética, 29
 sinais em beisebol, 125
 solução ótima, 3
 solução quase ótima, 3
 solução, 2–3

SQUARE-ROOT, 10
 subarranjo, 18
 subcadeia, 100
 subestrutura ótima, 97
 subsequência, 100
 subsequência comum, 100
 mas longa, 99–106
 subsequência comum mais longa, 99–106
 por programação dinâmica, 100–105
 tempo de execução de, 104, 106
 sufixo, 116

T

tabela hash (tabela de espalhamento), 150
 tamanho
 da entrada, 4
 de um clique, 163
 de uma cobertura de vértices, 166
 tempo de execução
 como caracterizar, 14–18
 da subsequência comum mais longa, 104, 106
 de busca binária, 26–27
 de busca linear, 14–18
 de caminhos mínimos de fonte única em um grafo acíclico
 dirigido, 76
 de correspondência de cadeias por autômato finito, 117–118
 de intercalação, 40
 de ordenação digital, 59
 de ordenação por contagem, 57–58
 de ordenação por inserção, 32
 de ordenação por intercalação, 40–41
 de ordenação por seleção, 28–29
 de ordenação topológica, 69
 de particionamento, 46
 de quicksort, 46–48
 de transformação de cadeias, 110
 do algoritmo de Bellman-Ford, 90–91
 do algoritmo de Dijkstra, 84–88
 do algoritmo de Floyd-Warshall, 96
 do código de Huffman, 140–141
 do método ingênuo de correspondência de cadeias, 113
 melhor caso, 16
 ordem de crescimento de, 4
 pior caso, 16
 tempo, 4
 teorema do número primo, 128
 terminação de uma invariante de laço, 18
 termo, 4
 termo de baixa ordem, 4, 15
 teste de primalidade
 AKS, 127
 Miller-Rabin, 127, 134
 texto cifrado, 120
 texto comum, 120
 tipo de dado abstrato, 85
TOPOLOGICAL-SORT, 66
 transformação de cadeias, 106–110
 por programação dinâmica, 108–110
 tempo de execução de, 110
 transitiva, 61
 3-CNF, 163
 trie, 150

U

último a entrar, primeiro a sair, 67
Unicode, 121
 uso de recurso, 3–5

V

variável
 de laço, 12
 em um procedimento, 12
 verificação de tempo polinomial, 156
 vértice, 64
 alvo, 74
 fonte, 74
 grau de entrada, 66
 grau de saída de, 66
 predecessor, 74
 vértice adjacente, 65
 vetor de inicialização, 124

W

widget, 168

X

XOR, 121