CORY ALTHOFF

CIENTISTA DA COMPUTAÇÃO *Autodidata*

Guia de estruturas de dados e algoritmos para o iniciante

novatec

WILEY

Cientista da Computação Autodidata

Guia de estruturas de dados e algoritmos para o iniciante

Cory Althoff

Novatec

Editor: Rubens Prates GRA20221011
Tradução: Aldir Coelho Corrêa da Silva
Revisão da tradução: Joel Saade
Revisão gramatical: Patrizia Zagni

ISBN impresso: 978-85-7522-837-1

ISBN ebook: 978-85-7522-838-8

Histórico de impressões:

Outubro/2022 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: novatec@novatec.com.br
Site: https://novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec
GRA20221011

Dedico este livro à minha esposa, Bobbi, e à minha filha, Luca.
Amo muito vocês!

Sumário

Capítulo 18 Próximos passos

Sobre o autor

Cory Althoff é um programador, palestrante e autor, cujo trabalho inclui os livros *Programador Autodidata* e *Cientista da Computação Autodidata*. Depois de se formar em ciência política, Cory aprendeu sozinho a programar, tornando-se um engenheiro de software no eBay. Os livros de Cory foram traduzidos para vários idiomas e ele foi destaque em publicações, como a Forbes e a CNBC. Mais de 250 mil desenvolvedores fazem parte da comunidade de programadores autodidatas que ele criou por meio de seu popular grupo no Facebook, blog, curso e newsletter. Cory mora na Califórnia com sua esposa e filha.

Sobre o editor técnico

Dr. Hannu Parviainen é um astrofísico que procura planetas fora do sistema solar, no Instituto de Astrofísica das Canárias, um dos principais institutos de astrofísica mundiais e sede do maior telescópio óptico existente atualmente. Antes, trabalhou durante vários anos como pesquisador pós-doutoral na Universidade de Oxford. Seus interesses básicos são computação científica e métodos numéricos modernos e tem mais de 20 anos de experiência no uso da linguagem de programação Python.

Agradecimentos

Gostaria de agradecer imensamente a todas as maravilhosas pessoas que tornaram este livro possível. Em primeiro lugar, à minha esposa, Bobbi, o amor da minha vida que sempre me apoiou. Ao meu pai, James Althoff, que cedeu grande parte do seu tempo me ajudando neste livro. O próximo agradecimento vai para Steve Bush: muito obrigado por ler meu livro e me dar feedback. Também gostaria de agradecer ao editor do projeto, Robyn Alvarez, e ao editor técnico, Hannu Parviainen. Para concluir, quero agradecer ao meu editor, Devon Lewis, por me ajudar a concluir este projeto e por ser incrivelmente flexível. Também preciso agradecer à minha filha, Luca, por ser a melhor filha do mundo e me inspirar a trabalhar com o maior afinco possível. Amo você, Luca! Não teria conseguido sem o enorme apoio que você me deu. Muito obrigado!

Introdução

Minha jornada para aprender a codificar começou quando me formei na faculdade, em Ciência Política. Após terminar os estudos, tentei conseguir um emprego. Não tinha as habilidades que os empregadores estavam procurando e, nesse meio-tempo, vi amigos que estudaram matérias mais práticas conseguirem empregos com altos salários. Enquanto isso ocorria, tentei ser contratado, sem conseguir, não tinha renda e me sentia um fracasso. Portanto, por viver no Vale do Silício e estar rodeado por programadores, decidi aprender a programar. Mal sabia que ia começar a jornada mais insana e gratificante da minha vida.

Esse esforço não foi minha primeira tentativa de aprender a codificar: já tinha tentado no passado sem sucesso. Durante meu primeiro ano na faculdade, tive uma aula de programação, achei difícil de entender e a abandonei rapidamente. Infelizmente, a maioria das escolas ensina Java como primeira linguagem de programação, que é uma linguagem difícil para iniciantes. Em vez de Java, decidi aprender Python sozinho, uma das linguagens mais fáceis para iniciantes. Apesar de aprender uma linguagem fácil de entender, mesmo assim quase desisti. Tive que coletar informações de várias fontes diferentes, o que foi frustrante. Também não ajudou o fato de me sentir isolado em minha jornada. Não tinha uma sala de aula repleta de alunos com os quais pudesse estudar e obter ajuda.

Estava quase desistindo quando comecei a passar mais tempo em comunidades de programação online como a Stack Overflow. Ingressar em uma comunidade me manteve motivado e me empolgou novamente. Houve muitos altos e baixos e, em certos momentos, pensei em desistir. Contudo, menos de um ano após ter tomado a fatídica decisão de aprender a programar, estava trabalhando como engenheiro de software no eBay. Um ano antes, teria sorte se conseguisse um emprego de atendimento ao cliente. Agora, estava recebendo 50 dólares por hora para programar para uma conhecida empresa de tecnologia. Não podia

acreditar! No entanto, o melhor não era o dinheiro. Ao tornar-me engenheiro de software, fiquei muito mais confiante. Após aprender a codificar, sentia-me como se pudesse fazer qualquer coisa.

Depois do eBay, comecei a trabalhar em uma startup em Palo Alto. Decidi, então, tirar férias e fazer uma viagem como mochileiro para o sudeste da Ásia. Estava no assento traseiro de um táxi passando pelas estreitas ruas de Seminyak, em Bali, na chuva, quando tive uma ideia. Em casa, as pessoas sempre me faziam perguntas relacionadas à minha experiência como engenheiro de software. Trabalhar como engenheiro de software no Vale do Silício não é incomum, mas era diferente de muitos de meus pares porque não tinha um diploma em Ciência da Computação.

Minha ideia era escrever um livro chamado *Programador Autodidata* (*The Self-Taught Programmer*): não só sobre programação, mas sobre tudo que aprendi para ser contratado como engenheiro de software. Em outras palavras, queria ajudar as pessoas a fazerem a mesma jornada que fiz. Logo, comecei a criar um roteiro para aspirantes a programadores autodidatas. Passei um ano escrevendo *Programador Autodidata* e publiquei-o por conta própria. Não tinha certeza se alguém o leria, provavelmente ninguém o faria, mas queria compartilhar minha experiência mesmo assim. Para minha surpresa, vendeu milhares de cópias nos primeiros meses. Com essas vendas, chegaram mensagens de pessoas do mundo todo que eram ou queriam se tornar programadores autodidatas.

Essas mensagens me inspiraram. Decidi resolver outro problema que encontrei ao aprender a programar: sentir-me sozinho na jornada. A solução foi criar um grupo no Facebook chamado Self-Taught Programmers, um local para os programadores ajudarem uns aos outros. Agora, tem mais de 60 mil membros e evoluiu para uma comunidade de suporte repleta de programadores autodidatas que ajudam uns aos outros a responder a perguntas, trocar informações e compartilhar histórias de sucesso. Se quiser fazer parte de nossa comunidade, ingressar nela acessando `https://facebook.com/groups/selftaughtprogrammers`. Você também pode se inscrever em minha newsletter, em `theselftaughtprogrammer.io`.

Quando publicava material online sobre trabalhar como engenheiro de software sem um diploma em Ciência da Computação, recebia alguns comentários negativos mencionando que é impossível trabalhar como programador sem um diploma. Algumas pessoas se irritavam e diziam: "O que vocês, programadores autodidatas, acham que estão fazendo? Vocês precisam de um diploma! Nenhuma empresa vai levá-los a sério!". Atualmente, os comentários são poucos e dispersos. Quando chegam, direciono quem os fez para o grupo Self-Taught Programmers. Temos programadores autodidatas trabalhando em empresas do mundo todo, em várias funções, de engenheiro de software júnior a engenheiro-chefe de software.

Enquanto isso, meu livro continuou a vender mais do que poderia imaginar e tornou-se até mesmo um popular curso da Udemy. Interagir com tantas pessoas maravilhosas que estão aprendendo a programar tem sido uma experiência incrível que me tornou mais humilde. Estou feliz por continuar a jornada com este livro, visto que dá continuidade ao meu primeiro livro, *Programador Autodidata*. Logo, se ainda não o leu, deve lê-lo antes, a não ser que já conheça os aspectos básicos da programação. Este livro presume que você sabe programar em Python, portanto, caso não saiba, leia meu primeiro livro, faça o curso da Udemy ou aprenda Python usando o recurso que achar melhor.

O que você aprenderá

Enquanto meu primeiro livro, *Programador Autodidata*, introduz a programação e as habilidades necessárias para aprender a programar profissionalmente, este livro é uma introdução à Ciência da Computação. Especificamente, é uma introdução às estruturas de dados e algoritmos. A Ciência da Computação é o estudo dos computadores e de como estes funcionam. Quando vamos à faculdade para nos tornar engenheiros de software, não nos formamos em programação, mas sim em Ciência da Computação. Os alunos estudam matemática, arquitetura dos computadores, compiladores, sistemas operacionais, estruturas de dados e algoritmos, programação de redes e muito mais.

Cada um desses tópicos poderia ser assunto de livros muito extensos e abordar todos eles não faz parte do escopo deste livro. A Ciência da Computação é uma área muito ampla. Você pode estudá-la a vida toda e ainda ter muito mais a aprender. Este livro não tem o objetivo de abordar tudo que você aprenderia se fosse à faculdade para obter um diploma de Ciência da Computação. Em vez disso, quero fornecer uma introdução a alguns dos conceitos essenciais da Ciência da Computação para você se sair bem em diferentes situações como programador autodidata.

Ao se tornar um programador autodidata, os dois assuntos mais importantes que precisará conhecer são estruturas de dados e algoritmos. Por isso, resolvi dedicar este livro a esses dois tópicos. Dividi o livro em duas partes. A Parte I é uma introdução aos algoritmos. Você aprenderá o que é um algoritmo, por que um algoritmo é melhor do que outro e conhecerá diferentes algoritmos, como os de busca linear e binária. A Parte II é uma introdução às estruturas de dados. Você aprenderá o que é uma estrutura de dados e estudará arrays, listas encadeadas, pilhas, tabelas hash, árvores binárias, heaps binários e grafos. Em seguida, concluo abordando o que você deve fazer após terminar este livro, incluindo as etapas seguintes que poderá executar e outros recursos que o ajudarão em sua jornada para aprender a programar.

Em meu livro anterior, expliquei que não faz sentido estudar Ciência da Computação antes de aprender a programar. No entanto, isso não significa que você pode ignorá-la. Será preciso estudar Ciência da Computação se quiser tornar-se um programador bem-sucedido. É simples: se não souber Ciência da Computação, não será contratado. Quase todas as empresas que empregam programadores os fazem passar por uma entrevista técnica como parte do processo de seleção. Todas as entrevistas técnicas enfocam o mesmo assunto: Ciência da Computação. Especificamente, concentram-se em estruturas de dados e algoritmos. Para ser contratado pelo Facebook, Google, Airbnb e todas as empresas mais visadas atualmente, tanto grandes quanto pequenas, é preciso passar por uma entrevista técnica focada em estruturas de dados e algoritmos. Se você não tem um bom conhecimento desses assuntos, não passará nas entrevistas técnicas. Em uma entrevista técnica não é possível improvisar.

Seu possível empregador lhe fará perguntas detalhadas sobre estruturas de dados, algoritmos e outros assuntos e você precisará saber as respostas se quiser ser contratado.

Além disso, após ser contratado para seu primeiro emprego, seu empregador e colaboradores esperarão que você conheça os aspectos básicos da Ciência da Computação. Se precisarem explicar para você por que um algoritmo O(*n***3) não é uma boa solução, não ficarão satisfeitos. Essa era a situação em que me encontrava quando consegui meu primeiro emprego de programação no eBay. Fazia parte de uma equipe com programadores muito talentosos de Stanford, Berkeley e do Cal Tech. Todos tinham um profundo conhecimento de Ciência da Computação e senti-me inseguro e deslocado. Como programador autodidata, estudar Ciência da Computação o ajudará a evitar essa situação.

Estudar estruturas de dados e algoritmos também o tornará um programador melhor. Os loops de feedback são a chave para quem deseja dominar uma habilidade. Um loop de feedback ocorre quando praticamos uma habilidade e obtemos feedback imediato sobre se fizemos um bom trabalho. Quando praticamos programação, não há loop de feedback. Por exemplo, se você criar um site, este poderá funcionar, mas com um código horrível. Não há loop de feedback para informar se o código é bom ou não. No entanto, quando estudamos algoritmos, não é isso que ocorre. Existem muitos algoritmos famosos na Ciência da Computação, o que significa que você pode escrever um código para resolver um problema, comparar seu resultado com o do algoritmo existente e saber instantaneamente se escreveu uma solução decente. Praticar com um loop de feedback positivo como esse melhorará suas habilidades de codificação.

O maior erro que cometi como programador autodidata iniciante, ao tentar entrar na indústria de software, foi não dedicar tempo suficiente estudando estruturas de dados e algoritmos. Se tivesse passado mais tempo estudando-os, minha jornada teria sido muito mais gerenciável. Você não precisa cometer esse erro!

Como mencionei, a Ciência da Computação é uma área muito ampla. Há

uma razão para os alunos passarem quatro anos estudando: há muito a aprender. No entanto, você pode não ter quatro anos para estudar Ciência da Computação. Felizmente, não é preciso.

Este livro aborda vários itens importantes que você precisa conhecer para ter uma carreira bem-sucedida como engenheiro de software. Sua leitura não substituirá o diploma de Ciência da Computação obtido após quatro anos. Contudo, se você o ler e praticar os exemplos, terá uma base sólida para passar em uma entrevista técnica. Começará a se sentir confortável em uma equipe de graduados em Ciência da Computação e também melhorará significativamente como programador.

Para quem é este livro?

Digamos que eu tenha convencido você de que os programadores autodidatas podem programar profissionalmente e que é preciso estudar Ciência da Computação, principalmente estruturas de dados e algoritmos. Isso significa que você não pode ler este livro a não ser que esteja aprendendo a programar fora da escola? Claro que não! Todos são bem-vindos na comunidade autodidata! Meu primeiro livro foi surpreendentemente popular entre alunos de universidades. Alguns professores do ensino superior chegaram a entrar em contato comigo e me dizer que estavam dando aulas de programação usando o meu livro.

Alunos universitários que estudam Ciência da Computação costumam me perguntar se devem desistir da faculdade. Meu objetivo é inspirar o máximo de pessoas a aprender a programar. Isso significa deixar que saibam que é possível programar profissionalmente, sem o diploma em Ciência da Computação. Se você estiver na faculdade estudando Ciência da Computação, isso também funcionará e não, não é preciso desistir. Permaneça na escola, jovem! Mesmo se estiver na escola, poderá fazer parte da comunidade autodidata aplicando nossa mentalidade de "sempre aprender" as suas tarefas escolares e fazendo de tudo para aprender ainda mais do que seus professores ensinam.

Como saber se você está pronto para estudar Ciência da Computação? É fácil. Se você sabe programar, está pronto! Escrevi este livro para qualquer

pessoa que queira aprender mais sobre Ciência da Computação. Independentemente de estar lendo-o para adquirir o conhecimento que lhe falta, preparar-se para uma entrevista técnica, sentir-se informado no seu trabalho ou tornar-se um programador melhor, eu o escrevi para você.

Histórias de sucesso de autodidatas

Fui contratado como engenheiro de software sem diploma e todo dia escuto histórias de sucesso de programadores autodidatas. Como programador autodidata, certamente você pode ter uma carreira bem-sucedida como engenheiro de software sem diploma. Sei que esse é um assunto delicado para algumas pessoas, logo, antes de passarmos para a Ciência da Computação, quero compartilhar algumas histórias de sucesso de programadores autodidatas de meu grupo do Facebook.

Matt Munson

O primeiro será Matt Munson, membro do grupo Self-Taught Programmers do Facebook. Segue sua história segundo suas próprias palavras:

> Tudo começou quando perdi meu emprego na Fintech. Para pagar as contas, passei a me dedicar a trabalhos incomuns: corte de lentes para óculos, reparo e regulagem de carros, assistente em bailes de carnaval e executando pequenos projetos paralelos de programação. Mesmo fazendo tudo o que podia, após alguns meses, perdi meu apartamento. Essa é a história de como evitei me tornar um morador de rua tornando-me um programador.
>
> Quando perdi o emprego, estava matriculado na universidade. Após perder o apartamento, continuei fazendo minhas tarefas escolares em meu carro e em uma barraca durante alguns meses. Minha família não podia ajudar. Ela não entendia que empregos de baixa renda não são suficientes para alimentar uma pessoa, manter o tanque cheio e, ao mesmo tempo, fornecer habitação. No entanto, não estava disposto a pedir ajuda a amigos. Em setembro, vendi meu carro, retirei a quantia que tinha em um fundo de pensão e viajei quase 3 mil quilômetros de

minha cidade natal em Helena, Montana, para me arriscar em Austin, Texas.

Em uma semana, tive duas ou três entrevistas, mas nenhuma empresa queria dar oportunidade a um sem-teto, qualificado ou não. Após alguns meses nessa rotina, amigos e estranhos começaram a fazer doações para a minha campanha GoFundMe para tentar me ajudar a me reerguer. A essa altura, comia uma vez ao dia e raramente era algo bom. A única solução para sair dessa situação seria tornar-me programador.

Finalmente, decidi fazer uma última tentativa. Enviei meu currículo para qualquer emprego para o qual tivesse, mesmo que remotamente, alguma chance de estar qualificado. No dia seguinte, uma pequena startup me chamou para uma entrevista. Fiz o melhor que pude para manter a aparência decente. Barbeei-me, usei roupas limpas, prendi o cabelo, tomei banho (uma tarefa difícil para os sem-teto) e me apresentei. Fui franco, expliquei minha situação, contei por que havia resolvido me arriscar em Austin, fiz o melhor que pude durante a entrevista para mostrar que talvez não estivesse no melhor dos momentos, mas, se me dessem uma oportunidade, faria tudo para mostrar que um dia poderia ser o melhor.

Fui embora achando que não tinha sido uma boa entrevista. Minha honestidade poderia ter prejudicado minhas chances, mas uma semana e meia depois, após pensar em desistir de tudo, a startup me chamou novamente para uma segunda entrevista.

Quando cheguei, só o diretor estava presente. Ele disse que ficou impressionado com minha honestidade e queria me dar uma chance. Disse que eu tinha uma boa base e era como uma caixa resistente, mas relativamente vazia. Ele achou que eu era suficientemente perseverante para manipular qualquer coisa que me passassem e aprenderia fazendo. Por fim, disse-se que eu começaria em 6 de dezembro.

Um ano depois, vivo em um apartamento muito melhor do que antes de me tornar programador. Sou respeitado por meus colaboradores que chegam a pedir minha opinião em assuntos relevantes da empresa. Você pode fazer ou ser qualquer coisa. Nunca tenha medo de

tentar, mesmo que isso signifique se arriscar quanto tudo estiver desabando.

Tianni Myers

A próxima é Tianni Myers, que leu *Programador Autodidata* e me enviou um email com a história a seguir sobre sua jornada para aprender a codificar fora da escola:

> Minha jornada como autodidata começou em uma aula de web design que eu tinha na universidade, quando tentava obter um diploma de graduação em Comunicação em Mídias Digitais. Na época, queria ser redatora e trabalhar em marketing. Minhas metas mudaram após decidir aprender a programar. Estou escrevendo para compartilhar minha história de autodidata e contar como em 12 meses passei de operadora de caixa a desenvolvedora web júnior.
>
> Comecei aprendendo os aspectos básicos de HTML e CSS no Code Academy. Foi assim que escrevi meu primeiro programa Python, um jogo de números; o computador selecionava um número aleatório e o usuário tinha três tentativas para adivinhar qual era o correto. Esse projeto e o Python me deixaram interessada em computadores.
>
> Minhas manhãs começavam às 4 horas, com uma xícara de café. Passava de seis a dez horas por dia lendo livros de programação e escrevendo código. Tinha 21 anos e trabalhava em regime de meio expediente na Goodwill para pagar as contas. Era muito feliz porque passava grande parte do dia fazendo o que gostava, ou seja, desenvolver usando várias linguagens de programação como minhas ferramentas.
>
> Um dia, estava na Indeed casualmente tentando conseguir um emprego. Não esperava obter resposta, mas recebi-a alguns dias depois de uma agência de marketing. Passei por uma avaliação em SQL na Indeed, seguida de uma entrevista por telefone, uma avaliação em codificação e, logo depois, uma entrevista presencial. Durante minha entrevista, o diretor de desenvolvimento web e dois desenvolvedores seniores examinaram minhas respostas à avaliação em codificação. Fiquei satisfeita porque gostaram muito de algumas respostas e

ficaram surpresos quando lhes disse que era autodidata. Eles me confessaram que algumas delas eram melhores que as dadas por desenvolvedores seniores que haviam feito a mesma avaliação. Duas semanas depois, fui contratada.

Se você puder se esforçar e aguentar a pressão, transformará seus sonhos em realidade.

Começando

Os exemplos de código deste livro estão em Python. Escolhi Python porque é uma das linguagens de programação mais fáceis de ler. No decorrer do livro, formatei os exemplos desta forma:

```
for i in range(100):
print("Hello, World!")
```

```
>> Hello, World!
>> Hello, World!
>> Hello, World!
```

O texto que vem depois de `>>` é a saída do shell interativo do Python. Reticências depois de uma saída (`...`) significam "e assim por diante". Se não houver `>>` depois de um exemplo, isso significará que o programa não produz nenhuma saída ou que estou explicando um conceito e a saída não é importante. Qualquer coisa que estiver em um parágrafo em `fonte monoespaçada` é algum tipo de código, saída de código ou jargão de programação.

Instalando Python

Para acompanhar os exemplos deste livro, você precisa ter o Python versão 3 instalado. É possível baixá-lo no Windows e Unix, em http://python.org/downloads. Se você estiver no Ubuntu, o Python 3 estará instalado por padrão. Certifique-se de baixar o Python 3 e não o Python 2. Alguns exemplos do livro não funcionarão se estiver usando o Python 2.

O Python está disponível para computadores de 32 e 64 bits. Se comprou seu computador depois de 2007, é provável que seja de 64 bits. Se não

tiver certeza, uma busca na internet poderá ajudá-lo a descobrir.

Se estiver no Windows ou em um Mac, baixe a versão do Python para 32 ou 64 bits, abra o arquivo e siga as instruções. Você também pode visitar `http://theselftaughtprogrammer.io/installpython` para ver vídeos explicando como instalar o Python em cada sistema operacional.

Solucionando problemas

Se estiver tendo problemas para instalar o Python, envie uma mensagem (em inglês) ao grupo Self-Taught Programmers do Facebook. Você pode encontrá-lo em `https://facebook.com/groups/selftaughtprogrammers`. Quando postar um código no grupo Self-Taught Programmers (ou em qualquer outro local online para pedir ajuda), certifique-se de inseri-lo em um gist do GitHub. Nunca envie uma screenshot de seu código. Para as pessoas o ajudarem, terão de executar seu programa por conta própria. Se você enviar uma screenshot, elas precisarão digitar todo o seu código manualmente, mas se enviar o código em um gist do GitHub, poderão copiá-lo e colá-lo rapidamente em seu IDE.

Desafios

Muitos capítulos deste livro terminam com um desafio de codificação para ser resolvido. Tais desafios têm o objetivo de testar seu conhecimento do material, torná-lo um programador melhor e ajudar a prepará-lo para uma entrevista técnica. Você pode encontrar as soluções de todos os desafios do livro no GitHub, em `https://github.com/calthoff/tstcs_challenge_solutions`.

Quando estiver lendo este livro e resolvendo os desafios, recomendo compartilhar suas vitórias com a comunidade de autodidatas usando `#selftaughtcoder` no Twitter. Sempre que achar que está fazendo progressos significativos em sua jornada para aprender a codificar, envie um tuíte motivacional usando `#selftaughtcoder` para outras pessoas da comunidade se motivarem com seu progresso. Fique à vontade para me marcar: `@coryalthoff`.

Seja insistente

Há um último item que desejo abordar antes de você se aprofundar na aprendizagem da Ciência da Computação. Se você está lendo este livro, já aprendeu sozinho a programar. Como você sabe, a parte mais desafiadora ao desenvolver uma nova habilidade como a programação não é a dificuldade do material: é ser insistente. Insistir em aprender coisas novas é algo que por anos me esforcei a fazer até finalmente aprender um truque que gostaria de compartilhar com você chamado *Não Quebre a Corrente*.

Jerry Seinfeld inventou o truque *Não Quebre a Corrente* quando estava estruturando sua primeira comédia stand-up. Primeiro, pendurou um calendário em sua sala. Em seguida, se escrevesse uma piada no fim de cada dia, marcava esse dia com um X vermelho (prefiro sinais de visto verdes) no calendário. É isso. O truque é somente esse e funciona muito bem.

Uma vez que você começar uma corrente (dois ou mais sinais de visto verdes em sequência), não vai querer quebrá-la. Dois sinais de visto verdes em sequência passarão a ser cinco sinais de visto verdes em sequência. Em seguida, 10. Depois, 20. Quanto mais longa ficar sua faixa de sinais de visto, mais difícil será quebrar a corrente. Suponhamos que seja o final do mês e você esteja examinando seu calendário. Você tem 29 sinais de visto verdes. Só precisa de mais um para ter um mês perfeito. Não há como não realizar sua tarefa nesse dia. Ou como Jerry Seinfeld descreve:

> Após alguns dias, você terá uma corrente. Continue assim e a corrente ficará mais longa a cada dia. Você gostará de olhar para a corrente, principalmente quando tiver feito tudo certo após algumas semanas. A única coisa que terá de fazer é não quebrar a corrente.

Minha obsessão em preservar uma de minhas correntes me levou a fazer coisas malucas, como ir à academia no meio da noite, para mantê-la intacta. Não há sensação melhor do que olhar para a página do calendário que contém seu primeiro mês perfeito e vê-lo preenchido com sinais de visto verdes. Se estiver se sentindo desmotivado, sempre poderá olhar para essa página e pensar no mês em que fez tudo certo.

É difícil terminar a leitura de livros técnicos. Perdi a conta de quantos abandonei no meio do caminho. Tentei ao máximo tornar este livro divertido e fácil de ler, mas, para ter certeza de que terminará a leitura, tente usar o truque *Não Quebre a Corrente*. Também fiz uma parceria com o site `monday.com` para criar um template chamado Self-Taught Programmer que registrarão suas sequências de codificação. Você pode experimentá-los em `https://hey.monday.com/CoryAlthoff`.

Está pronto para estudar Ciência da Computação?

Comecemos então!

PARTE I

Introdução aos algoritmos

Capítulo 1: O que é um algoritmo?
Capítulo 2: Recursão
Capítulo 3: Algoritmos de busca
Capítulo 4: Algoritmos de ordenação
Capítulo 5: Algoritmos de string
Capítulo 6: Matemática
Capítulo 7: Inspiração autodidata: Margaret Hamilton

CAPÍTULO 1

O que é um algoritmo?

Seja para revelar os segredos do universo, seja para apenas seguir uma carreira no século 21, a programação básica de computadores é uma habilidade essencial que precisa ser aprendida.

– Stephen Hawking

Um **algoritmo** é uma sequência de etapas que resolve um problema. Por exemplo, um algoritmo de como fazer ovos mexidos seria quebrar três ovos em uma tigela, batê-los, colocá-los em uma frigideira, aquecê-la em um fogão e e tirá-los da frigideira assim que eles já estiverem no ponto. Esta seção do livro é toda sobre algoritmos. Você conhecerá algoritmos que poderá usar para resolver problemas, como encontrar números primos. Também aprenderá a escrever um novo e elegante tipo de algoritmo e a buscar e ordenar dados.

Neste capítulo, você aprenderá a comparar dois algoritmos para que isso o ajude a analisá-los. É importante que os programadores saibam por que um algoritmo pode ser melhor do que outro, por que passam boa parte do tempo escrevendo algoritmos e decidindo que estruturas de dados usar com eles. Se você não souber por que deve escolher um algoritmo em vez de outro, não será um programador eficiente, logo este capítulo é crítico.

Embora os algoritmos sejam um conceito essencial na Ciência da Computação, os cientistas da computação ainda não entraram em acordo sobre uma definição formal. Há muitas definições concorrentes, mas a de Donald Knuth é uma das melhores entre as conhecidas. Ele descreve um algoritmo como um processo definido, eficaz e finito que recebe entradas e produz saídas de acordo com essas entradas.

- *Precisão* significa etapas claras, concisas e não ambíguas.
- *Eficácia* significa você poder executar com precisão cada operação para resolver o problema.
- *Finitude* significa o algoritmo parar após um número finito de etapas.

Algo que poderíamos acrescentar a essa lista é a *exatidão*. Um algoritmo deve sempre produzir a mesma saída para uma entrada específica e essa saída deve ser a resposta correta para o problema que ele resolve.

A maioria dos algoritmos, mas não todos, atende a esses requisitos e algumas exceções são importantes. Por exemplo, quando criamos um gerador de números aleatórios, o objetivo é gerar aleatoriedade para que ninguém consiga usar a entrada para adivinhar a saída. Além disso, muitos algoritmos em ciência de dados não são rigorosos quando se trata de exatidão. Pode ser suficiente um algoritmo estimar a saída, contanto que a incerteza da estimativa seja conhecida. Quase sempre, no entanto, os algoritmos devem atender a todos os requisitos anteriores. Se você escrever um algoritmo para fazer ovos mexidos, o usuário poderá não ficar satisfeito se, ocasionalmente, o algoritmo produzir um omelete ou ovos cozidos.

Analisando algoritmos

Em geral, podemos usar mais de um algoritmo para resolver um problema. Por exemplo, existem várias maneiras de ordenar uma lista. Se muitos algoritmos resolverem um problema, como saber qual é o melhor? É o mais simples? O mais rápido? O menor? Ou algum outro?

Uma maneira de avaliar um algoritmo é pelo seu tempo de execução (run time). O **tempo de execução** é quanto tempo o computador leva para executar um algoritmo escrito em uma linguagem de programação como Python. Por exemplo, aqui está um algoritmo em Python que conta de 1 a 5 e exibe cada número:

```
for i in range(1, 6):
print(i)
```

Você pode medir o tempo de execução desse algoritmo usando o módulo

interno `time` do Python para rastrear quanto tempo seu computador leva para executá-lo:

```
import time
start = time.time()
for i in range(1, 6):
print(i)
end = time.time()
print(end - start)
```

```
>> 1
>> 2
>> 3
>> 4
>> 5
>> 0.15141820907592773
```

Quando você executar seu programa, ele exibirá os números de 1 a 5 e mostrará o tempo que levou sua execução. Nesse caso, levou 0,15 segundo.

Agora, reexecute o programa:

```
import time
start = time.time()
for i in range(1, 6):
print(i)
end = time.time()
print(end - start)
```

```
>> 1
>> 2
>> 3
>> 4
>> 5
>> 0.14856505393981934
```

Na segunda vez que você executar seu programa, deverá ver um tempo de execução diferente. Se você executá-lo mais uma vez, verá outro tempo de execução. O tempo de execução do algoritmo continua mudando porque o poder de processamento disponível em seu computador quando executa o programa varia, o que afeta o tempo de execução.

O tempo de execução de seu algoritmo também seria diferente em outro

computador. Se você executá-lo em um computador com menos poder de processamento, será mais lento, enquanto em um computador mais poderoso, será mais veloz. Além disso, o tempo de execução desse programa será afetado pela linguagem de programação na qual você o escreveu. Por exemplo, o tempo de execução será mais rápido se você executar esse mesmo programa em C porque C consegue ser mais rápido do que Python.

Já que o tempo de execução de um algoritmo é afetado por tantas variáveis diferentes, como o poder de processamento do computador e a linguagem de programação, não é uma maneira eficaz de comparar dois algoritmos. Em vez disso, os cientistas da computação comparam algoritmos examinando o número de etapas que eles requerem. Você pode inserir o número de etapas envolvidas em um algoritmo, em uma fórmula que compare dois ou mais algoritmos, sem considerar a linguagem de programação ou o computador. Veja um exemplo. Este é o programa anterior que conta de 1 a 5:

```
for i in range(1, 6):
print(i)
```

Seu programa leva cinco etapas para ser concluído (percorre um loop cinco vezes e exibe `i` cada vez). Você pode expressar o número de etapas que o algoritmo requer com esta equação:

```
f(n) = 5
```

Se você tornar o programa mais complexo, a equação mudará. Por exemplo, você pode querer calcular a soma de todos os números que está exibindo:

```
count = 0
for i in range(1, 6):
print(i)
count += i
```

Agora, o algoritmo leva 11 etapas para ser concluído. Primeiro, atribui zero à variável `count`. Em seguida, exibe cinco números e incrementa cinco vezes (1 + 5 + 5 = 11).

Esta é a nova equação do seu algoritmo:

```
f(n) = 11
```

O que acontecerá se você alterar o 6 do código para uma variável?

```
count = 0
for i in range(1, n):
print(i)
count += i
```

A equação mudará para:

```
f(n) = 1 + 2n
```

Agora, o número de etapas que o algoritmo executará vai depender do valor de *n*. O 1 da equação representa a primeira etapa: `count = 0`. Em seguida, existem duas vezes *n* etapas. Por exemplo, se *n* for 5, `f(n)` = 1 + 2 × 5. Os cientistas da computação chamam a variável *n* de uma equação que descreve o número de etapas de um algoritmo de **tamanho do problema**. Nesse caso, podemos dizer que o tempo necessário para resolver um problema de tamanho *n* é 1 + 2*n* ou, em notação matemática, $T(n) = 1 + 2n$.

No entanto, uma equação que descreva o número de etapas de um algoritmo não é muito útil porque, entre outras coisas, nem sempre é possível contar confiavelmente o número de etapas. Por exemplo, se um algoritmo tiver muitas instruções condicionais, você não terá como saber de antemão qual delas será executada. A boa notícia é que, como cientista da computação, você não precisa se preocupar com o número exato de etapas de um algoritmo. O que é necessário saber é qual é o desempenho do algoritmo quando *n* aumenta. A maioria dos algoritmos tem bom desempenho com um conjunto de dados (dataset) pequeno, mas pode ser um desastre com conjuntos de dados maiores. Até mesmo o algoritmo mais ineficiente terá um bom desempenho se *n* for 1. Contudo, no mundo real, provavelmente *n* não será 1. Poderá ser igual a várias centenas de milhares, um milhão ou mais.

O detalhe importante que precisa ser conhecido em um algoritmo não é o número exato de etapas que ele executará, e sim uma estimativa do número de etapas a serem executadas quando *n* aumentar. À medida que *n* aumentar, uma parte da equação ofuscará o restante a ponto de

tudo o mais se tornar irrelevante. Veja este código Python:

```
def print_it(n):
# loop 1
for i in range(n):
print(i)
# loop 2
for i in range(n):
print(i)
for j in range(n):
print(j)
for h in range(n):
print(h)
```

Que parte desse programa é mais importante para determinar quantas etapas seu algoritmo executará para ser concluído? Você pode achar que as duas partes da função (o primeiro loop e o segundo loop que contém outros loops) são importantes. Afinal, se *n* for 10.000, seu computador exibirá muitos números nos dois loops.

No entanto, o código a seguir será irrelevante se você estiver considerando a eficiência do algoritmo:

```
# loop 1
for i in range(n):
print(i)
```

Para entender o porquê, precisamos examinar o que acontece quando *n* aumenta.

Aqui está a equação do número de etapas de seu algoritmo:

```
T(n) = n + n**3
```

Quando você tiver dois loops `for` aninhados executando *n* etapas, a equação passará para *n***2 (*n* elevado à segunda potência) porque se *n* for 10, será necessário executar 10 etapas duas vezes ou 10**2. Três loops `for` aninhados serão sempre *n***3 pela mesma razão. Nessa equação, quando *n* for 10, o primeiro loop do programa executará 10 etapas e o segundo executará 10^3 etapas, o que resulta em 1.000. Quando *n* for 1.000, o primeiro loop executará 1.000 etapas e o segundo loop executará 1.000^3, resultando em 1 bilhão.

Viu o que aconteceu? Quando *n* aumenta, a segunda parte do algoritmo

cresce tão rapidamente que a primeira parte torna-se irrelevante. Por exemplo, se você precisasse que esse programa trabalhasse com 100.000.000 de registros de banco de dados, não se preocuparia com quantas etapas a primeira parte da equação executa porque a segunda parte executará exponencialmente mais etapas. Com 100.000.000 de registros, a segunda parte do algoritmo executaria mais de um septilhão de etapas, o que seria 1 seguido de 24 zeros, logo não se trata de um algoritmo razoável para ser usado. As primeiras 100.000.000 de etapas não serão relevantes para a sua decisão.

Já que a parte importante de um algoritmo é a que cresce com a rapidez que *n* aumenta, os cientistas da computação usam a notação big O para expressar a eficiência do algoritmo em vez de uma equação T(*n*). A **notação big O** é uma notação matemática que descreve como os requisitos de tempo ou espaço de um algoritmo (você conhecerá os requisitos de espaço posteriormente) aumentam conforme o tamanho de *n* cresce.

Os cientistas da computação usam a notação *big O* para criar uma função de ordem de grandeza a partir de T(*n*). A **ordem de grandeza** é uma classe em um sistema de classificação no qual cada classe é muitas vezes maior ou menor do que a classe anterior. Em uma função de ordem de grandeza, usamos a parte de T(*n*) predominante na equação e ignoramos o restante. A parte de T(*n*) predominante na equação é a ordem de grandeza de um algoritmo. Estas são as classificações mais usadas para a ordem de grandeza na notação big O, ordenadas da melhor (mais eficiente) para a pior (menos eficiente):

- Tempo constante
- Tempo logarítmico
- Tempo linear
- Tempo log-linear
- Tempo quadrático
- Tempo cúbico
- Tempo exponencial

Cada ordem de grandeza descreve a complexidade de tempo de um algoritmo. A **complexidade de tempo** é o numero máximo de etapas que um algoritmo executa para ser concluído quando *n* aumenta.

Examinaremos cada ordem de grandeza.

Tempo constante

A ordem de grandeza mais eficiente chama-se *complexidade de tempo constante*. Um algoritmo é executado em **tempo constante** quando requer o mesmo número de etapas independentemente do tamanho do problema. A notação big O para a complexidade constante é O(1).

Suponhamos que você tivesse uma livraria online e todo dia desse um livro grátis para seu primeiro cliente. Você armazena os clientes em uma lista chamada `customers`. Seu algoritmo teria esta aparência:

```
free_books = customers[0]
```

A equação T(*n*) seria:

```
T(n) = 1
```

Seu algoritmo requer uma única etapa, não importando quantos clientes você tem. Se tiver 1.000 clientes, o algoritmo executará uma etapa. Se tiver 10.0000 clientes, também executará uma etapa, e se houver um trilhão de clientes, continuará sendo necessária apenas uma etapa.

Se você representar graficamente a complexidade de tempo constante com o número de entradas no eixo *x* e o número de etapas no eixo *y*, o gráfico será plano (Figura 1.1).

Como você pode ver, o número de etapas que o algoritmo leva para ser concluído não cresce quando o tamanho do problema aumenta. Logo, esse é o algoritmo mais eficiente que você pode escrever porque seu tempo de execução não muda quando os conjuntos de dados aumentam.

Figura 1.1: Complexidade constante.

Tempo logarítmico

O tempo logarítmico é a segunda complexidade de tempo mais eficiente. Um algoritmo é executado em **tempo logarítmico** quando seu tempo de execução cresce de acordo com o logaritmo do tamanho da entrada. Vemos essa complexidade de tempo em algoritmos como o de busca binária que pode descartar muitos valores a cada iteração. Se não ficou claro, não se preocupe porque discutiremos esse conceito com detalhes posteriormente no livro. O algoritmo logarítmico é expresso na notação big O como O(log n).

A Figura 1.2 mostra como fica a representação gráfica de um algoritmo logarítmico.

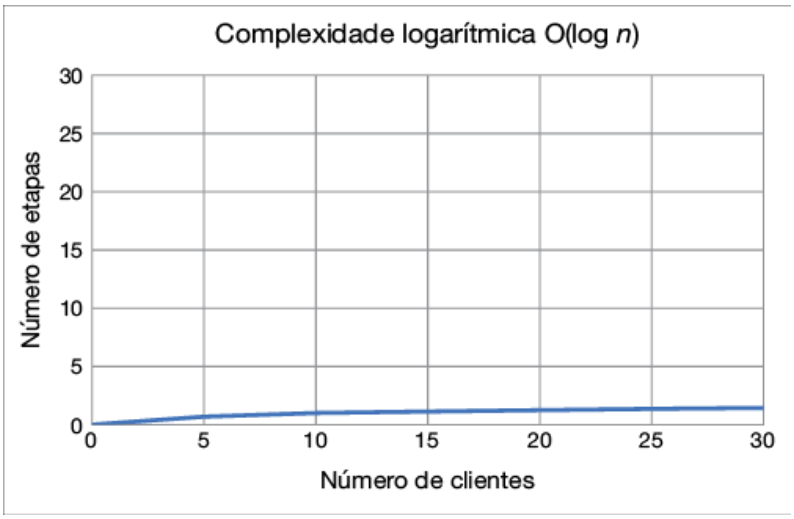

Figura 1.2: Complexidade logarítmica.

O número de etapas necessárias cresce mais lentamente em um algoritmo logarítmico quando o conjunto de dados aumenta.

Tempo linear

O próximo tipo de algoritmo mais eficiente é o que é executado em tempo linear. Um algoritmo executado em **tempo linear** cresce com a mesma proporção do tamanho do problema. O algoritmo linear é expresso na notação big O como O(*n*).

Suponhamos que você precisasse modificar seu programa de livros gratuitos para, em vez de dar um livro grátis para o primeiro cliente do dia, iterasse pela lista de clientes e desse a eles um livro grátis se seus nomes começassem com a letra *B*. Dessa vez, entretanto, a lista de clientes não está ordenada alfabeticamente. Agora, você será forçado a percorrer cada item da lista para encontrar os nomes que começam com *B*.

```
free_book = False
customers = ["Lexi", "Britney", "Danny", "Bobbi", "Chris"]
for customer in customers:
if customer[0] == 'B':
print(customer)
```

Quando sua lista de clientes tiver cinco itens, o programa levará cinco etapas para ser concluído. Para uma lista de 10 clientes, seu programa demandará 10 etapas; para 20 clientes, 20 etapas; e assim por diante.

Esta é a equação da complexidade de tempo desse programa:

```
f(n) = 1 + 1 + n
```

Na notação big O, você pode ignorar as constantes e se concentrar na parte predominante da equação:

```
O(n) = n
```

Em um algoritmo linear, quando *n* cresce, o número de etapas que ele executa aumenta de acordo com o aumento de *n* (Figura 1.3).

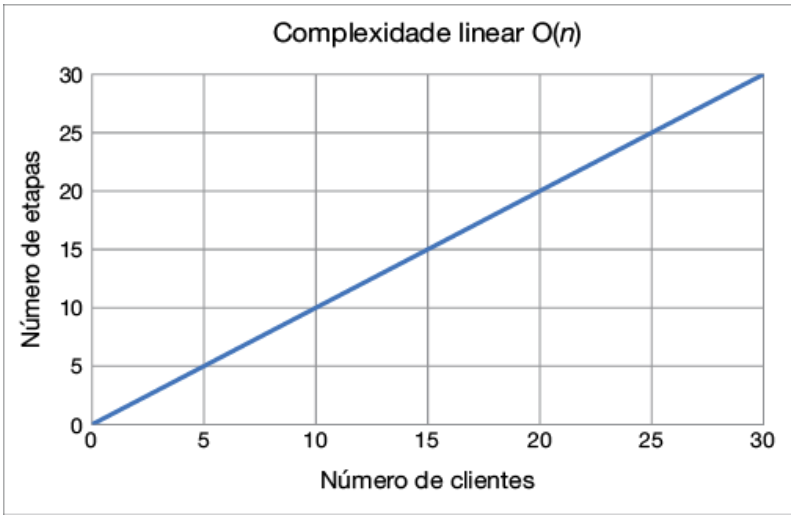

Figura 1.3: Complexidade linear.

Tempo log-linear

Um algoritmo executado em **tempo log-linear** cresce de acordo com uma combinação (multiplicação) das complexidades de tempo logarítmica e linear. Por exemplo, um algoritmo log-linear pode calcular uma operação O(log *n*) *n* vezes. Na notação big O, o algoritmo log-linear é expresso como O(*n* log *n*). Em geral, os algoritmos log-lineares dividem um conjunto de dados em partes menores e processam cada parte independentemente. Muitos algoritmos de ordenação mais eficientes que você conhecerá posteriormente, como a classificação por interpolação (merge sort), são log-lineares.

A Figura 1.4 mostra a aparência de um algoritmo log-linear quando o representamos em um gráfico.

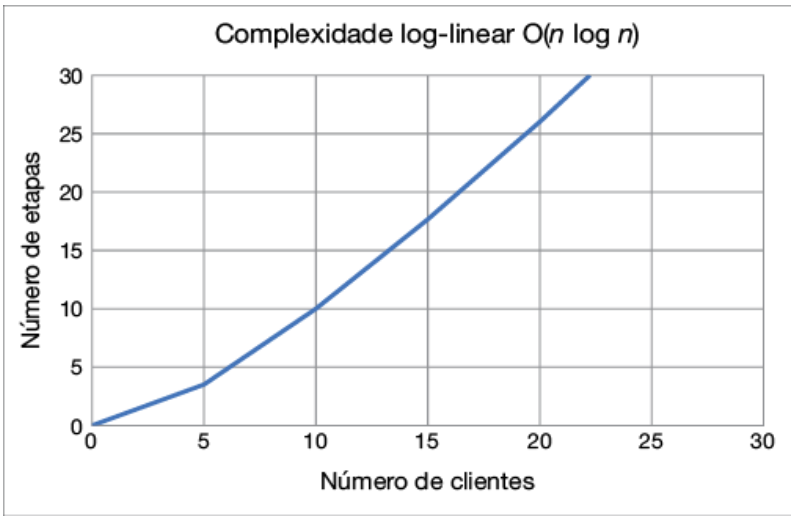

Figura 1.4: Complexidade log-linear.

Como você pode ver, a complexidade log-linear não é tão eficiente quanto a de tempo linear. No entanto, sua complexidade não cresce com a mesma rapidez da quadrática, que você conhecerá a seguir.

Tempo quadrático

Após o tempo log-linear, a próxima complexidade de tempo mais eficiente é a de tempo quadrático. Um algoritmo é executado em **tempo quadrático** quando seu desempenho é diretamente proporcional ao tamanho do problema ao quadrado. Na notação big O, você pode expressar um algoritmo quadrático como O(*n***2).

Aqui está um exemplo de um algoritmo com complexidade quadrática:

```
numbers = [1, 2, 3, 4, 5]
for i in numbers:
for j in numbers:
x = i * j
print(x)
```

Esse algoritmo multiplica cada número de uma lista de números por todos os outros números, armazena o resultado em uma variável e exibe-o.

Nesse caso, *n* é o tamanho da lista `numbers`. A equação da complexidade de tempo desse algoritmo é a seguinte:

```
f(n) = 1 + n * n * (1 + 1)
```

A parte (1 + 1) da equação vem da instrução de multiplicação e exibição. Estamos repetindo a instrução de multiplicação e exibição $n * n$ vezes com os dois loops `for` aninhados. Você pode simplificar a equação desta forma:

```
f(n) = 1 + (1 + 1) * n**2
```

que é o mesmo que a linha a seguir:

```
f(n) = 1 + 2 * n**2
```

Como você deve ter percebido, a parte $n**2$ da equação ofusca o restante, logo, na notação big O, a equação é:

```
O(n) = n**2
```

Quando representamos um algoritmo com complexidade quadrática em um gráfico, o número de etapas aumenta nitidamente à medida que o tamanho do problema aumenta (Figura 1.5).

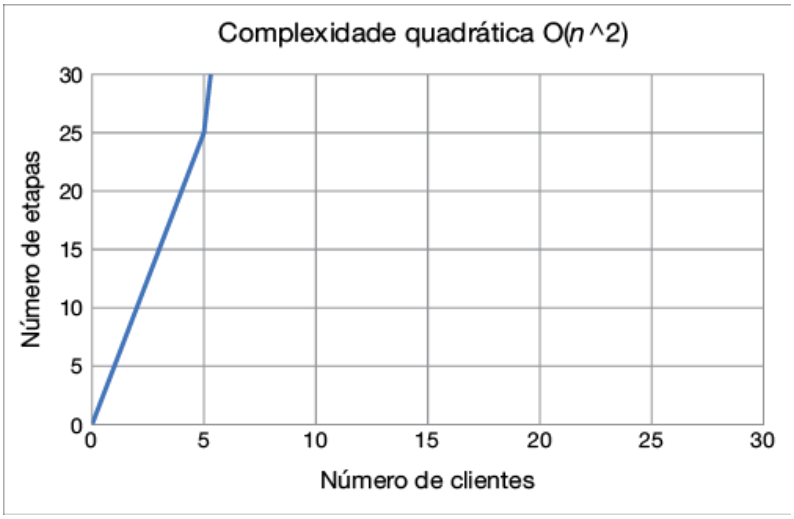

Figura 1.5: Complexidade quadrática.

Como regra geral, se seu algoritmo tiver dois loops aninhados indo de 1 a n (ou de 0 a n – 1), sua complexidade de tempo será no mínimo O($n**2$). Muitos algoritmos de ordenação como o da ordenação por inserção e por bolha (que você conhecerá posteriormente no livro) seguem o tempo quadrático.

Tempo cúbico

Após a complexidade quadrática, vem a complexidade de tempo cúbico. Um algoritmo é executado em **tempo cúbico** quando seu desempenho é diretamente proporcional ao tamanho do problema ao cubo. Na notação big O, um algoritmo cúbico é expresso como O(*n***3). Um algoritmo com complexidade cúbica é semelhante ao quadrático, exceto por *n* ser elevado à terceira potência em vez de à segunda.

Aqui está um algoritmo com complexidade de tempo cúbico:

```
numbers = [1, 2, 3, 4, 5]
for i in numbers:
for j in numbers:
for h in numbers:
x = i + j + h
print(x)
```

A equação desse algoritmo é a seguinte:

```
f(n) = 1 + n * n * n * (1 + 1)
```

Ou também pode ser:

```
f(n) = 1 + 2 * n**3
```

Como em um algoritmo com complexidade quadrática, a parte mais crítica dessa equação é *n***3, que cresce tão rapidamente que torna o restante da equação, mesmo se incluísse *n***2, irrelevante. Logo, na notação big O, a complexidade cúbica é expressa assim:

```
O(n) = n**3
```

Enquanto dois loops aninhados são sinal de complexidade de tempo quadrático, três loops aninhados indo de 0 a *n* significa que o algoritmo seguirá o tempo cúbico. Você deverá encontrar a complexidade de tempo cúbico se seu trabalho envolver ciência de dados ou estatística.

As complexidades de tempo tanto quadrático quanto cúbico são casos especiais de uma família maior de complexidades de tempo polinomial. Um algoritmo executado em **tempo polinomial** aumenta de escala seguindo o padrão O(*n****a*), onde *a* = 2 para o tempo quadrático e *a* = 3 para o tempo cúbico. Ao projetar algoritmos, devemos evitar o aumento

em escala polinomial, quando possível, porque os algoritmos podem ficar muito lentos à medida que *n* aumenta. Pode ser difícil evitar o aumento em escala polinomial, mas talvez sirva de consolo o fato de que certamente a complexidade polinomial não é o pior dos casos.

Tempo exponencial

O prêmio de pior complexidade de tempo vai para a complexidade exponencial. Um algoritmo executado em **tempo exponencial** contém uma constante elevada ao tamanho do problema. Em outras palavras, um algoritmo com complexidade de tempo exponencial leva *c* elevado à enésima potência em etapas para ser concluído. A notação big O da complexidade exponencial é O(*c****n*), onde *c* é uma constante. O valor da constante não importa. O que importa é que *n* está no expoente.

Felizmente, você não encontrará a complexidade exponencial com frequência. Um exemplo de complexidade exponencial envolvendo a tentativa de adivinhar uma senha numérica composta de *n* dígitos decimais com a verificação de cada combinação possível teria complexidade O(10***n*).

Aqui está um exemplo de adivinhação de senha com complexidade O(10***n*):

```
pin = 931
n = len(str(pin))
for i in range(10**n):
if i == pin:
print(i)
```

O número de etapas que esse algoritmo leva para ser concluído cresce muito rápido à medida que *n* aumenta. Quando *n* é 1, o algoritmo executa 10 etapas. Quando *n* é 2, executa 100 etapas. Quando *n* é 3, o algoritmo executa 1.000 etapas. Como você pode ver, inicialmente não parece que um algoritmo exponencial vai crescer muito rapidamente. No entanto, seu crescimento explode. Adivinhar uma senha com 8 dígitos decimais leva 100 milhões de etapas e adivinhar uma senha com 10 dígitos decimais leva mais de 10 bilhões de etapas. Por causa do aumento em escala

exponencial, é importante criar senhas longas. Se alguém tentar adivinhar sua senha usando um programa como esse, conseguirá adivinhá-la facilmente se a senha tiver quatro dígitos. Contudo, se sua senha tiver 20 dígitos, será impossível decifrá-la porque o programa levará mais tempo para ser executado do que o tempo de vida de uma pessoa.

Essa solução para adivinhar uma senha é um exemplo de algoritmo de força bruta. Um **algoritmo de força bruta** testa todas as opções possíveis. Em geral, não é eficiente e deve ser seu último recurso.

A Figura 1.6 compara a eficiência dos algoritmos que discutimos.

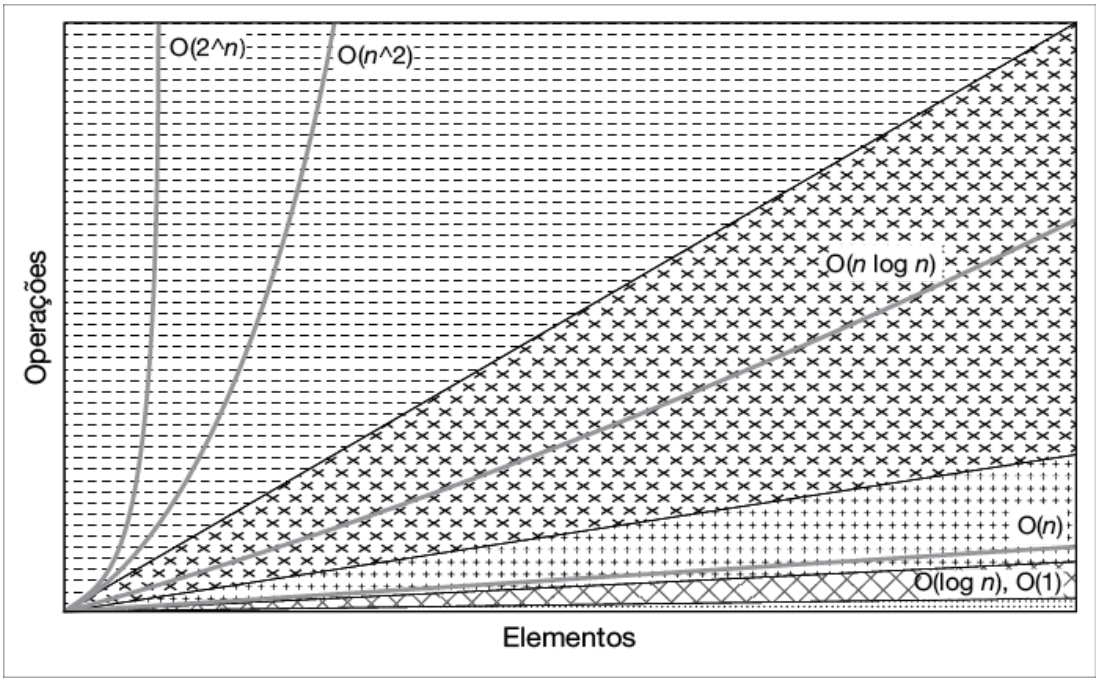

Figura 1.6: Gráfico das complexidades big O.

Complexidade de melhor caso versus de pior caso

O desempenho de um algoritmo pode mudar de acordo com diferentes fatores, como o tipo de dado com o qual você está trabalhando. Portanto, ao avaliar o desempenho de um algoritmo, é preciso considerar suas complexidades de melhor caso, pior caso e caso médio. A **complexidade de melhor caso** de um algoritmo indica como é seu desempenho com entradas ideais e a **complexidade de pior caso** é como ele se sai no pior

cenário possível. A **complexidade de caso médio** de um algoritmo representa como é seu desempenho na média.

Por exemplo, se você tiver de pesquisar item a item em uma lista, poderá ter sorte e encontrar o que está procurando após verificar o primeiro item. Essa seria a complexidade de melhor caso. No entanto, se o item que você está procurando não estiver na lista, será preciso pesquisar a lista inteira e teremos a complexidade de pior caso.

Se você tiver de pesquisar item a item em uma lista cem vezes, em média encontrará o que está procurando no tempo O(n/2), que é o mesmo que o tempo O(n) na notação big O. Ao comparar algoritmos, em geral começamos examinando a complexidade de caso médio. Se você quiser fazer uma análise mais profunda, também poderá comparar as complexidades de melhor caso e de pior caso.

Complexidade de espaço

Os computadores têm recursos finitos como a memória, logo, além de pensar na complexidade de tempo de um algoritmo, você deve considerar o uso de seus recursos. A **complexidade de espaço** é a quantidade de espaço na memória que o algoritmo demanda e inclui o espaço fixo, o espaço de estruturas de dados e o espaço temporário. O **espaço fixo** é a quantidade de memória que o programa requer e o **espaço de estruturas de dados** é a quantidade de memória que o programa precisa para armazenar o conjunto de dados, por exemplo, o tamanho de uma lista que você estiver pesquisando. A quantidade de memória usada pelo algoritmo para armazenar esses dados vai depender da quantidade de entradas que o problema demandar. O **espaço temporário** é a quantidade de memória que o algoritmo precisa para o processamento intermediário, por exemplo, se o algoritmo precisar copiar temporariamente uma lista para transferir dados.

Podemos aplicar os conceitos de complexidade de tempo que já vimos à complexidade de espaço. Por exemplo, você pode calcular um fatorial de n (um produto de todos os inteiros positivos menores que ou iguais a n) usando um algoritmo que tenha uma complexidade de espaço

constante, O(1):

```
x = 1
n = 5
for i in range(1, n + 1):
x = x * i
```

A complexidade de espaço é constante porque a quantidade de espaço que seu algoritmo precisa não cresce quando *n* aumenta. Se você decidisse armazenar todos os fatoriais até *n* em uma lista, o algoritmo teria uma complexidade de espaço linear, O(*n*):

```
x = 1
n = 5
a_list = []
for i in range(1, n + 1):
a_list.append(x)
x = x * i
```

A complexidade de espaço de seu algoritmo é O(*n*) porque a quantidade de espaço que ele usa cresce no mesmo ritmo de *n*.

Como na complexidade de tempo, o nível aceitável de complexidade de espaço de um algoritmo depende da situação. No entanto, quanto menor é o espaço que o algoritmo demanda, melhor é.

Por que isso é importante?

Como cientista da computação, você precisa conhecer as diferentes ordens de grandeza para otimizar seus algoritmos. Quando estiver tentando melhorar um algoritmo, deverá se concentrar em alterar sua ordem de grandeza em vez de melhorá-lo de outras maneiras. Por exemplo, suponhamos que você tenha um algoritmo O(*n***2) que use dois loops `for`. Em vez de otimizar o que ocorre dentro dos loops, é muito mais importante determinar se você pode rescrever seu algoritmo para não ter dois loops `for` aninhados, portanto uma ordem de grandeza menor.

Se puder resolver o problema escrevendo um algoritmo com dois loops `for` não aninhados, o algoritmo terá complexidade O(*n*), o que fará uma enorme diferença em seu desempenho. Essa alteração fará uma diferença muito maior no desempenho de seu algoritmo do que qualquer ganho em

eficiência que você possa obter ajustando um algoritmo O(*n***2). Contudo, também é importante pensar nos cenários de melhor e pior casos do algoritmo. Você pode ter um algoritmo O(*n***2) que, no cenário de melhor caso, apresente complexidade O(*n*) e os dados sejam típicos de seu cenário de melhor caso. Em uma situação dessas, o algoritmo pode ser uma boa opção.

As decisões que você tomar referentes aos algoritmos podem ter consequências importantes no mundo real. Por exemplo, suponhamos que você seja um desenvolvedor web responsável por escrever um algoritmo que responda à solicitação web de um cliente. Sua decisão de escrever um algoritmo constante ou quadrático poderá resultar na diferença entre o site ser carregado em menos de um segundo, deixando seu cliente satisfeito, e o carregamento levar mais de um minuto, o que poderá fazê-lo perder clientes antes de a solicitação ser carregada.

Vocabulário

algoritmo: sequência de etapas que resolve um problema.

algoritmo de força bruta: um tipo de algoritmo que testa todas as opções possíveis.

complexidade de caso médio: como é o desempenho de um algoritmo na média.

complexidade de espaço: quantidade de espaço na memória que um algoritmo necessita.

complexidade de melhor caso: como é o desempenho de um algoritmo com entradas ideais.

complexidade de pior caso: como é o desempenho de um algoritmo no pior cenário possível para ele.

complexidade de tempo: número máximo de etapas que um algoritmo leva para ser concluído quando *n* aumenta.

espaço de estruturas de dados: quantidade de memória que um programa requer para armazenar o conjunto de dados.

espaço fixo: quantidade de memória que um programa requer.

espaço temporário: quantidade de memória que um algoritmo requer para processamento intermediário, por exemplo, se seu algoritmo precisar copiar temporariamente uma lista para transferir dados.

notação big O: notação matemática que descreve como os requisitos de tempo ou espaço de um algoritmo aumentam à medida que o tamanho de *n* cresce.

ordem de grandeza: classe de um sistema de classificação no qual cada classe é muitas vezes maior ou menor do que a classe anterior.

tamanho do problema: a variável *n* de uma equação que descreve o número de etapas de um algoritmo.

tempo constante: um algoritmo é executado em tempo constante quando requer o mesmo número de etapas independentemente do tamanho do problema.

tempo cúbico: um algoritmo é executado em tempo cúbico quando seu desempenho é diretamente proporcional ao cubo do tamanho do problema.

tempo de execução: o tempo que o computador leva para executar um algoritmo escrito em uma linguagem de programação como Python.

tempo exponencial: um algoritmo é executado em tempo exponencial quando contém uma constante elevada ao tamanho do problema.

tempo linear: um algoritmo é executado em tempo linear quando cresce com a mesma proporção do tamanho do problema.

tempo log-linear: um algoritmo é executado em tempo log-linear quando seu crescimento se dá como uma combinação (multiplicação) das complexidades de tempos logarítmico e linear.

tempo logarítmico: um algoritmo é executado em tempo logarítmico quando seu tempo de execução cresce de acordo com o logaritmo do tamanho da entrada.

tempo polinomial: um algoritmo é executado em tempo polinomial quando sua escala aumenta seguindo o padrão O($n^{**}a$), onde $a = 2$ para

o tempo quadrático e $a = 3$ para o tempo cúbico.

tempo quadrático: um algoritmo é executado em tempo quadrático quando seu desempenho é diretamente proporcional ao quadrado do tamanho do problema.

Desafio

1. Encontre um programa que você escreveu no passado. Percorra-o e anote as complexidades de tempo de seus diferentes algoritmos.

CAPÍTULO 2

Recursão

Para entender a recursão, primeiro é preciso saber o que é repetição.

– Anônimo

Um **algoritmo iterativo** resolve problemas repetindo etapas continuamente, em geral usando um loop. A maioria dos algoritmos que você escreveu até agora na sua jornada de programação provavelmente é iterativa. A **recursão** é um método de solução de problemas em que resolvemos partes menores do problema até chegar a uma solução. Os algoritmos recursivos dependem de funções que chamam a si mesmas. Qualquer problema que você puder resolver com um algoritmo iterativo, também poderá resolver com um recursivo; no entanto, às vezes um algoritmo recursivo é uma solução mais elegante.

Um algoritmo recursivo é escrito dentro de uma função ou método que chama a si mesmo. O código dentro da função altera a entrada e passa uma entrada nova na próxima vez que a função chamar a si mesma. Portanto, a função deve ter um **caso base**: uma condição que encerre o algoritmo recursivo para que este não continue sendo executado eternamente. Cada vez que a função chamar a si própria, chegará mais perto do caso base. A condição do caso base acabará sendo satisfeita, o problema será resolvido e a função parará de chamar a si mesma. Um algoritmo que segue essas regras atende às três leis da recursão:

- Um algoritmo recursivo deve ter um caso base.
- Um algoritmo recursivo deve alterar seu estado e prosseguir em direção ao caso base.
- Um algoritmo recursivo deve chamar a si mesmo recursivamente.

Para ajudá-lo a entender como um algoritmo recursivo funciona,

examinaremos o cálculo do fatorial de um número usando um algoritmo recursivo e um iterativo. O **fatorial** de um número é o produto de todos os inteiros positivos menores que ou iguais ao número. Por exemplo, o fatorial de 5 é 5 × 4 × 3 × 2 × 1.

```
5! = 5 * 4 * 3 * 2 * 1
```

Aqui está um algoritmo iterativo que calcula o fatorial de um número, `n`:

```
def factorial(n):
the_product = 1
while n > 0:
the_product *= n
n = n - 1
return the_product
```

Sua função `factorial` recebeu o número `n`, que você está usando em seu cálculo.

```
def factorial(n):
```

Dentro da função, você definiu a variável `the_product`, e a definiu com 1. Você está usando `the_product` para registrar o produto, enquanto multiplica `n` pelos números que o antecedem, por exemplo, 5 * 4 * 3 * 2 * 1.

Em seguida, você usou um loop `while` para iterar para trás de `n` a 1, registrando, ao mesmo tempo, o produto.

```
while n > 0:
the_product *= n
n = n - 1
```

No fim de seu loop `while`, você retornará `the_product`, que contém o fatorial de `n`.

```
return the_product
```

Veja como escrever o mesmo algoritmo recursivamente:

```
def factorial(n):
if n == 0:
return 1
return n * factorial(n - 1)
```

Primeiro, você definiu uma função chamada `factorial`, que recebeu o número `n` como parâmetro. Em seguida, vem o caso base. Sua função chamará a si mesma repetidamente até `n` ser 0, ponto em que retornará `1` e

parará de chamar a si mesma.

```
if n == 0:
return 1
```

Enquanto a condição do caso base não for satisfeita, esta linha de código será executada:

```
return n * factorial(n - 1)
```

Como você pode ver, seu código chama a função `factorial`, ou seja, chama a si mesma. Se essa é a primeira vez que você vê um algoritmo recursivo, deve estar achando estranho e pode parecer até mesmo que esse código não funcionará. No entanto, garanto que funciona. Nesse caso, sua função `factorial` chama a si mesma e retorna o resultado. Contudo, não chama a si mesma com o valor de `n`; em vez disso, chama com o valor de `n` – 1. O parâmetro `n` acabará sendo menor do que 1, o que atenderá ao caso base:

```
if n == 0:
return 1
```

Este é o código completo que você precisa escrever para seu algoritmo recursivo que tem apenas quatro linhas:

```
def factorial(n):
if n == 0:
return 1
return n * factorial(n - 1)
```

Contudo, como funciona? Internamente, cada vez que a função chega a uma instrução `return`, insere-a em uma pilha (stack). Uma pilha é uma estrutura de dados sobre a qual você aprenderá mais na Parte II. É como uma lista em Python, mas removemos os itens na mesma ordem na qual os adicionamos. Suponhamos que você chamasse sua função recursiva `factorial` desta forma:

```
factorial(3)
```

Sua variável `n` começa como 3. A função verifica o caso base cujo resultado é `False`, logo o Python executa esta linha de código:

```
return n * factorial(n - 1)
```

O Python ainda não sabe o resultado de `n * factorial(n – 1)`,

portanto o insere na pilha.

```
# Pilha interna (não tente executar esse código)
# n = 3
[return n * factorial( n - 1)]
```

Em seguida, a função chama a si mesma novamente após subtrair 1 de `n`:

```
factorial(2)
```

A função verifica o caso base novamente, cujo resultado é `False`, logo o Python executa esta linha de código:

```
return n * factorial(n - 1)
```

Python continua não sabendo o resultado de `n * factorial(n - 1)`, então o coloca na pilha.

```
# Pilha interna
# n = 3 # n = 2
[return n * factorial( n - 1), return n * factorial( n - 1),]
```

Mais uma vez, a função chama a si mesma após subtrair 1 de `n`:

```
factorial(1)
```

Python ainda não sabe o resultado de `n * factorial(n - 1)` e coloca-o na pilha.

```
# Pilha interna
# n = 3 # n = 2 # n = 1
[return n * factorial( n - 1), return n * factorial( n - 1), return n *
factorial( n - 1),]
```

Novamente, a função chama a si mesma após subtrair 1 de `n`, mas, dessa vez, `n` é 0, o que significa que o caso base foi atendido, logo você retornará `1`.

```
if n == 0:
return 1
```

Python insere o valor de retorno na pilha novamente, só que dessa vez sabe o que está retornando: o número 1. Agora, a pilha interna de Python ficará assim:

```
# Pilha interna
# n = 3 # n = 2 # n = 1
[return n * factorial( n - 1), return n * factorial( n - 1), return n *
factorial( n - 1), 1]
```

Já que Python conhece o último resultado de `return`, pode calcular o resultado anterior e removê-lo da pilha. Em outras palavras, Python multiplica `1 * n` e `n` é 1.

```
1 * 1 = 1
```

A pilha interna de Python agora ficou assim:

```
# Pilha interna
# n = 3 # n = 2
[return n * factorial( n - 1), return n * factorial( n - 1), 1]
```

Mais uma vez, já que Python sabe o último resultado do `return`, pode calcular o resultado anterior e removê-lo da pilha.

```
2 * 1 = 2
```

A pilha interna de Python ficou assim:

```
# Pilha interna
# n = 3
[return n * factorial( n - 1), 2]
```

Para concluir, Python conhece o último resultado do `return`, pode calcular o resultado anterior, removê-lo da pilha e retornar a resposta.

```
3 * 2 = 6
# Internal stack
[return 6]
```

Como você pode ver, calcular o fatorial de um número é um exemplo perfeito de um problema que você pode resolver encontrando soluções para partes menores do mesmo problema. Sabendo disso e escrevendo um algoritmo recursivo, você criou uma solução elegante para calcular o fatorial de um número.

Quando usar a recursão

A frequência com que você usará a recursão em seus algoritmos é decisão sua. Qualquer algoritmo que você puder escrever recursivamente, também poderá escrever iterativamente. A principal vantagem da recursão é que é elegante. Como vimos anteriormente, sua solução iterativa para calcular fatoriais necessitou de seis linhas de código, enquanto a solução recursiva demandou apenas quatro. Uma desvantagem dos algoritmos recursivos é

que geralmente usam mais memória porque têm de armazenar dados na pilha interna do Python. As funções recursivas também podem ser mais difíceis de ler e depurar do que os algoritmos iterativos porque pode ser mais complicado seguir o que está ocorrendo em um algoritmo recursivo.

O emprego ou não da recursão para resolver um problema vai depender da situação. Por exemplo, o quanto o uso da memória é importante versus o quanto o algoritmo recursivo seria mais elegante do que um algoritmo iterativo correspondente. Posteriormente no livro, você verá outros exemplos nos quais a recursão oferece uma solução mais elegante do que um algoritmo iterativo, como quando percorremos uma árvore binária.

Vocabulário

algoritmo iterativo: algoritmo que resolve um problema repetindo etapas continuamente, em geral usando um loop.

caso base: condição que encerra um algoritmo recursivo para que não seja executado eternamente.

fatorial: produto de todos os inteiros positivos menores que ou iguais a um número.

recursão: método de solução de um problema no qual a solução depende de soluções para partes menores do mesmo problema.

Desafio

1. Exiba os números de 1 a 10 recursivamente.

CAPÍTULO 3

Algoritmos de busca

Um algoritmo tem de ser visto para ser entendido.

– Donald Knuth

Como programador profissional, você passará muito tempo trabalhando com dados. Caso se torne um desenvolvedor web ou de aplicações, exibirá dados para os usuários quando eles visitarem seu site ou aplicação. Com frequência, você terá de manipular os dados antes de exibi-los para eles. Caso se torne um cientista de dados, passará ainda mais tempo trabalhando com dados. Talvez a Netflix o contrate para usar seus dados a fim de melhorar o algoritmo de recomendação de filmes. Ou o Instagram poderá contratá-lo para analisar dados e ajudar a manter os usuários em sua plataforma por mais tempo.

Uma das tarefas mais básicas que um programador que trabalhe com dados precisa saber é como os buscar. Cientistas da computação buscam dados escrevendo um **algoritmo de busca**: um algoritmo que procure dados em um conjunto de dados. Um **conjunto de dados** é uma coleção de dados. Dois exemplos comuns de algoritmos de busca seriam as buscas linear e binária. Como programador profissional, provavelmente você não passará muito tempo implementando algoritmos de busca porque linguagens de programação, como Python, têm os próprios algoritmos internos. No entanto, aprender como codificar alguns algoritmos de busca o tornará um programador melhor, já que o ajudará a entender com mais profundidade conceitos básicos da ciência da computação, como ordens de grandeza linear e logarítmica. Também é crucial que você conheça esses algoritmos para saber quais algoritmos de busca internos do Python deve usar e como será seu desempenho com diferentes conjuntos de dados.

Neste capítulo, você aprenderá como procurar um número em uma lista usando dois algoritmos diferentes: busca linear e busca binária. Em seguida, após codificar por conta própria cada algoritmo de busca, mostrarei como executar a mesma busca usando ferramentas internas do Python.

Busca linear

Em uma **busca linear** (ou **busca sequencial**), percorremos cada elemento de um conjunto de dados e o comparamos com o número-alvo. Se a comparação encontrar uma ocorrência, o número estará na lista. Se o algoritmo terminar sem encontrar uma correspondência, o número não estará na lista.

Aqui está um algoritmo de busca linear em Python:

```
def linear_search(a_list, n):
for i in a_list:
if i == n:
return True
return False
a_list = [1, 8, 32, 91, 5, 15, 9, 100, 3]
print(linear_search(a_list, 91))
```

```
>> True
```

A primeira parte de seu programa chama a função `linear_search` e passa uma lista e o número a ser procurado, *n*:

```
a_list = [1, 8, 32, 91, 5, 15, 9, 100, 3]
print(linear_search(a_list, 91))
```

Nesse caso, *n* é 91, logo seu algoritmo está tentando ver se 91 está em `a_list`.

Em seguida, você usou um loop `for` para percorrer cada elemento de `a_list`:

```
for i in a_list:
```

Depois, usou uma instrução `if` para comparar cada elemento de `a_list` com `n`:

```
if i == n:
```

Se `n` for encontrado, será retornado `True`. Se você percorrer a lista e `n` não for encontrado, será retornado `False`:

```
for i in a_list:
if i == n:
return True
return False
```

Quando você executar seu programa, retornará `True` porque o número, *n* (nesse caso, 91), está em `a_list`:

```
print(linear_search(a_list, 91))
>> True
```

Se você executar novamente o programa, mas procurar 1003 em vez de 91, ele retornará `False` porque 1003 não está em `a_list`:

```
print(linear_search(a_list, 1003))
>> False
```

Quando usar uma busca linear

A complexidade de tempo de uma busca linear é O(*n*). No cenário de pior caso, em uma lista de dez itens, seu algoritmo executará dez etapas. O cenário de melhor caso de uma busca linear tem complexidade O(1) porque o item que você está procurando pode ser o primeiro item da lista, logo o algoritmo executará apenas uma etapa, já que parará assim que encontrar uma correspondência. Na média, uma busca linear executa *n*/2 etapas.

Você deve considerar o uso de busca linear quando seus dados não estiverem ordenados. **Dados ordenados** são dados organizados de maneira significativa. Por exemplo, você poderia ordenar uma lista de números sequencialmente (na ordem crescente ou decrescente):

```
# Lista não ordenada
the_list = [12, 19, 13, 15, 14, 10, 18]
# Lista ordenada na ordem crescente
the_list = [10, 12, 13, 14, 15, 18, 19]
```

Se seus dados estiverem ordenados, você poderá usar a busca binária, que é mais eficiente e será vista em breve.

Quando estiver programando no mundo real, em vez de escrever a própria busca linear, poderá usar a palavra-chave interna `in` do Python. Veja como executar uma busca linear em uma lista de números usando a palavra-chave `in` do Python:

```
unsorted_list = [1, 45, 4, 32, 3]
print(45 in unsorted_list)
```

```
>> True
```

Usando a palavra-chave `in` do Python, você executou uma busca linear em `unsorted_list` com apenas uma linha de código.

Nos exemplos que vimos até agora, só procuramos números. Você também pode usar uma busca linear para encontrar caracteres em strings. Em Python, é possível procurar um caractere em uma string usando uma busca linear como esta:

```
print('a' in 'apple')
```

Busca binária

A **busca binária** é outro algoritmo mais rápido para a busca de um número em uma lista. No entanto, você não pode usá-la em qualquer conjunto de dados porque ela só funciona quando os dados estão ordenados.

Uma busca binária procura um elemento em uma lista, dividindo-a em metades. Suponhamos que você tivesse a lista de números ordenada (do número mais baixo ao mais alto) mostrada na Figura 3.1 e estivesse procurando o número 19.

Figura 3.1: Conjunto de dados ordenado para uma busca binária.

A primeira etapa de uma busca binária é localizar o número do meio. Há sete itens nessa lista, logo o número do meio é 14 (Figura 3.2).

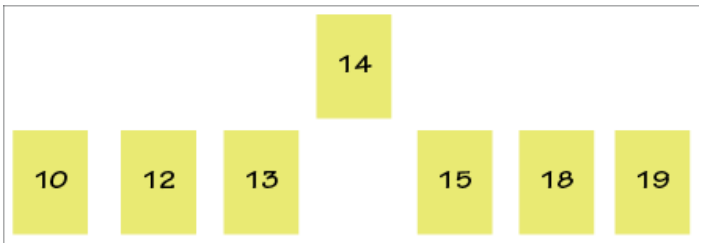

Figura 3.2: Primeiro a busca binária localiza o número do meio.

Já que 14 não é o número procurado, você dá prosseguimento.

A próxima etapa é determinar se o número que você está procurando é maior ou menor do que o número do meio. O número procurado, 19, é maior do que 14, portanto não é necessário pesquisar a metade inferior da lista. Você pode descartá-la. Agora, só sobrou a metade superior com três números para serem pesquisados (Figura 3.3).

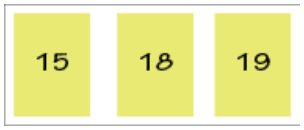

Figura 3.3: A próxima etapa de uma busca binária elimina a metade dos dados que não pode conter o número.

Em seguida, você repete o processo localizando o número do meio novamente, que agora é 18 (Figura 3.4).

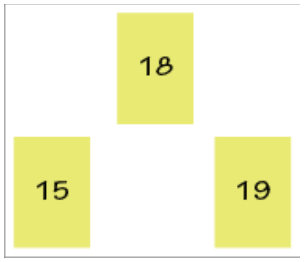

Figura 3.4: A busca binária encontra, então, o número do meio novamente.

Já que 18 não é o número procurado, você determina novamente se deve manter a parte inferior ou a superior da lista. Como 19 é maior do que 18, você mantém a metade superior e descarta a inferior.

Isso deixa sobrando apenas um número, 19, que é o que você está procurando (Figura 3.5). Se o número não fosse 19, você saberia que ele não está na lista.

Figura 3.5: Nossa busca binária encontrou o número.

Em uma busca linear, teríamos levado sete etapas para encontrar o número 19. Precisamos apenas de três etapas com uma busca binária, que é menos da metade do número de etapas.

Veja como implementar uma busca binária em Python:

```
def binary_search(a_list, n):
first = 0
last = len(a_list) - 1
while last >= first:
mid = (first + last) // 2
if a_list[mid] == n:
return True
else:
if n < a_list[mid]:
last = mid - 1
else:
first = mid + 1
return False
```

Sua função **`binary_search`** recebeu dois argumentos, **`a_list`** e **`n`** (o número-alvo):

```
def binary_search(a_list, n):
```

Você usou as variáveis **`first`** e **`last`** para registrar o começo e o fim da lista que está pesquisando. Inicialmente, atribuiu o valor 0 à variável **`first`**. Em seguida, atribuiu o tamanho da lista menos 1 à variável **`last`**. Você alterará o valor dessas variáveis ao dividir **`a_list`** em segmentos que diminuirão cada vez mais:

```
first = 0
last = len(a_list) - 1
```

O loop do seu algoritmo continuará sendo executado enquanto existirem itens na lista:

```
while last >= first:
```

Dentro do loop, você localizou o ponto intermediário de **`a_list`** somando

`first` e `last` e dividindo por 2:

```
mid = (first + last) // 2
```

A barra dupla é o **operador** denominado **divisão de piso** (floor division), que retorna o valor inteiro da divisão, arredondado para baixo. Por exemplo, 7 dividido por 2 é 3,5, mas, com esse operador, o resultado é 3. Como você usou operador `//`, logo `mid` será sempre um número inteiro porque os índices são sempre números inteiros.

Em seguida, você usou uma instrução condicional para verificar se o elemento do ponto intermediário da lista é o procurado. Se for, você retornará `True` porque encontrou o número que está procurando:

```
if a_list[mid] == n:
return True
```

Se o item do ponto intermediário de sua lista não for o item-alvo, você verificará se o item-alvo é maior ou menor do que o valor do ponto intermediário. Se o item-alvo for menor do que o valor do ponto intermediário, você atribuirá à variável `last` o valor do ponto intermediário menos 1, descartando a metade superior da lista do processamento posterior:

```
if n < a_list[mid]:
last = mid - 1
```

Se o item-alvo for maior do que o valor do ponto intermediário, você configurará o valor de `first` com o ponto intermediário mais um, descartando a metade inferior da lista do processamento posterior:

```
else:
first = mid + 1
```

Seu loop será, então, repetido em um segmento menor da lista que você criou usando as variáveis `first` e `last`:

```
mid = (first + last) // 2
```

Quando o loop terminar, a função retornará `False` porque se você foi até o fim da função, o número não faz parte do argumento iterável `a_list`:

```
return False
```

Quando usar a busca binária

Uma busca binária demanda tempo O(log *n*). É mais eficiente do que a busca linear porque não precisamos pesquisar uma lista inteira. Em vez disso, você pode descartar segmentos inteiros da lista sem pesquisá-los. A eficiência da busca binária faz grande diferença quando lidamos com grandes quantidades de dados. Por exemplo, suponhamos que você estivesse pesquisando uma lista com um milhão de números. Se executasse uma busca linear, esta poderia demandar um milhão de etapas para concluir a pesquisa. Por outro lado, com uma busca binária logarítmica, levaria apenas 20 etapas.

Examinaremos com mais detalhes o que significa um algoritmo ser logarítmico. A **exponenciação** é uma operação matemática escrita como *bn* (ou `b**n` em Python), na qual multiplicamos um número *b* por si mesmo *n* vezes. Nessa equação, o número *b* chama-se **base** e o número *n* é o **expoente**. O processo de exponenciação significa elevar *b* à potência *n*. Por exemplo, 2**2 = 2*2, 2**3 = 2*2*2 e assim por diante. Um **logaritmo** é a potência à qual devemos elevar um número para produzir outro número. Em outras palavras, é o inverso da exponenciação. Por exemplo, um logaritmo pode mostrar quantas vezes teríamos de multiplicar 2 por si mesmo para obter 8. Em notação matemática, essa questão seria representada por $\log_2(8)$. A solução para $\log_2(8)$ é 3 porque temos de multiplicar 2 por si mesmo 3 vezes para obter 8 (Figura 3.6).

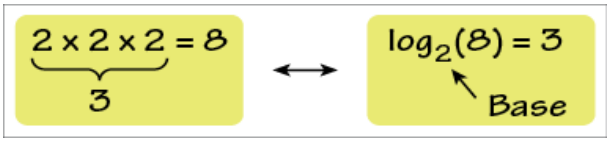

Figura 3.6: Notação exponencial versus notação logarítmica.

Em uma busca binária, na primeira vez que você dividir sua lista em 2, sobrarão *n*/2 itens nela. Após a segunda iteração, sobrarão *n*/2/2 itens e, após a terceira, *n*/2/2/2 itens. Explicando de outra forma, após sua primeira iteração em uma busca binária, haverá *n*/2*1 itens e, após a terceira iteração, *n*/2**3 itens. Logo, em termos mais gerais, após *x* iterações, você terá *n*/2***x* itens em sua lista.

Você pode usar um logaritmo para determinar quantas iterações uma busca binária demandará para encontrar um número em uma lista no cenário de pior caso. Por exemplo, suponhamos que você tivesse uma lista com 100 números e quisesse saber quantas iterações uma busca binária demandará para descobrir se um número não está nela. Para responder a isso, é preciso encontrar n em $2^{**}n = 100$, que é o mesmo que $\log_2(100)$. Você pode resolver essa equação intuitivamente por adivinhação. Poderia começar com um palpite de que n é 5, mas 2^{**5} é 32, que é muito baixo. Continuaria, então, tentando adivinhar: 2^{**6} é 64, que também é muito baixo, e 2^{**7} é 128, que é maior do que 100 e, portanto, é sua resposta. Em outras palavras, se você executar uma busca binária em uma lista com 100 itens e o item não estiver nela, seu algoritmo precisará de sete etapas para determinar isso, ou seja, $100/2/2/2/2/2/2/2 < 1$.

Quando você executar uma busca binária, esta dividirá sua lista pela metade a cada iteração, o que significa que o logaritmo que está descrevendo seu tempo de execução é de base 2. No entanto, na notação big O, a base de um logaritmo não importa porque você pode alterá-la multiplicando o logaritmo por uma constante. Os detalhes matemáticos não fazem parte do escopo deste livro, mas o que é importante saber é que a base do logaritmo não importa na notação big O. O importante é se um algoritmo é logarítmico, o que costuma ocorrer quando reduz a quantidade de cálculos pela metade ou por outra quantia significativa a cada iteração.

Pela eficiência da busca binária, se você tiver dados ordenados que precisar pesquisar, poderá ser melhor usá-la. Contudo, mesmo se tiver dados não ordenados, talvez valerá a pena classificá-los para se beneficiar dela. Por exemplo, se tiver uma lista grande e estiver pretendendo fazer muitas pesquisas, poderá ser benéfico ordenar seus dados uma única vez para acelerar enormemente cada uma das pesquisas que fará no futuro.

Como para a busca linear, o Python tem um módulo interno para a execução de uma busca binária, que você deve usar quando escrever aplicações do mundo real. A chave para escrever uma busca binária usando as ferramentas internas do Python é utilizar a função

`bisect_left` do módulo `bisect`, que encontra o índice de um elemento existente em uma lista ordenada empregando uma busca binária:

```
from bisect import bisect_left
sorted_fruits = ['apple', 'banana', 'orange', 'plum']
bisect_left(sorted_fruits, 'banana')
```

```
>> 1
```

Nesse caso, `bisect_left` retornou 1 porque `'banana'` está no índice 1 em `sorted_fruits`. Se o item que você está procurando não estiver em seu iterável ordenado, `bisect_left` retornará onde ele estaria se estivesse presente.

```
from bisect import bisect_left
sorted_fruits = ['apple', 'banana', 'orange', 'plum']
bisect_left(sorted_fruits, 'kiwi')
```

```
>> 2
```

Como você pode ver, `'kiwi'` não está em seu iterável ordenado, mas se estivesse, ocuparia o índice 2.

Já que `bisect_left` informa onde um item deveria estar caso não esteja presente, para verificar se um item se encontra em um iterável, você precisa ver se o índice existe dentro do iterável (`bisect` poderia retornar uma posição fora do iterável) e se o item do índice retornado por `bisect_left` é o valor procurado. Veja como usar `bisect_left` para executar uma busca binária em Python:

```
from bisect import bisect_left
def binary_search(an_iterable, target):
index = bisect_left(an_iterable, target)
if index <= len(an_iterable) and an_iterable[index] == target:
return True
return False
```

Se `bisect_left` retornar um índice existente no iterável e esse índice contiver o alvo, você retornará `True` porque o item existe no iterável. Caso contrário, não estará presente e você retornará `False`.

Procurando caracteres

Você sabe como procurar caracteres em uma lista usando ferramentas de

busca linear e binária internas do Python. No entanto, se precisasse escrever uma busca linear ou binária a partir do zero para procurar caracteres em vez de números? Para saber como procurar caracteres, você precisa conhecer melhor como um computador os armazena.

Um **conjunto de caracteres** é um mapeamento entre caracteres e números binários. Os cientistas da computação usam a codificação de caracteres para implementar diferentes conjuntos de caracteres. No **ASCII** (American Standard Code for Information Interchange ou Código Padrão Americano para o Intercâmbio de Informação), o computador mapeia cada letra do alfabeto para um número de sete bits. A Figura 3.7 mostra o relacionamento entre números binários e os diferentes caracteres no idioma inglês.

Por exemplo, o valor ASCII de *A* é 1000001 em binário (você aprenderá mais sobre os números binários posteriormente no livro) e 65 na base 10 (o sistema numérico que usamos diariamente). O dígito binário para *b* é 01100010. Letras maiúsculas, letras minúsculas, símbolos de pontuação, numerais e vários caracteres de controle que indicam ações, como quebra de página e nova linha, todos esses elementos têm códigos ASCII. Existem códigos ASCII para o intervalo de 0 a 9 porque os números de 0 a 9 na tabela ASCII não são valores numéricos. São caracteres para fins não matemáticos, como para expressar os numerais do endereço residencial 26 Broadway Street, Nova York. Já que o ASCII mapeia cada caractere para um número binário de 7 bits, só pode representar um máximo de 128 caracteres diferentes (2^7 é 128). No entanto, a maioria dos computadores estende o ASCII para 8 bits para que possa representar 256 caracteres.

Dec	Bin	Char	Dec	Bin	Char	Dec	Bin	Char	Dec	Bin	Char
0	0000 0000	[NUL]	32	0010 0000	space	64	0100 0000	@	96	0110 0000	`
1	0000 0001	[SOH]	33	0010 0001	!	65	0100 0001	A	97	0110 0001	a
2	0000 0010	[STX]	34	0010 0010	"	66	0100 0010	B	98	0110 0010	b
3	0000 0011	[ETX]	35	0010 0011	#	67	0100 0011	C	99	0110 0011	c
4	0000 0100	[EOT]	36	0010 0100	$	68	0100 0100	D	100	0110 0100	d
5	0000 0101	[ENQ]	37	0010 0101	%	69	0100 0101	E	101	0110 0101	e
6	0000 0110	[ACK]	38	0010 0110	&	70	0100 0110	F	102	0110 0110	f
7	0000 0111	[BEL]	39	0010 0111	'	71	0100 0111	G	103	0110 0111	g
8	0000 1000	[BS]	40	0010 1000	(	72	0100 1000	H	104	0110 1000	h
9	0000 1001	[TAB]	41	0010 1001	)	73	0100 1001	I	105	0110 1001	i
10	0000 1010	[LF]	42	0010 1010	*	74	0100 1010	J	106	0110 1010	j
11	0000 1011	[VT]	43	0010 1011	+	75	0100 1011	K	107	0110 1011	k
12	0000 1100	[FF]	44	0010 1100	,	76	0100 1100	L	108	0110 1100	l
13	0000 1101	[CR]	45	0010 1101	-	77	0100 1101	M	109	0110 1101	m
14	0000 1110	[SO]	46	0010 1110	.	78	0100 1110	N	110	0110 1110	n
15	0000 1111	[SI]	47	0010 1111	/	79	0100 1111	O	111	0110 1111	o
16	0001 0000	[DLE]	48	0011 0000	0	80	0101 0000	P	112	0111 0000	p
17	0001 0001	[DC1]	49	0011 0001	1	81	0101 0001	Q	113	0111 0001	q
18	0001 0010	[DC2]	50	0011 0010	2	82	0101 0010	R	114	0111 0010	r
19	0001 0011	[DC3]	51	0011 0011	3	83	0101 0011	S	115	0111 0011	s
20	0001 0100	[DC4]	52	0011 0100	4	84	0101 0100	T	116	0111 0100	t
21	0001 0101	[NAK]	53	0011 0101	5	85	0101 0101	U	117	0111 0101	u
22	0001 0110	[SYN]	54	0011 0110	6	86	0101 0110	V	118	0111 0110	v
23	0001 0111	[ETB]	55	0011 0111	7	87	0101 0111	W	119	0111 0111	w
24	0001 1000	[CAN]	56	0011 1000	8	88	0101 1000	X	120	0111 1000	x
25	0001 1001	[EM]	57	0011 1001	9	89	0101 1001	Y	121	0111 1001	y
26	0001 1010	[SUB]	58	0011 1010	:	90	0101 1010	Z	122	0111 1010	z
27	0001 1011	[ESC]	59	0011 1011	;	91	0101 1011	[	123	0111 1011	{
28	0001 1100	[FS]	60	0011 1100	<	92	0101 1100	\	124	0111 1100	l
29	0001 1101	[GS]	61	0011 1101	=	93	0101 1101	]	125	0111 1101	}
30	0001 1110	[RS]	62	0011 1110	>	94	0101 1110	^	126	0111 1110	~
31	0001 1111	[US]	63	0011 1111	?	95	0101 1111	_	127	0111 1111	[DEL]

Figura 3.7: Tabela ASCII.

Embora você possa representar os 256 caracteres do alfabeto latino com o ASCII, este não suporta caracteres suficientes para lidar com o texto de outros sistemas de escrita, como o japonês ou o mandarim. Para resolver esse problema, cientistas da computação desenvolveram o conjunto de caracteres **Unicode**. **Codificar caracteres** significa atribuir um número aos caracteres para representação digital. O **UTF-8** é um dos métodos de codificação de caracteres que os cientistas da computação usam para implementar o conjunto de caracteres Unicode.

Em vez de usar 7 ou 8 bits como o ASCII, o UTF-8 usa até 32 bits para codificar cada caractere, o que permite representar mais de um milhão de caracteres. O UTF-8 é compatível com o ASCII porque usa a mesma representação em bits do alfabeto latino. Por exemplo, tanto o ASCII quanto o UTF-8 representam um *A* maiúsculo com 1000001.

Você pode usar a função interna `ord()` do Python para obter o valor ASCII de um caractere:

```
print(ord('a'))
>> 97
```

Como você pode ver, o valor ASCII para *a* é 97 (na base 10).

A função `ord()` é útil quando precisamos trabalhar diretamente com os códigos ASCII subjacentes de diferentes caracteres. Para modificar a busca binária que você escreveu anteriormente para procurar caracteres, você tem de obter e comparar os valores ASCII dos caracteres. A cada passagem pelo loop, você verificará se o código ASCII de cada caractere é mais alto, mais baixo ou igual ao código ASCII do caractere que está procurando. Em vez de mostrar a solução aqui, desafio-o a codificá-la no fim deste capítulo.

Agora, você sabe como a busca linear e a busca binária funcionam e quando as usar se estiver procurando dados. Embora a busca binária seja muito eficiente, não é a maneira mais rápida de procurar dados. Na Parte II, você aprenderá como procurar dados usando uma tabela hash e saberá por que é o tipo de busca mais eficiente.

Vocabulário

algoritmo de busca: algoritmo que procura dados em um conjunto de dados.

ASCII: conjunto de caracteres do American Standard Code for Information Interchange.

base: o *b* da equação de exponenciação (b^n).

busca binária: outro algoritmo para a busca de um número em uma lista, mais rápido do que uma busca linear.

busca linear: algoritmo de busca que percorre cada elemento de um conjunto de dados e o compara com o número-alvo.

codificação de caracteres: atribuir um número a caracteres para representação digital.

conjunto de caracteres: mapeamento entre caracteres e números binários.

conjunto de dados: uma coleção de dados.

dados ordenados: dados organizados de maneira significativa.

expoente: o *n* da equação de exponenciação (b^n).

exponenciação: operação matemática expressa como b^n (ou `b**n` em Python), na qual multiplicamos um número *b* por si mesmo *n* vezes.

logaritmo: potência à qual devemos elevar um número para produzir outro número.

operador // (floor division – divisão de piso): operador que retorna o valor inteiro da divisão, arredondado para baixo.

Unicode: conjunto de caracteres que pode armazenar mais caracteres do que o ASCII.

UTF-8: um dos métodos de codificação de caracteres que os cientistas da computação usam para implementar o conjunto de caracteres Unicode.

Desafio

1. Dada uma lista de palavras em ordem alfabética, escreva uma função que execute a busca binária de uma palavra e retorne se ela está na lista.

CAPÍTULO 4

Algoritmos de ordenação

Acho que a ordenação por bolha (bubble sort) seria o caminho errado a tomar.

– Barack Obama

Como cientista da computação, além de procurar dados, com frequência você terá de ordená-los. **Ordenar dados** significa organizá-los de maneira significativa. Por exemplo, se você tivesse uma lista de números, poderia ordená-los do menor para o maior (ordem crescente). Ou suponhamos que você estivesse desenvolvendo uma aplicação que registrasse os livros que cada usuário leu. Em uma aplicação assim, talvez você queira permitir que o usuário veja seus livros ordenados de diferentes maneiras. Poderia dar a ele a opção de ver os livros ordenados do mais curto ao mais longo, do mais antigo ao mais recente ou do mais recente ao mais antigo.

Existem muitos algoritmos de ordenação que podem ajudá-lo a ordenar dados e cada um tem vantagens e desvantagens. Por exemplo, alguns algoritmos de ordenação funcionam melhor em situações específicas, como no caso de um iterável estar quase ordenado. Neste capítulo, você conhecerá a ordenação por bolha (bubble sort), a ordenação por inserção (insertion sort) e a ordenação por intercalação (merge sort). Outras ordenações populares são a ordenação rápida (quick sort), a ordenação de Donald Shell (shell sort) e a ordenação por heap (heap sort). Há vários algoritmos de ordenação que raramente você vai usar. Logo, após ensinar algumas ordenações, passarei o restante do capítulo mostrando como usar as funções internas do Python para ordenar dados, algo que você empregará com frequência em programação no mundo real.

Quando estiver desenvolvendo programas no mundo real, quase sempre

será mais aconselhável usar as funções de ordenação internas da sua linguagem de programação. Você não deve implementar os algoritmos de ordenação clássicos discutidos aqui (exceto em raras situações), porque linguagens de programação modernas, como Python, têm funções de ordenação internas mais rápidas. No entanto, conhecer alguns dos algoritmos de ordenação clássicos o ajudará a entender melhor a complexidade de tempo e ensinará conceitos que você usará em situações que não sejam de ordenação, como a etapa de intercalação de uma ordenação por intercalação.

Ordenação por bolha (bubble sort)

A **ordenação por bolha** é um algoritmo de ordenação no qual percorremos uma lista de números, comparamos cada número com o número seguinte e os trocamos se estiverem fora de ordem. Os cientistas da computação a chamam de ordenação por bolha porque os números de valores mais altos flutuam para o fim da lista e os números de valores mais baixos passam para o início da lista à medida que o algoritmo avança.

Digamos que você tivesse a lista a seguir:

```
[32, 1, 9, 6]
```

Primeiro, você compara 32 e 1:

```
[32, 1, 9, 6]
```

Trinta e dois é maior, logo você faz a troca:

```
[1, 32, 9, 6]
```

Em seguida, você compara 32 e 9:

```
[1, 32, 9, 6]
```

Trinta e dois é maior, então você faz a troca novamente:

```
[1, 9, 32, 6]
```

Para concluir, você compara 32 e 6:

```
[1, 9, 32, 6]
```

Mais uma vez, você faz a troca:

```
[1, 9, 6, 32]
```

Como você pode ver, 32 flutuou para o fim da lista. No entanto, sua lista ainda não está em ordem porque 9 e 6 não estão nos locais corretos. Logo, o algoritmo começa do início novamente e compara 1 e 9:

```
[1, 9, 6, 32]
```

Nada acontece porque 1 não é maior do que 9. Em seguida, ele compara 9 e 6:

```
[1, 9, 6, 32]
```

Nove é maior do que 6, portanto você faz a troca e a lista agora está em ordem:

```
[1, 6, 9, 32]
```

Em uma ordenação por bolha, o número maior passa para o fim da lista no final da primeira iteração do algoritmo, mas se o número menor estiver no fim, serão necessárias várias iterações para o algoritmo movê-lo para o começo da lista. Nesse exemplo, 32 entrou no fim da lista após uma única iteração. Contudo, suponhamos que a lista começasse assim:

```
[32, 6, 9, 1]
```

Nesse caso, serão necessárias quatro iterações para mover o 1 do final para o início da lista.

Pode ser útil você usar um visualizador de ordenação por bolha para entender melhor como esse algoritmo funciona. Existem muitos visualizadores de ordenação por bolha disponíveis na internet, os quais podem ajudá-lo a compreender melhor como o algoritmo opera. Recomendo fazer isso para os algoritmos de ordenação que você conhecerá neste capítulo.

Veja como escrever um algoritmo de ordenação por bolha em Python:

```
def bubble_sort(a_list):
list_length = len(a_list) - 1
for i in range(list_length):
for j in range(list_length):
if a_list[j] > a_list[j + 1]:
a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
return a_list
```

Sua função `bubble_sort` recebeu uma lista de números chamada `a_list` como parâmetro:

```
def bubble_sort(a_list):
```

Dentro da função, você está obtendo o tamanho da lista, diminuindo 1 e salvando o resultado em `list_length` para controlar o número de iterações que seu algoritmo executará:

```
list_length = len(a_list) - 1
```

A função tem dois loops aninhados para você percorrer a lista e fazer comparações:

```
for i in range(list_length):
for j in range(list_length)
```

Dentro do loop `for` interno, você usou uma instrução `if` para comparar o número atual com o número que vem depois dele, adicionando 1 ao índice do número atual:

```
if a_list[j] > a_list[j + 1]:
```

Esta linha de código é o número atual:

```
a_list[j]
```

Esta é o número seguinte da lista:

```
a_list[j + 1]
```

Se o número atual for maior do que o número seguinte, você os trocará de posição. A sintaxe Python a seguir permite trocar dois itens sem carregar um deles em uma variável temporária:

```
a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
```

Todas as comparações do algoritmo ocorrem dentro do loop interno. O loop externo só existe para manter o algoritmo sendo executado por quantas trocas forem necessárias para colocar a lista em ordem. Veja, por exemplo, a lista do começo da seção:

```
[32, 1, 9, 6]
```

Após uma única iteração do loop interno, a lista ficou assim:

```
[1, 9, 6, 32]
```

Lembre-se, entretanto, de que essa lista não está em ordem. Se você só

tivesse o loop interno, o algoritmo terminaria prematuramente e a lista não ficaria em ordem. Por isso, você precisa do loop externo: para executar o loop interno do algoritmo desde o início e repeti-lo até a lista estar em ordem.

Você pode melhorar a eficiência da ordenação por bolha incluindo `- i` no segundo loop `for`. Desta forma, o loop interno não comparará os dois últimos números na primeira passagem; na segunda passagem, não comparará os três últimos números e assim por diante.

```
def bubble_sort(a_list):
list_length = len(a_list) - 1
for i in range(list_length):
for j in range(list_length - i):
if a_list[j] > a_list[j + 1]:
a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
return a_list
```

Você não precisa fazer essas comparações porque os números mais altos flutuam para o fim da lista. No exemplo do começo desta seção, você viu que 32 flutuou para o fim da lista após a primeira iteração da ordenação. Isso significa que o número maior estará no fim da lista após a primeira iteração, o segundo número maior estará na penúltima posição após a segunda iteração etc., ou seja, você não precisa comparar os outros números com eles e pode encerrar seu loop mais cedo. Por exemplo, examine a lista do começo desta seção:

```
[32, 1, 9, 6]
```

Após uma iteração completa do loop interno, ela ficará assim:

```
[1, 9, 6, 32]
```

Após uma iteração, o número maior passou para o fim da lista, logo você não precisa comparar os números com 32, porque sabe que ele é o maior número da lista.

Na segunda iteração do loop interno, o segundo valor maior passará para sua posição final como penúltimo etc.

```
[1, 6, 9, 32]
```

Portanto, a cada iteração do loop interno, o loop terminará mais cedo.

Você também pode tornar uma ordenação por bolha mais eficiente, adicionando uma variável que registre se seu algoritmo fez alguma troca no loop interno. Se você percorrer um loop interno sem trocas, sua lista estará ordenada e será possível sair do loop e retornar a lista sem nenhum processamento adicional.

```
def bubble_sort(a_list):
list_length = len(a_list) - 1
for i in range(list_length):
no_swaps = True
for j in range(list_length - i):
if a_list[j] > a_list[j + 1]:
a_list[j], a_list[j + 1] = a_list[j + 1], a_list[j]
no_swaps = False
if no_swaps:
return a_list
return a_list
```

Nesse caso, você adicionou uma variável chamada `no_swaps` que começa como `True` no início do loop interno. Se você trocar dois números de lugar dentro do loop interno, ela terá o valor `False`. Se percorrer o loop interno e `no_swaps` continuar sendo `True`, a lista estará ordenada e você poderá encerrar seu algoritmo. Essa pequena alteração fará sua ordenação por bolha ser executada de maneira significativamente mais rápida quando uma lista estiver inicialmente quase ordenada.

Quando usar a ordenação por bolha

Nessa implementação da ordenação por bolha, seu algoritmo está ordenando números, mas você também pode escrever uma ordenação por bolha (ou de qualquer outro tipo) que ordene strings. Por exemplo, você poderia modificar uma ordenação por bolha para ordenar strings alfabeticamente pela primeira letra de cada palavra.

A principal vantagem da ordenação por bolha é a facilidade de implementação, o que a torna um bom ponto de partida para o ensino de algoritmos de ordenação. Já que a ordenação por bolha depende de dois loops `for` aninhados, sua complexidade de tempo é O(n**2), o que significa que embora possa ser aceitável para pequenos conjuntos de

dados, não é uma opção eficiente para conjuntos de dados maiores.

A ordenação por bolha também é estável. Uma **ordenação estável** é aquela que não interfere em nenhuma sequência, a não ser na especificada pela chave da ordenação. Por exemplo, suponhamos que você tivesse um banco de dados contendo registros para esses quatro animais:

```
Akita
Urso
Tigre
Albatroz
```

Se você ordenar pela primeira letra, esta será a saída esperada:

Ordenação estável

```
Akita
Albatroz
Urso
Tigre
```

Essa saída é um exemplo de ordenação estável porque Akita e Albatroz estão na mesma ordem da lista original, ainda que a ordenação só tenha levado em consideração a primeira letra de cada nome. Uma ordenação instável poderia romper a ordem original dos nomes dos dois animais que começam com *A* e Albatroz viria antes de Akita, embora Akita venha antes de Albatroz na lista original.

Ordenação instável

```
Albatroz
Akita
Urso
Tigre
```

Em outras palavras, em uma ordenação estável, quando existem duas chaves iguais, os itens mantêm sua ordem original.

Apesar de sua estabilidade, como a ordenação por bolha tem complexidade O(n**2) e existem outras ordenações mais eficientes (que

veremos em seguida), provavelmente você não verá ninguém usando-a fora da sala de aula.

Ordenação por inserção (insertion sort)

Uma **ordenação por inserção** é um algoritmo de ordenação no qual ordenamos os dados como ordenaríamos as cartas de um baralho. Primeiro, dividimos uma lista de números em duas metades: uma metade esquerda ordenada e uma metade direita não ordenada. Em seguida, classificamos a metade esquerda da mesma forma que ordenaríamos uma mão repleta de cartas. Por exemplo, quando ordenamos uma mão com cinco cartas na ordem crescente, percorremos as cartas uma a uma, inserindo cada carta à direita das cartas menores que ela.

Veja como um algoritmo de ordenação por inserção funciona em uma lista com quatro elementos: 6, 5, 8 e 2. O algoritmo começa com o segundo item da lista, que é 5:

```
[6, 5, 8, 2]
```

Em seguida, você compara o item atual com o anterior da lista. Seis é maior do que 5, logo você os troca de lugar:

```
[5, 6, 8, 2]
```

Agora, a metade esquerda da lista está ordenada, mas a metade direita não:

```
[5, 6, 8, 2]
```

Você passa, então, para o terceiro item da lista. Seis não é maior do que 8, portanto você não troca 8 e 6:

```
[5, 6, 8, 2]
```

Já que a metade esquerda da lista está ordenada, você não precisa comparar 8 e 5:

```
[5, 6, 8, 2]
```

Em seguida, você compara 8 e 2:

```
[5, 6, 8, 2]
```

Já que 8 é maior do que 2, você percorre cada item da metade esquerda

ordenada da lista, comparando 2 com cada número até este chegar à posição inicial e a lista inteira estar ordenada:

```
[2, 5, 6, 8]
```

Veja como codificar uma ordenação por inserção em Python:

```
def insertion_sort(a_list):
for i in range(1, len(a_list)):
value = a_list[i]
while i > 0 and a_list[i - 1] > value:
a_list[i] = a_list[i - 1]
i = i - 1
a_list[i] = value
return a_list
```

Você começou definindo uma função `insertion_sort` que recebe uma lista de números como entrada:

```
def insertion_sort(a_list):
```

A função usa um loop `for` para percorrer cada item da lista. Em seguida, usa um loop `while` para as comparações que o algoritmo faz quando da inclusão de um novo número no lado esquerdo classificado da lista:

```
for i in range(1, len(a_list)):
...
while i > 0 and a_list[i - 1] > value:
...
```

O loop `for` começa com o segundo item da lista (índice 1). Dentro do loop, você está registrando o número atual na variável `value`:

```
for i in range(1, len(a_list)):
value = a_list[i]
```

O loop `while` move os itens da metade direita não ordenada da lista para a metade esquerda ordenada. Continuará fazendo isso enquanto duas condições forem verdadeiras: `i` deve ser maior do que 0 e o item anterior da lista deve ser maior do que o item que vem depois dele. A variável `i` deve ser maior do que 0 porque o loop `while` compara dois números. Se `i` for 0, isso significará que o algoritmo está no primeiro item da lista e não haverá um número anterior com o qual o comparar:

```
while i > 0 and a_list[i - 1] > value:
```

Seu loop `while` só será executado se o número de `value` for menor do que o item anterior da lista. Dentro do loop `while`, você está movendo o valor maior para a direita na lista. Em seguida, o algoritmo tenta descobrir onde deve colocar o valor menor na metade esquerda ordenada. Você decrementou `i` para que o loop possa fazer outra comparação e ver se o número menor precisa ser deslocado ainda mais para a esquerda:

```
while i > 0 and a_list[i - 1] > value:
a_list[i] = a_list[i - 1]
i = i - 1
```

Quando o loop `while` terminar, você inserirá o número atual de `value` na posição correta da metade esquerda ordenada da lista:

```
a_list[i] = value
```

Examinaremos como seu algoritmo de ordenação por inserção funciona com a lista a seguir:

```
[6, 5, 8, 2]
```

Na primeira iteração do loop `for`, `i` é 1 e `value` é 5:

```
for i in range(1, len(a_list)):
value = a_list[i]
```

O código em negrito a seguir é `True` porque `i` > 0 e 6 > 5, logo loop `while` será executado:

```
while i > 0 and a_list[i - 1] > value:
a_list[i] = a_list[i - 1]
i = i - 1
a_list[i] = value
```

Este código:

```
while i > 0 and a_list[i - 1] > value:
a_list[i] = a_list[i - 1]
i = i - 1
a_list[i] = value
```

alterará a lista deste formato:

```
[6, 5, 8, 2]
```

para este:

```
[6, 6, 8, 2]
```

Este código:

```
while i > 0 and a_list[i - 1] > value:
a_list[i] = a_list[i - 1]
i = i - 1
a_list[i] = value
```

está subtraindo 1 de `i`. Agora, a variável `i` é 0, o que significa que o loop `while` não será executado novamente:

```
while i > 0 and a_list[i - 1] > value:
a_list[i] = a_list[i - 1]
i = i - 1
a_list[i] = value
```

Em seguida, esta linha de código:

```
while i > 0 and a_list[i - 1] > value:
a_list[i] = a_list[i - 1]
i = i - 1
a_list[i] = value
```

alterará sua lista deste formato:

```
[6, 6, 8, 2]
```

para este:

```
[5, 6, 8, 2]
```

Você ordenou os dois primeiros itens da lista e tudo o que o algoritmo tem de fazer é repetir o mesmo processo para ordenar a metade direita que contém 8 e 2.

Quando usar a ordenação por inserção

Como a ordenação por bolha, a ordenação por inserção é estável e tem complexidade O(*n***2), logo não é muito eficiente. No entanto, ao contrário do que ocorre com a ordenação por bolha, cientistas da computação usam ordenações por inserção em aplicações do mundo real. Por exemplo, uma ordenação por inserção pode ser uma opção eficiente se você tiver dados quase classificados. Quando uma lista está ordenada ou quase ordenada, a complexidade de tempo da ordenação por inserção é O(*n*), o que é muito eficiente.

Suponhamos que você tivesse a lista a seguir:

```
[1, 2, 3, 4, 5, 7, 6]
```

Como você pode ver, todos os números estão ordenados, exceto os dois últimos. Já que o segundo loop (o loop `while`) de uma ordenação por inserção só é executado quando dois números estão fora de ordem, a ordenação dessa lista quase ordenada usando a ordenação por inserção só levará oito etapas, o que é linear porque o segundo loop só precisa ser executado uma vez.

Ordenação por intercalação (merge sort)

Uma **ordenação por intercalação** é um algoritmo de ordenação recursivo que divide continuamente uma lista pela metade até existir(em) uma ou mais listas contendo um único item e, em seguida, combina-as novamente na ordem correta. Veja como funciona. Primeiro, usamos a recursão para dividir continuamente a lista pela metade até a lista original tornar-se uma ou mais sublistas contendo um único número (Figura 4.1).

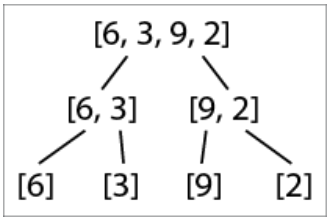

Figura 4.1: Primeira parte de uma ordenação por intercalação.

Listas que contêm um único item são consideradas ordenadas por definição. Quando você tiver listas ordenadas contendo um item cada uma, intercalará duas sublistas de cada vez comparando o primeiro item de cada lista. Intercalar listas significa combiná-las ordenadamente.

Primeiro, você intercalará [6] e [3] e, depois, [9] e [2]. Nesse caso, cada lista contém apenas um número, logo você comparará os dois números e inserirá o menor no começo da nova lista intercalada e o maior no fim. Agora, você tem duas listas ordenadas:

```
[3, 6], [2, 9]
```

Em seguida, intercalará estas duas listas:

```
# não intercaladas
[3, 6], [2, 9]
#intercalada
[]
```

Primeiro, você comparará 3 e 2. Já que 2 é menor, entrará na lista intercalada:

```
# não intercaladas
[3, 6], [9]
#intercalada
[2]
```

Agora, comparará 3 e 9. Já que 3 é menor, entrará na lista intercalada:

```
# não intercaladas
[6], [9]
#intercalada
[2, 3]
```

Para concluir, você comparará 6 e 9. Como 6 é menor, entrará na lista intercalada: Em seguida, você inserirá 9 na lista intercalada:

```
# não intercaladas
[], []
#intercalada
[2, 3, 6, 9]
```

Agora que terminou todas as intercalações, você tem uma única lista ordenada.

Veja como implementar uma ordenação por intercalação em Python:

```
def merge_sort(a_list):
if len(a_list) > 1:
mid = len(a_list) // 2
left_half = a_list[:mid]
right_half = a_list[mid:]
merge_sort(left_half)
merge_sort(right_half)
left_ind = 0
right_ind = 0
alist_ind = 0
while left_ind < len(left_half) and right_ind < len(right_half):
if left_half[left_ind] <= right_half[right_ind]:
a_list[alist_ind] = left_half[left_ind]
left_ind += 1
```

```
else:
a_list[alist_ind] = right_half[right_ind]
right_ind += 1
alist_ind += 1
while left_ind < len(left_half):
a_list[alist_ind]=left_half[left_ind]
left_ind += 1
alist_ind += 1
while right_ind < len(right_half):
a_list[alist_ind]= right_half[right_ind]
right_ind += 1
alist_ind += 1
```

Esta parte do algoritmo é responsável por dividir as listas em sublistas:

```
if len(a_list) > 1:
mid = len(a_list) // 2
left_half = a_list[:mid]
right_half = a_list[mid:]
merge_sort(left_half)
merge_sort(right_half)
left_ind = 0
right_ind = 0
alist_ind = 0
```

Esta parte é responsável por intercalar duas listas:

```
left_ind = 0
right_ind = 0
alist_ind = 0
while left_ind < len(left_half) and right_ind < len(right_half):
if left_half[left_ind] <= right_half[right_ind]:
a_list[alist_ind] = left_half[left_ind]
left_ind += 1
else:
a_list[alist_ind] = right_half[right_ind]
right_ind += 1
alist_ind += 1
while left_ind < len(left_half):
a_list[alist_ind]=left_half[left_ind]
left_ind += 1
alist_ind += 1
while right_ind < len(right_half):
a_list[alist_ind]= right_half[right_ind]
right_ind += 1
```

```
alist_ind += 1
```

A recursão é a chave desse algoritmo. Discutiremos, posteriormente, neste capítulo como a segunda metade do algoritmo que intercala listas funciona. Agora, examinaremos a função da parte recursiva chamada a chamada. Suponhamos que você começasse com a lista do início desta seção:

```
[6, 3, 9, 2]
```

Na primeira vez que você chamar sua função `merge_sort`, ela criará três variáveis:

```
# Chamada de função 1
a_list = [6, 3, 9, 2]
left_half = [6, 3]
right_half = [9, 2]
```

Você passará `a_list` para a função e este código criará as outras duas variáveis dividindo `a_list` em uma metade esquerda e uma metade direita:

```
mid = len(a_list) // 2
left_half = a_list[:mid]
right_half = a_list[mid:]
```

Em seguida, a função chama recursivamente a si própria e passa `left_half` como parâmetro:

```
merge_sort(left_half)
```

Agora, Python criará mais três variáveis:

```
# Chamada de função 2
a_list = [6, 3]
left_half = [6]
right_half = [3]
```

O item importante que precisamos entender é que a variável `left_half` da primeira chamada de função e a variável `a_list` da segunda chamada *apontam para a mesma lista*. Isso significa que quando você alterar `left_half` na segunda chamada de função, isso alterará `a_list` na primeira chamada.

Agora, o código chamará a si mesmo novamente, mas, dessa vez, `left_half` é `[6]` e o caso base interromperá as recursões:

```
if len(a_list) > 1:
```

Python passa, então, para a linha seguinte de código:

```
merge_sort(right_half)
```

A variável `right_half` é `[3]`, logo o caso base não deixará a função chamar a si própria novamente.

```
if len(a_list) > 1:
```

Python chama, então, o código de intercalação. O código intercala `left_half`, que é `[6]`, e `right_half`, que é `[3]`, e salva o resultado modificando `a_list`, que agora está ordenada:

```
# Chamada de função 2
a_list = [3, 6]
left_half = [6]
right_half = [3]
```

Como vimos anteriormente, quando você modificar `a_list` na chamada de função 2, também estará alterando a variável `left_half` na chamada de função 1.

Originalmente, o estado de Python durante a chamada de função 1 tinha esta aparência:

```
# Chamada de função 1
a_list = [6, 3, 9, 2]
left_half = [6, 3]
right_half = [9, 2]
```

No entanto, na chamada de função 2, você modificou `a_list`, o que também alterou `left_half` na chamada de função 1. Agora, o estado da chamada de função 1 ficou assim:

```
# Chamada de função 1
a_list = [3, 6, 9, 2]
left_half = [3, 6]
right_half = [9, 2]
```

Essa alteração é importante porque, como você sabe, quando usamos recursão, Python retorna a estados anteriores quando alcançamos o caso base. Como você pode ver, agora `left_half` está ordenada.

O algoritmo chama a si próprio recursivamente mais uma vez, o mesmo processo ocorre e `right_half` é ordenada. Isso significa que, agora, o

estado da função 1 é:

```
# Chamada de função 1
a_list = [3, 6, 9, 2]
left_half = [3, 6]
right_half = [2, 9]
```

Podemos ver que `right_half` está ordenada no estado da chamada de função 1. Agora, quando o algoritmo voltar ao estado da função 1, intercalará `left_half` e `right_half` que, chamadas de funções recursivas anteriores, já ordenaram. Logo, quando o algoritmo chegar a esse ponto e chamar seu código de intercalação, a função terá duas listas ordenadas, `left_half` e `right_half`, e só precisará intercalá-las para produzir a lista ordenada final.

Chegou a hora de examinarmos o código que intercala duas listas. O código de intercalação começa com três variáveis que você inicializou com 0:

```
left_ind = 0
right_ind = 0
alist_ind = 0
```

Você usará essas variáveis para registrar os índices de três listas: `left_half`, `right_half` e `a_list`.

Este código compara cada primeiro item de `left_half` com cada primeiro item de `right_half` e insere o número menor em sua posição correta em `a_list`:

```
while left_ind < len(left_half) and right_ind < len(right_half):
if left_half[left_ind] <= right_half[right_ind]:
a_list[alist_ind] = left_half[left_ind]
left_ind += 1
else:
a_list[alist_ind] = right_half[right_ind]
right_ind += 1
alist_ind += 1
```

O código terminará cada intercalação e também poderá manipular uma situação em que as duas listas que estiverem sendo intercaladas estiverem desniveladas:

```
while left_ind < len(left_half):
a_list[alist_ind] = left_half[left_ind]
left_ind += 1
alist_ind += 1
while right_ind < len(right_half):
a_list[alist_ind]= right_half[right_ind]
right_ind += 1
alist_ind += 1
```

Por exemplo, suponhamos que você estivesse intercalando estas duas listas:

```
[2], [6, 4]
```

O Python chamaria o código de intercalação com estas variáveis:

```
left_ind = 0
right_ind = 0
alist_ind = 0
a_list = [6, 3]
left_half = [6]
right_half = [3]
```

As expressões `left_ind < len(left_half)` e `right_ind < len(right_half)` serão `True`, logo você entrará no loop `while`. O código:

```
left_half[left_ind] <= right_half[right_ind]:
```

produzirá `6 <= 3`, que é `False`, portanto o código descrito a seguir será executado:

```
if left_half[left_ind] <= right_half[right_ind]:
a_list[alist_ind] = left_half[left_ind]
left_ind += 1
else:
a_list[alist_ind] = right_half[right_ind]
right_ind += 1
alist_ind += 1
```

Agora, as variáveis serão:

```
left_ind = 0
right_ind = 1
alist_ind = 1
a_list = [3, 3]
left_half = [6]
```

```
right_half = [3]
```

Na próxima passagem do loop `while`, o código a seguir não será executado porque `right_ind` não será menor do que o tamanho de `right_half`:

```
while left_ind < len(left_half) and right_ind < len(right_half):
if left_half[left_ind] <= right_half[right_ind]:
a_list[alist_ind] = left_half[left_ind]
left_ind += 1
else:
a_list[alist_ind] = right_half[right_ind]
right_ind += 1
alist_ind += 1
```

Em seguida, este código será executado porque `left_ind` é menor do que o tamanho de `left_half`:

```
while left_ind < len(left_half):
a_list[alist_ind] = left_half[left_ind]
left_ind += 1
alist_ind += 1
while right_ind < len(right_half):
a_list[alist_ind]= right_half[right_ind]
right_ind += 1
alist_ind += 1
```

Agora, suas variáveis são:

```
left_ind = 1
right_ind = 1
alist_ind = 2
a_list = [3, 6]
left_half = [6]
right_half = [3]
```

Sua lista está ordenada e a intercalação foi concluída.

Quando usar a ordenação por intercalação

A ordenação por intercalação é um exemplo de algoritmo do tipo divisão e conquista. Um **algoritmo de divisão e conquista** é aquele que divide um problema recursivamente em dois ou mais subproblemas relacionados até se tornarem suficientemente simples para serem resolvidos com facilidade.

Em uma ordenação por intercalação, dividimos uma lista em sublistas até cada sublista ter apenas um item, o que facilita a solução porque uma lista com um único item é considerada ordenada por definição. O tempo de execução da ordenação por intercalação é O(n * log n) porque embora a divisão da lista inicial em sublistas seja logarítmica, o algoritmo requer tempo linear para manipular cada item das sublistas enquanto as intercala.

Pela complexidade de tempo log-linear, a ordenação por intercalação é um dos algoritmos de ordenação mais eficientes e amplamente usados pelos cientistas da computação. Por exemplo, o Python usa a ordenação por intercalação em seus algoritmos de ordenação internos, que você conhecerá em breve. Como a ordenação por bolha e a ordenação por inserção, a ordenação por intercalação também é estável.

Você também pode aplicar o conceito de intercalação da ordenação por intercalação a situações que não sejam de ordenação. Por exemplo, suponhamos que você estivesse em uma sala de aula com 50 alunos. Cada aluno tem uma quantia em dinheiro aleatória no bolso de até 1 dólar. Qual é a melhor maneira de somar todas as quantias? A primeira resposta na qual você deve ter pensado seria o professor consultar cada aluno, perguntar quanto ele tem e fazer a soma total. Essa solução é uma busca linear e seu tempo de execução é O(n). Em vez de executarmos uma busca linear, seria mais eficiente os alunos fazerem uma intercalação. Veja como funciona: cada aluno irá até o aluno que estiver próximo a ele, pegará seu dinheiro e juntará com o seu (terá de se lembrar do novo total). Os alunos que não tiverem dinheiro deixarão a sala. Em seguida, você repetirá esse processo até haver apenas um aluno e a quantia total existente na sala tiver sido obtida. Em vez de 50 etapas, o uso de uma intercalação para a contagem da quantia em dinheiro total existente na sala demandou apenas 6 etapas.

Algoritmos de ordenação do Python

O Python tem duas funções de ordenação internas: `sorted` e `sort`. As funções de ordenação do Python usam o Timsort, um algoritmo de

ordenação híbrido que combina a ordenação por intercalação e a ordenação por inserção. Um **algoritmo de ordenação híbrido** é aquele que combina dois ou mais algoritmos que resolvem o mesmo problema, selecionando um (dependendo dos dados) ou alternando entre eles no decorrer da execução. A combinação das ordenações por inserção e por intercalação do Timsort o torna um algoritmo eficiente, por isso, em geral, usamos os algoritmos de ordenação internos do Python em vez de tentar codificar um por conta própria.

A função `sorted` do Python permite ordenar qualquer iterável, contanto que o Python possa comparar os dados existentes nele. Por exemplo, você pode chamar `sorted` com uma lista de inteiros e o Python retornará uma nova lista com os inteiros da lista original ordenados em ordem crescente:

```
a_list = [1, 8, 10, 33, 4, 103]
print(sorted(a_list))
```

```
>> [1, 4, 8, 10, 33, 103]
```

Se você chamar `sorted` com uma lista de strings, o Python retornará uma nova lista ordenada alfabeticamente pela primeira letra de cada string:

```
a_list = ["Guido van Rossum", "James Gosling", "Brendan Eich", "Yukihiro
Matsumoto"]
print(sorted(a_list))
```

```
>> ['Brendan Eich', 'Guido van Rossum', 'James Gosling', 'Yukihiro Matsumoto']
```

A função `sorted` do Python tem um parâmetro opcional chamado `reverse`, logo, se quiser ordenar seu iterável na ordem decrescente, bastará passar `reverse=True`:

```
a_list = [1, 8, 10, 33, 4, 103]
print(sorted(a_list, reverse=True))
```

```
>> [103, 33, 10, 8, 4, 1]
```

`Sorted` também tem um parâmetro chamado `key` que permite passar uma função. O Python chamará essa função em cada item do iterável e usará o resultado para ordená-lo. Por exemplo, você poderia passar a função `len`, que ordenará a lista de strings pelo tamanho:

```
a_list = ["onehundred", "five", "seventy", "two"]
print(sorted(a_list, key=len))
```

```
>> ['two', 'five', 'seventy', 'onehundred']
```

A outra função do Python para ordenação é `sort`, que tem os mesmos parâmetros opcionais de `sorted`, mas, ao contrário de `sorted`, `sort` só funciona com listas. Além disso, ao contrário de `sorted`, `sort` faz a ordenação alterando a lista original em vez de retonar uma nova. Aqui há um exemplo do uso de `sort` em uma lista:

```
a_list = [1, 8, 10, 33, 4, 103]
a_list.sort()
print(a_list)
```

```
>> [1, 4, 8, 10, 33, 103]
```

Como você pode ver, o Python ordenou os números na lista original na ordem crescente.

Vocabulário

algoritmo de divisão e conquista: algoritmo que divide recursivamente um problema em dois ou mais subproblemas relacionados até se tornarem suficientemente simples para serem resolvidos com facilidade.

algoritmo de ordenação híbrido: algoritmo que combina dois ou mais algoritmos que resolvem o mesmo problema, selecionando um (dependendo dos dados) ou alternando-se entre eles no decorrer da execução.

ordenação estável: ordenação que não interfere em nenhuma sequência, a não ser a especificada por sua chave.

ordenação por bolha: algoritmo de ordenação no qual percorremos uma lista de números, comparamos cada número com o número seguinte e os trocamos de lugar se estiverem fora de ordem.

ordenação por inserção: algoritmo de ordenação no qual classificamos os dados como ordenamos as cartas de um baralho.

ordenação por intercalação: algoritmo de ordenação recursivo que divide continuamente uma lista pela metade até haver uma ou mais listas contendo um único item e, em seguida, reúne-as novamente na ordem correta.

ordenar dados: organizar dados de maneira significativa.

Desafio

1. Pesquise e escreva um algoritmo de ordenação que não seja por bolha, por inserção nem por intercalação.

CAPÍTULO 5

Algoritmos de string

Uma das habilidades mais importantes que qualquer empreendedor deve aprender é programar um computador. Será uma habilidade crítica se você quiser iniciar uma startup de tecnologia, mas um conhecimento básico de codificação é útil mesmo em áreas tradicionais, porque os softwares estão mudando tudo.

– Redi Hoffman

Neste capítulo, você aprenderá a resolver algumas das questões mais comuns baseadas em strings apresentadas em entrevistas técnicas. Você não deve precisar encontrar anagramas em seu emprego de engenharia de software, mas aprender a detectá-los o ensinará a resolver problemas usando conceitos como o da ordenação, que é algo que será preciso executar. Além disso, outros tópicos que você aprenderá neste capítulo, como aritmética modular e compreensão de lista (*list comprehension*), serão úteis em sua programação diária.

Detecção de anagramas

Duas strings são **anagramas** quando contêm as mesmas letras, mas não necessariamente na mesma ordem (letras maiúsculas e minúsculas não importam). Por exemplo, *Car* e *arc* são anagramas. O segredo para determinar se duas strings são anagramas é ordená-las. Se as strings ordenadas forem iguais, serão anagramas. Veja como escrever um algoritmo que determina se duas strings são anagramas:

```
def is_anagram(s1, s2):
s1 = s1.replace(' ','').lower()
s2 = s2.replace(' ','').lower()
if sorted(s1) == sorted(s2):
```

```
return True
else:
return False
s1 = 'Emperor Octavian'
s2 = 'Captain over Rome'
print(is_anagram(s1,s2))
```

```
>> True
```

Anagramas podem conter várias palavras e incluir letras maiúsculas e minúsculas. Logo, você começará sua função removendo os espaços das strings e convertendo todos os caracteres em letras minúsculas:

```
s1 = s1.replace(' ','').lower()
s2 = s2.replace(' ','').lower()
```

Em seguida, você ordenará as duas strings e comparará o resultado para determinar se são iguais. Se forem, as strings serão anagramas e você retornará `True`. Caso contrário, não serão anagramas e você retornará `False`.

```
if sorted(s1) == sorted(s2):
return True
else:
return False
```

Já que seu algoritmo para detectar anagramas depende da função interna `sorted` do Python, seu tempo de execução é O(*n* log *n*).

Detecção de palíndromos

Um **palíndromo** é uma palavra igual quando lida de trás para frente ou normalmente. *Hannah*, *mom*, *wow* e *racecar* são exemplos de palíndromos. Existem várias maneiras de determinar se uma string é um palíndromo. Uma maneira é copiar a string, invertê-la e compará-la com a string original. Se forem iguais, a string será um palíndromo.

Veja como inverter uma string em Python:

```
print("blackswan"[::-1])
```

```
>> nawskcalb
```

Esta é a maneira de verificar se uma string é um palíndromo:

```
def is_palindrome(s1):
```

```
if s1.lower() == s1[::-1].lower():
return True
return False
```

Primeiro, você usará a função interna `lower` do Python para converter as duas strings em minúsculas, para que não afete sua comparação. Em seguida, usará a sintaxe de fatiamento do Python para inverter a string e compará-la com a string original:

```
if s1.lower() == s1[::-1].lower():
```

Se as duas strings forem iguais, a string será um palíndromo e você retornará `True`:

```
return True
```

Caso contrário, as strings não são palíndromos e você retornará `False`:

```
Return False
```

A parte mais lenta do algoritmo de detecção de palíndromos é a sintaxe Python para a inversão de uma lista. Já que Python tem de verificar cada item da lista para invertê-la, o tempo de execução da inversão de uma lista é O(*n*), o que faz o tempo de execução do algoritmo também ser O(*n*).

Último dígito

Outra questão comum em entrevistas é retornar o dígito da parte mais à direita de uma string. Por exemplo, dada a string `"Buy 1 get 2 free"`, sua função deve retornar o número 2.

Uma maneira elegante de resolver esse problema é usando o recurso de compreensão de lista (*list comprehension*) do Python. Uma **compreensão de lista** é a sintaxe Python para a criação de uma lista nova e alterada a partir de um iterável existente (como outra lista).

Aqui está a sintaxe de uma compreensão de lista:

```
new_list = [expression(i) for i in iterable if filter(i)]
```

`iterable` é o iterável que você está usando para criar sua nova lista. `expression(i)` é uma variável que armazena cada elemento de `iterable`. Por exemplo, nesta expressão regular, `c` contém cada caractere da string `"selftaught"`.

```
print([c for c in "selftaught"])
```

```
>> ['s', 'e', 'l', 'f', 't', 'a', 'u', 'g', 'h', 't']
```

Como você pode ver, Python retorna uma lista que contém todas as letras da string original "`selftaught`".

`filter(i)` permite fazer alterações no iterável original. Por exemplo, você pode criar um filtro que só adicione itens ao iterável se atenderem aos seus requisitos:

```
print([c for c in "selftaught" if ord(c) > 102])
```

```
>> ['s', 'l', 't', 'u', 'g', 'h', 't']
```

A função interna `ord` do Python retorna o código ASCII de uma letra. Nesse caso, você adicionou um filtro que só adiciona caracteres ao iterável se seu código ASCII for maior do que 102 (a letra *f*). Podemos ver quer a nova lista não tem as letras *e*, *f* nem *a*.

Você pode usar a função `isdigit` do Python para filtrar qualquer coisa, exceto números:

```
s = "Buy 1 get 2 free"
nl = [c for c in s if c.isdigit()]
print(nl)
```

```
>> ['1', '2']
```

Agora que você sabe como usar uma compreensão de lista para encontrar todos os dígitos de uma string, há apenas mais uma etapa para a busca do dígito da extrema direita, que é usar um índice negativo para obter o último dígito de sua nova lista:

```
s = "Buy 1 get 2 free"
nl =[c for c in s if c.isdigit()][-1]
print(nl)
```

```
>> 2
```

Primeiro, a compreensão de lista retornará uma lista com todos os dígitos da string. Em seguida, você usará um índice negativo para obter o último dígito da nova lista de números, que é o dígito da extrema direita da string original.

Como você pode ver, é possível usar uma compreensão de lista para transformar três ou quatro linhas de código em uma elegante linha única,

o que o ajudará a escrever códigos curtos e legíveis quando estiver programando profissionalmente.

Já que seu algoritmo percorre cada caractere da string para verificar se é um dígito e também inverte a lista, seu tempo de execução é O(*n*).

Cifra de César

Uma **cifra** é um algoritmo para criptografia ou descriptografia. Júlio César, o famoso general e político romano, protegia suas mensagens confidenciais usando uma cifra engenhosa. Primeiro, ele escolhia um número; depois, deslocava cada letra de acordo com esse número. Por exemplo, se ele escolhesse o número 3, a string *abc* passaria a ser *def*.

Se o deslocamento ultrapassasse o fim do alfabeto, ele voltava para a letra inicial. Por exemplo, se ele precisasse deslocar *z* em duas posições, *z* passaria a ser *b*.

A **aritmética modular** é o segredo para a codificação de uma cifra de César. É um tipo de aritmética no qual os números retornam quando alcançam um valor específico. Você já deve estar familiarizado com ela porque sabe ver as horas (Figura 5.1).

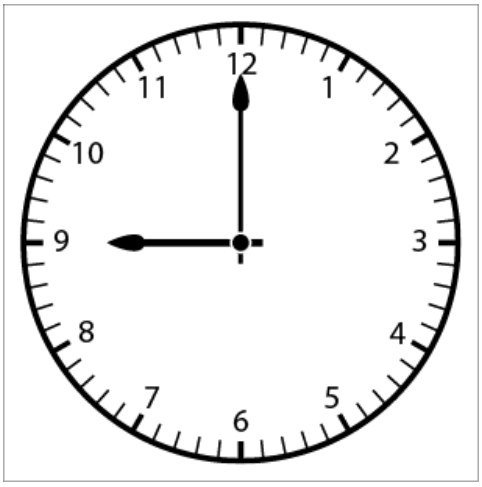

Figura 5.1: Você usa a aritmética modular para ver as horas.

Por exemplo, suponhamos que haja um voo de Nova York para Lima, Peru, que saia às 21 horas. Suponhamos que ambas as cidades tenham o mesmo fuso horário e o voo leve 8 horas. A que horas o voo chegará?

Nove mais 8 são 17, mas um relógio de 12 horas não exibe 17. Para determinar a hora da chegada, é preciso somar 9 e 8 (17) e executar a operação de módulo com 12 e o resultado.

```
17 % 12
```

Dezessete pode ser dividido por 12 uma vez e o resto é 5, o que significa que o voo chegará às 5 horas (Figura 5.2).

A aritmética modular será útil quando você estiver escrevendo algum programa que envolva tempo. Por exemplo, se estiver criando um site para processar tempos de voos, poderá usar a aritmética modular para descobrir a que horas um voo chegará.

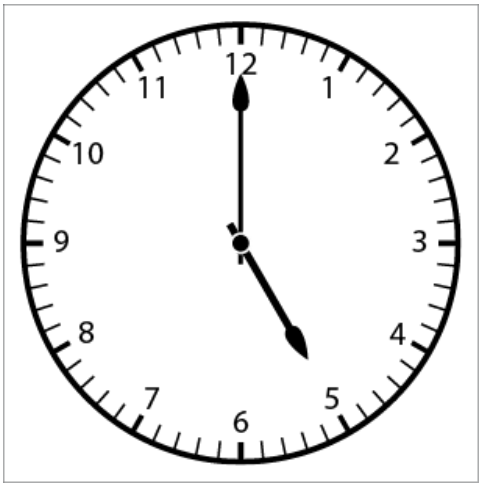

Figura 5.2: O resultado de 8 horas depois das 9 é 5.

Agora que você sabe como a aritmética modular funciona, poderá codificar uma cifra de César escrevendo uma função que recebe uma string e um número, de acordo com o qual cada letra será deslocada e exibirá uma string nova criptografada:

```
import string
def cipher(a_string, key):
uppercase = string.ascii_uppercase
lowercase = string.ascii_lowercase
encrypt = ''
for c in a_string:
if c in uppercase:
new = (uppercase.index(c) + key) % 26
encrypt += uppercase[new]
```

```
elif c in lowercase:
new = (lowercase.index(c) + key) % 26
encrypt += lowercase[new]
else:
encrypt += c
return encrypt
```

Sua função, `cipher`, recebeu dois parâmetros: `a_string`, que é a string que você deseja criptografar, e `key`, o número de posições segundo o qual cada letra será deslocada.

Você começou usando o módulo interno `string` do Python para criar duas strings que contêm todos os caracteres do alfabeto tanto em maiúsculas quanto em minúsculas:

```
import string
def cipher(a_string, key):
uppercase = string.ascii_uppercase
lowercase = string.ascii_lowercase
```

Se você fizer a exibição em maiúsculas e minúsculas, a saída será esta:

```
>> 'abcdefghijklmnopqrstuvwxyz'
>> 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Em seguida, criou a variável `encrypt`, que começa como uma string vazia, mas posteriormente conterá a string criptografada:

```
encrypt = ''
```

Você percorreu, então, a string, registrando cada caractere na variável `c`:

```
for c in a_string:
```

Se o caractere for maiúsculo, você encontrará seu índice em `uppercase` (que, lembre-se, é `ABCDEFGHIJKLMNOPQRSTUVWXYZ`). Em seguida, adicionará `key` a esse índice, o que fornecerá o novo caractere criptografado. Por exemplo, se o caractere for *A* e `key` for 2, primeiro você obterá o índice de *A* em `uppercase`, que é 0, e depois somará 2. O índice 2 em `uppercase` será a letra *C*.

No entanto, há um problema no uso desse método para a obtenção do novo caractere. O que acontecerá se você deslocar *Z* em uma ou mais posições? A letra *Z* ficará no índice 25. Se você adicionar 2 a 25, obterá o índice 27, que não existe. Já que *Z* é a última letra do alfabeto, será preciso

ir até o começo para obter o novo caractere criptografado. Nesse caso, você precisará obter o índice 2, que é *C* (índice 0 + 2).

Você resolve isso usando o módulo 26 na soma do índice inicial de cada caractere com `key`. Primeiro, você obteve o índice inicial do caractere em `uppercase`. Depois, adicionou `key` e executou a operação módulo 26.

```
if c in uppercase:
new = (uppercase.index(c) + key) % 26
```

Esse código funciona porque você está usando a aritmética modular para "voltar" sempre que alcança um valor específico. Nesse caso, você "está voltando" sempre que o índice excede 25.

Quando tiver o novo índice do caractere criptografado, você o usará para saber que caractere ele representa em `uppercase` e o armazenará na variável `encrypt`:

```
encrypt += uppercase[new]
```

Se o caractere for minúsculo, você fará a mesma coisa, mas usará `lowercase`:

```
elif c in lowercase:
new = (lowercase.index(c) + key) % 26
encrypt += lowercase[new]
```

Se o caractere não estiver em `uppercase` nem `lowercase`, isso significará que se trata de um caractere especial, logo você o adicionará a `encrypt` sem fazer alterações:

```
else:
encrypt += c
```

No fim do loop, você retornará a nova string criptografada:

```
return encrypt
```

Já que seu algoritmo só precisa percorrer cada letra da string para criptografá-la, seu tempo de execução é O(*n*).

Vocabulário

anagramas: duas strings que contêm as mesmas letras, mas não necessariamente na mesma ordem (letras maiúsculas e minúsculas não

importam).

aritmética modular: tipo de aritmética em que os números voltam ao início quando alcançam um valor específico.

cifra: algoritmo para criptografia ou descriptografia.

compreensão de lista: sintaxe Python para a criação de uma lista nova alterada a partir de um iterável existente (como outra lista).

palíndromo: palavra que é igual tanto quando lida de trás para a frente quanto quando lida normalmente.

Desafio

1. Use uma compreensão de lista para retornar uma lista com todas as palavras da relação a seguir que tenham mais de quatro caracteres: `["selftaught", "code", "sit", "eat", "programming", "dinner", "one", "two", "coding", "a", "tech"]`.

CAPÍTULO 6

Matemática

Não se preocupe muito com suas dificuldades em matemática.
Posso lhe assegurar que as minhas são ainda maiores.

– Albert Einstein

Neste capítulo, você aprenderá algumas noções básicas de matemática que o ajudarão a passar em uma entrevista técnica e a melhorar como programador. Embora procurar números primos talvez não o ajude em sua programação diária, conhecer os diferentes algoritmos que você pode usar para encontrá-los o tornará um programador melhor. Embora testar nossa habilidade no uso do operador módulo em algoritmos seja uma das "pegadinhas" mais comuns em entrevistas técnicas, esse operador também pode ser útil em aplicações do mundo real. Para concluir, o capítulo introduzirá o conceito de condições limite (boundary conditions). Se você criar aplicações sem pensar em condições limite, provavelmente produzirá um site ou uma aplicação repleto de erros inesperados, logo é importante saber quais são essas condições e como se preparar para elas.

Sistema binário

Os computadores "pensam" em binário. Um **número binário** é um número no sistema de numeração de base 2. Um **sistema de numeração** é um sistema de escrita que representa números. Na base 2, os números só têm dois dígitos: 0 e 1. Em binário, um dígito chama-se **bit**, que é a abreviação de binary digit (dígito binário). O sistema de numeração que você está acostumado a usar para contar chama-se de base 10 e tem 10 dígitos (de 0 a 9). A **base** de um sistema de numeração é o número de

dígitos que o sistema tem. Os sistemas de numeração binário e decimal não são os únicos que existem. Também existem outros sistemas de numeração, como o de base 16, chamado de *sistema hexadecimal*, que é popular entre os programadores.

Aqui estão alguns exemplos de números binários:

```
100
1000
101
1101
```

Quando olhamos para esses números, não sabemos se estão na base 2 ou na base 10. Por exemplo, o primeiro número, **`100`**, poderia ser 100 na base 10 ou 4 na base 2.

Existem várias notações que você pode usar para mostrar que um número está na base 2. Em geral, os cientistas da computação inserem um **`b`** antes de um número para indicar que ele está na base 2. Aqui, há outras maneiras de indicar que um número está na base 2:

```
100b
1000₂
%100
0b100
```

O **valor posicional** é o valor numérico que um dígito tem em razão de sua posição em um número. Por exemplo, um número de quatro dígitos tem valores posicionais que representam milhares, centenas, dezenas e unidades. O número 1452 representa um milhar, mais quatro centenas, mais cinco dezenas, mais duas unidades (Figura 6.1).

Figura 6.1: Valores posicionais do número 1452 na base 10.

No sistema decimal, cada valor posicional é uma potência de 10. O valor posicional da extrema direita é 10 elevado a 0, que é 1. O valor posicional seguinte é 10 elevado a 1, que é 10. O próximo valor posicional é 10 elevado a 2 (10 × 10), que é 100. O valor seguinte é 10 elevado a 3 (10 × 10

× 10), que é 1000 (Figura 6.2).

Figura 6.2: As potências de 10 usadas nos valores posicionais na base 10.

Você pode expressar o número 1452 como uma equação usando seus valores posicionais:

```
(1 * 10 ** 3) + (4 * 10 ** 2) + (5 * 10 ** 1) + (2 * 10 ** 0) = 1452
```

Ou pode visualizá-lo da seguinte forma:

```
1 * 10 ** 3 = 1 * 1000 = 1000 +
4 * 10 ** 2 = 4 * 100 = 400 +
5 * 10 ** 1 = 5 * 10 = 50 +
2 * 10 ** 0 = 2 * 1 = 2
_____________
1452
```

O sistema de numeração binário funciona da mesma maneira que o sistema decimal, exceto por haver apenas dois dígitos, 0 e 1, e os valores posicionais serem potências de 2, em vez de potências de 10.

O valor posicional da extrema direita é 2 elevado a zero, que é 1. O valor posicional seguinte é 2 elevado a 1, que é 2. O próximo valor posicional é 2 elevado a 2, que é 4 (2 × 2). O valor seguinte é 2 elevado a 3, que é 8 (2 × 2 × 2) (Figura 6.3).

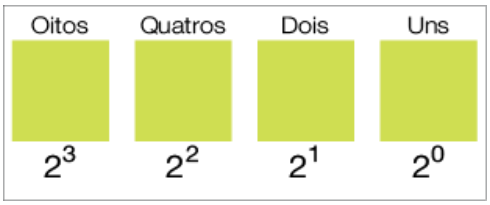

Figure 6.3: Potências de 2 usadas nos valores posicionais na base 2.

Aqui está uma equação que representa o número 1101 na base 2:

```
(1 * 2 ** 3) + (1 * 2 ** 2) + (0 * 2 ** 1)
+ (1 * 2 ** 0) =
8 + 4 + 0 + 1 = 13
```

Ou:

```
1 * 2 ** 3 = 1 * 8 = 8 +
1 * 2 ** 2 = 1 * 4 = 4 +
0 * 2 ** 1 = 0 * 2 = 0 +
1 * 2 ** 0 = 1 * 1 = 1

____
13
```

Como você pode ver, 1101 em binário é o número 13 na base 10.

No sistema decimal, quando contamos, começamos em zero: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Ao chegar aqui, ficamos sem dígitos. Para representar o próximo número, como você sabe, 10 é criado com o uso de 2 dígitos. Representamos 10 com um 1 seguido de um 0.

Quando contamos em binário, também começamos com 0.

```
0
```

Assim como no sistema decimal, o próximo número é 1.

```
1
```

Depois de 1, ficamos sem dígitos. Isso significa que você terá de usar dois dígitos para representar o número 2, da mesma forma que temos de usar dois dígitos no sistema decimal para representar o número 10.

Em binário, representamos o número 2 com um 1 e um 0:

```
10
```

O 0 representa a ausência de 1s e o 1, uma quantidade igual a 2.

Como o número 3 é representado no sistema binário?

```
11
```

O primeiro 1 significa uma quantidade igual a 1 e o segundo 1, uma quantidade igual a 2. Quando somamos 2 e 1, obtemos 3.

O próximo é o número 4 que, em binário, escreve-se assim:

```
100
```

O primeiro 0 significa a ausência de uns, o segundo 0 significa a ausência de dois e o 1 significa uma quantidade igual a 4. Some tudo e você obterá 4.

Operadores bitwise

Em geral, quando programamos e lidamos com números, trabalhamos com inteiros e números de ponto flutuante (floats) como 100 e 10,5. No entanto, em certas situações, é útil trabalhar com números binários. Por exemplo, trabalhar com números binários pode ajudá-lo a resolver problemas rapidamente como determinar se um número é uma potência de 2.

Você pode usar o método `bin` para trabalhar com números binários em Python.

```
print(bin(16))
>> 0b10000
```

Quando você exibir `bin(16)`, o resultado será `0b10000`, que é 16 em binário. Como vimos anteriormente, `0b` permite saber que o número está no sistema binário.

O **operador bitwise (bit a bit)** é um operador que podemos usar em uma expressão com dois números binários. Por exemplo, o operador bitwise AND do Python executa a aritmética booleana bit a bit. Quando os dois bits são 1 (`True`), Python retorna 1; caso contrário, retorna 0 (`False`). A lógica de cada bit no operador bitwise AND é a mesma da palavra-chave `and` em Python. Como você sabe, quando usamos a palavra-chave `and` do Python, se os dois lados da expressão forem `True`, Python retornará `True`.

```
print(1==1 and 2==2)
>> True
```

Se os dois lados forem `False`, retornará `False`.

```
print(1==2 and 2==3)
>> False
```

Se um dos lados for `True` e o outro for `False`, também retornará `False`.

```
print(1==1 and 1==2)
>> False
```

Examinaremos um exemplo de uso do operador bitwise AND. Suponhamos que você tivesse dois inteiros, 2 e 3. Dois em binário é 0b10 e 3 é 0b11. O primeiro bit de 2 é 0 e o primeiro bit de 3 é 1.

```
10 # 2
11 # 3
—
0
```

A aplicação do operador bitwise AND a esses bits produz 0 porque há um `True` e um `False`, o que retorna `False` (lembre-se: 0 é `False` e 1 é `True`). A aplicação do operador AND ao segundo conjunto de bits produz 1 porque os dois dígitos binários são `True`, logo Python retornará `True`.

```
10 # 2
11 # 3
—
10
```

Nesse caso, a operação bitwise AND produziu 0b10, que é o número 2 em binário (você aprenderá por que isso é útil em breve).

Em Python, o operador bitwise AND é o símbolo ampersand (`&`). Veja como usar esse operador com os números binários 0b10 e 0b11 em Python:

```
print(0b10 & 0b11)
>> 2
```

Não é preciso usar um operador bitwise como AND com números binários.

```
print(2 & 3)
>> 2
```

Aqui, você executou a operação bitwise AND com números decimais, mas Python usa os valores binários de 2 e 3 para executá-la.

Python também tem um operador bitwise OR, que opera bit a bit e retorna 1 quando um ou os dois bits são `True` e retorna `False` quando os dois são `False`, como ocorre com a palavra-chave `or`. Por exemplo, vejamos o que acontece quando usamos o operador bitwise OR com os números 2 e 3. A aplicação do operador aos dois primeiros bits produz 1 porque um dos bits é `True`.

```
10 # 2
11 # 3
—
```

```
1
```

Quando usamos o operador bitwise OR no segundo conjunto de bits, Python retorna 1 porque os dois bits são `True` (1).

```
10 # 2
11 # 3
—
11
```

Como você pode ver, o resultado da operação bitwise OR com 2 e 3 é 0b11, que é o número decimal 3.

Em Python, o operador bitwise OR é o símbolo pipe (`|`).

```
print(2 | 3)
```

```
>> 3
```

Os operadores binários que vimos até agora estão entre os mais comuns; no entanto, existem outros operadores binários, como o bitwise XOR, o bitwise NOT, o bitwise de deslocamento para a direita e o bitwise de deslocamento para a esquerda, que você pode conhecer melhor na documentação do Python.

Examinaremos algumas situações em que os operadores bitwise são úteis. Você pode usar o operador bitwise AND para verificar se um número é par ou ímpar. Um número par como 2 sempre tem um 0 no final, enquanto o número 1 sempre termina com 1 (só contém um dígito binário: 1).

```
10 # 2
1 # 1
```

Quando você usar o operador bitwise AND com um número par e com 1, o Python sempre retornará `False`, porque o número par terminará com um 0 e 1 só tem um dígito binário: 1.

```
10 # 2
1 # 1
--
0
```

Por outro lado, quando o usar com um número ímpar e 1, o Python sempre retornará `True`, porque o número ímpar terminará com 1 e o número 1 só tem um dígito binário: 1.

```
11 #3
1 #1
--
1
```

Já que em binário 1 só tem um dígito, não importa se o número que você está verificando se é par tem um único dígito binário ou mil. Como 1 só tem um dígito binário, você fará apenas uma comparação: entre o último dígito binário do número e 1.

Veja como verificar se um número é par ou ímpar usando o operador bitwise AND em Python:

```
def is_even(n):
return not n & 1
```

Em sua função `is_even`, você está retornando `not n & 1`. Seu código `n & 1` usa o operador bitwise AND com `n` e `1`. Em seguida, você está usando `not` para alterar o resultado para o oposto do que ele seria, porque quando executar a operação bitwise AND com um número par e 1, ela retornará `False`, o que significa que o número é par. Nesse caso, você deseja que sua função retorne `True` para indicar que o número é par, logo usou `not` para alterar `True` para `False` e `False` para `True`.

Você também pode usar o operador bitwise AND para verificar se um inteiro é uma potência de 2. Todo número que é uma potência de 2 tem um único 1 em sua representação binária porque o sistema binário é de base 2, o que significa que qualquer potência de 2 terá um único 1. Por exemplo, o número 8 é 0b1000 em binário. Inversamente, um número que for uma unidade menor do que uma potência de 2 conterá apenas bits 1. O número 7, que é uma unidade menor do que 8, é representado por 0b111 em binário.

Quando você aplicar o operador bitwise AND a esses dois números binários, verá que o binário resultante só terá zeros quando o primeiro número for uma potência de 2.

```
1000 # 8
0111 # 7
————
0000
```

Se o número não for uma potência de 2, pelo menos um dígito binário será igual a 1.

```
0111 # 7
0110 # 6
____
0001
```

Veja como usar o operador bitwise AND em Python para determinar se um número é uma potência de 2:

```
def is_power(n):
if n & (n - 1) == 0:
return True
return False
```

Sua função `is_power` está recebendo o número em questão. Dentro da função, você usou uma instrução `if` para verificar se o uso do operador bitwise AND com `n` e `n - 1` resulta em 0. Se resultar, `n` é uma potência de 2 e você retornará `True`. Caso contrário, `n` não é potência de 2 e `False` será retornado.

FizzBuzz

O FizzBuzz é um dos desafios mais clássicos que aparecem em entrevistas. Uma vez ouvi um relato sobre um engenheiro que foi entrevistado para a posição de engenheiro consultor de software e lhe pediram para resolver o FizzBuzz. Ele não conseguiu e ficou muito constrangido. No entanto, não se preocupe; você vai aprender agora como o resolver, portanto isso não acontecerá.

Este é o desafio FizzBuzz: escreva um programa que exiba os números de 1 a 100. Se o número for múltiplo de 3, exiba "Fizz". Se o número for múltiplo de 5, exiba "Buzz". Se o número for múltiplo de 3 e 5, exiba "FizzBuzz".

O segredo desse desafio é usar o operador módulo, que divide dois valores e retorna o resto. Se o resto for 0, você saberá que o dividendo (o número que foi dividido) é múltiplo do divisor (o número pelo qual foi dividido). Por exemplo, `6 % 3` divide 6 por 3 e retorna 0.

```
print(6 % 3)
```

```
>> 0
```

Já que não há resto, você sabe que 6 é múltiplo de 3.

No cálculo de 7 % 3, haverá resto, logo você saberá que 7 não é múltiplo de 3.

```
print(7 % 3)
```

```
>> 1
```

Para resolver o FizzBuzz, você percorrerá os números de 1 a 100 e usará o módulo para verificar se cada número é múltiplo de 3 e de 5, só de 3 ou só de 5.

Você fará o seguinte:

```
def fizzbuzz(n):
for i in range(1, n + 1):
if i % 3 == 0 and i % 5 == 0:
print('FizzBuzz')
elif i % 3 == 0:
print('Fizz')
elif i % 5 == 0:
print('Buzz')
else:
print(i)
```

Ainda que você queira encontrar os números de 1 a 100, é melhor passar um número `n`, em vez de embutir 100 no código. Passar `n` tornará seu programa flexível se quiser executá-lo com um número diferente. Portanto, sua função, `fizzbuzz`, recebe `n` como parâmetro.

```
def fizzbuzz(n):
```

Primeiro, você usará um loop `for` para percorrer cada número de 1 a `n` + 1.

```
for i in range(1, n + 1):
```

Em seguida, usará uma instrução condicional com o operador módulo para determinar se o número `i`, é divisível tanto por 3 quanto por 5. Se for, você exibirá “FizzBuzz”.

```
if i % 3 == 0 and i % 5 == 0:
print('FizzBuzz')
```

Depois, usará outra instrução condicional para verificar se o número é divisível por 3. Se for, você exibirá "Fizz".

```
elif i % 3 == 0:
print('Fizz')
```

Agora, usará mais uma instrução condicional para verificar se o número é divisível por 5. Se for, você exibirá "Buzz".

```
elif i % 5 == 0:
print('Buzz')
```

Se o número não for divisível por 3, por 5 nem por ambos, você o exibirá.

```
else:
print(i)
```

Quando você executar seu programa, verá que os números divisíveis por 3, como 6 e 27, exibem "Fizz". Os números divisíveis por 5, como 10 e 85, exibem "Buzz" e os números divisíveis por ambos, como 15 e 30, exibem "FizzBuzz".

```
>> 1 2 Fizz 4 Buzz Fizz 7 8...Buzz Fizz 97 98 Fizz Buzz
```

Seu algoritmo executa `n` etapas, o que significa que é linear. Se você passar `100`, o algoritmo demandará 100 etapas, e se passar `1000`, demandará 1.000 etapas.

O operador módulo foi a chave para a solução desse problema. Contudo, não é útil apenas em entrevistas técnicas. Também será útil quando você estiver desenvolvendo aplicações do mundo real. Por exemplo, suponhamos que você tivesse um arquivo de texto com 50.000 linhas e conseguisse inserir 49 linhas em uma página.

Quantas linhas de texto caberão na última página? A última página terá 20 linhas porque `50.000 % 49 = 20`. Se você tivesse um banco de dados com 20.000 itens e quisesse fazer algo com um a cada 2 itens? Uma maneira de fazer isso seria percorrer cada item e só alterar os itens com índice par.

Máximo divisor comum

O **máximo divisor comum** é o maior inteiro positivo que divide

igualmente dois ou mais inteiros. Neste capítulo, dados dois inteiros, como 20 e 12, você aprenderá como encontrar seu máximo divisor comum.

No caso de 20 e 12, você pode dividi-los igualmente por 1, 2 e 4. Já que 4 é o maior número, é o máximo divisor comum.

Divisores de 20: 1, 2, **4**, 5, 10.

Divisores de 12: 1, 2, 3, **4**, 6.

Um algoritmo que poderia encontrar o máximo divisor comum de dois números seria um que verificasse todos os divisores possíveis para ver quais dividem os números igualmente. Por exemplo, para encontrar o máximo divisor comum de 20 e 12, você poderia começar dividindo-os por 1, depois por 2, depois por 3 etc. Não é preciso testar nenhum número que seja maior do que o menor dos dois números porque um número maior do que o menor número não conseguirá fazer a divisão igualmente. Um número maior do que 12, por exemplo, não dividirá 12 igualmente.

Aqui está o código Python para seu algoritmo:

```
def gcf(i1, i2):
gcf = None
if i1 > i2:
smaller = i2
else:
smaller = i1
for i in range(1, smaller + 1):
if i1 % i == 0 and i2 % i == 0:
gcf = i
return gcf
gcf(20, 12)
```

Sua função `gfc` recebe como parâmetro os dois inteiros positivos entre os quais você está procurando o máximo divisor comum.

```
def gcf(i1, i2):
```

Dentro da função, você determinará qual dos dois inteiros é menor e o atribuirá à variável `smaller` para parar de testar divisores quando atingir esse número.

```
if i1 > i2:
```

```
smaller = i2
else:
smaller = i1
```

Em seguida, usará um loop `for` para testar cada divisor de 1 ao valor da variável `smaller` mais um (para também testar o menor número).

```
for i in range(1, smaller + 1):
```

Depois, usará uma instrução `if` para ver se o divisor divide igualmente os dois inteiros. Se dividir, você atribuirá o divisor à variável `gcf`.

```
if i1 % i == 0 and i2 % i == 0:
gcf = div
```

No entanto, só porque você encontrou um divisor comum, isso não significa que tenha encontrado o máximo divisor comum. Se encontrar outro divisor maior, configurará `gcf` com ele na próxima vez que executar o loop. Desta forma, quando seu loop terminar, `gfc` conterá o maior divisor.

Contudo, há um problema em seu código. E se um dos inteiros for 0?

```
print(gcf(0, 22))
>> None
```

O programa retornará a resposta errada quando um dos inteiros for 0.

O fato de o código não conseguir manipular 0 é um exemplo de **condição limite**: uma entrada que não é a que você esperava que o programa recebesse. No cálculo do máximo divisor comum entre dois números, se um dos inteiros for 0, o máximo divisor comum será o outro inteiro. Por exemplo, o máximo divisor comum de 0 e 12 é 12.

Quando você estiver escrevendo algoritmos, deverá sempre considerar que tipo de entrada inesperada pode invalidá-los. Nesse caso, o algoritmo está incorreto quando a entrada é 0. Veja como modificar seu programa para que retorne a saída correta se um dos inteiros for 0:

```
def gcf(i1, i2):
if i1 == 0:
return i2
if i1 == 0:
return i1
if i1 > i2:
```

```
smaller = i2
else:
smaller = i1
for divisor in range(1, smaller + 1):
if(i1 % divisor == 0) and (i2 % divisor == 0):
gcf = divisor
return gcf
```

O programa também não consegue manipular números negativos, logo você precisa adicionar um teste no começo para assegurar que os dois números são positivos.

```
def gcf(i1, i2):
if i1 < 0 or i2 < 0:
raise ValueError("Numbers must be positive.")
if i1 == 0:
return i2
if i1 == 0:
return i
if i1 > i2:
smaller = i2
else:
smaller = i1
for divisor in range(1, smaller+1):
if(i1 % divisor == 0) and (i2 % divisor == 0):
gcf = divisor
return gcf
```

O código do máximo divisor comum é linear porque o algoritmo resolve o problema em `n` etapas. Não é ruim ser linear, mas há uma maneira melhor de resolver esse problema.

Algoritmo de Euclides

Uma solução mais eficiente para a busca do máximo divisor comum chama-se algoritmo de Euclides. Primeiro, dividimos um número, `x`, por outro número, `y`, para encontrar o resto. Em seguida, dividimos novamente, usando o resto como `y` e o `y` anterior como o novo `x`. Continuaremos esse processo até o resto ser 0. O último divisor será o máximo divisor comum.

Por exemplo, para encontrar o máximo divisor comum de 20 e 12, você

começará dividindo 20 por 12 (ou 12 por 20) e obterá o resto 8. Depois, dividirá 12 pelo resto e 12 dividido por 8 produzirá um resto igual a 4. Agora, você dividirá 8 por 4. Dessa vez, não há resto, o que significa que 4 é o máximo divisor comum.

20 / 12 = 1 com 8 como resto.

12 / 8 = 1 com 4 como resto.

8 /4 = 2 com 0 como resto.

Aqui está o algoritmo de Euclides em Python:

```
def gcf(x, y):
if y == 0:
x, y = y, x
while y != 0:
x, y = y, x % y
return x
```

Sua função `gcf` recebe como parâmetros os dois números entre os quais você está procurando o máximo divisor comum.

Na primeira linha de código, você resolverá uma condição limite. Se `y` for 0, Python lançará uma exceção posteriormente em seu programa, tentando fazer a divisão por 0. Logo, se `y` for 0, você trocará os conteúdos das variáveis `x` e `y` para resolver esse problema.

```
if y == 0:
x, y = y, x
```

Em seguida, criará um loop `while` que iterará até `y` ser 0.

```
while y != 0:
```

Dentro do loop `while`, essa instrução trocará os valores de `x` e `y` por `y` e pelo resto de `x` dividido por `y`.

```
x, y = y, x % y
```

Quando o loop `while` terminar, isso significará que `x % y` retornou um resto igual a 0. Você retornará, então, o valor de `x`, que contém o máximo divisor comum de `x` e `y`.

```
return x
```

Você pode usar matemática para comprovar que esse algoritmo é logarítmico em vez de linear, melhorando significativamente seu

desempenho com números grandes em relação ao algoritmo original na busca do máximo divisor comum de dois números.

Números primos

Um **número primo** é um inteiro positivo divisível apenas por si mesmo e por 1. Números como 2, 3, 5, 7 e 11 são exemplos de números primos. Nesta seção, você aprenderá a escrever uma função para determinar se um número é ou não primo.

Veja como o fazer:

```
def is_prime(n):
for i in range(2, n):
if n % i == 0:
return False
return True
```

Sua função `is_prime` está recebendo o número que você deseja verificar se é ou não primo (`n)`.

```
def is_prime(n):
```

Em seguida, você usou um loop `for` para percorrer cada número de 2 a `n`. Você começou com 2 (e não com 1) porque os números primos são divisíveis por 1.

```
for i in range(2, n):
```

Se `n` for 10, seu código percorrerá de 2 a 9, o que significa que 10 será ignorado (você quer que isso aconteça porque os números primos também são divisíveis por si mesmos).

Você usou, então, o operador de módulo para verificar se há resto na divisão de `n` por `i`. Se não houver resto, você encontrou um divisor que não é 1 nem o número. Isso significa que `n` não é um número primo, logo você retornará `False`.

```
if n % i == 0:
return False
```

Se você percorrer seu intervalo de números sem encontrar um divisor, isso significará que `n` é primo e `True` será retornado.

```
return True
```

Já que seu algoritmo leva `n` etapas para ser concluído, é um algoritmo linear.

Você pode melhorar o algoritmo terminando o loop na raiz quadrada de `n` em vez de em `n - 1`.

```
import math
def is_prime(n):
for i in range(2, int(math.sqrt(n)) + 1):
if n % i == 0:
return False
return True
```

Vejamos por que você pode parar na raiz quadrada de `n`. Se `a * b == n`, `a` ou `b` tem de ser menor que ou igual à raiz quadrada de `n`. Por quê? Porque se tanto `a` quanto `b` forem maiores do que a raiz quadrada de `n`, `a * b` terá de ser maior do que `n`, portanto não poderá ser igual a `n`. Logo, você não pode ter dois números, `a` e `b`, em que os dois são maiores do que a raiz quadrada de `n`, e multiplicá-los para serem iguais a `n`, já que `a * b` será maior do que `n`. Como o divisor deve ser menor que ou igual à raiz quadrada de `n`, você não precisará fazer a verificação até `n`. Em vez disso, poderá parar no primeiro inteiro maior do que a raiz quadrada de `n`. Se você não encontrar um número menor ou igual à raiz quadrada de `n` que divida `n` igualmente, não encontrará outro número que o faça.

Você pode modificar seu programa facilmente para exibir uma lista de todos os números primos de um intervalo de números.

```
def is_prime(n):
for i in range(2, int(math.sqrt(n)) + 1):
if n % i == 0:
return False
return True
def find_primes(n):
return [i for i in range(2, n) if is_prime(i)]
```

Primeiro, você criou uma nova função chamada `find_primes`, que recebe como parâmetro `n`, que é o número até o qual números primos estão sendo procurados.

Em seguida, usou uma compreensão de lista para percorrer de 2 a `n`, adicionando itens à sua nova lista somente se `is_prime` retornar `True`.

```
return [is_prime(i) for i in range(2, n)]
```

Seu algoritmo que encontra todos os números primos em um intervalo de números chama a função linear `is_prime` dentro de um loop, o que significa que tem complexidade de tempo O(n**2) e não é muito eficiente. No entanto, o algoritmo que você aprendeu neste capítulo não é o único para a busca de números primos. Existem outros algoritmos mais complicados para a busca de números primos com complexidade de tempo mais eficiente que você também pode usar.

Vocabulário

base: a base de um sistema de numeração é o número de dígitos que o sistema tem.

bit: em binário, um dígito é chamado de bit, que é a abreviação de binary digit.

condição limite: entrada diferente da entrada padrão que esperamos que o programa receba.

máximo divisor comum: maior inteiro positivo que divide igualmente dois ou mais inteiros.

número binário: número do sistema de numeração de base 2.

número primo: inteiro positivo divisível somente por si mesmo e por 1.

operador bitwise: operador que podemos usar em uma expressão com dois números binários.

sistema de numeração: sistema de escrita para a representação de números.

valor posicional: valor numérico que um dígito tem em razão de sua posição em um número.

Desafio

1. Pesquise e crie outra maneira de encontrar número primos.

CAPÍTULO 7

Inspiração autodidata: Margaret Hamilton

A todos os meus amigos que têm irmãos menores que estão ingressando na Universidade ou no Ensino Médio – Meu principal conselho é: vocês precisam aprender a programar.

– Mark Zuckerberg

Atualmente, existem muitos recursos que ensinam a programar; é fácil nos esquecermos de que nem sempre foi assim. Margaret Hamilton, uma das codificadoras originais da missão espacial Apollo e uma das maiores programadoras autodidatas de todos os tempos, deixou sua marca no mundo da tecnologia muito antes de os cursos de programação estarem amplamente disponíveis.

Embora Hamilton tenha frequentado a universidade (ela graduou-se como bacharel em Matemática, na Universidade de Michigan), suas habilidades de programação foram obtidas de forma totalmente autodidata. Nos anos de 1950 e 1960, todos eram autodidatas porque a ciência da computação como a conhecemos ainda não existia, logo os programadores tinham de descobrir o que fazer por conta própria. Na época, não existia nem mesmo o termo *engenharia de software*: Hamilton ajudou a cunhá-lo! Após se graduar na Universidade em 1960, ela começou sua carreira em programação no MIT, em um software chamado Project Whirlwind para prever o tempo. Enquanto esteve no MIT, ajudou a criar o código do primeiro computador portátil do mundo.

O sucesso de Hamilton com o Project Whirlwind a levou a um cargo na SAGE, uma organização que ajudava a detectar possíveis ataques aéreos soviéticos durante a Guerra Fria. Se você for fã de *Jornada nas Estrelas,*

deve conhecer a história de Kobayashi Maru – um exercício de treinamento para cadetes da frota estelar no qual era impossível ser bem-sucedido. Futuros oficiais podiam não conseguir vencer o jogo, mas mostravam traços de caráter importantes por meio de suas escolhas. Hamilton enfrentou uma situação como a de Kobayashi Maru na vida real e, como o infame Capitão Kirk, conseguiu superá-la. Todos os novatos da SAGE recebiam um programa praticamente indecifrável e eram solicitados a fazê-lo funcionar. O programador que o criou chegou até a escrever comentários em grego e latim para tornar o desafio mais difícil. Hamilton foi a primeira engenheira a colocá-lo em funcionamento, o que assegurou seu lugar dentro da organização.

Sua habilidade para resolver problemas difíceis a fez ser contratada para a missão Apollo da Nasa, o que teve como ápice o pouso na lua realizado por Neil Armstrong, Buzz Aldrin e Michael Collins, em 1969. Embora os astronautas tenham recebido a maior parte do crédito pela histórica missão, mais de 400 mil pessoas deram suas contribuições e Hamilton foi essencial para o seu sucesso.

Um dos feitos mais significativos de sua equipe foi o desenvolvimento de um sistema que dava alertas sobre emergências. Esse processo foi uma parte crítica do sucesso da aterrissagem na lua e beneficiou-se enormemente da insistência de Hamilton para a execução de testes rigorosos. Dr. Paul Curto, que a indicou para um NASA Space Act Award, disse que seu trabalho continha a "base para um design de software altamente confiável" porque, até hoje, ninguém encontrou um bug no software da Apollo.

Em 22 de novembro de 2016, o presidente Barack Obama concedeu a Hamilton a Medalha Presidencial da Liberdade pelo reconhecimento de suas excepcionais realizações na engenharia de software e solidificou sua posição como uma das maiores programadoras autodidatas de todos os tempos.

PARTE II

Estruturas de dados

CAPÍTULO 8

O que é uma estrutura de dados?

Algoritmos + estruturas de dados = programas.
– Niklaus Wirth

Uma **estrutura de dados** é uma maneira de organizar dados em um computador para que os programadores possam usá-los eficazmente em seus programas. No decorrer deste livro, você usou algumas estruturas de dados internas do Python, como as listas e os dicionários, para buscar dados, ordená-los e em várias outras tarefas. Esta seção do livro ensinará mais detalhes sobre as estruturas de dados e como as usar. Você também conhecerá novos tipos de estruturas de dados com as quais talvez ainda não esteja familiarizado, como os arrays, as listas encadeadas, as pilhas, as filas, as árvores, os heaps, os grafos e as tabelas hash. Cada uma dessas estruturas de dados tem vantagens e desvantagens. A melhor estrutura de dados a ser usada em um programa vai depender do tipo de problema que você estiver tentando resolver e o que estiver tentando otimizar. Na Parte II deste livro, você conhecerá as vantagens e desvantagens de diferentes estruturas de dados para, quando estiver desenvolvendo aplicações, decidir qual é a melhor a ser usada. Além disso, aprenderá a responder às perguntas mais comuns que os entrevistadores fazem sobre estruturas de dados para, quando chegar a hora de fazer uma entrevista técnica, passar com louvor.

Você não conseguirá ser um bom programador sem conhecimento sólido das estruturas de dados porque programar significa escrever algoritmos e escolher a estrutura de dados certa para eles. Por isso, Niklaus Wirth criou a famosa expressão: "Algoritmos + estruturas de dados = programas". O algoritmo diz ao computador o que fazer e a estrutura de dados informa a ele como armazenar os dados do algoritmo. Linux

Torvalds, o criador do Linux, reforçou ainda mais a importância das estruturas de dados com seu famoso argumento: "Na verdade, diria que a diferença entre um bom e um mau programador está em se ele considera mais importante seu código ou suas estruturas de dados. Programadores ruins se preocupam com o código. Bons programadores se preocupam com estruturas de dados e seus relacionamentos". Quero que você seja um bom programador. Por isso me dedicarei a ensinar estruturas de dados no restante deste livro.

Um **tipo de dado abstrato** é uma descrição de uma estrutura de dados, enquanto uma estrutura de dados é uma implementação real. Por exemplo, uma lista é um tipo de dado abstrato: descreve uma estrutura de dados que contém um grupo de itens no qual cada um tem uma posição em relação aos outros. A lista também tem operações para a manipulação de seus itens, como para adicioná-los e removê-los. Quando você usa uma lista do Python, está usando uma estrutura de dados, e não um tipo de dado abstrato, porque uma estrutura de dados é a implementação real de um tipo de dado abstrato. O Python pode ter duas estruturas de lista diferentes, implementadas de maneiras totalmente distintas, ambas baseadas no tipo de dado abstrato da lista.

Cientistas da computação classificam as estruturas de dados de acordo com diferentes propriedades, por exemplo, se são lineares ou não lineares. Uma **estrutura de dados linear** organiza os elementos em uma sequência, enquanto uma **estrutura de dados não linear** vincula os dados de maneira não sequencial. As listas do Python são uma estrutura de dados linear: cada elemento pode ter um único elemento antes e um único elemento depois dele. Ao contrário, os grafos (sobre os quais você também saberá mais posteriormente) são estruturas não lineares nas quais cada elemento pode se conectar com vários outros elementos.

Percorrer uma estrutura de dados significa ir do primeiro ao último elemento da estrutura. Em uma estrutura de dados linear, podemos percorrer facilmente do primeiro ao último elemento sem backtracking (mover-se para trás na estrutura de dados), enquanto em uma estrutura de dados não linear geralmente temos de executar o backtracking.

Costuma ser mais eficiente acessar elementos individuais em uma estrutura de dados linear porque as estruturas de dados não lineares demandam backtracking ou recursão para o acesso a um único elemento. Já que podemos percorrer facilmente uma estrutura de dados linear, normalmente fazer uma alteração em cada elemento da estrutura é mais fácil do que em cada elemento de uma estrutura não linear. Como não precisamos executar o backtracking para visitar cada elemento, as estruturas de dados lineares são mais fáceis de projetar e usar. Mesmo assim, as estruturas de dados não lineares podem ser melhores (e mais eficientes) para certos tipos de problemas. Por exemplo, você pode usar estruturas de dados não lineares para armazenar e acessar dados cujo armazenamento seria ineficiente em uma estrutura linear, como as conexões de uma rede social.

Os cientistas da computação também classificam as estruturas de dados baseando-se em se são estáticas ou dinâmicas. Uma **estrutura de dados estática** tem tamanho fixo, enquanto o tamanho de uma **estrutura de dados dinâmica** pode crescer ou diminuir em tamanho. Em geral, definimos o tamanho de uma estrutura de dados estática quando a criamos no programa. Depois que isso é feito, o tamanho da estrutura de dados é fixo e não pode ser alterado, mas podemos alterar os valores dos seus dados. Python não tem estruturas de dados estáticas, no entanto linguagens de programação de nível mais baixo, como C, têm.

Um problema das estruturas de dados estáticas é o fato de ser preciso alocar uma quantidade de memória específica para elas. A **memória do computador** é o local onde a máquina armazena dados. Existem diferentes tipos de memória e detalhar todos não faz parte do escopo deste livro. Contudo, para este capítulo, você pode considerar a memória como um local no qual seu computador armazenará dados com um endereço para poder referenciá-los posteriormente. Se o número de elementos com o qual você estiver trabalhando for menor do que o tamanho alocado, o espaço na memória será desperdiçado. Além disso, pode não ser possível adicionar mais elementos do que os que caberiam no espaço alocado para a estrutura de dados. Em geral, a única maneira de adicionar mais elementos a uma estrutura de dados estática é alocando

memória para uma nova estrutura suficientemente grande para conter os elementos antigos e os novos e, depois, copiar a estrutura antiga para essa memória recém-alocada com os novos elementos. Portanto, estruturas de dados estáticas não serão melhores se você não souber antecipadamente o número de elementos que terá de armazenar. Contudo, se souber quantos elementos de dados deseja armazenar e se esse número não mudar, uma estrutura de dados estática terá um desempenho melhor do que uma dinâmica. Por exemplo, se você estiver mantendo uma lista de funções de desfazer (undo) para um programa que permita aos usuários desfazer até 10 operações, uma estrutura de dados estática será mais apropriada.

Muitas estruturas de dados podem ser estáticas ou dinâmicas. Por exemplo, em geral os arrays (que discutiremos no Capítulo 9) são estruturas de dados estáticas, mas várias linguagens de programação modernas, como Python, oferecem arrays dinâmicos (listas).

Ao contrário do que ocorre com as estruturas de dados estáticas, podemos alterar facilmente o tamanho de uma estrutura de dados dinâmica. Para uma estrutura de dados dinâmica, o computador aloca memória adicional à medida que incluímos mais elementos. Quando removemos elementos, o computador libera essa memória para outros dados. Com seu tamanho flexível, as estruturas de dados dinâmicas permitem adicionar e remover elementos de dados eficientemente e fazem uso dos recursos da memória de maneira eficaz. No entanto, o acesso a elementos de estruturas de dados dinâmicas pode ser lento em comparação com o de estruturas de dados estáticas e o armazenamento de um número específico de elementos nas estruturas de dados dinâmicas pode consumir mais memória do que o armazenamento do mesmo número de elementos em estruturas de dados estáticas. Na manipulação de uma quantidade de dados desconhecida, principalmente em situações em que o espaço na memória seja limitado, as estruturas de dados dinâmicas costumam ser uma melhor opção.

A menos que você esteja escrevendo código de baixo nível para um sistema operacional ou algum projeto em que seja preciso adicionar qualquer otimização possível ao desempenho, provavelmente não será

necessário dedicar muito tempo pensando na escolha de uma estrutura de dados estática ou dinâmica. Em vez disso, você deve gastar mais tempo decidindo se deve usar uma estrutura de dados linear ou não linear e, após tomar a decisão, qual estrutura de dados linear ou não linear usar. Como vimos anteriormente, as diferentes estruturas de dados têm vantagens e desvantagens. Quase sempre as vantagens e desvantagens estão relacionadas à sua eficácia na inserção, exclusão, busca e ordenação de dados e com que eficiência usam espaço na memória. Por exemplo, é muito eficiente verificar se um item existe em um dicionário Python, mesmo se tiver bilhões de dados. Contudo, não é tão eficiente quanto procurar um dado em um grafo. No Capítulo 9, veremos mais detalhes sobre quando e como usar as estruturas de dados.

Vocabulário

estrutura de dados: maneira de organizar dados em um computador para os programadores usá-los eficientemente em seus programas.

estrutura de dados estática: estrutura de dados com tamanho fixo.

estrutura de dados dinâmica: estrutura de dados cujo tamanho pode crescer ou diminuir.

estrutura de dados linear: estrutura de dados que organiza os elementos de dados em sequência.

estrutura de dados não linear: estrutura de dados que vincula os dados de maneira não sequencial.

memória do computador: local onde seu computador armazena os dados.

percorrer: ir do primeiro ao último elemento de uma estrutura de dados.

Desafio

1. Escreva uma lista com todas as estruturas de dados que você usa em Python.

CAPÍTULO 9

Arrays

Saudamos os codificadores, designers e programadores que estão trabalhando arduamente em suas mesas e encorajamos todos os alunos que ainda não decidiram se devem assistir a uma aula de Ciência da Computação a tentarem.

– Michael Bloomberg, ex-prefeito da cidade de Nova York

Uma **lista** é um tipo de dado abstrato que descreve uma estrutura de dados que armazena valores ordenados. Em geral, as listas têm uma operação para criar uma nova lista vazia, para verificar se a lista está vazia, para inserir um item no início, para inserir um item no final e para acessar um elemento em um índice. Você já está familiarizado com as listas do Python, uma das várias implementações diferentes do tipo de dado abstrato lista, que é um tipo de array. Neste capítulo, você aprenderá mais sobre os arrays.

Um **array** é uma estrutura de dados que armazena elementos com índices em um bloco de memória contíguo. Em geral, os arrays são homogêneos e estáticos. Uma **estrutura de dados homogênea** pode conter apenas um tipo de dado, como inteiros ou strings. Uma estrutura de dados estática não pode ser redimensionada após sua criação. Quando criamos um array em uma linguagem de programação de baixo nível como C, decidimos quantos elementos de um tipo de dado específico queremos armazenar nele. O computador atribui, então, um bloco de memória para o array, de acordo com o número de elementos e de quanta memória um elemento desse tipo requer. Esse bloco é composto de itens armazenados um após o outro na memória do computador.

Uma lista Python é um **array heterogêneo de tamanho variável**. Um **array de tamanho variável** tem um tamanho que pode ser alterado após

sua criação. Um **array heterogêneo** pode conter diferentes tipos de dados em vez de apenas um tipo. Guido van Rossum escreveu o Python em C, mas o Python oculta as complexidades da criação e manipulação de arrays. Fornece uma estrutura de dados de lista que podemos usar sem nos preocupar com a atribuição de seu tamanho antecipadamente ou a especificação dos tipos de dados que esta pode armazenar.

A Figura 9.1 mostra um exemplo de como um computador armazena um array na memória.

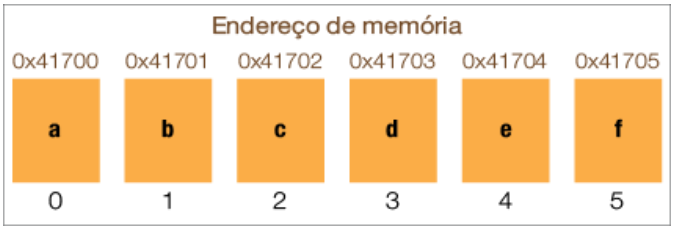

Figura 9.1: Exemplo de dados de um array.

Você pode acessar cada elemento desse array com um índice inteiro único. Em geral, o primeiro índice de um array é 0, no entanto diferentes linguagens de programação podem usar esquemas de indexação distintos. Tanto Python quanto C usam a indexação baseada em zero, mas algumas outras linguagens, como MATLAB e Fortran, usam a indexação de base 1 (o primeiro elemento tem índice 1). Algumas linguagens permitem até mesmo utilizar qualquer valor inteiro como índice do primeiro elemento. A localização do primeiro elemento de um array na memória é chamada de seu **endereço base**. Quando o computador precisar adicionar um novo item a um array, calculará o local na memória onde deve inseri-lo usando esta fórmula:

```
endereço_base + índice * tamanho_do_tipo_de_dado
```

Pegará o endereço base e o somará ao índice multiplicado por quanto espaço na memória o tipo de dado do array ocupa. O computador também usará essa fórmula quando precisar encontrar um item em um array.

Os arrays podem ser unidimensionais ou multidimensionais. Em um **array unidimensional**, acessamos cada elemento com um índice inteiro:

```
array = [1, 2, 3]
print(array[0])
```

```
>> 1
```

Em um **array multidimensional**, acessamos cada elemento individual com dois índices (um índice inteiro para cada dimensão):

```
multi_array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(array[1][2])
```

```
>> 6
```

Independentemente de um array ser unidimensional ou multidimensional, o computador armazenará seus elementos em um bloco de memória contíguo e usará uma fórmula matemática que mapeará o índice para um local na memória para acessá-los.

Desempenho do array

Você pode acessar e modificar qualquer elemento individual de um array em tempo constante. Por exemplo, suponhamos que estivesse procurando os dados do índice 3 de um array. Nesse caso, o computador precisará obter informações de um único local na memória, mesmo se houver um milhão de elementos no array. Pesquisar um array não ordenado tem complexidade O(*n*) porque é preciso verificar cada item. No entanto, geralmente podemos ordenar os arrays, como quando temos uma lista de nomes ou endereços, ou um grande conjunto de números, caso em que a busca no array pode ter complexidade O(log *n*). A Figura 9.2 mostra os tempos de execução dos arrays.

Estrutura de dados	Complexidade de tempo							
	Média				Pior			
	Acesso	Busca	Inserção	Exclusão	Acesso	Busca	Inserção	Exclusão
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Pilha	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Fila	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Lista encadeada	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Tabela Hash	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)
Árvore de busca binária	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)

Figura 9.2: Tempos de execução das operações dos arrays.

Se os arrays são tão eficientes, você deve estar se perguntando por que não pode usá-los em todas as situações. Embora seja muito rápido acessar e modificar os elementos individuais de um array, alterar a forma do array de alguma maneira (adicionando ou excluindo itens) tem complexidade O(*n*). Já que o computador armazena os elementos de um array em um único bloco de memória contíguo, inserir um elemento significa que você precisará deslocar todos os elementos que vêm depois do elemento adicionado, o que não é eficiente. Por exemplo, suponhamos que você tivesse um array que se parecesse com o da Figura 9.3.

Figura 9.3: Array armazenado na memória do computador.

Veja na Figura 9.4 o que ocorrerá quando você adicionar *z* depois de *a* e *b*.

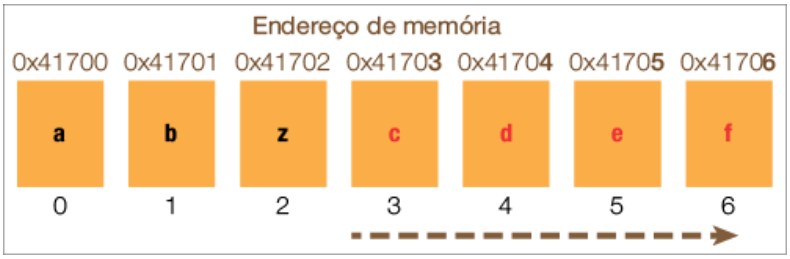

Figura 9.4: Geralmente adicionar dados a um array significa alterar muitos locais na memória.

Tornar *z* o terceiro item dessa lista requer que o computador desloque quatro elementos para diferentes localizações na memória.

Ser necessário deslocar elementos não é um problema em arrays pequenos, mas se o programa tiver de adicionar itens a locais aleatórios em um array grande (principalmente perto do início), o computador poderá passar muito tempo copiando a memória. Esse problema é pior para arrays estáticos, porque o computador não pode garantir que os endereços de memória que vêm após o bloco que ele reservou para o

array estarão livres. Isso significa que quando você adicionar um elemento a um array em uma linguagem como C, poderá ter de reservar um novo bloco de memória para o array, copiar todos os itens do bloco anterior para o novo bloco, adicionar o novo elemento e liberar o bloco anterior. O Python adiciona itens às suas listas mais eficientemente por meio de um processo chamado **superalocação**: reservar mais memória do que o array precisa e registrar quantos elementos está armazenando e o espaço que não está sendo usado.

Como programador, você usará arrays com frequência em seus programas. Deve considerar usar um array sempre que precisar armazenar e acessar dados sequenciais. Por exemplo, digamos que estivesse programando um jogo como *Call of Duty* e quisesse criar uma página que mostrasse a pontuação dos 10 melhores jogadores. Poderia usar um array para registrar facilmente os 10 melhores jogadores, suas pontuações e a ordem que ocupam classificando o jogador de pontuação mais alta no índice 0 e o jogador de pontuação mais baixa no índice 9. Os arrays são uma das estruturas de dados mais importantes para cálculos matemáticos. Se você tiver de lidar com grandes quantidades de dados numéricos, vai acabar se familiarizando com os arrays. Os cientistas da computação também usam arrays para implementar outras estruturas de dados. Em capítulos posteriores, você aprenderá como implementar as estruturas de dados pilha e fila usando arrays.

Os sistemas de tempo de execução (run-time) de várias linguagens de programação usam arrays para implementar estruturas de nível mais alto como as listas, que os programadores Python utilizam muito. Arrays como os do pacote NumPy (Numerical Python) da linguagem Python são úteis para aplicações matemáticas e científicas, aplicações financeiras, estatística e assim por diante. Os arrays do NumPy suportam operações matemáticas como a multiplicação de matrizes, que são usadas, por exemplo, em aplicações gráficas para o dimensionamento, a conversão e a rotação de objetos gráficos. Em sistemas operacionais, em geral os arrays manipulam qualquer operação que lide com uma sequência de dados, como o gerenciamento de memória e o armazenamento em buffer.

Um array não é a melhor opção para grandes conjuntos de dados aos quais tenhamos a intenção de adicionar mais dados com frequência porque a inclusão de itens em um array tem complexidade O(*n*). Nessa situação, as listas encadeadas, que você conhecerá no Capítulo 10, costumam ser uma opção melhor. Quando você inserir um item em um array, este alterará o índice de outros elementos, logo, se for necessário que os dados mantenham o mesmo índice, provavelmente um dicionário Python será uma escolha melhor.

Criando um array

Quando programamos em Python, normalmente usamos uma lista quando precisamos de um array. No entanto, se você precisar do desempenho de um array homogêneo, poderá utilizar a classe `array` interna de Python. Veja como funciona:

```
import array
arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
print(arr[1])
>> 1.5
```

Primeiro, você importará o módulo interno `array` do Python:

```
import array
```

Em seguida, passará dois parâmetros em `array.array`. O primeiro parâmetro dirá ao Python que tipo de dado você deseja que seu array armazene. Nesse caso, `f` representa float (um tipo de dado para números decimais em Python), mas você também pode criar um array com outros tipos de dados de Python. O segundo parâmetro é uma lista Python contendo os dados que você deseja inserir em seu array.

```
arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
```

Após criar seu array, você poderá usá-lo como uma lista de Python:

```
print(arr[1])
```

Contudo, se tentar adicionar dados ao array que não tenham o tipo de dado passado inicialmente, verá um erro:

```
arr[1] = 'hello'
>> TypeError: "must be real number, not str"
```

O pacote NumPy do Python também oferece um array que você poderá usar para fazer cálculos tão rapidamente quanto em uma linguagem de programação de nível mais baixo como C. Você pode aprender mais sobre a criação de arrays com o NumPy, lendo sua documentação em https://numpy.org/.

Movendo zeros

Em uma entrevista técnica, você pode ter de localizar todos os zeros de uma lista e empurrá-los para o final, deixando os elementos restantes na ordem original. Por exemplo, suponhamos que você tivesse esta lista:

```
[8, 0, 3, 0, 12]
```

Você a pegaria e retornaria uma nova lista com todos os zeros no final, desta forma:

```
[8, 3, 12, 0, 0]
```

Veja como esse problema pode ser resolvido em Python:

```
def move_zeros(a_list):
zero_index = 0
for index, n in enumerate(a_list):
if n != 0:
a_list[zero_index] = n
if zero_index != index:
a_list[index] = 0
zero_index += 1
return(a_list)
a_list = [8, 0, 3, 0, 12]
move_zeros(a_list)
print(a_list)
```

Primeiro, você criará uma variável chamada `zero_index` e a inicializará com 0:

```
zero_index = 0
```

Depois, percorrerá cada número de `a_list`, usando a função `enumerate` para registrar tanto o índice quanto o número atual da lista:

```
for index, n in enumerate(a_list):
```

Em seguida, vem este código, que só será executado se `n` não for igual a 0:

```
if n != 0:
a_list[zero_index] = n
if zero_index != index:
a_list[index] = 0
zero_index += 1
```

Quando `n` não for igual a 0, você usará o índice armazenado em `zero_index` para substituir o que houver em `zero_index` por `n`. Depois, verificará se `zero_index` e `index` não têm mais o mesmo número. Se não tiverem o mesmo número, isso significará que havia um zero anteriormente na lista, portanto você substituirá qualquer que seja o número do índice atual por 0 e incrementará `zero_index` em 1 unidade.

Examinaremos por que isso funciona. Digamos que esta fosse sua lista e seu algoritmo só alcançasse o primeiro zero, o que significa que `index` é 1.

```
[8, 0, 3, 0, 12]
```

Já que o número é zero, dessa vez, quando percorrermos o loop, este código não será executado:

```
if n != 0:
a_list[zero_index] = n
if zero_index != index:
a_list[index] = 0
zero_index += 1
```

Ou seja, você não incrementou `zero_index`. Na próxima vez que executar o loop, `index` será 2, `n` será 3 e sua lista continuará sendo a mesma.

```
[8, 0, 3, 0, 12]
```

Como dessa vez `n` não é 0, este código será executado:

```
if n != 0:
a_list[zero_index] = n
if zero_index != index:
a_list[index] = 0
zero_index += 1
```

Quando esta parte de seu código for executada:

```
a_list[zero_index] = n
```

alterará a lista deste formato:

```
[8, 0, 3, 0, 12]
```

para este:

```
[8, 3, 3, 0, 12]
```

Em seguida, esta linha de código:

```
if zero_index != index:
a_list[index] = 0
```

alterará sua lista deste formato:

```
[8, 3, 3, 0, 12]
```

para este:

```
[8, 3, 0, 0, 12]
```

Seu código trocou o zero do fim da lista pelo próximo número diferente de zero em direção ao início da lista.

Agora, seu algoritmo alcançará um zero novamente e isso também ocorrerá.

```
[8, 3, 0, 0, 12]
```

As variáveis `zero_index` e `index` não são mais iguais e `zero_index` é 3, que é o índice do 0 mais distante em relação ao início da lista. Na próxima vez, `n` será 12, logo esse código será executado novamente:

```
if n != 0:
a_list[zero_index] = n
if zero_index != index:
a_list[index] = 0
zero_index += 1
```

Este código:

```
a_list[zero_index] = n
```

alterará sua lista deste formato:

```
[8, 3, 0, 0, 12]
```

para este:

```
[8, 3, 12, 0, 12]
```

Este código:

```
if zero_index != index:
a_list[index] = 0
```

a alterará deste formato:

```
[8, 3, 12, 0, 12]
```

para este:

```
[8, 3, 12, 0, 0]
```

Como você pode ver, agora os zeros estão no fim da lista, com o restante dos números antecedendo-os em sua ordem original.

O algoritmo contém um loop principal que itera pelos elementos de `a_list`, o que significa que sua complexidade de tempo é O(*n*).

Combinando duas listas

Ao se preparar para uma entrevista técnica, você precisa saber como combinar duas listas, algo que terá de fazer com frequência em sua programação diária. Por exemplo, suponhamos que você tivesse uma lista de filmes:

```
movie_list = [ "Interstellar", "Inception",
"The Prestige", "Insomnia",
"Batman Begins" ]
```

e uma lista de classificações:

```
ratings_list = [1, 10, 10, 8, 6]
```

Você quer combinar esses dois conjuntos de dados em uma única lista de tuplas contendo o título de cada filme e sua classificação desta forma:

```
[('Interstellar', 1),
('Inception', 10),
('The Prestige', 10),
('Insomnia', 8),
('Batman Begins', 6)]
```

É possível usar a função interna `zip` do Python para combinar essas listas:

```
print(list(zip(movie_list, ratings_list)))
```

```
>> [('Interstellar', 1), ('Inception', 10), ('The Prestige', 10), ('Insomnia',
8), ('Batman Begins', 6)]
```

A função `zip` recebe um ou mais iteráveis e retorna um objeto `zip` contendo dados de cada iterável que você, então, transformará em uma lista. A saída é uma lista de tuplas, com cada tupla contendo o nome de

um filme associado à sua classificação.

Encontrando as duplicidades em uma lista

Outra questão comum em entrevistas técnicas é a busca de itens duplicados em uma lista, o que você também terá de fazer com frequência em programação no mundo real. Uma solução seria comparar cada item com os demais da lista. Infelizmente, comparar cada item com todos os outros itens requer dois loops aninhados e a complexidade é O(*n***2). Os conjuntos do Python fornecem uma maneira melhor de procurar duplicidades. Um **conjunto** é uma estrutura de dados que não pode conter elementos duplicados. Se um conjunto tiver uma string como `'Kanye West'`, será impossível adicionar outra instância de `'Kanye West'` a ele.

Veja como criar um conjunto e adicionar dados:

```
a_set = set()
a_set.add("Kanye West")
a_set.add("Kendall Jenner")
a_set.add("Justin Bieber")
print(a_set)
```

```
>> {'Kanye West', 'Kendall Jenner', 'Justin Bieber'}
```

Seu código cria um conjunto com três strings: `"Kanye West"`, `"Kendall Jenner"` e `"Justin Bieber"`.

Agora tente adicionar uma segunda instância de `"Kanye West"` ao conjunto:

```
a_set = set()
a_set.add('Kanye West')
a_set.add('Kanye West')
a_set.add('Kendall Jenner')
a_set.add('Justin Bieber')
print(a_set)
```

```
>> {'Kanye West', 'Kendall Jenner', 'Justin Bieber'}
```

Como você pode ver, o conjunto ainda tem apenas três itens. Python não adicionou uma segunda instância de `'Kanye West'` porque é uma duplicidade.

Já que os conjuntos não podem conter duplicidades, você pode adicionar os itens de um iterável a um conjunto um a um, e se o tamanho não mudar, saberá que o item que está tentando adicionar é uma duplicidade.

Veja uma função que usa conjuntos para procurar as duplicidades de uma lista:

```
def return_dups(an_iterable):
dups = []
a_set = set()
for item in an_iterable:
l1 = len(a_set)
a_set.add(item)
l2 = len(a_set)
if l1 == l2:
dups.append(item)
return dups
a_list = [
"Susan Adams",
"Kwame Goodall",
"Jill Hampton",
"Susan Adams"]
dups = return_dups(a_list)
print(dups)
```

A lista que você está processando contém quatro elementos com uma única duplicidade: `"Susan Adams"`.

A função `return_dups` recebeu um iterável chamado `an_iterable` como parâmetro:

```
def return_dups(an_iterable):
```

Dentro da função, você criou uma lista vazia chamada `dups` para armazenar as duplicidades:

```
dups = []
```

Em seguida, criou um conjunto vazio chamado `a_set`:

```
a_set = set()
```

Você usou um loop `for` para percorrer cada item de `an_iterable`:

```
for item in an_iterable:
```

Depois, obteve o tamanho do conjunto, adicionou um item de

`an_iterable` e verificou se o tamanho mudou:

```
l1 = len(a_set)
a_set.add(item)
l2 = len(a_set)
```

Se o tamanho de seu conjunto não mudou, o item atual é uma duplicidade, logo você o acrescentará no final da lista `dups`:

```
if l1 == l2:
dups.append(item)
```

Aqui está o programa completo:

```
def duplicates(an_iterable):
dups = []
a_set = set()
for item in an_iterable:
l1 = len(a_set)
a_set.add(item)
l2 = len(a_set)
if l1 == l2:
dups.append(item)
return dups
a_list = [
'Susan Adams',
'Kwame Goodall',
'Jill Hampton',
'Susan Adams']
dups = duplicates(a_list)
print(dups)
```

```
>> ['Susan Adams']
```

Quando você executar sua função e passar um iterável com duplicidades, ela exibirá a lista `dups` contendo todas as duplicidades.

Encontrando a interseção de duas listas

Mais um desafio comum das entrevistas técnicas envolve escrever uma função que encontre a interseção de duas listas, o que também será útil em sua programação diária. Por exemplo, digamos que você tivesse uma lista com os números sorteados na loteria e outra lista contendo os números mais comuns sorteados até hoje.

```
this_weeks_winners = [2, 43, 48, 62, 64, 28, 3]
most_common_winners = [1, 28, 42, 70, 2, 10, 62, 31, 4, 14]
```

O objetivo é descobrir quantos números sorteados atuais estão na lista dos sorteados mais comuns.

Uma maneira de resolver esse problema seria com o uso de uma compreensão de lista (*list comprehension*) para a criação de uma terceira lista e a utilização de um filtro para verificarmos se cada valor de `list1` também está em `list2`:

```
def return_inter(list1, list2):
list3 = [v for v in list1 if v in list2]
return list3
list1 = [2, 43, 48, 62, 64, 28, 3]
list2 = [1, 28, 42, 70, 2, 10, 62, 31, 4, 14]
print(return_inter(list1, list2))
```

```
>> [2, 62, 28]
```

Como você pode ver, os números 2, 62 e 28 estão nas duas listas.

Esta linha de código usa um loop `for` para percorrer `list1` e só acrescenta o item à nova lista se o valor também estiver em `list2`.

```
list3 = [v for v in list1 if v in list2]
```

A palavra-chave `in` do Python procura um valor em um iterável. Já que estamos lidando com listas não ordenadas, o Python executará uma busca linear quando usarmos a palavra-chave `in` dentro da list comprehension. Por você estar usando a palavra-chave `in` dentro de um loop (a primeira parte da list comprehension), a complexidade de tempo do algoritmo é O(n**2).

Outra opção seria o uso de um conjunto na solução desse problema. Em Python, os conjuntos têm uma função `intersection` que retorna qualquer elemento que apareça em dois ou mais conjuntos.

Você pode converter facilmente as listas em conjuntos desta forma:

```
set1 = set(list1)
set2 = set(list2)
```

Após converter suas listas em conjuntos, poderá aplicar a função `intersection` para descobrir onde os dois conjuntos têm itens duplicados. Aqui está a sintaxe que chama a função `intersection` em

dois conjuntos:

```
set1.intersection(set2)
```

Em seguida, você converterá o conjunto da interseção novamente em uma lista usando a função `list`:

```
list(set1.intersection(set2))
```

Reunindo tudo, obteremos:

```
def return_inter(list1, list2):
set1 = set(list1)
set2 = set(list2)
return list(set1.intersection(set2))
list1 = [2, 43, 48, 62, 64, 28, 3]
list2 = [1, 28, 42, 70, 2, 10, 62, 31, 4, 14]
new_list = return_inter(list1, list2)
print(new_list)
```

```
>> [2, 28, 62]
```

A primeira linha define uma função chamada `return_inter` que recebe duas listas como parâmetros:

```
def return_inter(list1, list2):
```

Depois, você converteu as listas em conjuntos:

```
set1 = set(list1)
set2 = set(list2)
```

Em seguida, chamou a função `intersection` e encontrou as duplicidades:

```
list(set1.intersection(set2))
```

Para concluir, você converteu o conjunto novamente em uma lista e retornou o resultado:

```
return list(set1.intersection(set2))
```

Não há a limitação de uso da função `intersection` somente com dois conjuntos. Você pode chamá-la com quantos conjuntos quiser. Este código encontra os elementos comuns de quatro conjuntos:

```
(s1.intersection(s2, s3, s4))
```

Vocabulário

array: estrutura de dados que armazena elementos com índices em um bloco de memória contíguo.

array de tamanho variável: array cujo tamanho pode mudar após sua criação.

array heterogêneo: array que pode conter diferentes tipos de dados.

array heterogêneo de tamanho variável: array cujo tamanho pode mudar após sua criação e que também pode armazenar vários tipos de dados.

array multidimensional: array no qual acessamos cada elemento usando uma tupla como índice.

array unidimensional: array no qual acessamos cada elemento usando um índice inteiro.

conjunto: estrutura de dados que não pode conter elementos duplicados.

endereço base: Localização de memória do primeiro elemento de um array na memória.

estrutura de dados homogênea: estrutura de dados que só pode armazenar elementos de um tipo de dado, como um inteiro ou float.

lista: tipo de dado abstrato que descreve uma estrutura de dados que armazena valores ordenados.

superalocação: reservar mais memória para uma lista do que seria estritamente necessário e registrar quantos elementos a lista está armazenando e o espaço não usado que esta possui.

Desafio

1. Dado um array chamado `an_array` de números inteiros não negativos, retorne um array composto de todos os elementos pares seguidos de todos os elementos ímpares de `an_array`.

CAPÍTULO 10

Listas encadeadas

Aprender a escrever programas expande a mente, ajuda a pensar melhor e cria uma maneira de vermos as coisas que considero úteis em todas as áreas.

– Bill Gates, cofundador da Microsoft

Uma **lista encadeada** é outra implementação do tipo de dado abstrato lista. Como em um array, podemos inserir no final, inserir no início, procurar e excluir itens em uma lista encadeada. No entanto, os elementos não têm índices porque o computador não armazena os itens de uma lista encadeada em localizações sequenciais na memória. Em vez disso, a lista encadeada contém uma cadeia de **nós**, com cada nó contendo dados e a localização do próximo nó na cadeia. Os dados de cada nó que armazenam a localização do próximo nó na lista encadeada chamam-se **ponteiros**. O primeiro nó de uma lista encadeada chama-se **cabeça (head)**. Em geral, o último elemento contém um ponteiro que aponta para `None`, assim sabemos que se trata do último nó da lista (Figura 10.1).

Figura 10.1: Uma lista encadeada é uma cadeia de nós.

Ao contrário do que ocorre com o array, o computador pode armazenar os nós de uma lista encadeada em localizações não consecutivas na memória (Figura 10.2).

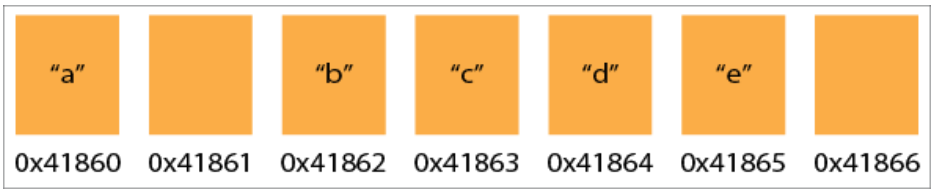

Figura 10.2: Uma lista encadeada não precisa armazenar os nós em localizações consecutivas na memória.

Na Figura 10.2, o computador está armazenando o caractere *a* no local 41860 da memória. Não precisa armazenar o elemento seguinte da lista, *b*, na próxima localização sequencial (41861). Em vez disso, pode armazenar *b* em qualquer localização da memória. Nesse caso, o computador está armazenando-o na localização de memória 41862.

Cada nó contém o ponteiro do endereço do próximo nó da lista, o que conecta todos os elementos (Figura 10.3). O primeiro elemento da lista encadeada, *a*, contém o ponteiro do endereço 41862 da memória, que é onde fica *b*, o segundo elemento. O elemento *b* contém o ponteiro da localização do endereço do próximo nó, *c* etc. Esse modelo cria uma sequência de elementos, todos mapeados em conjunto.

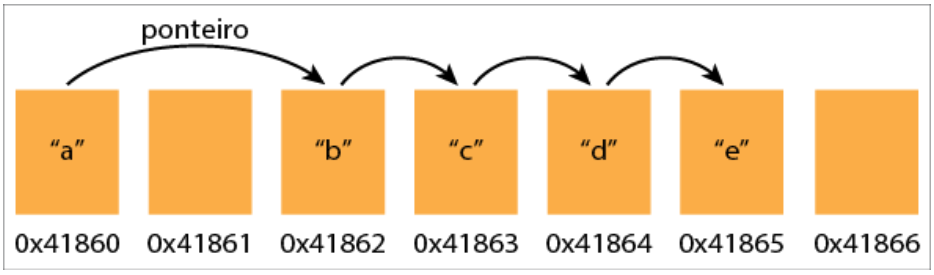

Figura 10.3: Ponteiros mapeiam os nós de uma lista encadeada.

Quando você inserir um elemento em uma lista encadeada, seu computador não precisará deslocar nenhum dado porque só terá de ajustar dois ponteiros. Por exemplo, a Figura 10.4 mostra o que acontece quando adicionamos o elemento *f* à lista depois de *a*.

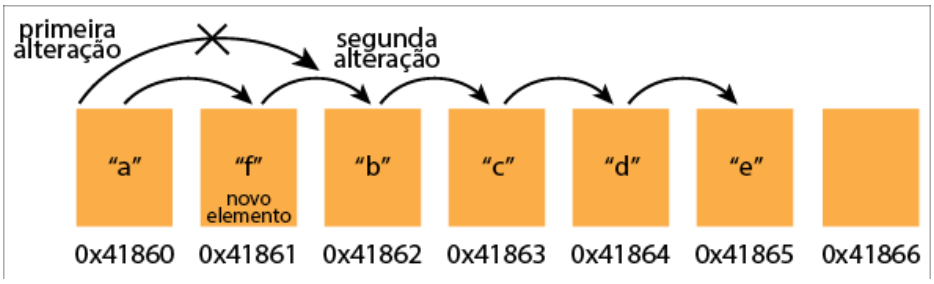

Figura 10.4: Inserir um elemento em uma lista encadeada requer ajuste em dois ponteiros.

O computador altera o ponteiro de *a* para a localização de *f* na memória e adiciona um ponteiro em *f*, apontando para o próximo item, *b* (Figura 10.4). Nada mais precisa mudar.

Existem muitos tipos de listas encadeadas. A lista da Figura 10.4 é uma lista encadeada simples. Uma **lista encadeada simples (singly linked list)** é um tipo de lista encadeada com ponteiros que só apontam para o próximo elemento. Em uma **lista duplamente encadeada (doubly linked list)**, cada nó contém dois ponteiros: um apontando para o próximo nó e o outro apontando para o nó anterior. Isso permite nos movermos por uma lista duplamente encadeada nas duas direções (Figura 10.5).

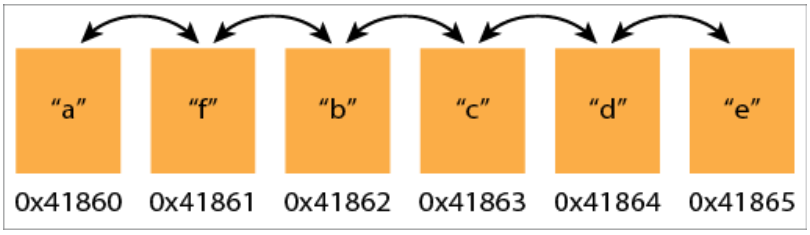

Figura 10.5: Uma lista duplamente encadeada tem ponteiros que seguem nas duas direções.

Só podemos iterar por uma lista encadeada simples começando na cabeça e indo até o fim. Por uma lista duplamente encadeada, podemos iterar da cabeça ao fim, mas também podemos voltar pelos nós.

Em uma **lista encadeada circular**, o último nó aponta de volta para o primeiro nó, o que nos permite ir do último elemento diretamente para o início da lista (Figura 10.6). Esse tipo de estrutura de dados é útil em aplicações que executam ciclos repetidos percorrendo dados que não têm um ponto inicial nem final claro. Por exemplo, você poderia usar uma lista

encadeada circular para rastrear os participantes de um jogo online de todos contra todos ou em um ambiente de pool de recursos em que os usuários se revezassem utilizando fatias alocadas de tempo de CPU. Uma lista encadeada contém um **ciclo** quando algum nó aponta para trás para um nó anterior.

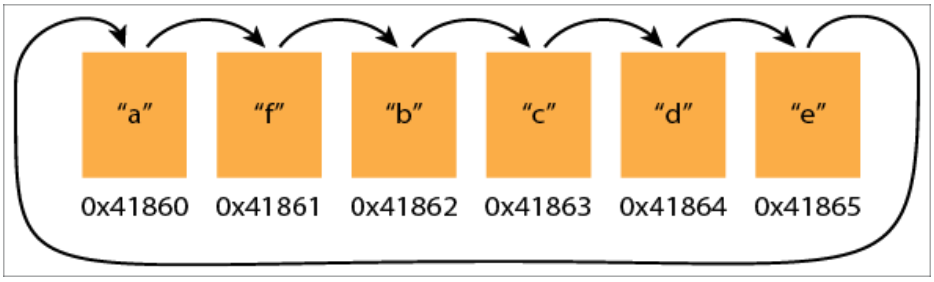

Figura 10.6: Uma lista encadeada circular aponta do fim novamente para a cabeça.

Desempenho da lista encadeada

Em um array, podemos acessar um item em tempo constante se soubermos seu índice. No entanto, a única maneira de acessar um elemento em uma lista encadeada é procurando-o em uma busca linear, o que tem complexidade O(*n*) (Figura 10.7). Por outro lado, adicionar ou remover um nó em uma lista encadeada tem complexidade O(1), enquanto inserir e excluir itens em um array tem complexidade O(*n*). Essa diferença é a vantagem mais significativa para o uso de uma lista encadeada em vez de um array. Para concluir, pesquisar uma lista encadeada, assim como um array, também tem complexidade O(*n*).

Estrutura de dados	Complexidade de tempo							
	Média				Pior			
	Acesso	Busca	Inserção	Exclusão	Acesso	Busca	Inserção	Exclusão
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Pilha	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Fila	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Lista encadeada	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Tabela Hash	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)
Árvore de busca binária	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)

Figura 10.7: Tempos de execução das operações das listas encadeadas.

Conceitualmente, uma lista encadeada é igual a uma lista ou um array da linguagem Python e você pode usar uma em qualquer situação em que os usaria. Como sabemos, adicionar ou remover dados em um array tem complexidade O(*n*), logo você deve considerar usar um array se estiver escrevendo um algoritmo que adicione ou remova dados com frequência, porque adicionar dados a uma lista encadeada tem complexidade O(1). Os sistemas de gerenciamento de memória dos sistemas operacionais utilizam muito listas encadeadas, assim como o fazem os bancos de dados e os sistemas empresariais de contabilidade e transações financeiras e comerciais. Você também pode usar uma lista encadeada para criar outras estruturas de dados. Por exemplo, poderia usar uma lista encadeada para criar duas estruturas de dados que veremos em capítulos posteriores: pilhas e filas. Por fim, as listas encadeadas são essenciais para a tecnologia blockchain existente por trás do movimento web 3.0, que preconiza o uso de criptomoedas. Os próprios blockchains são semelhantes às listas encadeadas e alguns as utilizam em sua tecnologia.

Embora as listas encadeadas sejam úteis em algumas situações, também apresentam várias desvantagens. A primeira delas é que cada nó deve conter um ponteiro que aponte para o nó seguinte. Os ponteiros de uma lista encadeada requerem recursos do sistema, logo as listas encadeadas demandam mais memória do que os arrays. Se o tamanho dos dados que você estiver armazenando em um único nó for pequeno, como o de um único inteiro, o tamanho de uma lista encadeada poderá ter duas vezes o

tamanho de um array contendo os mesmos dados.

A segunda desvantagem é que a lista encadeada não permite acesso aleatório. Em ciência da computação, **acesso aleatório** é quando podemos acessar dados aleatoriamente em tempo constante. Por exemplo, não podemos pular para o terceiro elemento de uma lista encadeada como poderíamos fazer em um array. Temos de começar na cabeça da lista e seguir cada ponteiro até alcançar o terceiro elemento. Embora isso possa ser uma desvantagem, existem algumas versões mais avançadas das listas encadeadas que resolvem esse problema.

Crie uma lista encadeada

Existem muitas maneiras de implementar uma lista encadeada em Python. Uma delas é definindo classes que representem a lista encadeada e seus nós. Veja como definir uma classe para representar um nó:

```
class Node:
def __init__(self, data, next=None):
self.data = data
self.next = next
```

Sua classe tem duas variáveis: a primeira, `data`, contém dados e a segunda, `next`, contém o nó seguinte da lista. Em Python, não precisamos lidar diretamente com os endereços da memória (como na linguagem de programação C) porque manipula isso automaticamente. Quando criamos um objeto, como a instância de uma classe chamada `Node`, Python retorna um ponteiro (ou uma referência) que aponta para ele. Na verdade, esse ponteiro é o endereço de onde estão os dados reais na memória do computador. Quando você atribuir objetos a variáveis em Python, estará lidando com ponteiros (referências), portanto poderá vincular facilmente os objetos porque Python fará todo o trabalho subjacente.

Em seguida, defina uma classe para representar a lista encadeada, com uma variável de instância chamada `head` para conter a cabeça da lista:

```
class LinkedList:
def __init__(self):
self.head = None
```

Dentro de sua classe `LinkedList`, você criará um método `append` que adicionará um novo nó à lista:

```
class LinkedList:
def __init__(self):
self.head = None
def append(self, data):
if not self.head:
self.head = Node(data)
return
current = self.head
while current.next:
current = current.next
current.next = Node(data)
```

O método `append` recebe os dados como parâmetro, cria um novo nó com eles e adiciona-o à lista encadeada.

Se a lista ainda não tiver uma cabeça, você criará um novo nó que passará a ser a cabeça da lista encadeada:

```
if not self.head:
self.head = Node(data)
return
```

Caso contrário, se a lista já tiver uma cabeça, você encontrará o último nó na lista encadeada, criará um novo e o atribuirá à variável de instância `next`. Para fazer isso, criará uma variável chamada `current` e a atribuirá à cabeça da lista:

```
current = self.head
```

Em seguida, usará um loop `while` que continuará sendo executado enquanto `current.next` não for `None`, porque você estará no fim da lista encadeada:

```
while current.next:
```

Dentro do loop `while`, você definirá continuamente `current` como `current.next` até `current` ser `None` (e o fim da lista ser alcançado) e o loop terminar:

```
while current.next:
current = current.next
```

Agora, a variável `current` contém o último nó da lista, logo você criará

um novo nó e o atribuirá a `current.next`:

```
current.next = Node(data)
```

Aqui, há um exemplo do uso de `append` para a inclusão de novos nós na lista encadeada:

```
a_list = LinkedList()
a_list.append("Tuesday")
a_list.append("Wednesday")
```

Você também pode adicionar o método `__str__` à classe `LinkedList` para exibir facilmente todos os nós da lista:

```
def __str__ (self):
node = self.head
while node is not None:
print(node.data)
node = node.next
a_list = LinkedList()
a_list.append("Tuesday")
a_list.append("Wednesday")
print(a_list)
```

```
>> Tuesday
>> Wednesday
```

Em Python, `__str__` é um método mágico. Quando definimos `__str__` dentro de uma classe, Python o chama quando o objeto é exibido.

Embora o Python não tenha listas encadeadas internas, tem uma estrutura de dados interna chamada *deque* que usa listas encadeadas internamente. A utilização da estrutura de dados interna *deque* nos permite tirar proveito da eficiência de uma lista encadeada sem ser preciso codificar uma:

```
from collections import deque
d = deque()
d.append('Harry')
d.append('Potter')
for item in d:
print(item)
```

```
>> 'Harry'
>> 'Potter'
```

Faça uma busca em uma lista encadeada

Você pode modificar levemente o método `append` da classe `LinkedList` da seção anterior para procurar um item em uma lista encadeada:

```
def search(self, target):
current = self.head
while current.next:
if current.data == target:
return True
else:
current = current.next
return False
```

Seu método chamado `search` recebe um parâmetro chamado `target`, que são os dados que você está procurando. Você percorrerá a lista encadeada e, se os dados do nó atual coincidirem com o valor de target, será retornado `True`:

```
if current.data == target:
return True
```

Se os dados do nó atual não coincidirem, você configurará `current` com o próximo nó da lista encadeada e continuará iterando:

```
else:
current = current.next
```

Se você alcançar o fim da lista encadeada sem encontrar uma ocorrência, saberá que os dados não existem na lista e retornará `False`:

```
return False
```

Você pode ver esse algoritmo em ação criando uma lista encadeada de 20 números aleatórios com valores de 1 a 30 e procurando o número 10:

```
import random
a_list = LinkedList()
for i in range(0, 20):
j = random.randint(1, 30)
a_list.append(j)
print(j, end= " ")
```

Removendo um nó de uma lista encadeada

Remover um nó de uma lista encadeada é outra questão comum em entrevistas técnicas. Também podemos usar busca linear para encontrar um nó em uma lista encadeada e excluí-lo. Um nó pode ser excluído com a alteração do ponteiro do nó anterior para que não aponte mais para o nó que desejamos remover (Figura 10.8).

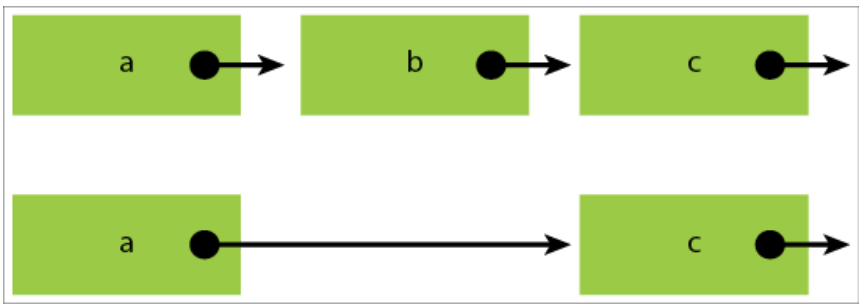

Figura 10.8: Para remover um nó, altere o ponteiro do nó anterior.

Veja como remover um nó de uma lista encadeada:

```
def remove(self, target):
if self.head == target:
self.head = self.head.next
return
current = self.head
previous = None
while current:
if current.data == target:
previous.next = current.next
previous = current
current = current.next
```

Seu método `remove` recebe um parâmetro, `target`, que são os dados que o nó que você deseja remover contém.

Dentro do método, primeiro você está manipulando o que ocorrerá se o nó a ser excluído for a cabeça da lista:

```
if self.head == target:
self.head = self.head.next
return
```

Se ele for a cabeça, você configurará `self.head` com o próximo nó da lista e retornará.

Caso contrário, você percorrerá a lista encadeada, registrando o nó atual e

o nó anterior nas variáveis `current` e `previous`:

```
current = self.head
previous = None
```

Em seguida, usará um loop `while` para percorrer a lista encadeada. Se encontrar os dados que está procurando, configurará `previous.next` com `current.next`, o que removerá o nó da lista:

```
while current:
if current.data == target:
previous.next = current.next
previous = current
current = current.next
```

Inverta uma lista encadeada

Você também precisa saber inverter uma lista encadeada. Para invertê-la, percorra-a, registrando tanto o nó atual quanto o nó anterior. Em seguida, faça o nó atual apontar para o anterior. Depois de alterar todos os ponteiros da lista encadeada, terá sido invertida (Figura 10.9).

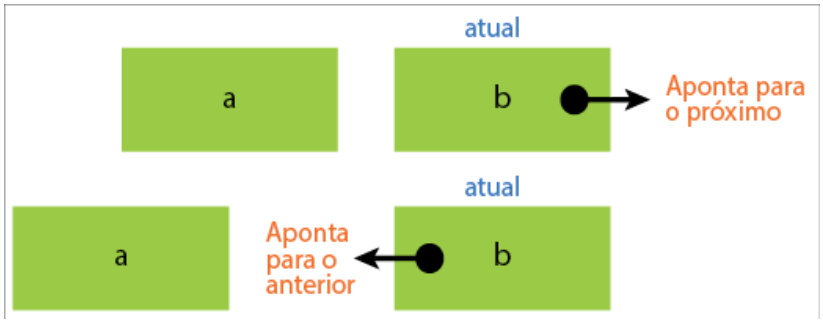

Figura 10.9: Invertendo uma lista encadeada.

Examinaremos o código que inverte uma lista encadeada:

```
def reverse_list(self):
current = self.head
previous = None
while current:
next = current.next
current.next = previous
previous = current
current = next
self.head = previous
```

Primeiro, use um loop `while` para percorrer a lista encadeada, utilizando as variáveis `current` e `previous` para registrar o nó atual e o anterior.

Dentro do loop `while`, atribua `current.next` à variável `next` para salvar esses dados quando atribuir `current.next` a `previous` na linha seguinte. Após configurar `current.next` com previous, você terá invertido o ponteiro para esse nó:

```
next = current.next
current.next = previous
```

Agora, basta configurar `previous` com `current` e `current` com `next` para continuar percorrendo a lista encadeada e alterando os ponteiros restantes:

```
previous = current
current = next
```

Após alterar todos os ponteiros, configure `self.head` com `previous`. Você está configurando a cabeça com `previous` e não com `current` porque, quando alcançar o fim da lista encadeada, `current` será `None` e `previous` conterá o que costumava ser o último nó, que foi transformado no primeiro nó ao ser configurado com a cabeça da lista.

Encontrando o ciclo de uma lista encadeada

Anteriormente, você aprendeu que o último elemento de uma lista encadeada circular aponta de volta para a cabeça da lista (Figura 10.6). Outra questão comum em entrevistas é detectar se uma lista encadeada contém um ciclo. Em outras palavras, verificar se o último item da lista aponta para algum item em vez de ter `None` como valor de sua variável "next". Um algoritmo que detecta o ciclo de uma lista encadeada chama-se *algoritmo da tartaruga e da lebre*. Nele, percorremos a lista encadeada com duas velocidades diferentes, registrando os nós em uma variável `slow` e em uma variável `fast`. Se a lista encadeada for um círculo, a variável `fast` acabará alcançando a variável `slow` e as duas variáveis serão iguais. Se isso ocorrer, você saberá que a lista encadeada é circular. Se o fim da lista for alcançado sem que esse fato ocorra, ficará claro que esta não contém um ciclo.

Veja como você pode implementar o algoritmo da tartaruga e da lebre:

```
def detect_cycle(self):
slow = self.head
fast = self.head
while True:
try:
slow = slow.next
fast = fast.next.next
if slow is fast:
return True
except:
return False
```

Comece com duas variáveis, `fast` e `slow`:

```
slow = self.head
fast = self.head
```

Em seguida, crie um loop infinito:

```
while True:
```

Dentro do loop infinito, atribua o próximo nó da lista encadeada a `slow` e o nó posterior a ele a `fast`. Insira esse código dentro de um bloco `try` porque se a lista encadeada não for circular, `fast` acabará sendo `None`, o que significa que você chamará `fast.next.next` em `None`, o que causará um erro. O bloco `try` também impedirá que o programa falhe se a entrada for uma lista vazia ou uma lista não circular com um único item.

```
try:
slow = slow.next
fast = fast.next.next
```

Agora, verifique se `slow` e `fast` são o mesmo objeto. Você não está verificando se os valores dos dois nós da lista encadeada são iguais porque os mesmos dados podem aparecer em mais de um nó. Em vez disso, use a palavra-chave `is` para saber se os dois nós são o mesmo objeto. Se forem, retorne `True` porque a lista encadeada será circular.

```
if slow is fast:
return True
```

Se houver um erro, isso significará que você chamou `.next.next` em `None`, ou seja, a lista encadeada não é circular e `False` será retornado.

Vocabulário

acesso aleatório: torna possível o acesso aos dados aleatoriamente em tempo constante.

cabeça: primeiro nó de uma lista encadeada.

ciclo: quando algum nó de uma lista encadeada aponta para o nó anterior.

lista duplamente encadeada: um tipo de lista encadeada no qual cada nó contém dois ponteiros, um apontando para o nó seguinte e o outro apontando para o nó anterior, o que permite percorrê-la nas duas direções.

lista encadeada: uma implementação do tipo de dados abstrato lista.

lista encadeada circular: um tipo de lista encadeada no qual o último nó aponta para o primeiro, o que nos permite ir do último ao primeiro elemento da lista.

lista encadeada simples: um tipo de lista encadeada com ponteiros que apontam apenas para o elemento seguinte.

nó: parte de uma estrutura de dados que contém trechos de dados e pode se conectar com outros trechos de dados.

ponteiro: dados de cada nó que contêm a localização do próximo nó de uma lista encadeada.

Desafios

1. Crie uma lista encadeada contendo os números de 1 a 100. Em seguida, exiba cada nó da lista.
2. Crie duas listas encadeadas: uma contendo um ciclo e a outra sem ciclo. Certifique-se de que cada lista tenha um método `detect_cycle` que identifique se ela tem um ciclo. Chame `detect_cycle` em cada lista.

CAPÍTULO 11

Pilhas (Stacks)

Se você quiser criar e ser um visionário, provavelmente trabalhará com tecnologia.

– Steph Curry

Uma **pilha (stack)** é um tipo de dado abstrato e uma estrutura de dados linear que só nos permite remover o elemento inserido mais recentemente. Poderíamos imaginar uma pilha usando como analogia uma pilha de livros ou pratos. Só podemos adicionar ou remover um livro no topo da pilha. Para alcançar o terceiro livro, primeiro temos de remover todos os que estão acima dele.

A pilha é um exemplo de estrutura de dados LIFO (last-in, first-out; último a entrar, primeiro a sair). Uma **estrutura de dados último a entrar, primeiro a sair** é aquela em que o último item inserido na estrutura de dados é o primeiro a sair dela. Já que só podemos acessar seus conteúdos um a um, a pilha também é um exemplo de **estrutura de dados de acesso limitado**: um tipo de estrutura de dados que nos força a acessar suas informações em uma ordem específica.

As pilhas têm duas operações principais: inserção (pushing) e remoção (popping) (Figura 11.1). **Inserir** um item em uma pilha significa colocar um novo item nela. **Remover** um item de uma pilha significa retirar seu último item. As pilhas também podem ter operações adicionais, como a de **examinar (peeking)**, que significa verificar o elemento do topo de uma pilha sem removê-lo.

As pilhas podem ser limitadas (bounded) ou ilimitadas (unbounded). Uma **pilha limitada** é aquela que limita quantos itens podemos adicionar a ela, enquanto uma **pilha ilimitada** não restringe quantos itens podem

ser adicionados. Se você ainda estiver confuso no que diz respeito à diferença entre um tipo de dado abstrato e uma estrutura de dados, a pilha poderá ajudá-lo a entender. O tipo de dado abstrato pilha descreve a ideia de uma estrutura de dados que só nos permite acessar o item mais recente inserido nela. No entanto, existem várias maneiras de criar uma estrutura de dados como essa. Por exemplo, você poderia criar uma pilha definindo uma classe que usasse internamente uma lista encadeada ou um array para registrar os itens. Quando você estiver escrevendo o código de uma pilha usando um array ou uma lista encadeada, terá passado da ideia abstrata de pilha para uma estrutura de dados: a implementação real de um tipo de dado abstrato.

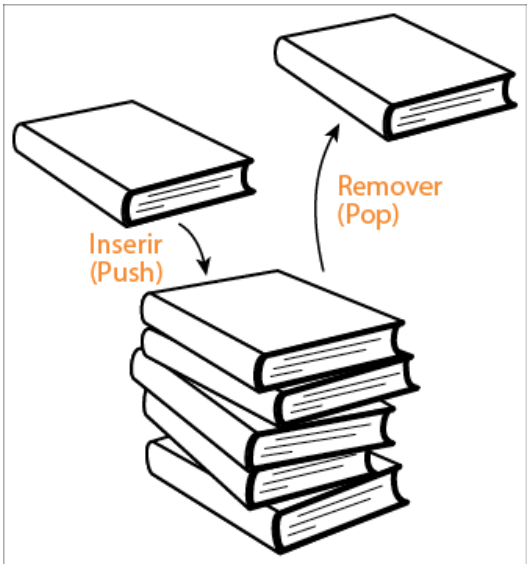

Figura 11.1: Os dados podem ser inseridos em uma pilha ou removidos dela.

Quando usar pilhas

Inserir e remover elementos em uma pilha são operações de complexidade O(1). Embora as pilhas sejam eficientes para a inclusão e a remoção de dados, não são tão eficientes para operações que demandem acessar a pilha inteira (Figura 11.2). Por exemplo, suponhamos que você precisasse exibir os conteúdos de uma pilha. Uma solução seria exibir cada objeto quando você o removesse. Exibir cada elemento quando for retirado da pilha tem complexidade O(*n*). No entanto, isso também

produzirá uma lista de objetos na ordem inversa. Sua pilha ficará vazia porque você terá removido todos os itens. Outra solução seria remover cada elemento da pilha original à medida que você os adicionasse a uma pilha temporária. Depois, você poderia exibir cada elemento quando o removesse da pilha temporária e o colocasse de volta na original. Contudo, essa solução requer mais recursos porque é preciso armazenar todos os dados em uma pilha temporária. Ela tem complexidade O(2*n*): demoraria duas vezes mais do que exibir os itens de um array.

Estrutura de dados	Complexidade de tempo							
	Média				Pior			
	Acesso	Busca	Inserção	Exclusão	Acesso	Busca	Inserção	Exclusão
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Pilha	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Fila	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Lista encadeada	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Tabela Hash	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)
Árvore de busca binária	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)

Figura 11.2: Tempos de execução das operações das pilhas.

As pilhas são uma das estruturas de dados mais frequentemente usadas em computação. Cientistas da computação usam pilhas para implementar algoritmos de busca em largura (breadth-first search algorithms) para procurar dados em árvores e grafos (que você conhecerá em capítulos posteriores). Os sistemas de tempo de execução (run-time) de linguagens como Python e Java utilizam uma pilha internamente para manipular chamadas de funções. Os compiladores usam pilhas para fazer o parsing de expressões, principalmente quando existem expressões que empregam pares de parênteses aninhados, como em expressões aritméticas padrão, ou pares de colchetes ou chaves aninhados.

Cientistas da computação também usam pilhas nos algoritmos de backtracking encontrados em machine learning (aprendizado de máquina) e em outras áreas da inteligência artificial. Já que tanto a inclusão quanto a remoção de elementos em uma pilha têm complexidade

O(1), serão uma ótima opção sempre que você estiver adicionando e removendo elementos de dados com frequência. Por exemplo, em geral programas que precisam de um mecanismo de desfazer (undo) utilizam uma pilha ou duas para manipular as operações "desfazer" e "refazer". Navegadores web, por exemplo, costumam usar duas pilhas para retroceder e avançar em seu histórico de navegação. Como o acesso a cada elemento de uma pilha tem complexidade O(*n*), elas não são a melhor opção para algoritmos que precisem acessar cada dado de um conjunto de dados continuamente.

Criando uma pilha

Como já vimos, existem algumas maneiras de usar uma pilha em Python. Uma delas é criando uma classe `Stack` e gerenciando seus dados internamente com um array:

```
class Stack:
def __init__(self):
self.items = []
def push(self, data):
self.items.append(data)
def pop(self):
return self.items.pop()
def size(self):
return len(self.items)
def is_empty(self):
return len(self.items) == 0
def peek(self):
return self.items[-1]
```

Dentro do método `__init__` da classe `Stack`, defina uma variável de instância chamada `items` e configure-a com uma lista vazia. É nessa lista que você registrará os itens da pilha:

```
class Stack:
def __init__(self):
self.items = []
```

Em seguida, defina o método `push` da pilha. Você usará o método interno `append` do Python para adicionar novos dados ao fim de `items`:

```
def push(self, data):
self.items.append(data)
```

O próximo método da pilha é `pop`. Dentro de `pop`, você usará o método interno `pop` do Python para retornar o item adicionado mais recentemente à pilha:

```
def pop(self):
return self.items.pop()
```

O método seguinte da classe `Stack` chama-se `size` e usa o método `len` para retornar o tamanho da pilha:

```
def size(self):
return len(self.items)
```

O método `is_empty` verifica se a pilha está vazia:

```
def is_empty(self):
return len(self.items) == 0
```

Para concluir, o último método chama-se `peek` e retorna o último item da pilha.

```
def peek(self):
return self.items[-1]
```

Você também pode implementar uma classe `Stack` usando uma lista encadeada interna. Veja como criar uma pilha simples (apenas com push e pop) utilizando uma lista encadeada:

```
class Node:
def __init__(self, data):
self.data = data
self.next = None
class Stack:
def __init__(self):
self.head = None
def push(self, data):
node = Node(data)
if self.head is None:
self.head = node
else:
node.next = self.head
self.head = node
def pop(self):
if self.head is None:
```

```
raise IndexError('pop from empty stack')
poppednode = self.head
self.head = self.head.next
return poppednode.data
```

Primeiro, defina uma classe `Node` para representar os nós da lista encadeada interna da pilha:

```
class Node:
def __init__(self, data):
self.data = data
self.next = None
```

Dentro da classe `Stack`, defina uma variável de instância para a cabeça da lista encadeada:

```
class Stack:
def __init__(self):
self.head = None
```

Em seguida, defina um método chamado `push`. Dentro de `push`, crie um novo nó. Se a lista encadeada não tiver uma cabeça, você a atribuirá ao novo nó. Caso contrário, torne esse nó a cabeça da lista:

```
def push(self, data):
node = Node(data)
if self.head is None:
self.head = node
else:
node.next = self.head
self.head = node
```

Agora, defina um método chamado `pop`:

```
def pop(self):
if self.head is None:
raise IndexError('pop from empty stack')
poppednode = self.head
self.head = self.head.next
return poppednode.data
```

Se alguém tentar remover algo de sua lista quando estiver vazia, você lançará uma exceção:

```
if self.head is None:
raise IndexError('pop from empty stack')
```

Caso contrário, removerá o primeiro item da lista encadeada e o

retornará:

```
poppednode = self.head
self.head = self.head.next
return poppednode.data
```

Aqui, há um exemplo de criação de uma pilha com o uso desse código e da inserção e remoção de itens:

```
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
for i in range(3):
print(stack.pop())
```

```
>> 3
>> 2
>> 1
```

Por fim, você também pode usar uma lista como pilha. Veja como funciona:

```
stack = []
print(stack)
stack.append('Kanye West')
print(stack)
stack.append('Jay-Z')
print(stack)
stack.append('Chance the Rapper')
print(stack)
stack.pop()
print(stack)
```

```
>> []
>> ['Kanye West']
>> ['Kanye West', 'Jay-Z']
>> ['Kanye West', 'Jay-Z', 'Chance the Rapper']
>> ['Kanye West', 'Jay-Z']
```

As listas do Python vêm com os métodos `append` e `pop`. O método `append` adiciona um item ao fim da lista, o que é o mesmo que inserir um item em uma pilha. O método `pop` remove um item do fim da lista. Se você não especificar que item será removido, removerá o último item.

Na primeira vez que a pilha foi exibida, estava vazia porque você ainda não havia adicionado nada:

```
>> []
```

Você inseriu três itens – `'Kanye West'`, `'Jay-Z'` e `'Chance the Rapper'` – na pilha com estas linhas de código:

```
stack.append('Kanye West')
stack.append('Jay-Z')
stack.append('Chance the Rapper')
```

Depois, removeu o último elemento, `'Chance the Rapper'`, deixando os dois primeiros:

```
stack.pop()
```

Por isso, na última vez que a pilha foi exibida, `'Chance the Rapper'` estava faltando:

```
>> ['Kanye West', 'Jay-Z']
```

É claro que quando você usar uma lista Python como pilha, não ficará restrito a remover os itens na ordem em que os inseriu. Logo, se quiser impor essa restrição, terá de criar uma classe `Stack`.

Usando pilhas para inverter strings

Uma questão comum em entrevistas para vagas de programador Python é inverter uma string de três maneiras diferentes. Se você está familiarizado com o Python, sabe que pode inverter uma string desta forma:

```
a_string[::-1]
```

ou também assim:

```
''.join(reversed('a string'))
```

No entanto, no que diz respeito a inverter uma string de uma terceira maneira, você pode não saber o que fazer. A chave está em algo que vimos neste capítulo: é possível usar uma pilha para inverter uma string porque quando removemos caracteres de uma pilha, estes são extraídos na ordem inversa (Figura 11.3).

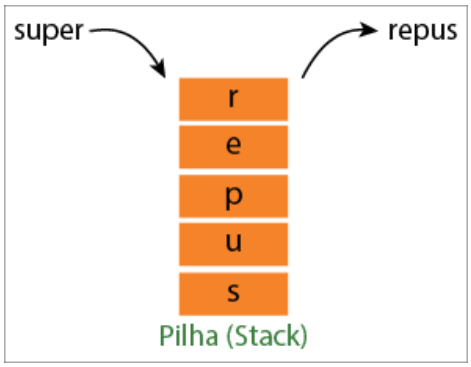

Figura 11.3: Se você remover os caracteres de `super`*, obterá* `repus`.

Veja como usar uma pilha para inverter uma string:

```
def reverse_string(a_string):
stack = []
string = ""
for c in a_string:
stack.append(c)
for c in a_string:
string += stack.pop()
return string
print(reverse_string("Bieber"))
```

```
>> "rebeiB"
```

A função `reverse_string` recebe uma string como parâmetro:

```
def reverse_string(a_string):
```

Dentro da função, use um loop `for` para inserir cada caractere na pilha:

```
for c in a_string:
stack.append(c)
```

Em seguida, use outro loop para percorrer a pilha e adicionar cada caractere à variável `a_string` à medida que for removendo-os:

```
for c in a_string:
string += stack.pop()
```

Para concluir, retorne a string invertida:

```
return string
```

Pilha mínima

Outro desafio comum em entrevistas técnicas é projetar uma estrutura de

dados que suporte operações de pilha, como a inserção e a remoção, e inclua um método que retorne o menor elemento. Todas as operações de pilha devem ter complexidade O(1). O segredo para a solução desse desafio é usar duas pilhas: uma principal e uma mínima. A pilha principal registrará todas as operações de inserção e remoção e a pilha mínima (min stack) registrará o menor elemento. Aprender como resolver esse problema não é útil apenas para obter um bom resultado em uma entrevista técnica: saber como criar uma pilha que registre o menor número será útil em várias situações que você encontrará em sua programação diária.

Veja como implementar uma pilha que registre o elemento mínimo em Python:

```
class MinStack():
def __init__(self):
self.main = []
self.min = []
def push(self, n):
if len(self.main) == 0:
self.min.append(n)
elif n <= self.min[-1]:
self.min.append(n)
else:
self.min.append(self.min[-1])
self.main.append(n)
def pop(self):
self.min.pop()
return self.main.pop()
def get_min(self):
return self.min[-1]
```

Primeiro, defina uma classe chamada `MinStack`. Dentro de `__init__`, defina duas variáveis de instância: `main` e `min`. Atribua as duas variáveis a listas vazias. Você usará `main` para registrar a pilha principal e `min` para registrar o menor elemento.

```
class MinStack():
def __init__(self):
self.main = []
self.min = []
```

Depois, defina o método `push` da pilha. Dentro de `push`, verifique se `self.main` está vazia, porque se estiver, seja qual for o valor de `n`, será o menor número da pilha. Se `self.main` estiver vazia, acrescente `n` a `min`.

```
def push(self, n):
if len(self.main) == 0:
self.min.append(n)
```

Se `self.main` não estiver vazia, verifique se `n` é menor que ou igual ao último item de `self.min`. O último item de `self.min` precisa ser sempre o menor número da pilha, logo se `n` for menor que ou igual ao último item de `self.min`, acrescente `n` a `self.min`:

```
elif n <= self.min[-1]:
self.min.append(n)
```

Se `n` não for menor que ou igual ao último item de `self.min` (portanto, for maior), acrescente o último item de `self.min` a `self.min`:

```
else:
self.min.append(self.min[-1])
```

Acrescentar o último item de `self.min` a `self.min` manterá o número de itens de `self.main` igual ao de `self.min` para rastrear o menor número da pilha.

Vejamos um exemplo. Quando você inserir o primeiro número na pilha, as duas pilhas internas ficarão assim:

```
min_stack = MinStack()
min_stack.push(10)
print(min_stack.main)
print(min_stack.min)
>> [10]
>> [10]
```

Aqui está o que acontecerá quando você inserir um número maior na pilha:

```
min_stack.push(15)
print(min_stack.main)
print(min_stack.min)
>> [10, 15]
>> [10, 10]
```

Observe que `min_stack.main` é uma pilha normal, armazenando os itens na ordem em que são inseridos:

```
>> [10, 15]
```

No entanto, `min_stack.min` não registra os itens conforme entram na pilha. Em vez disso, registra o número que for o menor. Nesse caso, tem 10 duas vezes:

```
>> [10, 10]
```

Quinze não está em `min_stack.min` porque nunca será o menor item da pilha.

Quando você chamar `get_min`, retornará o último item de `self.min`, que é o menor número da pilha:

```
print(min_stack.get_min())
>> 10
```

Aqui, está retornando 10.

Após você remover um item, suas duas pilhas ficarão assim:

```
min_stack.pop()
print(min_stack.main)
print(min_stack.min)
>> [10]
>> [10]
```

Quando chamar `get_min` pela segunda vez, o método retornará 10 novamente:

```
print(min_stack.get_min())
>> 10
```

Quando você chamar `pop` pela última vez, as pilhas ficarão vazias:

```
min_stack.pop()
print(min_stack.main)
print(min_stack.min)
>> []
>> []
```

Como você pode ver, `self.min` registrou o menor número da pilha sem em momento algum precisar registrar o número 15.

Parênteses empilhados

Uma vez, quando estava sendo entrevistado em uma startup, fizeram-se resolver o seguinte problema: "Dada uma string, use uma pilha para verificar se tem parênteses balanceados, o que significa que sempre que houver um parêntese de abertura, deverá haver um parêntese de fechamento".

```
(str(1)) # Balanceados
print(Hi!)) # Não balanceados
```

Infelizmente, não forneci uma boa resposta. Sua reação inicial a esse problema poderia ser resolvê-lo rapidamente, fazendo o parsing da string e usando um contador para parênteses de abertura e outro para parênteses de fechamento. Se os contadores forem iguais no fim da string, os parênteses estarão balanceados. No entanto, o que aconteceria se você encontrasse uma string como a seguinte?

```
a_string = ")( )("
```

Nesse caso, sua solução não funcionará.

Uma maneira melhor de resolver esse problema seria usando uma pilha. Primeiro, percorremos cada caractere da string e, se um parêntese estiver aberto, será inserido na pilha. Se for de fechamento, verificaremos se já existe um parêntese de abertura na pilha. Se não houver, isso significará que a string não está balanceada. Se houver, removeremos um parêntese de abertura da pilha. Se houver um número igual de parênteses de abertura e fechamento na string, a pilha estará vazia no fim do loop. Se a pilha não estiver vazia é porque não há o mesmo número de parênteses de abertura e fechamento.

Vejamos o código:

```
def check_parentheses(a_string):
stack = []
for c in a_string:
if c == "(":
stack.append(c)
if c == ")":
if len(stack) == 0:
return False
```

```
else:
stack.pop()
return len(stack) == 0
```

A função `check_parentheses` recebeu como parâmetro a string na qual queremos procurar parênteses balanceados:

```
def check_parentheses(a_string):
```

Dentro da função, crie uma pilha usando uma lista:

```
stack = []
```

Use um loop `for` para percorrer os caracteres de `a_string`:

```
for c in a_string:
```

Se o caractere for um parêntese de abertura, insira-o na pilha:

```
if c == "(":
stack.append(c)
```

Se o símbolo for um parêntese de fechamento e a pilha estiver vazia, retorne `False` porque não há um parêntese de abertura correspondente, o que significa que a string não está balanceada. Se houver parênteses de abertura na pilha, remova um para fazer o balanceamento com o parêntese de fechamento.

```
if c == ")":
if len(stack) == 0:
return False
else:
stack.pop()
```

Quando o loop `for` terminar, retorne se o tamanho da pilha for ou não zero:

```
return len(stack) == 0
```

Se a função retornar `True`, os parênteses estarão balanceados; caso contrário, será porque não estão.

Saber como resolver esse problema não é útil apenas em entrevistas técnicas. Os compiladores de linguagens como Python e Java têm códigos como esse para o parsing e a avaliação de expressões. Se algum dia você precisar criar a própria linguagem de programação ou escrever um código para fazer o parsing de dados com símbolos de abertura e fechamento,

poderá escrever um código assim para fazer a avaliação.

Vocabulário

estrutura de dados de acesso limitado: tipo de estrutura de dados que força o acesso a suas informações em uma ordem específica.

estrutura de dados último a entrar, primeiro a sair: estrutura de dados na qual o último item inserido é o primeiro a ser removido.

inserção: colocar um novo item em uma pilha.

inspeção: examinar o elemento do topo de uma pilha sem removê-lo.

pilha: tipo de dado abstrato e estrutura de dados linear que só permite remover o elemento mais recente adicionado.

pilha ilimitada: pilha que não limita o número de itens que podem ser adicionados a ela.

pilha limitada: pilha que limita o número de itens que podem ser adicionados a ela.

remoção: extração do último item de uma pilha.

Desafios

1. Modifique seu programa de string balanceada para verificar se dois parênteses, `()`, e duas chaves, `{}`, estão balanceados em uma string.
2. Projete uma pilha de valor máximo que permita que você insira, remova e rastreie o número mais alto de sua pilha em tempo O(1).

CAPÍTULO 12

Filas (Queues)

Todos os estudantes deveriam ter a oportunidade de aprender a programar. A Ciência da Computação é a base da criatividade e da expressão na era moderna. Os programadores de computador do futuro revolucionarão a medicina.

– Anne Wojcicki

Uma **fila (queue)** é um tipo de dado abstrato e uma estrutura de dados linear na qual só podemos adicionar itens ao final e removê-los do início (Figura 12.1). O tipo de dado abstrato fila descreve uma estrutura de dados que funciona como as filas dos caixas de uma mercearia: a primeira pessoa da fila é a primeira a pagar e quem vai chegando entra no final.

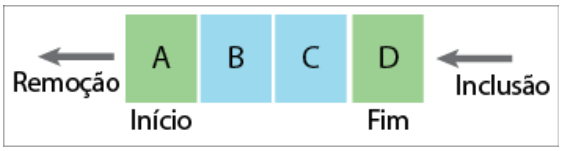

Figura 12.1: Em uma fila, adicionamos itens ao final e os removemos do início.

A fila é um exemplo de estrutura de dados FIFO (first-in, first-out; primeiro a entrar, primeiro a sair). Como o nome sugere, em uma **estrutura de dados primeiro a entrar, primeiro a sair**, o primeiro item a entrar na estrutura é o primeiro a ser removido. As filas, como as pilhas, são estruturas de dados de acesso limitado.

As filas têm duas operações principais: o enfileiramento (enqueueing) e o desenfileiramento (dequeueing) (Figura 12.2). **Enfileirar** significa adicionar um item a uma fila, enquanto **desenfileirar** significa remover um item. Enfileiramos elementos no final da fila e os desenfileiramos no

início.

Existem várias maneiras de implementar o tipo de dado abstrato fila como uma estrutura de dados. Por exemplo, como ocorre com as pilhas, podemos implementar uma estrutura de dados fila usando um array ou uma lista encadeada. Além disso, como as pilhas, as filas podem ser limitadas ou ilimitadas. Uma **fila limitada** é aquela que limita o número de itens que podemos adicionar a ela, enquanto uma **fila ilimitada** não limita quantos itens podem ser adicionados. Você pode criar uma fila limitada usando um array ou uma fila ilimitada usando uma lista encadeada (também é possível implementar uma fila limitada com o uso de uma lista encadeada se registrarmos o número de itens armazenados nela).

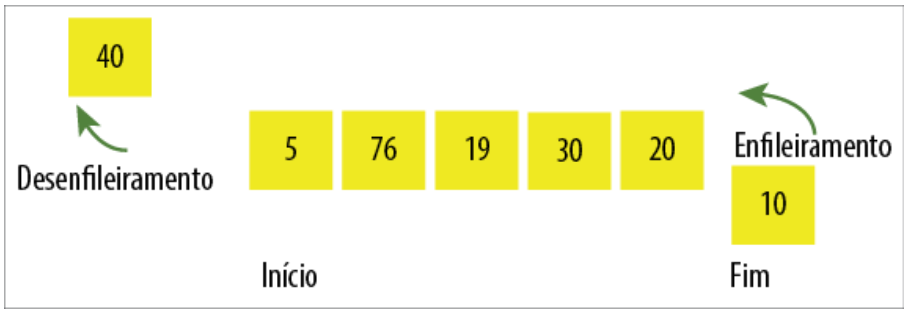

Figura 12.2: As principais operações das filas são o enfileiramento e o desenfileiramento.

Quando usar filas

Como as pilhas, as filas são eficientes para a inclusão ou a remoção de dados (Figura 12.3). Tanto o enfileiramento quanto o desenfileiramento têm complexidade O(1), independentemente do tamanho da fila. Também como as pilhas, as filas não são eficientes no acesso a dados individuais porque a busca de um item demanda percorrer seus elementos. Isso significa que tanto acessar um item quanto fazer uma busca em uma fila tem complexidade O(*n*).

Estrutura de dados	Complexidade de tempo							
	Média				Pior			
	Acesso	Busca	Inserção	Exclusão	Acesso	Busca	Inserção	Exclusão
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Pilha	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Fila	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Lista encadeada	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Tabela Hash	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)
Árvore de busca binária	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)

Figura 12.3: Tempos de execução das operações da fila.

Como programador, você usará as filas com frequência. Serão uma estrutura de dados ideal quando você estiver programando qualquer coisa relacionada com o padrão primeiro a chegar, primeiro a ser atendido. Por exemplo, uma fila seria útil para a programação de um sistema telefônico automatizado que colocasse quem ligou em uma fila, quando todos os operadores disponíveis estivessem ocupados e depois conectasse quem fez a primeira chamada, seguido dos próximos que discaram. Os sistemas operacionais usam filas para manipular solicitações de gravações de dados em uma unidade de disco rígido, fazer o streaming de áudio e vídeo e enviar e receber pacotes de rede. Por outro lado, os servidores web usam filas para manipular solicitações recebidas.

Sempre que aparecer a mensagem "buffering", isso significará que provavelmente o sistema de software que você está usando está esperando que dados recebidos sejam adicionados à sua fila para processamento. Por exemplo, para assegurar um streaming suave, em geral os sistemas de streaming de áudio e vídeo definem uma fila para dados recebidos. Suponhamos que você estivesse assistindo a um filme na Netflix. O software de sua TV responsável por exibi-lo pode ter de aguardar algum tempo antes de começar o filme a fim de preencher sua fila com os dados de vídeo que a Netflix está enviando. Quando o software da televisão permitir que você comece a assistir, mais dados chegarão e o sistema os adicionará à fila. O uso de uma fila permite que o reprodutor de vídeo retire pacotes do início dela em um intervalo de tempo fixo e constante,

suavizando a experiência de visualização, mesmo se os dados recebidos chegarem com uma taxa inconstante. Se pacotes de dados em demasia chegarem com muita rapidez, o sistema os manterá na fila até estar pronto para eles. Se os pacotes chegarem muito lentamente, o sistema poderá continuar reproduzindo os que estiverem na fila até ela terminar. O ideal é que isso não ocorra, mas quando você encontrar uma mensagem "buffering", isso terá ocorrido porque a fila ficou sem dados e o streaming não poderá continuar até a fila começar a ser preenchida novamente.

Imaginemos como seria um programa como esse. Provavelmente, teria um loop que seria executado até você ter terminado de assistir ao programa. Dentro do loop, haveria um algoritmo. Esse algoritmo seria responsável por adicionar dados à fila e remover e exibir os dados para o usuário na forma de vídeo. Juntos, o algoritmo e uma estrutura de dados apropriada (uma fila) são tudo que é necessário para você fazer o streaming de um filme para sua televisão ou laptop e ilustram melhor porque programas = estruturas de dados + algoritmos.

Criando uma fila

Existem várias maneiras de implementar uma fila em Python. Uma delas seria definindo uma classe `Queue` que usasse uma lista encadeada para registrar dados internamente. Veja como criar uma fila usando uma lista encadeada:

```
class Node:
def __init__(self, data, next=None):
self.data = data
self.next = next
class Queue:
def __init__(self):
self.front = None
self.rear = None
self._size = 0
def enqueue(self, item):
self._size += 1
node = Node(item)
if self.rear is None:
self.front = node
```

```
self.rear = node
else:
self.rear.next = node
self.rear = node
def dequeue(self):
if self.front is None:
raise IndexError('pop from empty queue')
self._size -= 1
temp = self.front
self.front = self.front.next
if self.front is None:
self.rear = None
return temp.data
def size(self):
return self._size
```

Primeiro, defina uma classe `Node` para representar os nós da lista encadeada interna da fila:

```
class Node:
def __init__(self, data, next=None):
self.data = data
self.next = next
```

Dentro da fila, registre seus itens inicial e final nas variáveis `self.front` e `self.rear`. Você deve registrar o início e o fim da fila para executar enfileiramentos e desenfileiramentos em tempo constante. Registre também o tamanho da fila na variável `self._size`:

```
def __init__(self):
self.front = None
self.rear = None
self._size = 0
```

Em seguida, defina um método chamado `enqueue` que adicione um item ao fim da fila:

```
def enqueue(self, item):
self._size += 1
node = Node(item)
if self.rear is None:
self.front = node
self.rear = node
else:
self.rear.next = node
```

```
self.rear = node
```

Seu método receberá como parâmetro os dados que você deseja armazenar na fila:

```
def enqueue(self, item):
```

Dentro de `enqueue`, primeiro incremente `self._size` em 1 unidade porque você está adicionando um novo item à fila. Depois, crie um novo nó para armazenar o item na lista encadeada interna:

```
self._size += 1
node = Node(item)
```

Se `self.rear` for `None`, significa que a fila está vazia, portanto configure `self.front` e `self.rear` com o nó que acabou de criar (já que há apenas um item na fila, este representa tanto o início quanto o fim). Caso contrário, atribua o novo nó a `self.rear.next` para adicioná-lo à lista encadeada interna. Em seguida, atribua o novo nó a `self.rear`, para que fique no fim da fila:

```
if self.rear is None:
self.front = node
self.rear = node
else:
self.rear.next = node
self.rear = node
```

Agora, defina um método chamado `dequeue` para remover um item do início da fila:

```
def dequeue(self):
if self.front is None:
raise IndexError('pop from empty queue')
self._size -= 1
temp = self.front
self.front = self.front.next
if self.front is None:
self.rear = None
return temp.data
```

A primeira linha de código do método lançará uma exceção se você tentar desenfileirar um item quando a fila estiver vazia:

```
if self.front is None:
raise IndexError('pop from empty queue')
```

Quando você chamar **dequeue**, removerá e retornará o item do início da fila. Para fazê-lo, armazene o nó inicial (**self.front)** em **temp** de modo a referenciá-lo posteriormente após o remover da lista encadeada interna:

```
temp = self.front
```

Em seguida, remova o item inicial da lista encadeada interna, definindo **self.front** como **self.front.next**:

```
self.front = self.front.next
```

Se não houver mais itens na fila após você remover o item inicial, configure **self.rear** com **None** porque não haverá mais um item no fim da fila:

```
if self.front is None:
self.rear = None
```

O último método a ser definido chama-se **size** e retorna o número de itens da fila:

```
def size(self):
return self._size
```

Com esses três métodos, você criou uma fila simples usando uma lista encadeada com a qual poderá adicionar e remover dados e verificar seu tamanho. Agora, poderá usar sua fila desta forma:

```
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.size())
for i in range(3):
print(queue.dequeue())
```

```
>> 3
>> 1
>> 2
>> 3
```

No código anterior, você criou uma fila, adicionou os números 1, 2 e 3 e exibiu o tamanho e cada item dela.

Examinemos o que ocorrerá dentro da classe de fila quando você executar esse programa. Quando **enqueue** for chamado, não haverá itens na fila,

logo você adicionará um nó à lista encadeada interna e ele será o início e o fim da fila (Figura 12.4).

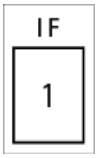

Figura 12.4: Quando houver um item na fila, ele será o início e o fim.

Em seguida, adicione o número 2 à fila. Agora, existem dois nós na lista encadeada interna e o nó que contém 1 não é mais o fim da fila: o nó que contém 2 passou a ser o fim (Figura 12.5).

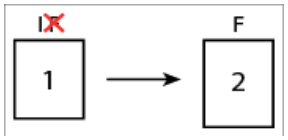

Figura 12.5: Agora o nó que contém 1 está no início e o nó que contém 2 está no fim.

Para concluir, adicione um número 3 à fila. Agora, existem três nós na lista encadeada interna e o nó que contém 2 não é mais o fim: o nó que contém 3 sim (Figura 12.6).

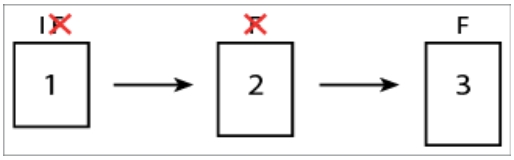

Figura 12.6: O nó que contém 1 está no início e o que contém 3 está no fim.

Quando você chamar **dequeue** pela primeira vez, removerá o nó que contém 1. Agora, o nó que contém 2 está no início da fila (Figura 12.7).

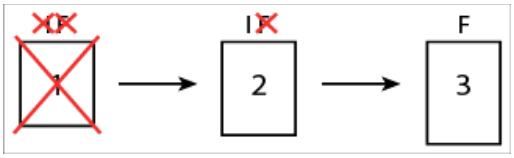

Figura 12.7: Quando você desenfileirar o número 1, o início mudará para o nó que contém 2.

Quando chamar **dequeue** pela segunda vez, removerá o nó que contém 2.

O nó que contém 3 passará a ser o início e o fim da fila (Figura 12.8).

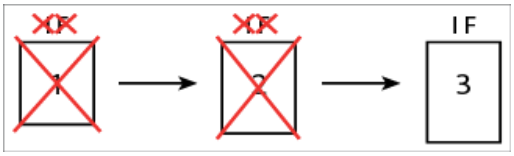

Figura 12.8: Quando você executar outro desenfileiramento, sobrará apenas um item, logo será o início e o fim.

Quando você chamar **`dequeue`** pela terceira vez, removerá o nó que contém 3, sua fila ficará vazia e tanto **`self.front`** quanto **`self.rear`** apontarão para **`None`** (Figura 12.9).

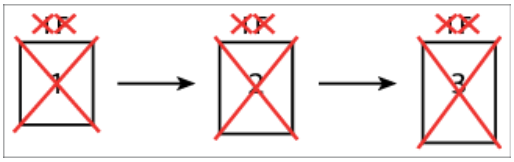

Figura 12.9: Agora sua fila está vazia.

Classe Queue interna do Python

O Python também tem uma classe interna que cria uma fila para podermos usar. Veja como funciona:

```
from queue import Queue
q = Queue()
q.put('a')
q.put('b')
q.put('c')
print(q.qsize())
for i in range(3):
print(q.get())
```

```
>> 3
>> a
>> b
>> c
```

Primeiro, importe **`Queue`** a partir do módulo **`queue`**:

```
from queue import Queue
```

Em seguida, crie uma fila chamando o método **`Queue`**:

```
q = Queue()
```

Adicione três strings à fila usando o método interno `put`:

```
q.put('a')
q.put('b')
q.put('c')
```

Verifique o tamanho da fila usando o método interno `qsize`:

```
print(q.qsize())
```

Por fim, use um loop `for` para remover todos os itens da fila e exibi-los:

```
for i in range(3):
print(q.get())
```

Crie uma fila usando duas pilhas

Uma questão que costuma aparecer em entrevistas técnicas é a criação de uma fila com o uso de duas pilhas. Veja como fazer isso:

```
class Queue:
def __init__(self):
self.s1 = []
self.s2 = []
def enqueue(self, item):
while len(self.s1) != 0:
self.s2.append(self.s1.pop())
self.s1.append(item)
while len(self.s2) != 0:
self.s1.append(self.s2.pop())
def dequeue(self):
if len(self.s1) == 0:
raise Exception("Cannot pop from empty queue")
return self.s1.pop()
```

Primeiro, defina uma classe `Queue` com duas pilhas internas, `self.s1` e `self.s2`:

```
class Queue:
def __init__(self):
self.s1 = []
self.s2 = []
```

Agora, defina um método chamado `enqueue` para adicionar um novo item à fila:

```
def enqueue(self, item):
while len(self.s1) != 0:
self.s2.append(self.s1.pop())
self.s1.append(item)
while len(self.s2) != 0:
self.s1.append(self.s2.pop())
```

Quando você adicionar um novo item à fila, terá de colocá-lo no fim da primeira pilha. Já que só podemos adicionar itens ao início de uma pilha, para colocar algo no fim da primeira pilha, você terá de remover tudo que existe nela, adicionar o novo item e, então, trazer todo o restante de volta.

Nesse caso, você removerá o conteúdo da primeira pilha, inserirá tudo na segunda pilha, adicionará o novo item à primeira pilha (quando estiver vazia) e trará tudo novamente para ela. Ao terminar, a primeira pilha terá todos os itens originais, mais o novo item no final:

```
while len(self.s1) != 0:
self.s2.append(self.s1.pop())
self.s1.append(item)
while len(self.s2) != 0:
self.s1.append(self.s2.pop())
```

Agora que você definiu `enqueue`, crie um método chamado `dequeue` para remover um item da fila:

```
def dequeue(self):
if len(self.s1) == 0:
raise Exception("Cannot pop from empty queue")
return self.s1.pop()
```

Primeiro, verifique se `self.s1` está vazia. Se estiver, isso significará que o usuário está tentando remover um item de uma fila vazia, logo você lançará uma exceção:

```
if len(self.s1) == 0:
raise Exception("Cannot pop from empty queue")
```

Caso contrário, remova o item do início da primeira pilha e retorne-o:

```
return self.s1.pop()
```

Nessa implementação de uma fila, o enfileiramento tem complexidade de tempo O(*n*) porque temos de percorrer cada item da pilha. Por outro lado, o desenfileiramento é O(1) já que só temos de remover o último item da

pilha interna.

Vocabulário

enfileirar: adicionar um item a uma fila.

estrutura de dados primeiro a entrar, primeiro a sair: estrutura de dados na qual o primeiro item a entrar é o primeiro a ser removido.

desenfileirar: remover um item de uma fila.

fila: estrutura de dados linear semelhante a uma pilha.

fila ilimitada: fila que não limita o número de itens que podemos adicionar a ela.

fila limitada: fila que limita o número de itens que podemos adicionar a ela.

Desafio

1. Implemente uma fila usando duas pilhas, mas faça com que o enfileiramento tenha complexidade O(1).

CAPÍTULO 13

Tabelas hash

Para cientistas e pensadores autodidatas, como Charles Darwin, Srinivasa Ramanujan, Leonardo da Vinci, Michael Faraday, eu mesmo e muitos outros, a educação é uma contínua viagem de descoberta. Para nós, a educação é uma busca eterna por conhecimento e sabedoria.

– Abhijit Naskar

Um **array associativo** é um tipo de dado abstrato que armazena pares chave-valor com chaves únicas. Um **par chave-valor** é composto de dois elementos de dados mapeados em conjunto: uma chave e um valor. A **chave** é o elemento de dados que usamos para recuperar o valor. O **valor** é o elemento de dados cuja recuperação depende da chave. Como programador Python, você deve estar familiarizado com os pares chave-valor, já que têm usado dicionários Python.

Existem muitas implementações de um array associativo, mas, neste capítulo, você conhecerá as tabelas hash. Uma **tabela hash** é uma estrutura de dados linear que armazena pares chave-valor com chaves únicas, o que significa que não podemos armazenar chaves duplicadas em uma tabela hash. A diferença entre um array associativo e uma tabela hash é que o primeiro é um tipo de dado abstrato, enquanto a tabela hash é uma estrutura de dados, portanto é uma implementação de um array associativo. O Python implementa dicionários usando tabelas hash.

Quando programamos, o sistema com o qual executamos nosso programa armazena os dados em uma tabela hash dentro de uma estrutura de dados de array. Quando adicionamos dados à tabela hash, o computador usa uma função hash para determinar em que local do array devem ser armazenados. Uma **função hash** é um código que recebe uma chave como entrada e gera um inteiro para ser usado no mapeamento da

chave da tabela hash para um índice de array que o computador usa para armazenar o valor. O índice que uma função hash gera chama-se **valor hash**. Você pode armazenar qualquer tipo de dado como valor em uma tabela hash, mas a chave deve ser algo que a função hash possa transformar em um índice, como um inteiro ou uma string. Esse processo torna a recuperação de valores em uma tabela hash incrivelmente eficiente, algo que você conhecerá melhor posteriormente.

Vejamos, rapidamente, como os dicionários Python funcionam. Pares chave-valor são armazenados em um dicionário Python. As chaves não podem ser duplicadas, mas os valores sim. Aqui, há um exemplo de armazenamento de um par chave-valor em um dicionário Python:

```
a_dict = {}
a_dict[1776] = 'Independence Year'
```

Agora, você pode usar a chave 1776 para procurar o valor `'Independence Year'` desta forma:

```
print(a_dict[1776])
>> 'Independence Year'
```

Examinaremos como uma função hash fictícia funciona determinando a localização de várias chaves, que, nesse exemplo, serão inteiros. Tudo que veremos aqui o computador faz em segundo plano quando um dicionário é usado em Python. Suponhamos que você tivesse uma tabela hash com sete slots e quisesse armazenar vários inteiros neles (Figura 13.1). (Nesse caso, para fins ilustrativos, estamos lidando apenas com as chaves e não com as chaves e seus valores).

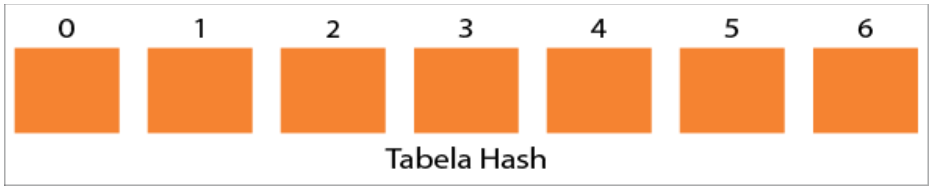

Figura 13.1: Uma tabela hash armazena pares chave-valor em um array.

O primeiro número que você precisa armazenar é 86. Para armazenar 86 na tabela hash, você precisa de uma função hash. Uma função hash simples pegaria cada número e executaria uma operação de módulo (divisão para obtenção do resto) pelo número de slots disponíveis

(Figura 13.2). Por exemplo, para obter um valor hash para 86, você calcularia 86 % 7. O resultado de 86 % 7 é 2, o que significa que você deve inserir 86 no índice dois do array que está usando para armazenar dados da tabela hash.

O próximo número que você precisa inserir na tabela hash é 90, logo calculará 90 % 7, que é 6. Deverá inserir, então, 90 no índice seis do array (Figura 13.3).

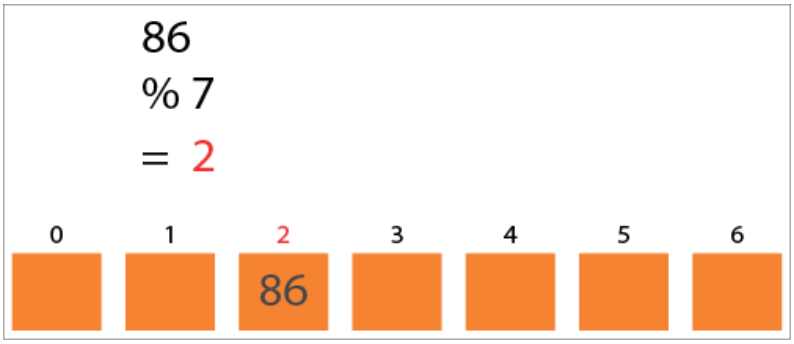

Figura 13.2: Para armazenar 86 na tabela hash, você executou a operação de módulo pelo número de slots e obteve 2.

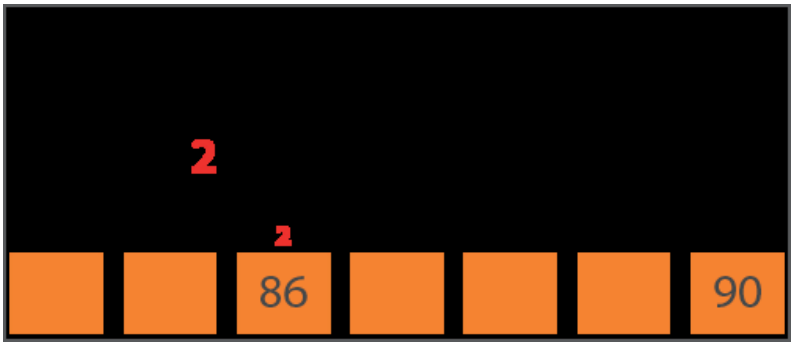

Figura 13.3: Para armazenar 90 na tabela hash, você executou a operação de módulo pelo número de slots e obteve 6.

Para concluir, você precisa adicionar os números 21, 29, 38, 39 e 40 à tabela hash. Veja o que acontece quando usamos o módulo 7 com esses números:

21 % 7 = 0

29 % 7 = 1

38 % 7 = 3

39 % 7 = 4

40 % 7 = 5

Quando você adicionar esses números à tabela hash, ficará semelhante à da Figura 13.4.

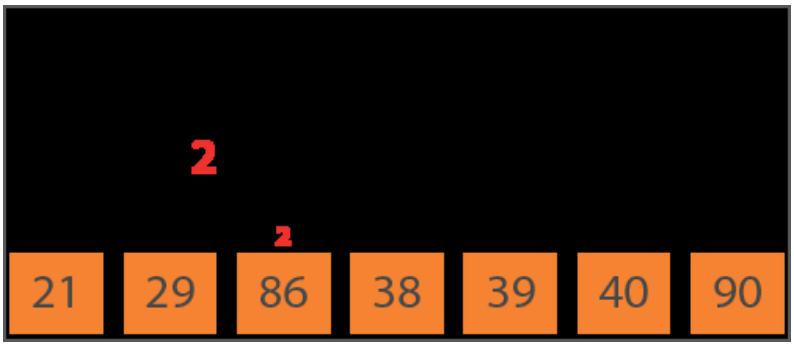

Figura 13.4: Tabela hash após a inclusão de todos os números.

Até agora, a inclusão de dados na tabela hash ocorreu como planejado. Suponhamos que você precisasse adicionar 30 à tabela hash. Já que 30 % 7 é 2, 30 entraria no slot 2. No entanto, há um problema porque 86 já está nesse slot. Dois números contendo um hash para o mesmo slot são uma **colisão**. Para resolver essa colisão, você pode inserir 30 no próximo slot vazio. Essa solução funcionará, mas quando você tiver de encontrar 30, precisará usar uma função hash para descobrir sua localização no array, examinar o slot 3, verificar que não contém 30 e examinar os slots subsequentes até encontrá-lo, o que adiciona complexidade de tempo. Existem outras maneiras de manipular colisões, como manter listas (em geral, listas encadeadas) em cada local e inserir cada par em colisão na lista associada ao local original da colisão. Quando criamos uma tabela hash, o que queremos é usar o número correto de slots e uma função hash que produza menos colisões. Contudo, ao programar em Python, não precisamos nos preocupar com colisões porque os dicionários as manipulam para nós.

Como mencionei, no exemplo anterior não estamos armazenando pares chave-valor. Você pode modificar o exemplo para armazenar pares chave-valor usando dois arrays: um para armazenar chaves e outro para valores. Assim, se estivesse mapeando `self` para `taught`, a função hash

transformaria `self` no índice de um array. Você armazenaria `self` nesse índice no array para chaves e `taught` nesse índice no array para valores.

Quando usar tabelas hash

Ao contrário das outras estruturas de dados que você conheceu até agora (e das que conhecerá posteriormente), em média, procurar dados em uma tabela hash tem complexidade O(1). Em média, a inserção e a exclusão de dados em uma tabela hash também têm complexidade O(1). As colisões podem prejudicar a eficiência das tabelas hash, fazendo a busca, a inserção e a exclusão terem complexidade O(*n*) no cenário de pior caso. Mesmo assim, as tabelas hash são uma das estruturas mais eficientes para o armazenamento de grandes conjuntos de dados. A razão para as tabelas hash serem tão eficientes é que para determinar se um dado se encontra na tabela hash, só precisamos executar a função hash com o dado e verificar o índice no array, o que leva apenas uma etapa. A Figura 13.5 mostra o tempo de execução das operações de uma tabela hash. Não inclui tempo de execução para a coluna **Acesso** das tabelas hash porque estas não permitem acessar seu enésimo item como ocorre em um array ou uma lista encadeada.

Estrutura de dados	Complexidade de tempo							
	Média				Pior			
	Acesso	Busca	Inserção	Exclusão	Acesso	Busca	Inserção	Exclusão
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Pilha	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Fila	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Lista encadeada	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Tabela Hash	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)
Árvore de busca binária	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)

Figura 13.5: Tempos de execução das operações da tabela hash.

Anteriormente, você conheceu os algoritmos de busca e viu que se ordenar seus dados poderá executar uma busca binária, que é significativamente mais rápida do que uma busca linear. Também soube

que havia outra maneira de procurar dados ainda mais eficiente do que a busca binária e que a conheceria posteriormente. Esse método é a busca de dados em uma tabela hash, que tem complexidade O(1), o que significa que procurar dados em uma tabela hash é a maneira mais rápida possível de buscar dados. A possibilidade de procurar dados em tempo constante em vez de ser preciso fazer uma busca linear ou binária faz uma grande diferença no trabalho com grandes conjuntos de dados.

Como programador, você usará tabelas hash com frequência. Por exemplo, se você for um desenvolvedor web, trabalhará muito com **JSON (JavaScript Object Notation)**, um formato de intercâmbio de dados. Muitas APIs enviam dados em JSON, que podem ser transformados facilmente em um dicionário Python. Uma **API (application programming interface; interface de programação de aplicações)** é um programa que permite que as aplicações se comuniquem umas com as outras. Sempre que você trabalhar com um banco de dados de chave-valor em Python, usará dicionários. O popular sistema de controle de versões Git usa os valores hash de uma função hash criptográfica para armazenar diferentes versões dos dados de projetos. Os sistemas operacionais costumam usar tabelas hash para ajudar no gerenciamento da memória. A própria linguagem Python usa dicionários (tabelas hash) para armazenar nomes e valores de variáveis de objetos.

Você deve considerar usar uma tabela hash sempre que tiver uma grande quantidade de dados e precisar acessar itens de dados individuais rapidamente. Por exemplo, suponhamos que você estivesse escrevendo um programa que tivesse de fazer uma busca em um dicionário de inglês ou quisesse criar uma aplicação que demandasse acesso rápido ao número de telefone de alguém em uma agenda telefônica com centenas de milhares ou milhões de entradas. As tabelas hash são apropriadas a essas duas situações. Em geral, são uma estrutura de dados adequada sempre que precisamos de acesso aleatório e rápido aos dados. No entanto, se você costuma trabalhar com dados em ordem sequencial, um array ou uma lista encadeada pode ser uma opção melhor.

Caracteres de uma string

Quando estiver resolvendo um problema, considere usar uma tabela hash em sua solução porque são muito eficientes. Por exemplo, digamos que um entrevistador lhe pedisse para contar todos os caracteres de uma string. Uma solução para esse problema seria usar um dicionário Python. Você pode armazenar cada caractere como uma chave em seu dicionário e o número de vezes que ele ocorre na string como um valor. Veja como funciona:

```
def count(a_string):
a_dict = {}
for char in a_string:
if char in a_dict:
a_dict[char] += 1
else:
a_dict[char] = 1
print(a_dict)
```

Sua função `count` recebeu uma string como parâmetro (a string cujos caracteres você deseja contar):

```
def count(a_string):
```

Dentro da função, crie um dicionário:

```
a_dict = {}
```

Em seguida, use um loop `for` para percorrer a string:

```
for char in a_string:
```

Se o caractere já existir no dicionário, incremente seu valor em 1 unidade:

```
if char in a_dict:
a_dict[char] += 1
```

Se o caractere ainda não estiver no dicionário, adicione-o como uma nova chave com o valor 1, já que essa é a primeira vez que apareceu:

```
else:
a_dict[char] = 1
```

Quando você exibir o dicionário no final, ele conterá cada letra da string, assim como quantas vezes esta ocorre:

```
print(a_dict)
```

Vejamos o que ocorrerá quando você executar sua função. Quando você chamar a função e passar a string `"Hello"`, na primeira iteração o programa adicionará ao dicionário um `H` maiúsculo como chave e o número 1 como seu valor, o que significa que seu dicionário ficará assim:

```
{"H": 1}
```

Na próxima iteração, o caractere será `e`. Como `e` ainda não está no dicionário, adicione-o como uma chave com o valor 1. Agora, o dicionário ficou assim:

```
{"H": 1, "e": 1}
```

Faça o mesmo para a letra `l`. O dicionário está assim:

```
{"H": 1, "e": 1, "l": 1}
```

Na próxima iteração, o caractere será `l` novamente. Dessa vez, o caractere estará no dicionário, logo incremente o valor da sua chave em 1 unidade. Agora, o dicionário contém:

```
{"H": 1, "e": 1, "l": 2}
```

Esse processo continuará até você percorrer todos os caracteres da string. Quando o loop terminar e você exibir o dicionário, teremos:

```
{'H': 1, 'e': 1, 'l': 2, 'o': 1}
```

Usando essa abordagem, você não só resolveu o problema, mas também o resolveu em tempo O(*n*), empregando uma tabela hash (nesse caso, *n* é o número de caracteres da string).

Soma de dois números

Outra questão comum em entrevistas técnicas para a qual podemos usar uma tabela hash na solução se chama *soma de dois números (two sum)*. Nesse desafio, o entrevistador pedirá que você retorne os índices de dois números de uma lista não ordenada cuja soma resulte em um valor-alvo. Só um par fornecerá como resultado o número-alvo. Não é permitido usar o mesmo número da lista duas vezes.

Por exemplo, digamos que o valor-alvo fosse 5 e você tivesse a lista de números a seguir:

```
[1, 2, 3, 4, 7]
```

Aqui, a soma dos números com posições de índice 1 e 2 tem como resultado o número-alvo (5), logo a resposta é o índice 1 e o índice 2 (2 + 3 = 5).

Uma maneira de resolver esse problema é usando força bruta para percorrer a lista, somar cada par de números e verificar se o resultado da soma é 5. Veja como codificar uma solução de força bruta para o problema:

```
def two_sum_brute(the_list, target):
index_list = []
for i in range(0, len(the_list)):
for j in range(i, len(the_list)):
if the_list[i] + the_list[j] == target:
return [the_list[i], the_list[j]]
```

Sua solução usa dois loops aninhados. O loop externo percorre a lista de números usando `i`, enquanto o loop interno também a percorre usando `j`. Você está utilizando as duas variáveis para criar pares, como −1 e 2, 2 e 3 etc., e verificar se sua soma resulta no número-alvo. A solução de força bruta é simples, mas não é eficiente. Já que o algoritmo requer dois loops aninhados para percorrer cada combinação, tem complexidade O(n**2).

Uma maneira mais eficiente de resolver esse problema é usando um dicionário. É assim que podemos resolver o desafio da soma de dois números com um dicionário:

```
def two_sum(a_list, target):
a_dict = {}
for index, n in enumerate(a_list):
rem = target - n
if rem in a_dict:
return index, a_dict[rem]
else:
a_dict[n] = index
```

A função `two_sum` recebe dois argumentos, uma lista de números e o número-alvo que sua soma deve produzir:

```
def two_sum(a_list, target):
```

Dentro da função, crie um dicionário:

```
a_dict = {}
```

Em seguida, chame `enumerate` na lista, o que permitirá que você a percorra registrando ao mesmo tempo cada número e seu índice:

```
for index, n in enumerate(a_list):
```

Agora, subtraia `n` do número-alvo:

```
rem = target - n
```

O resultado será o número ao qual o número atual precisa corresponder para que a soma seja o número-alvo que você está procurando. Se o número da variável `rem` estiver no dicionário, você saberá que encontrou a resposta, logo retorne o índice atual e procure o índice do número que armazenou usando a chave `rem` dentro do dicionário:

```
return index, a_dict[rem]
```

Se o número da variável `rem` não estiver no dicionário, adicione-o, inserindo `n` como a chave e seu índice como o valor:

```
else:
a_dict[n] = index
```

Examinaremos como funciona. Suponhamos que você executasse o programa com esta lista e um número-alvo igual a 5:

```
[1, 2, 3, 4, 7]
```

Na primeira execução do loop, `n` será -1 e não haverá nada no dicionário, portanto você adicionará -1 no índice 0 ao dicionário. Na segunda execução do loop, *n* será 2 e `rem` será 3 (5 - 2), então, dessa vez, você adicionará 2 no índice 1 ao dicionário. Na terceira execução do loop, `n` será 3, o que significa que `rem` é 2. Você já inseriu 2 no dicionário, logo encontrou a resposta.

Ao contrário da abordagem de força bruta para a solução do problema, essa solução é O(*n*) porque, ao usar uma tabela hash, você não precisará mais usar dois loops `for` aninhados, o que é muito mais eficiente.

Vocabulário

API (application programming interface; interface de programação de aplicações): programa que permite que as aplicações se comuniquem

umas com as outras.

array associativo: tipo de dados abstrato que armazena pares chave-valor com chaves únicas.

chave: dado usado para a recuperação do valor.

colisão: quando o hash de dois números leva ao mesmo slot.

JSON (JavaScript Object Notation): formato de intercâmbio de dados.

par chave-valor: dois elementos de dados mapeados juntos: uma chave e um valor.

função hash: código que recebe uma chave como entrada e gera um dado único que mapeia a chave para o índice de um array usado pelo computador para armazenar o valor.

tabela hash: estrutura de dados linear que armazena pares chave-valor com chaves únicas.

valor: dado cuja chave é usada para sua recuperação.

valor hash: valor único que uma função hash produz.

Desafio

1. Dada uma string, remova todas as palavras duplicadas. Por exemplo, dada a string

   `"I am a selftaught programmer looking for a job as a programmer.",`

   sua função deve retornar

   `"I am a selftaught programmer looking for a job as a."`.

CAPÍTULO 14

Árvores binárias

As pessoas mais poderosas são as que nunca param de aprender.

– Rejoice Denhere

Até agora, todas as estruturas de dados que vimos eram lineares. Nos próximos capítulos, você conhecerá algumas estruturas de dados e tipos de dados abstratos não lineares essenciais. O primeiro que discutiremos é a **árvore**, um tipo de dado abstrato não linear composto de nós conectados em uma estrutura hierárquica (Figura 14.1). As operações comuns executadas em árvores são a inserção, a busca e a exclusão.

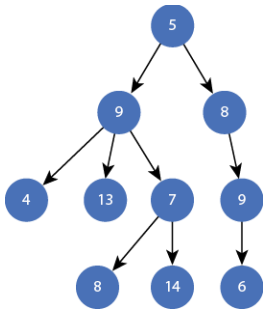

Figura 14.1: Exemplo de uma estrutura de dados em árvore.

Existem vários tipos de estruturas de dados em árvore: árvores gerais, árvores AVL, árvores rubro-negras, árvores binárias, árvores de busca binária, entre outros. Neste capítulo, você conhecerá as árvores gerais, as árvores de busca binária e as árvores binárias, com ênfase nas árvores binárias. Abordar todos os tipos de árvore não faz parte do escopo deste livro, mas recomendo que você aprenda mais sobre os outros tipos por conta própria.

Uma árvore geral é uma estrutura de dados que começa com um nó no topo. Esse nó chama-se **nó-raiz**. Cada nó conectado abaixo de um nó em uma árvore é seu **nó-filho**. Um nó com um ou mais nós-filhos chama-se

nó-pai. Os **nós-irmãos** compartilham o mesmo pai. A conexão entre dois nós de uma árvore chama-se **aresta (edge)** (Figura 14.2).

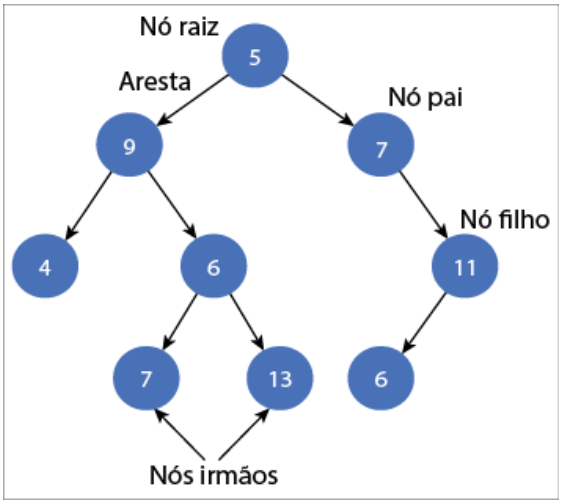

Figura 14.2: Árvore com um nó-raiz, nós-pais, nós-filhos e arestas.

Você pode se mover de um nó para outro, contanto que compartilhem uma aresta (Figura 14.3).

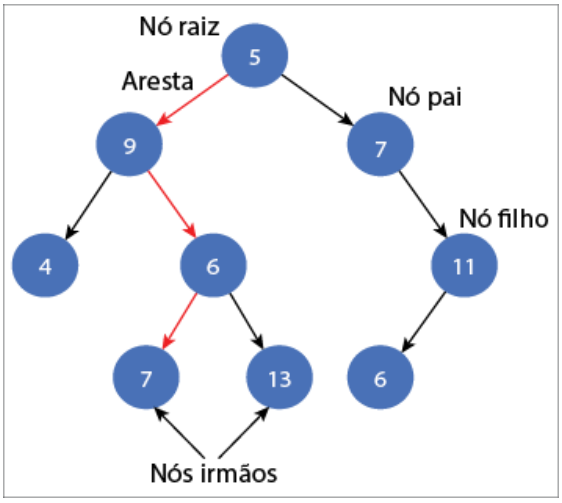

Figura 14.3: Caminho que percorre uma árvore.

Cada nó de uma árvore, exceto o raiz, tem um único nó-pai. Um nó sem nós-filhos chama-se **nó-folha** e um nó com nós-filhos chama-se **nó-ramo (branch node)**.

Uma **árvore binária** é uma estrutura de dados de árvore na qual cada nó só pode ter dois nós-filhos (também chamados somente de filho ou

filhos). Cada nó de uma árvore binária (exceto o raiz) é o filho esquerdo ou direito de um nó-pai (Figura 14.4).

Todo o restante em uma árvore binária é igual a uma árvore geral; a única diferença é o limite de nós-filhos.

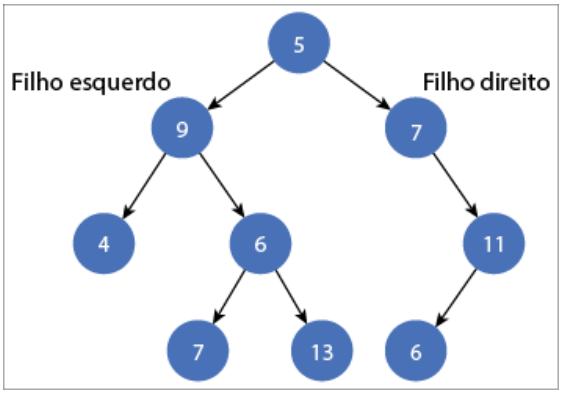

Figura 14.4: Em uma árvore binária, um nó-pai só pode ter dois nós-filhos.

Uma **árvore de busca binária** é uma estrutura de dados em árvore na qual cada nó tem apenas dois filhos e a árvore armazena seus nós em uma ordem na qual o valor de cada nó é maior do que qualquer valor de sua subárvore esquerda e menor do que qualquer valor de sua subárvore direita (Figura 14.5). Como ocorre com as chaves das tabelas hash, não podemos armazenar valores duplicados em uma árvore de busca binária. Você pode contornar essa restrição e manipular valores duplicados adicionando um campo de contagem nos objetos de nó da árvore para registrar o número de ocorrências de um valor específico.

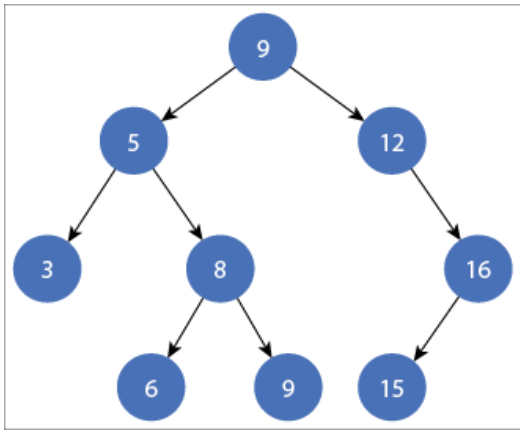

Figura 14.5: Exemplo de árvore de busca binária.

Ao contrário do que ocorre com estruturas de dados lineares, como os arrays e as listas encadeadas, nem sempre podemos percorrer uma árvore sem backtracking. Você pode alcançar qualquer nó de uma árvore começando pelo nó-raiz, mas, uma vez que tiver saído dele, só poderá alcançar os descendentes desse nó. Os **descendentes** de um nó são seus filhos, os filhos deles, os filhos desses filhos etc. Por exemplo, a Figura 14.6 mostra uma árvore simples com um nó-raiz A, os nós-folhas B, D e E e um nó-ramo C. O nó A tem dois filhos (B e C) e o nó C também tem dois filhos (D e E). Os nós B, C, D e E são os descendentes do nó A.

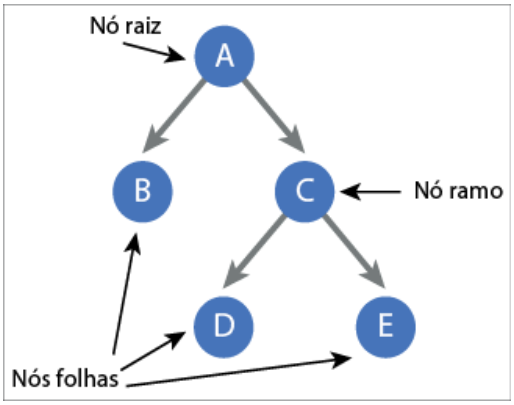

Figura 14.6: Árvore simples mostrando o nó-raiz, A, e seus descendentes.

Se você começar no nó raiz e mover-se apenas para os nós-filhos direitos, percorrerá os nós A, C e E. Se, em vez disso, mover-se apenas para os nós-filhos esquerdos, percorrerá os nós A e B. Você deve ter notado que nenhuma dessas varreduras inclui o nó D. Para chegar ao nó D, primeiro você precisa mover-se para o filho direito do nó A e, depois, para o filho esquerdo do nó C. Em outras palavras, para chegar ao nó D nessa árvore binária, é preciso executar o backtracking.

Quando usar árvores

Inserir, excluir e procurar dados em uma árvore geral ou binária são operações de complexidade O(*n*). As árvores de busca binária são mais eficientes: todas as três operações são logarítmicas em uma árvore de busca binária porque podemos fazer uma busca binária para inserir, excluir e procurar nós (Figura 14.7).

Estrutura de dados	Complexidade de tempo							
	Média				Pior			
	Acesso	Busca	Inserção	Exclusão	Acesso	Busca	Inserção	Exclusão
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Pilha	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Fila	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Lista encadeada	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Tabela Hash	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)
Árvore de busca binária	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)

Figura 14.7: Tempos de execução das operações das árvores de busca binária.

Você deve estar se perguntando por que usar uma árvore geral ou binária se todas as operações são lineares. Até mesmo as árvores de busca binária que permitem procurar dados logaritmicamente são mais lentas do que as tabelas hash, então por que usar árvores? As árvores permitem armazenar informações hierárquicas que seriam difíceis ou impossíveis de representar em uma estrutura de dados linear como um array. Por exemplo, suponhamos que você quisesse representar os diretórios de seu computador programaticamente. Digamos que você tivesse uma pasta chamada `Documentos` com 10 pastas e cada uma dessas pastas tivesse 20 pastas, que, por sua vez, tivessem 4 pastas cada uma etc. Representar o relacionamento entre as pastas do computador e em que diretório um usuário está seria confuso e desafiador com um array, mas é fácil com uma árvore (Figura 14.8).

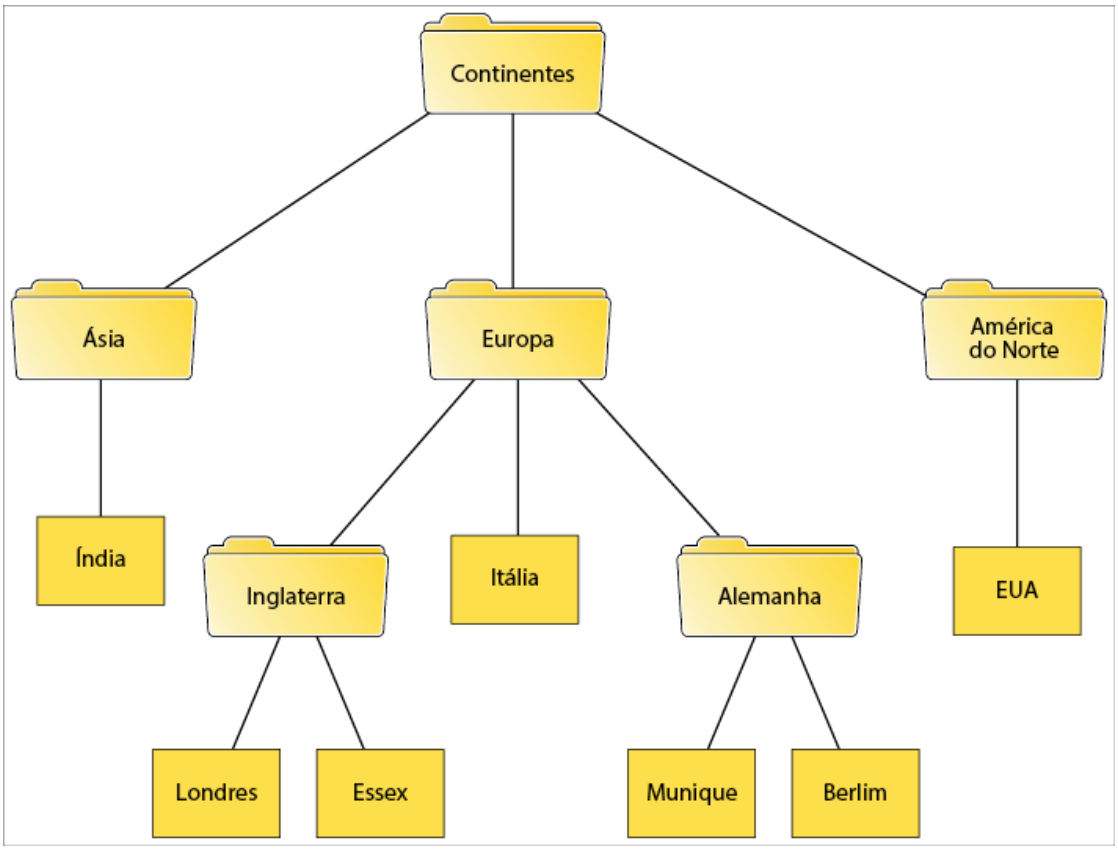

Figura 14.8: Exemplo de pastas em uma árvore.

Os documentos HTML e XML são outro exemplo de dados hierárquicos que os cientistas da computação representam com árvores. O **HTML** é uma linguagem de marcação (markup) que podemos usar para criar páginas web. O **XML** é uma linguagem de marcação para documentos. Podemos aninhar tags HTML e XML, logo, em geral, os programadores as armazenam como árvores nas quais cada nó representa um único elemento do código HTML ou XML. Quando programamos o front-end de um site, a linguagem de programação JavaScript dá acesso ao DOM (document object model). O **document object model** é uma interface independente de linguagem que modela um documento XML ou HTML como uma árvore (Figura 14.9).

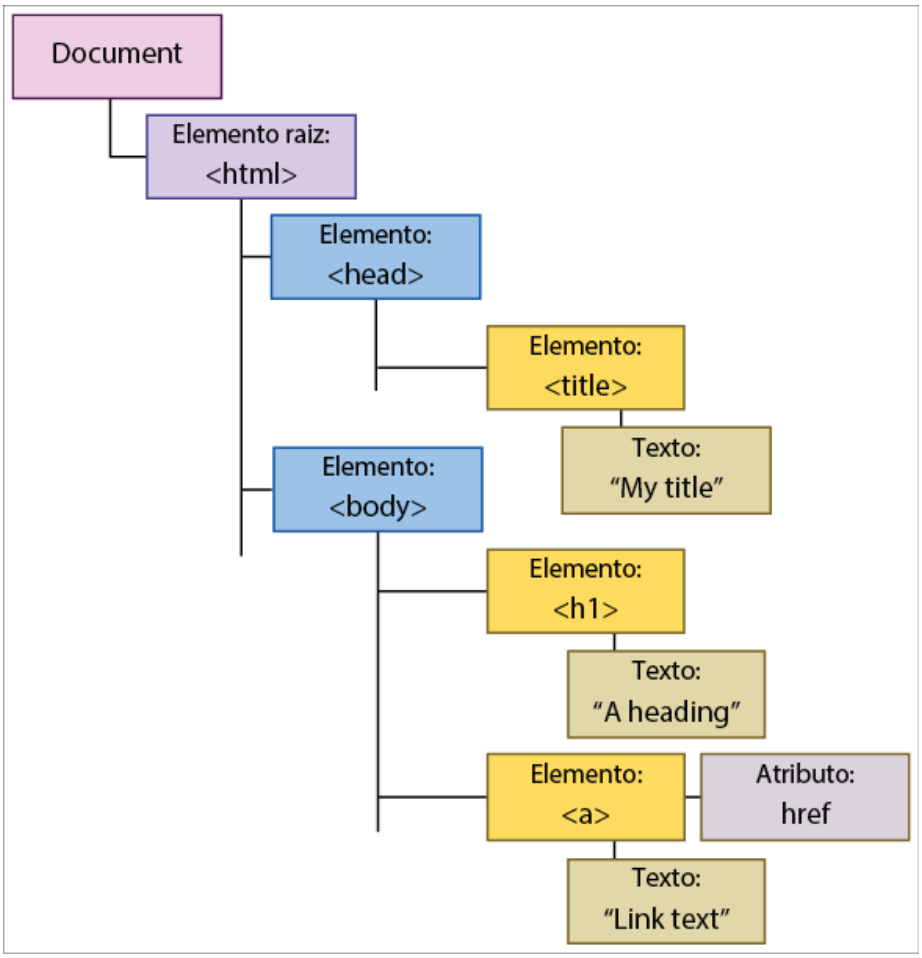

Figura 14.9: O document object model.

Você pode usar uma árvore para fazer o parsing de, por exemplo, expressões aritméticas. Poderia avaliar uma expressão como 2 + 3 * 4 criando uma árvore como a da Figura 14.10. Você avaliaria a parte inferior da árvore (3 * 4) e subiria um nível para calcular a solução final (2 + 12). Uma árvore como a da Figura 14.10 chama-se árvore de parsing (parse tree). Uma **árvore de parsing** é uma árvore ordenada que armazena os dados de acordo com algum tipo de sintaxe, como as regras de avaliação de uma expressão.

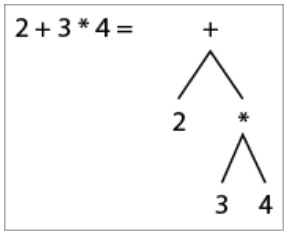

Figura 14.10: Árvore para avaliação de uma expressão matemática.

Representar dados hierárquicos não é a única razão para cientistas da computação usarem árvores. Como você sabe, é possível fazer uma busca em uma árvore binária ordenada em tempo logarítmico. Embora uma busca logarítmica não seja tão rápida como uma busca de tempo constante em uma tabela hash, as árvores de busca binária oferecem algumas vantagens em relação às tabelas hash. A primeira é o uso da memória. Em razão das colisões, as tabelas hash podem ser 10 ou mais vezes maiores do que a quantidade de dados armazenados nelas. Por outro lado, as árvores de busca binária não consomem nenhuma memória adicional. Além disso, uma árvore de busca binária nos permite percorrer rapidamente os dados tanto na ordem da ordenação quanto na ordem inversa, o que não podemos fazer em uma tabela hash porque não armazenam dados em ordem.

Criando uma árvore binária

Veja como criar uma árvore binária em Python:

```
class BinaryTree:
def __init__(self, value):
self.key = value
self.left_child = None
self.right_child = None
def insert_left(self, value):
if self.left_child == None:
self.left_child = BinaryTree(value)
else:
bin_tree = BinaryTree(value)
bin_tree.left_child = self.left_child
self.left_child = bin_tree
def insert_right(self, value):
if self.right_child == None:
self.right_child = BinaryTree(value)
else:
bin_tree = BinaryTree(value)
bin_tree.right_child = self.right_child
self.right_child = bin_tree
```

Primeiro, defina uma classe chamada `BinaryTree` para representar sua árvore. `BinaryTree` tem três variáveis de instância: `key`, `left_child` e `right_child`. A variável `key` armazena os dados do nó (por exemplo, um inteiro), `left_child` registra o filho esquerdo do nó e `right_child` registra seu filho direito. Quando você criar um nó-filho para a árvore, criará uma instância da classe `BinaryTree`, que também terá uma chave, um filho esquerdo e um filho direito. Cada nó-filho é uma subárvore. Uma **subárvore** é um nó de uma árvore, sem ser o nó-raiz, e seus descendentes. A subárvore pode ter outras subárvores.

Em seguida, defina um método chamado `insert_left` para criar um nó-filho e inseri-lo no lado esquerdo da árvore:

```
def insert_left(self, value):
if self.left_child = None:
self.left_child = BinaryTree(value)
else:
bin_tree = BinaryTree(value)
bin_tree.left_child = self.right_child
bin_tree.left_child = bin_tree
```

Primeiro, o método verifica se `self.left_child` é `None`. Se for, você criará uma nova classe `BinaryTree` e a atribuirá à `self.left_child`:

```
if self.left_child = None:
self.left_child = BinaryTree(value)
```

Caso contrário, criará um novo objeto `BinaryTree`, atribuirá qualquer objeto `BinaryTree` que estiver atualmente em `self.left_child` ao `self.left_child` da nova `BinaryTree` e atribuirá a nova `BinaryTree` a `self.left_child`:

```
else:
bin_tree = BinaryTree(value)
bin_tree.left_child = self.left_child
self.left_child = bin_tree
```

Após definir o método `insert_left`, defina também um método chamado `insert_right`, que fará o mesmo que `insert_left`, mas adicionará o novo nó no lado direito da árvore binária:

```
def insert_right(self, value):
if self.right_child == None:
```

```
self.right_child = BinaryTree(value)
else:
bin_tree = BinaryTree(value)
bin_tree.right_child = self.right_child
self.right_child = bin_tree
```

Agora, você pode criar uma árvore binária e adicionar nós a ela desta forma:

```
tree = BinaryTree(1)
tree.insert_left(2)
tree.insert_right(3)
tree.insert_left(4)
tree.left_child.insert_right(6)
tree.insert_right(5)
```

Esse código cria a árvore binária da Figura 14.11.

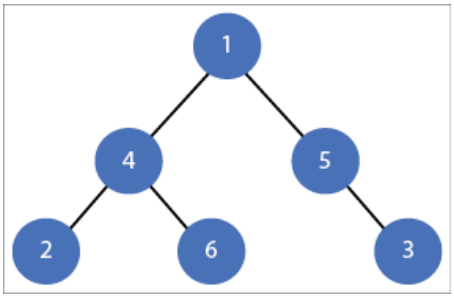

Figura 14.11: Árvore binária com cinco nós.

Percorrendo uma árvore pela largura

Como vimos anteriormente, nem sempre é possível percorrer uma árvore movendo-se de nó em nó sem backtracking. No entanto, isso não significa que não possamos procurar dados em uma árvore. Para procurar dados em uma árvore, temos de visitar cada nó e ver se este contém as informações que estamos procurando. Existem várias maneiras de visitar cada nó de uma árvore binária. Uma delas é **percorrendo pela largura**: método para o acesso a cada nó de uma árvore no qual os nós são visitados nível a nível. Por exemplo, na árvore binária da Figura 14.12, a raiz fica no nível 0 e, depois, vêm os níveis 1, 2 e 3.

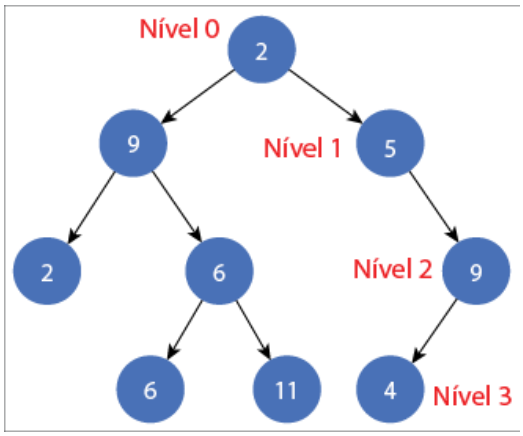

Figura 14.12: Níveis de uma árvore binária.

Quando percorremos uma árvore pela largura para fazer uma busca, isso se chama **busca em largura**. Começamos uma busca em largura na raiz da árvore (nível 0) e avançamos nível a nível, visitando cada nó em cada nível até alcançar o último nível. Você pode codificar uma busca em largura usando duas listas para registrar o nível atual e o nível seguinte da árvore. À medida que for visitando cada nó da lista atual, verifique se este contém os dados que você está procurando e adicione seus filhos à "próxima" lista. Quando chegar a hora de passar para o próximo nível, mude de lista. Veja como procurar um número em uma árvore binária usando uma busca em largura:

```
class BinaryTree:
def __init__(self, value):
self.key = value
self.left_child = None
self.right_child = None
def insert_left(self, value):
if self.left_child == None:
self.left_child = BinaryTree(value)
else:
bin_tree = BinaryTree(value)
bin_tree.left_child = self.left_child
self.left_child = bin_tree
def insert_right(self, value):
if self.right_child == None:
self.right_child = BinaryTree(value)
else:
bin_tree = BinaryTree(value)
```

```
bin_tree.right_child = self.right_child
self.right_child = bin_tree
def breadth_first_search(self, n):
current = [self]
next = []
while current:
for node in current:
if node.key == n:
return True
if node.left_child:
next.append(node.left_child)
if node.right_child:
next.append(node.right_child)
current = next
next = []
return False
```

Seu método, `breadth_first_search`, recebeu o parâmetro `n`, que é o dado a ser procurado:

```
def breadth_first_search(self, n):
```

Agora, defina duas listas. Use a primeira, `current`, para registrar os nós do nível atual que você está pesquisando. A segunda, `next`, será usada para registrar os nós do nível seguinte. Você também pode adicionar `self` a `current` para o algoritmo começar pesquisando pela raiz da árvore (nível 0):

```
current = [self]
next = []
```

O loop `while` continuará sendo executado enquanto `current` tiver nós para serem pesquisados.

```
while current:
```

Em seguida, use um loop `for` para percorrer cada nó de `current`.

```
for node in current:
```

Se o valor do nó coincidir com `n` (o valor que você está procurando), retorne `True`:

```
if node.key == n:
return True
```

Caso contrário, acrescente os nós-filhos esquerdo e direito à lista `next`, se

não forem `None`, para que sejam pesquisados quando você fizer a busca no nível seguinte:

```
if node.left_child:
next.append(node.left_child)
if node.right_child:
next.append(node.right_child)
```

Agora, no fim do loop `while`, troque as listas `current` e `next`. A lista de nós que seriam pesquisados em seguida será a lista a ser pesquisada agora e você faz de `next` uma lista vazia:

```
current = next
next = []
```

Se o loop `while` terminar, retorne `False` porque sua busca em largura não encontrou `n` na árvore:

```
return False
```

Mais buscas em árvores

Uma busca em largura não é a única maneira de percorrer uma árvore binária: você também pode fazer uma busca em profundidade. Em uma **busca em profundidade**, visitamos todos os nós de uma árvore binária, indo o mais profundamente possível em uma direção antes de passar para o próximo irmão. A busca em profundidade oferece três maneiras de visitar cada nó: em pré-ordem, em pós-ordem e em ordem simétrica. As implementações das três abordagens são semelhantes, mas seus usos são diferentes.

Suponhamos que você tivesse a árvore binária mostrada na Figura 14.13. Na busca em pré-ordem, você começaria com a raiz e seguiria para a esquerda e, depois, para a direita.

Aqui está o código para uma busca em pré-ordem em uma árvore:

```
def preorder(tree):
if tree:
print(tree.key)
preorder(tree.left_child)
preorder(tree.right_child)
```

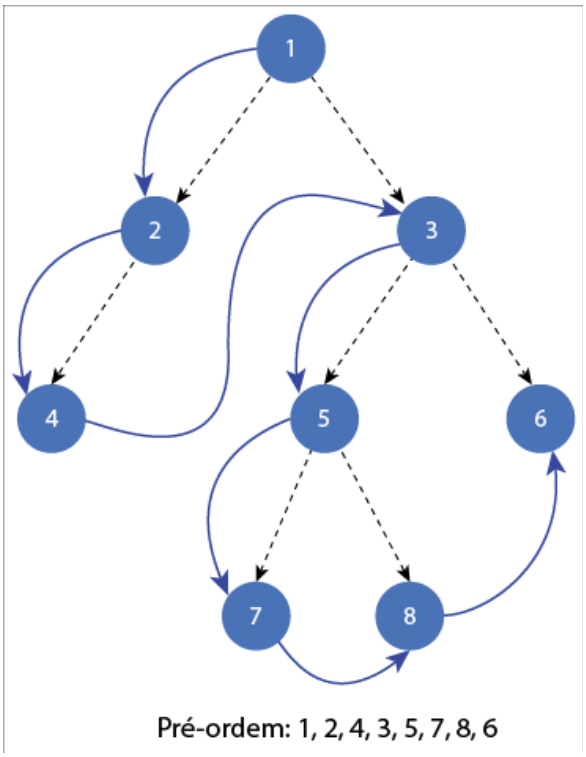

Figura 14.13: Um livro representado como uma árvore.

A função chama a si mesma recursivamente até alcançar seu caso base, que é esta linha de código:

```
if tree:
```

A linha de código a seguir exibe o valor de cada nó da árvore:

```
print(tree.key)
```

Essas linhas de código chamam `preorder` nos filhos esquerdo e direito de cada nó da árvore:

```
preorder(tree.left_child)
preorder(tree.right_child)
```

Essa busca deve ser familiar para você porque é semelhante ao que fizemos quando escrevemos a ordenação por intercalação no Capítulo 4. Quando codificamos a ordenação por intercalação, adicionamos uma chamada recursiva à metade esquerda de uma lista seguida por uma chamada recursiva à metade direita. O algoritmo chamava a si mesmo com a metade esquerda até que tivéssemos uma lista com apenas um item. Ele chamava o código recursivo para separar a metade direita da lista sempre que isso ocorria. Quando alcançávamos o caso base,

subíamos um nível na pilha recursiva e mesclávamos as duas listas com o código escrito abaixo das duas chamadas recursivas. O algoritmo de ordenação por intercalação é semelhante a uma busca em pré-ordem, mas chama-se *busca em pós-ordem*. A diferença entre uma busca em pós-ordem e uma busca em pré-ordem é que em uma busca em pós-ordem exibimos o valor de cada nó (ou fazemos outra coisa) depois das chamadas recursivas:

```
def postorder(tree):
if tree:
postorder(tree.left_child)
postorder(tree.right_child)
print(tree.key)
```

Em uma busca em pós-ordem, percorremos a árvore começando pela esquerda, depois passamos para a direita e terminamos na raiz (Figura 14.14).

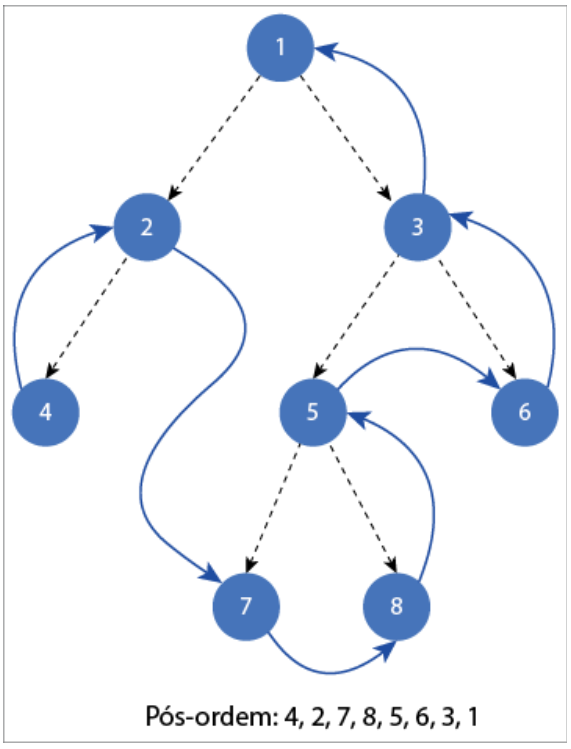

Figura 14.14: Busca em pós-ordem em uma árvore.

Se imaginarmos a busca em pós-ordem como um algoritmo de ordenação por intercalação, um nó seria exibido sempre que houvesse uma intercalação.

Para concluir, temos a busca em ordem simétrica:

```
def inorder(tree):
if tree:
inorder (tree.left_child)
print(tree.key)
inorder (tree.right_child)
```

Uma busca em ordem simétrica é como uma busca em pré-ordem ou em pós-ordem, mas exibimos o valor do nó (ou fazemos qualquer outra coisa) entre as duas chamadas recursivas. Quando você usar uma busca em ordem simétrica, mover-se-á pela árvore da esquerda para a raiz e, depois, para a direita (Figura 14.15).

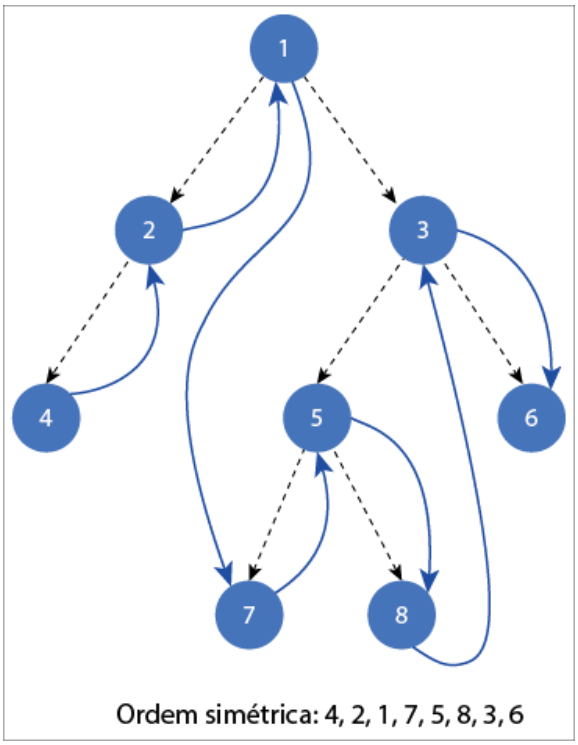

Figura 14.15: Busca em uma árvore em ordem simétrica.

Inverta uma árvore binária

Max Howell é o criador do Homebrew, um gerenciador de pacotes popular. Em um caso famoso, ele foi entrevistado na Google para uma vaga de engenheiro de software e foi rejeitado. Após a entrevista, ele tuítou o seguinte: "Google: 90% de nossos engenheiros usam o software

que você criou (Homebrew), mas você não consegue inverter uma árvore binária em um quadro branco, então se dane". **Inverter uma árvore binária** significa trocar todos os nós dela. Em outras palavras, cada nó direito passará a ser um nó esquerdo e cada nó esquerdo será um nó direito. Nesta seção, você aprenderá como inverter uma árvore binária para não ter o mesmo destino de Max Howell em uma entrevista técnica.

Para inverter uma árvore binária, você terá de visitar cada nó e registrar seus filhos para trocá-los. Uma maneira de fazer isso é usando uma busca em largura, que permitirá que você registre facilmente cada filho esquerdo e cada filho direito e troque-os.

Veja o código que inverte uma árvore binária:

```
class BinaryTree:
def __init__(self, value):
self.key = value
self.left_child = None
self.right_child = None
def insert_left(self, value):
if self.left_child == None:
self.left_child = BinaryTree(value)
else:
bin_tree = BinaryTree(value)
bin_tree.left_child = self.left_child
self.left_child = bin_tree
def insert_right(self, value):
if self.right_child == None:
self.right_child = BinaryTree(value)
else:
bin_tree = BinaryTree(value)
bin_tree.right_child = self.right_child
self.right_child = bin_tree
def invert(self):
current = [self]
next = []
while current:
for node in current:
if node.left_child:
next.append(node.left_child)
if node.right_child:
next.append(node.right_child)
```

```
tmp = node.left_child
node.left_child = node.right_child
node.right_child = tmp
current = next
next = []
```

O código é o mesmo da busca em largura na qual procuramos um número, mas, em vez de verificar se o valor de um nó é `n`, você trocará os filhos direito e esquerdo a cada iteração.

Para fazê-lo, primeiro você terá de salvar `node.left_child` em uma variável temporária chamada `tmp`. Em seguida, configure `node.left_child` com `node.right_child` e `node.right_child` com `tmp`, o que trocará os dois nós-filhos:

```
tmp = node.left_child
node.left_child = node.right_child
node.right_child = tmp
```

Quando o algoritmo terminar, você terá invertido com sucesso sua árvore binária.

Uma maneira mais elegante de inverter uma árvore binária é usando a busca em profundidade, o que você pode fazer nos desafios.

Vocabulário

aresta: conexão entre dois nós em uma árvore.

árvore: tipo de dado abstrato não linear composto de nós conectados em uma estrutura hierárquica.

árvore binária: estrutura de dados de árvore na qual cada nó só pode ter dois filhos.

árvore de busca binária: estrutura de dados em árvore na qual cada nó só pode ter dois filhos e a árvore armazena seus nós de forma ordenada em que o valor de cada nó é maior do que o valor do seu filho esquerdo e menor do que o valor de seu filho direito.

árvore de parsing: árvore ordenada que armazena dados de acordo com algum tipo de sintaxe, como as regras de avaliação de uma expressão.

busca em largura: quando percorremos uma árvore em largura para fazer

uma busca.

busca em profundidade: visitar todos os nós de uma árvore binária, descendo o máximo possível em profundidade, na mesma direção, antes de passar para o próximo irmão.

descendentes: os filhos de um nó, seus filhos, os filhos desses filhos etc.

document object model: interface independente de linguagem que modela um documento XML ou HTML como uma árvore.

HTML: linguagem de marcação que podemos usar para criar páginas web.

inversão de uma árvore binária: troca de todos os nós da árvore.

nó-folha: nó sem nós-filhos.

nó-filho: nó conectado a um nó-pai acima dele em uma árvore.

nó-pai: nó com um ou mais nós-filhos.

nó-raiz: nó do topo de uma árvore.

nó-ramo: nó com nós-filhos.

nós-irmãos: nós que compartilham o mesmo pai.

percorrer pela largura: método que visita cada nó de uma árvore acessando-os nível a nível.

subárvore: nó de uma árvore, que não é o nó-raiz, e seus descendentes.

XML: linguagem de marcação para documentos.

Desafios

1. Adicione um método chamado `has_leaf_nodes` ao seu código de árvore binária. O método deve retornar `True` se a árvore tiver nós-filhos, e `False`, se não tiver.
2. Inverta uma árvore binária usando uma busca em profundidade.

CAPÍTULO 15

Heaps binários

Acho que a ascensão do Google, do Facebook e da Apple é prova de que existe um lugar para a Ciência da Computação como algo para solucionar os problemas que as pessoas enfrentam todo dia.

– Eric Schmidt

Uma **fila de prioridade** é um tipo de dado abstrato que descreve uma estrutura de dados na qual cada dado tem uma prioridade. Ao contrário de uma fila, que libera os itens usando o padrão primeiro a entrar, primeiro a ser servido, a fila de prioridade serve os elementos por prioridade. Primeiro, remove os dados de prioridade mais alta, seguidos daqueles de prioridade mais alta subsequentes (ou o oposto com o valor menor vindo primeiro). O heap é uma das muitas implementações da fila de prioridade. Um **heap** é uma estrutura de dados baseada em árvore na qual cada nó registra duas informações: um valor e sua prioridade. O valor do nó de um heap chama-se **chave**. Embora a chave de um nó e sua prioridade possam não estar relacionados, se seu dado for um valor numérico, como um inteiro ou caractere, você também poderá usá-lo como prioridade. Neste capítulo, usarei a chave dos heaps que veremos para também representar a prioridade.

Cientistas da computação criam heaps usando árvores. Existem muitos tipos de heaps (dependendo do tipo de árvore usada para criá-lo), mas, neste capítulo, você conhecerá os heaps binários. Um **heap binário** é aquele criado com uma árvore binária (Figura 15.1).

Existem dois tipos de heaps binários: heaps máximos (max heaps) e heaps mínimos (min heaps). A prioridade do nó-pai de um **heap máximo** sempre é maior que ou igual à prioridade de qualquer nó-filho e o nó de prioridade mais alta é a raiz da árvore. Por exemplo, a Figura 15.2 mostra um heap máximo com os inteiros 1, 2, 3, 4, 6, 8 e 10.

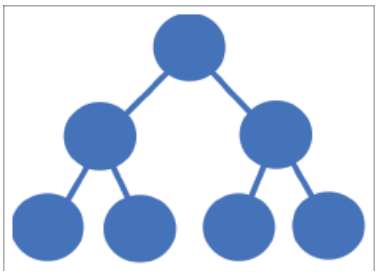

Figura 15.1: Criamos um heap binário usando uma árvore binária.

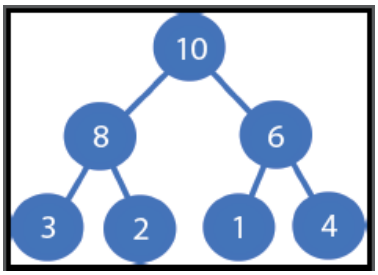

Figura 15.2: Um heap máximo tem o nó de prioridade mais alta como raiz.

A prioridade do nó-pai de um **heap mínimo** é sempre menor que ou igual à prioridade de qualquer nó-filho e o nó de prioridade mais baixa é a raiz da árvore. A Figura 15.3 mostra um heap mínimo com os mesmos inteiros do heap máximo da Figura 15.2.

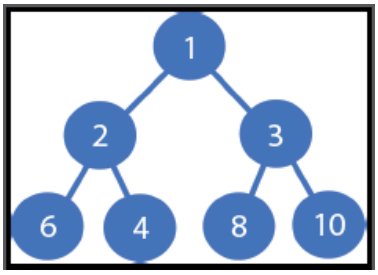

Figura 15.3: Um heap mínimo tem o nó de prioridade mais baixa como raiz.

Em um heap binário, a ordem (de prioridade mínima ou máxima) só é aplicável a um nó-pai e seus filhos. Não há ordem entre nós-irmãos. Como você pode ver na Figura 15.3, os irmãos não estão em ordem (6 e 4).

Cientistas da computação usam o termo **heapificar** para se referir à criação de um heap com base em uma estrutura de dados como um array . Por exemplo, digamos que você tivesse um array de chaves não ordenadas como este:

```
["R", "C", "T", "H", "E", "D", "L"]
```

Para heapificar esses dados, primeiro adicione cada dado como um nó a uma árvore binária. Comece no topo da árvore e, depois, forneça nós-filhos da esquerda para a direita em cada nível subsequente, como mostrado na Figura 15.4.

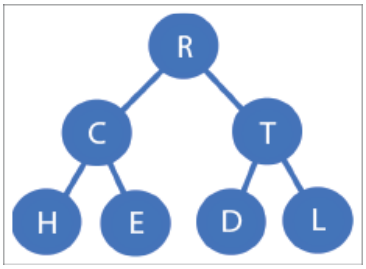

Figura 15.4: Resultado da heapificação de um array.

Em seguida, balanceie o heap. **Balancear um heap** significa reordenar as chaves fora de ordem. Nesse caso, comece com o último nó-pai (T) e compare-o com seus nós-folhas. Se algum dos nós-folhas tiver um valor menor do que o do nó-pai, troque-o com o nó-pai (Figura 15.5).

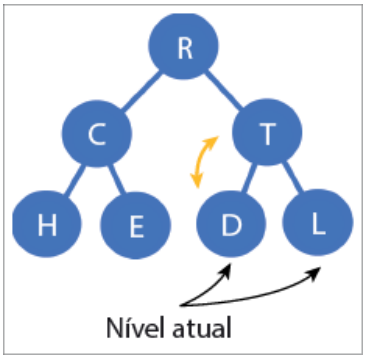

Figura 15.5: Trocando valores para balancear um heap.

Aqui, D é o menor dos três nós (T, D e L), portanto troque-o com o seu nó-pai T (Figura 15.6).

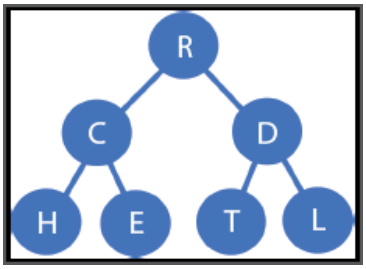

Figura 15.6: Trocar D e T é a primeira etapa para o balanceamento desse

heap.

Em seguida, passe para o penúltimo pai e seus nós-folhas (C, H e E). C vem antes tanto de H quanto de E, então não faça nenhuma troca no lado esquerdo da árvore (Figura 15.7).

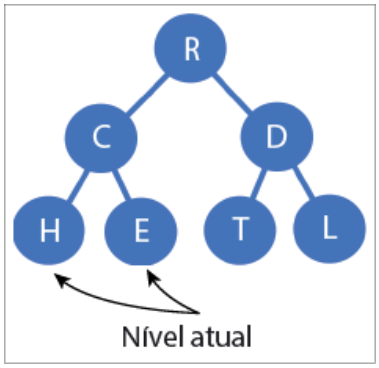

Figura 15.7: O lado esquerdo do heap já estava balanceado.

Agora, suba um nível e compare novamente (Figura 15.8).

C tem o valor mais baixo entre os nós R, C e D, logo troque C e R. C passou a ser a raiz da árvore (Figura 15.9).

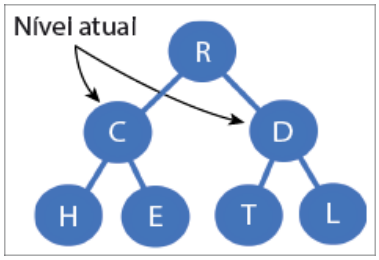

Figura 15.8: Balanceando a árvore no próximo nível.

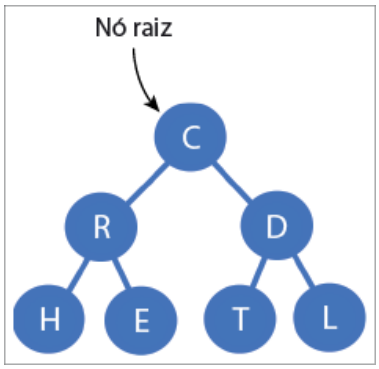

Figura 15.9: Agora C é o nó-raiz do heap binário.

Agora você "desce" o nó R comparando seu valor com o de seus nós-

folhas. Se o nó R tiver um valor maior que o de seus nós-folhas, troque-os e compare o valor de R com o dos novos nós-folhas. Continue fazendo isso enquanto R tiver um valor maior do que algum de seus nós-folhas ou até alcançar o nível mais baixo do heap. E vem antes de R, então os troque (Figura 15.10).

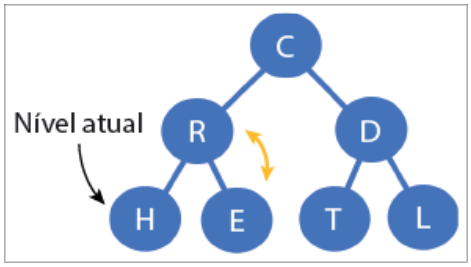

Figura 15.10: O nó R descerá pela árvore enquanto tiver um valor maior do que algum de seus nós-folhas.

O heap está balanceado (Figura 15.11).

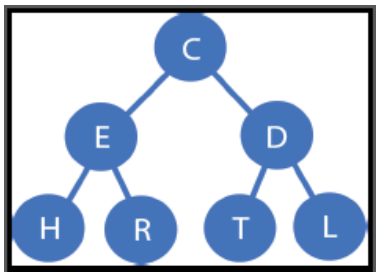

Figura 15.11: Um heap balanceado.

Em geral, cientistas da computação armazenam os heaps em arrays. Você pode armazenar um heap em uma lista Python, distribuindo as chaves em uma lista baseada na sua posição na árvore (Figura 15.12).

Figura 15.12: Array com chaves em índices de acordo com sua posição na árvore.

Dentro da lista, o nó-raiz do heap está no índice 0. Seu filho esquerdo está no índice 1 e seu nó-filho direito está no índice 2. Você pode usar uma equação matemática para encontrar a posição do filho de um nó. Para

qualquer nó, k, o índice de seu filho esquerdo é igual a $2k + 1$ e o do filho direito é $2k + 2$. Por exemplo, a equação que encontra o filho direito de C é 2 * 0 + 2, que é igual a 2. Isso significa que o filho direito do nó do índice 0 está no índice 2 (Figura 15.13).

Figura 15.13: O filho direito da raiz está no índice 2.

Quando usar heaps

Você pode encontrar o valor máximo ou mínimo de um heap máximo ou mínimo em tempo constante, mas a remoção do nó mínimo de um heap mínimo ou do nó máximo de um heap máximo é logarítmica porque depois que você remover o item, terá de balancear os nós restantes. A inserção de dados em um heap é logarítmica e a busca de dados é O(n).

Os heaps serão úteis sempre que você tiver de executar tarefas de acordo com prioridades. Por exemplo, seu sistema operacional poderia usar um heap para registrar diferentes tarefas e alocar recursos dependendo da prioridade de cada uma. Você também pode usar um heap para implementar o algoritmo de Dijkstra para encontrar o caminho mais curto entre dois nós em um grafo. O algoritmo de Dijkstra, que você conhecerá no Capítulo 16, poderá ajudá-lo a resolver problemas de rotas e a determinar como chegar de uma cidade a outra e fazer o roteamento em redes de computador. Cientistas da computação também usam heaps em um algoritmo de ordenação chamado *heapsort (ordenação por heap)*.

Criando um heap

O Python tem uma função de biblioteca chamada `heapq` que facilita a criação de um heap mínimo. Aqui, há um programa que usa a função `heapify` da biblioteca `heapq` para a heapificação de uma lista de sete elementos:

```
from heapq import heapify
```

```
a_list = ['R', 'C', 'T', 'H', 'E', 'D', 'L']
heapify(a_list)
print(a_list)
```

```
>> ['C', 'E', 'D', 'H', 'R', 'T', 'L']
```

Primeiro, importe a função `heapify` da biblioteca `heapq`. Em seguida, passe uma lista para a função `heapify`. É possível verificar que quando exibirmos a lista heapificada, será um heap mínimo armazenado em uma lista Python.

Você pode usar a função `heappop` da biblioteca `heapq` para extrair uma chave de um heap e rebalanceá-lo. Veja como remover a chave-raiz de um heap e balancear as chaves restantes:

```
from heapq import heapify, heappop
a_list = ['R', 'C', 'T', 'H', 'E', 'D', 'L']
heap = heapify(a_list)
print(a_list)
heappop(a_list)
print("After popping")
print(a_list)
```

```
>> ['C', 'E', 'D', 'H', 'R', 'T', 'L']
>> After popping
>> ['D', 'E', 'L', 'H', 'R', 'T']
```

Primeiro, importe `heapify` e `heappop` do módulo `heapq`:

```
from heapq import heapify, heappop
```

Em seguida, crie um heap, passando sua lista para a função `heapify`, e exiba-o:

```
a_list = ['R', 'C', 'T', 'H', 'E', 'D', 'L']
heap = heapify(a_list)
print(a_list)
```

Agora, use a função `heappop` para remover o elemento mínimo do heap e exiba o resultado:

```
heappop(a_list)
print("After popping")
print(a_list)
```

Você pode usar um loop `while` para remover todos os elementos de um heap. Veja como criar um heap e remover todas as suas chaves:

```
from heapq import heapify, heappop
a_list = ['D', 'E', 'L', 'H', 'R', 'T']
heapify(a_list)
while len(a_list) > 0:
print(heappop(a_list))
```

Primeiro, crie um heap:

```
a_list = ['D', 'E', 'L', 'H', 'R', 'T']
heapify(a_list)
```

Em seguida, use um loop `while` para remover todas as chaves:

```
while len(a_list) > 0:
print(heappop(a_list))
```

A biblioteca `heapq` também tem uma função chamada `heappush` que insere uma chave em um heap e o rebalanceia. Veja como usar `heappush` para inserir um item no heap:

```
from heapq import heapify, heappush
a_list = ['D', 'E', 'L', 'H', 'R', 'T']
heapify(a_list)
heappush(a_list, "Z")
print(a_list)
>> ['D', 'E', 'L', 'H', 'R', 'T', 'Z']
```

O Python fornece uma função interna apenas para heaps mínimos, mas você pode criar facilmente um heap máximo para valores numéricos multiplicando cada valor por -1. Um heap máximo com strings como chaves é mais difícil de implementar. Em vez de usar a biblioteca `heapq`, você terá de criar o heap usando uma classe ou codificá-lo por conta própria.

Para concluir, você pode usar `heapq` para manipular pares de prioridade e valor com o armazenamento de tuplas cujo primeiro elemento seja a prioridade e o segundo seja o valor, que pode ser qualquer coisa. Veremos um exemplo disso no Capítulo 16 quando você codificar o algoritmo de Dikijstra.

Emendando cordas com custo mínimo

Você pode usar heaps para resolver problemas que surgirem em sua

programação diária e também os encontrados em entrevistas técnicas. Por exemplo, em uma entrevista técnica, você pode receber uma lista com diferentes tamanhos de cordas e ser solicitado a emendar todas elas, duas de cada vez, na ordem que resultar no menor custo total. O custo de emendar duas cordas é a sua soma e o custo total é a soma da emenda de todas as cordas. Por exemplo, digamos que você receba esta lista de cordas:

```
[5, 4, 2, 8]
```

Primeiro, você pode emendar 8 e 2, depois 4 e 10 e, então, 5 e 14. Quando somar cada custo, obterá 43:

```
[5, 4, 2, 8] # 8 + 2 = 10
[5, 4, 10] # 10 + 4 = 14
[5, 14] # 5 + 14 = 19
# 10 + 14 + 19 = 43
```

No entanto, se você emendar as cordas em uma ordem diferente, obterá outra resposta. Para obter a resposta correta, será preciso emendar sempre as duas cordas menores desta forma:

```
[5, 4, 2, 8] # 4 + 2 = 6
[5, 8, 6] # 6 + 5 = 11
[8, 11] # 8 + 11 = 19
# 6 + 11 + 19 = 36
```

O custo total ao resolvermos o problema é 36, que é a resposta correta.

Você pode usar um heap mínimo para escrever uma função que resolva esse problema. Veja como fazer isso:

```
from heapq import heappush, heappop, heapify
def find_min_cost(ropes):
heapify(ropes)
cost = 0
while len(ropes) > 1:
sum = heappop(ropes) + heappop(ropes)
heappush(ropes, sum)
cost += sum
return cost
```

Primeiro, defina uma função `find_min_cost` que receba sua lista de cordas como parâmetro:

```
def find_min_cost(ropes):
```

Em seguida, use `heapify` para transformar `ropes` em um heap mínimo e defina uma variável chamada `cost` para registrar o custo total da adição de todas as cordas:

```
heapify(ropes)
cost = 0
```

Agora, crie um loop `while` que seja executado enquanto o tamanho de `ropes` for maior do que 1:

```
while len(ropes) > 1:
```

Dentro do loop, use `heappop` para obter os dois valores mais baixos do heap e somá-los. Em seguida, use `heappush` para inserir sua soma novamente no heap. Por fim, adicione a soma ao custo:

```
sum = heappop(ropes) + heappop(ropes)
heappush(ropes, sum)
cost += sum
```

Quando o loop terminar, você retornará `cost`, que conterá o custo mais baixo da emenda de todas as cordas:

```
return cost
```

Vocabulário

balancear um heap: reordenar as chaves fora de ordem.

chave: valor de um nó em um heap.

fila de prioridade: tipo de dado abstrato que descreve uma estrutura de dados na qual cada dado tem uma prioridade.

heap: estrutura de dados baseada em árvore na qual cada nó registra dois dados: os dados propriamente ditos e sua prioridade.

heap binário: heap que usa uma árvore binária como sua estrutura de dados subjacente.

heap máximo: heap no qual a prioridade do nó-pai é sempre maior que ou igual à prioridade de qualquer nó- filho e o nó com prioridade mais alta é a raiz da árvore.

heap mínimo: heap no qual a prioridade do nó-pai é sempre menor do

que ou igual à prioridade de qualquer nó-filho e o nó com prioridade mais baixa é a raiz da árvore.

heapificar: criar um heap com base em uma estrutura de dados como um array.

Desafio

1. Escreva uma função que receba uma árvore binária como parâmetro e retorne `True`, se for um heap mínimo, e `False`, se não for.

CAPÍTULO 16

Grafos

Aprender a codificar lhe dará um grande impulso para o futuro, independentemente de quais forem seus planos profissionais. Também o tornará uma pessoa extremamente interessante!

– Max Levchin

Um **grafo** é um tipo de dado abstrato no qual um dado se conecta a um ou mais outros dados. Cada dado de um grafo é chamado de **vértice** ou *nó*. Um vértice tem um nome que chamamos de chave. O vértice pode ter dados adicionais chamados **payload**. A conexão entre os vértices de um grafo chama-se **aresta**. As arestas do grafo podem conter um **peso**: o custo do trajeto entre os vértices. Por exemplo, se você criasse um grafo para representar os dados de um mapa, cada vértice poderia ser uma cidade e o peso entre dois vértices seria a distância entre eles (Figura 16.1).

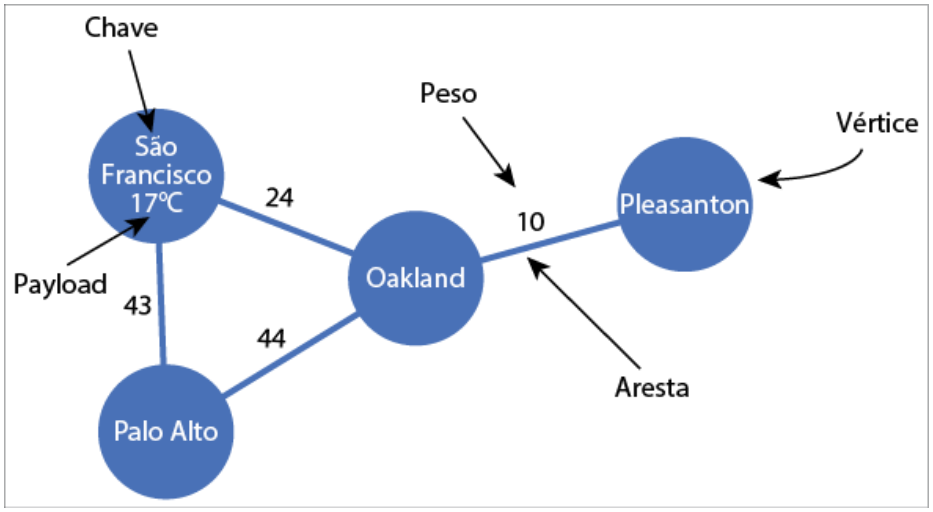

Figura 16.1: Um grafo contém vértices, arestas, payloads e peso.

Existem vários tipos de grafos, que incluem grafos direcionados, não direcionados e completos. Um **grafo direcionado** é aquele em que cada

aresta tem uma direção associada a ela e só podemos nos mover entre dois vértices nessa direção. Normalmente, a conexão entre dois vértices se dá em uma única direção, mas você também pode transformar uma aresta em uma conexão bidirecional. Um grafo direcionado seria uma excelente escolha para representar uma rede social com seguidores (como o Twitter). Por exemplo, você poderia usar um grafo direcionado para representar que está seguindo Lebron James no Twitter, mas ele não o está seguindo. Quando desenhamos um grafo direcionado, geralmente representamos as arestas com setas mostrando a direção para a qual podemos nos mover (Figura 16.2).

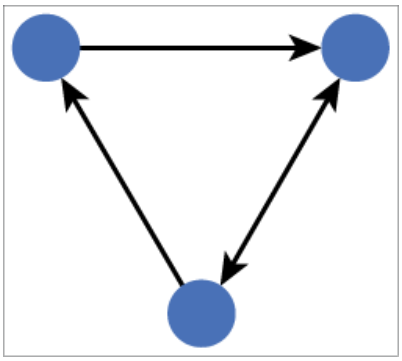

Figura 16.2: Um grafo direcionado segue uma direção específica.

Um **grafo não direcionado** é aquele em que as arestas são bidirecionais, o que significa que podemos nos mover em qualquer direção entre dois vértices conectados. Pode ser considerado uma conexão de duas vias, semelhante ao relacionamento entre amigos em uma rede social como o Facebook. Por exemplo, se Logan é amigo de Hadley no Facebook, então Hadley é amigo de Logan. Quando desenhamos um grafo não direcionado, geralmente usamos arestas sem setas (Figura 16.3).

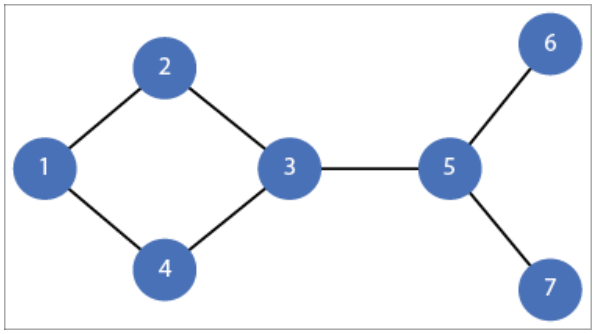

Figura 16.3: Um grafo não direcionado pode se mover nas duas direções.

Um **grafo completo** é aquele em que cada vértice está conectado a todos os outros vértices (Figura 16.4).

Em um **grafo incompleto**, alguns vértices, mas não todos, estão conectados (Figura 16.5).

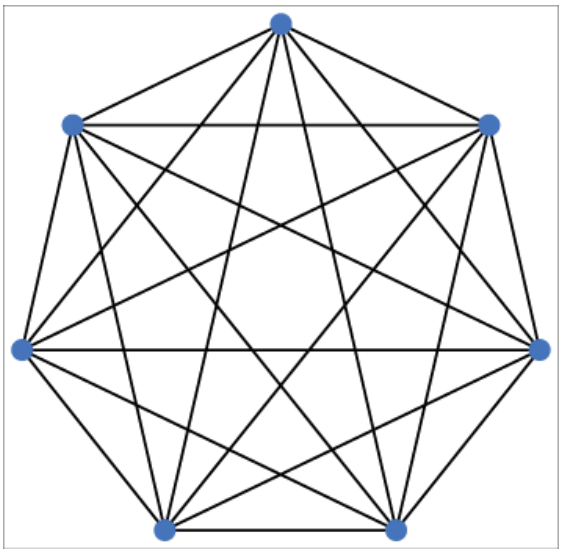

Figura 16.4: Um grafo completo tem conexões entre todos os vértices.

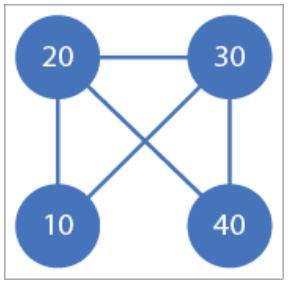

Figura 16.5: Um grafo incompleto tem alguns vértices conectados.

Um **caminho** em um grafo é uma sequência de vértices conectados por arestas (Figura 16.6). Por exemplo, digamos que você tivesse um grafo representando uma cidade. Um caminho entre Los Angeles e São Francisco seria uma série de arestas (representando estradas) a serem usadas no trajeto entre as duas cidades.

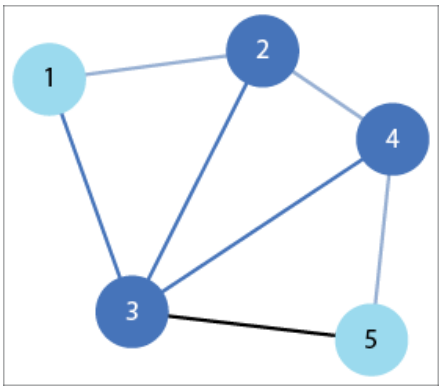

Figura 16.6: Um caminho em um grafo segue uma sequência específica.

Um **ciclo** é um caminho em um grafo que começa e termina no mesmo vértice. Um **grafo acíclico** não contém um ciclo. Consulte a Figura 16.7.

Muitos desses conceitos devem ser familiares porque você já conhece as árvores. Uma árvore é um tipo restrito de grafo. Árvores têm direção (o relacionamento pai-filho) e não contêm ciclos, o que as torna grafos acíclicos direcionados com uma restrição: um filho só pode ter um pai.

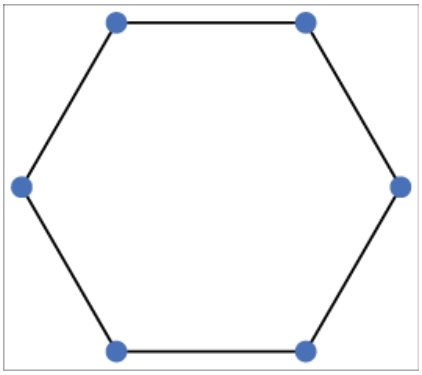

Figura 16.7: Exemplo de um grafo que contém um ciclo.

Existem várias maneiras de criar grafos programaticamente. Por exemplo, você pode usar uma lista de arestas, uma matriz de adjacências ou uma lista de adjacências. Uma **lista de arestas** é uma estrutura de dados na qual representamos cada aresta de um grafo com dois vértices que se conectam. Por exemplo, a Figura 16.8 é um grafo com quatro vértices.

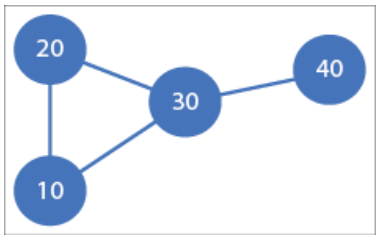

Figura 16.8: Grafo com quatro vértices.

Você pode representá-lo com uma lista de arestas desta forma:

```
[
[10, 20]
[10, 30]
[20, 10]
[20, 30]
[30, 10]
[30, 20]
[30, 40]
[40, 30]
]
```

Essa lista de arestas é uma lista de listas e cada lista contém dois vértices do grafo que se conectam.

Você também pode representar um grafo com uma matriz de adjacências. Uma **matriz de adjacências** é um array bidimensional de linhas e colunas que contém os vértices de um grafo. Em uma matriz de adjacências, usamos a interseção de cada linha e coluna para representar uma aresta. Tradicionalmente, usamos 1 para representar vértices que se conectam e 0 para exibir vértices não conectados. Quando dois vértices estão conectados, são **adjacentes**. A Figura 16.9 mostra como representar o mesmo grafo com uma matriz de adjacências:

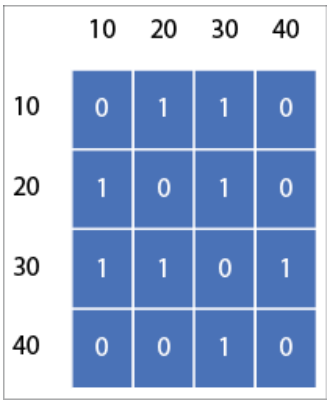

Figura 16.9: Matriz de adjacências do grafo da Figura 16.8.

Um problema das matrizes de adjacências é a esparsidade ou as células vazias. Existem oito células vazias nesse exemplo. As matrizes de adjacências não são uma maneira muito eficiente de armazenar dados porque acabam tendo um grande número de células vazias, o que é um uso ineficiente da memória do computador.

Para concluir, você também pode representar um grafo com uma lista de adjacências. Uma **lista de adjacências** é uma coleção de listas não ordenadas, cada uma representando as conexões de um único vértice. A seguir, temos o mesmo grafo da Figura 16.8 como uma lista de adjacências:

```
{
10: [20, 30],
20: [10, 30],
30: [10, 20, 40],
40: [30]
}
```

Como você pode ver, o nó 10 se conecta ao 20 e ao 30, o nó 20 se conecta ao 10 e ao 30, e assim por diante.

Quando usar grafos

Como sabemos, existem muitas implementações diferentes para os grafos. Adicionar um vértice e uma aresta a um grafo geralmente tem complexidade O(1). O tempo de execução de busca, exclusão e outros algoritmos em um grafo depende de sua implementação e das estruturas

de dados usadas nela: arrays, listas encadeadas, tabelas hash etc. Em geral, o desempenho das operações básicas em grafos depende do número de vértices, do número de arestas ou de alguma combinação desses dois números, já que os grafos lidam basicamente com duas coisas: os itens do grafo (vértices) e as conexões (arestas) entre esses itens.

Os grafos são úteis em muitas situações. Por exemplo, engenheiros de software de empresas de mídia social, como Instagram e Twitter, usam os vértices de um grafo para representar pessoas e as arestas para representar as associações entre elas. Programadores também usam grafos para construir redes, geralmente representando os dispositivos com vértices e as arestas indicando conexões wireless ou com fio entre esses dispositivos. Você pode usar grafos para criar mapas, com cada vértice representando cidades e outros destinos e as arestas representando estradas, rotas de ônibus ou rotas aéreas entre esses destinos. Além disso, programadores usam grafos para encontrar o caminho mais rápido entre os destinos. Os grafos também são úteis na computação gráfica. Você pode usar os vértices e as arestas dos grafos para representar os pontos, as linhas e os planos de formas 2D e 3D (Figura 16.10).

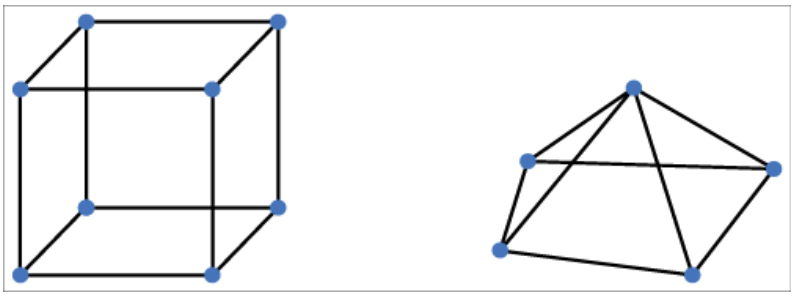

Figura 16.10: Grafos podem representar formas 3D.

Os algoritmos dos mecanismos de busca costumam usar grafos para determinar rankings de busca de acordo com a conectividade da busca e os resultados. Os sistemas operacionais e os sistemas de linguagens de programação também usam grafos no gerenciamento da memória.

Criando um grafo

Veja como criar uma lista de adjacências em Python:

```
class Vertex:
def __init__(self, key):
self.key = key
self.connections = {}
def add_adj(self, vertex, weight=0):
self.connections[vertex] = weight
def get_connections(self):
return self.connections.keys()
def get_weight(self, vertex):
return self.connections[vertex]
class Graph:
def __init__(self):
self.vertex_dict = {}
def add_vertex(self, key):
new_vertex = Vertex(key)
self.vertex_dict[key] = new_vertex
def get_vertex(self, key):
if key in self.vertex_dict:
return self.vertex_dict[key]
return None
def add_edge(self, f, t, weight=0):
if f not in self.vertex_dict:
self.add_vertex(f)
if t not in self.vertex_dict:
self.add_vertex(t)
self.vertex_dict[f].add_adj(self.vertex_dict[t], weight)
```

Primeiro, defina uma classe de vértice, como fez anteriormente com os nós quando criou listas encadeadas:

```
class Vertex:
def __init__(self, key):
self.key = key
self.connections = {}
def add_adj(self, vertex, weight=0):
self.connections[vertex] = weight
```

Sua classe `Vertex` tem duas variáveis de instância: : `self.key` e `self.connections`. A primeira variável, `key`, representa a chave do vértice e a segunda, `connections`, é um dicionário no qual você armazenará os vértices aos quais cada vértice é adjacente:

```
def __init__(self, key):
self.key = key
```

```
self.connections = {}
```

A classe `Vertex` tem um método chamado `add_adj` que recebe um vértice como parâmetro e torna-o adjacente ao vértice no qual você chamou o método adicionando a conexão a `self.connections`. O método também receberá um peso se você quiser adicioná-lo ao relacionamento:

```
def add_adj(self, vertex, weight=0):
self.connections[vertex] = weight
```

Agora, defina uma classe chamada `Graph`, a qual tem uma variável de instância, `self.vertex_dict`, que armazena os vértices de cada grafo:

```
def __init__(self):
self.vertex_dict = {}
```

O método `add_vertex` da classe adiciona um novo vértice a um grafo, primeiro criando um vértice e, depois, mapeando a chave passada pelo usuário como parâmetro para o novo vértice dentro de `self.vertex_dict`:

```
def add_vertex(self, key):
new_vertex = Vertex(key)
self.vertex_dict[key] = new_vertex
```

O próximo método, `get_vertex`, recebe uma chave como parâmetro e verifica `self.vertex_dict` para saber se o vértice é o grafo:

```
def get_vertex(self, key):
if key in self.vertex_dict:
return self.vertex_dict[key]
return None
```

Por fim, a classe de grafo tem um método chamado `add_edge` que adiciona uma aresta entre dois vértices do grafo:

```
def add_edge(self, f, t, weight=0):
if f not in self.vertex_dict:
self.add_vertex(f)
if t not in self.vertex_dict:
self.add_vertex(t)
self.vertex_dict[f].add_adj(self.vertex_dict[t], weight)
```

Agora, você pode criar um grafo e adicionar vértices a ele desta forma:

```
graph = Graph()
graph.add_vertex("A")
```

```
graph.add_vertex("B")
graph.add_vertex("C")
graph.add_edge("A", "B", 1)
graph.add_edge("B", "C", 10)
vertex_a = graph.get_vertex("A")
vertex_b = graph.get_vertex("B")
```

Nesse exemplo, para simplificar as coisas, dois vértices diferentes não podem ter a mesma chave.

Algoritmo de Dijkstra

Quando trabalhamos com grafos, geralmente temos de encontrar o caminho mais curto entre dois vértices. Um dos algoritmos mais famosos da ciência da computação chama-se **algoritmo de Dijkstra** e você pode usá-lo para encontrar o caminho mais curto entre o vértice de um grafo e todos os outros vértices. O conhecido cientista da computação Edsger Dijkstra o inventou mentalmente, em apenas 20 minutos, sem papel ou caneta.

Vejamos como o algoritmo de Dijkstra funciona. Primeiro, você deve selecionar o vértice inicial, que será aquele para o qual você encontrará o caminho mais curto até todos os outros vértices do grafo. Digamos que você tivesse um grafo semelhante ao da Figura 16.11.

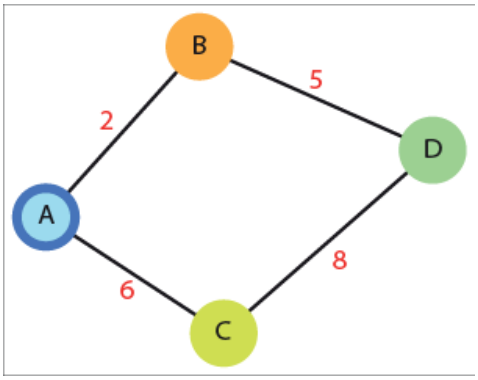

Figura 16.11: Grafo com quatro vértices.

Se A for o vértice inicial, no fim do programa você terá um dicionário contendo todos os vértices do grafo e o caminho mais curto do vértice inicial (A) até cada vértice.

```
{
"A": 0,
"B": 2,
"C": 6,
"D": 7,
}
```

Como você pode ver na Figura 16.11, o caminho mais curto de A a D é 7 porque o trajeto do vértice A ao vértice B e, depois, ao D é igual a 7 (2 + 5) e o do vértice A ao vértice C e, depois, ao D é 14 (6 + 8).

No começo do algoritmo, configure o caminho do vértice inicial até ele mesmo com zero e defina as extensões de todos os outros caminhos com infinito (Figura 16.12).

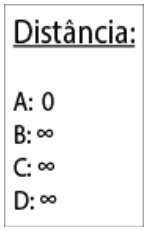

Figura 16.12: Configure o caminho do vértice inicial até ele mesmo com zero e os outros caminhos com infinito.

O segredo do algoritmo é a fila de prioridade. Você usará uma fila de prioridade para fazer uma busca em largura no grafo. Também a usará para registrar os vértices e suas distâncias a partir do vértice inicial. Examinaremos como isso funciona com o grafo anterior.

Seu algoritmo começa com o vértice inicial A na fila de prioridade. Você registrará todos os caminhos mais curtos em um dicionário. Nele, configure a distância do vértice A até ele mesmo com 0 e todas as outras distâncias com infinito (Figura 16.13). Você ainda não visitou nenhum vértice. Visitar um vértice significa removê-lo da fila de prioridade e caso não encontre um caminho mais curto do vértice inicial até ele, examine em todos os vértices adjacentes caminhos mais curtos até o vértice inicial. Se encontrar um caminho mais curto, insira esse vértice adjacente na fila de prioridade.

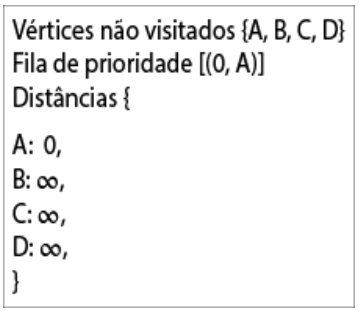

Figura 16.13: Como se encontram as estruturas de dados de seu algoritmo quando ele começa.

Nesse momento, a fila de prioridade tem apenas um vértice, logo remova o vértice A e sua distância até o vértice inicial (0) da fila de prioridade e verifique se já encontrou um caminho mais curto até esse vértice. Quando você estiver examinando um novo vértice da fila de prioridade, se já tiver encontrado um caminho mais curto do ponto inicial até ele, não precisará fazer mais nada. Nesse caso, você não encontrou um caminho mais curto até o vértice A, portanto percorra todos os vértices adjacentes a ele.

Em seguida, calcule as distâncias entre os vértices adjacentes e o vértice inicial. Faça isso adicionando a distância que obteve na fila de prioridade ao peso do vértice adjacente (sua distância a partir do vértice que você removeu da fila de prioridade). Já que a fila de prioridade contém duas informações para cada vértice existente nela – o vértice e sua distância do vértice inicial –, independentemente de quanto você se afastar do vértice inicial, poderá calcular facilmente a distância entre um novo vértice adjacente e o vértice inicial, adicionando a distância que obteve na fila de prioridade ao peso dos vértices adjacentes.

Se a distância entre o vértice adjacente e o vértice inicial for mais curta do que qualquer um dos caminhos que você encontrou até agora, adicione o novo caminho ao dicionário e insira o vértice adjacente na fila de prioridade. Nesse caso, você inseriu os dois vértices adjacentes ao vértice A (B e C) na fila de prioridade e adicionou seus caminhos ao dicionário (Figura 16.14).

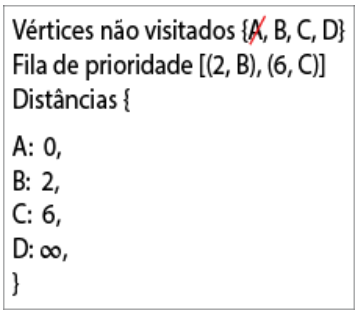

Figura 16.14: Estruturas de dados após a visita ao vértice A.

Agora, remova o vértice B da fila de prioridade porque tem prioridade mais alta (o caminho mais curto até o vértice inicial). Você ainda não encontrou um caminho mais curto de B ao vértice inicial, então continue visitando esse vértice. Procure caminhos mais curtos em todos os seus vértices adjacentes, adicione ao dicionário qualquer caminho mais curto que encontrar e atualize a fila de prioridade. B só tem um vértice adjacente com caminho mais curto (D), logo atualize D com 7 em seu dicionário e adicione D e sua distância até o vértice inicial à fila de prioridade (Figura 16.15).

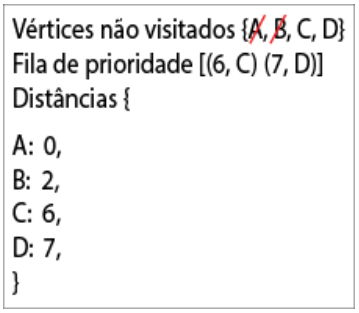

Figura 16.15: Estruturas de dados após a visita ao vértice B.

Em seguida, remova o vértice C da fila de prioridade porque tem o caminho mais curto da fila. C também é adjacente a D, mas, como vimos anteriormente, sua distância a partir do vértice inicial é 14 e você já encontrou um caminho mais curto para D, portanto não adicione D à fila de prioridade novamente (Figura 16.16). Ignorar caminhos mais longos (não os visitar de novo) é o que torna esse algoritmo tão eficiente.

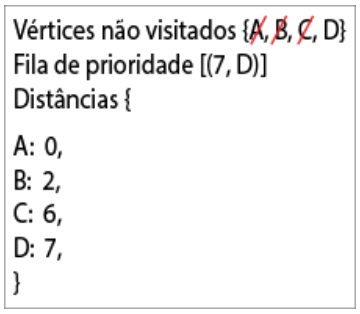

Figura 16.16: Estruturas de dados após a visita ao vértice C.

O vértice D não é adjacente a nenhum outro vértice, logo, após você removê-lo, o algoritmo estará concluído (Figura 16.17).

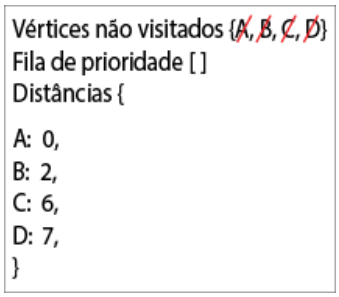

Figura 16.17: Estruturas de dados após a visita ao vértice D.

A seguir, temos o código que implementa o algoritmo de Dijkstra em Python. Nesse caso, espera um grafo como dicionário de dicionários em vez da classe `Graph` que você codificou anteriormente no capítulo.

```
import heapq
def dijkstra(graph, starting_vertex):
distances = {vertex: float('infinity') for vertex in graph}
distances[starting_vertex] = 0
pq = [(0, starting_vertex)]
while len(pq) > 0:
current_distance, current_vertex = heapq.heappop(pq)
if current_distance > distances[current_vertex]:
continue
for neighbor, weight in graph[current_vertex].items():
distance = current_distance + weight
if distance < distances[neighbor]:
distances[neighbor] = distance
heapq.heappush(pq, (distance, neighbor))
return distances
graph = {
```

```
'A': {'B': 2, 'C': 6},
'B': {'D': 5},
'C': {'D': 8},
'D': {},
}
dijkstra(graph, 'A')
print(dijkstra(graph, 'A'))
```

Primeiro, importe `heapq` porque o algoritmo usa um heap como fila de prioridade. A função `dijkstra` retorna um dicionário contendo os caminhos mais curtos a partir do vértice inicial. Ela recebe dois parâmetros: um grafo e o vértice a partir do qual você deseja encontrar os caminhos mais curtos:

```
import heapq
def dijkstra(graph, starting_vertex):
```

Nessa implementação, você passará uma lista de adjacências como esta:

```
graph = {
'A': {'B': 2, 'C': 6},
'B': {'D': 5},
'C': {'D': 8},
'D': {},
}
```

Quando chamar a função `dijkstra`, passará o grafo e uma string para representar um vértice inicial desta forma:

```
dijkstra(graph, 'A')
```

O vértice inicial deve ser um vértice do grafo.

Dentro da função, crie um dicionário chamado `distances` para armazenar os caminhos do vértice inicial até cada outro vértice do grafo. No fim do algoritmo, esse dicionário conterá o caminho mais curto do vértice inicial até todos os outros vértices. Crie o dicionário usando uma compreensão de dicionário (*dictionary comprehension*), que é semelhante a uma compreensão de lista (*list comprehension*), mas para dicionários. Sua compreensão de dicionário mapeará cada vértice para `float('infinity')`: a representação do Python para infinito. Você mapeará cada vértice para o infinito porque o algoritmo compara os tamanhos dos caminhos e, no início, estes são desconhecidos, portanto

use infinito para representar isso:

```
distances = {vertex: float('infinity') for vertex in graph}
```

Quando você passar o dicionário (representando um grafo) anterior para `dijkstra`, o código produzirá um dicionário como este:

```
{'A': inf, 'B': inf, 'C': inf, 'D': inf}
```

Em seguida, configure a distância do vértice inicial (o vértice a partir do qual você está encontrando os caminhos mais curtos) até ele mesmo com zero, já que a distância entre um vértice e ele mesmo é zero:

```
distances[starting_vertex] = 0
```

Agora, crie uma lista (você a usará como fila de prioridade) que, inicialmente, contenha o vértice inicial e a distância de si próprio (zero):

```
pq = [(0, starting_vertex)]
```

Em seguida, vem o código que visita os vértices da fila de prioridade. Use um loop `while` que seja executado enquanto ainda houver um ou mais vértices na fila de prioridade. Use esse loop `while` para visitar todos os vértices do seu grafo:

```
while len(pq) > 0:
```

Dentro do loop `while`, remova a distância entre o vértice inicial e o vértice atual da fila de prioridade e salve-os nas variáveis `current_distance` e `current_vertex`. O vértice atual é o vértice da fila de prioridade que tem a menor distância a partir do vértice inicial. A fila de prioridade servirá automaticamente o vértice com a menor distância sempre que você remover um novo vértice dela (porque a fila de prioridade é um heap mínimo):

```
current_distance, current_vertex = heapq.heappop(pq)
```

Você só deseja processar um vértice se ainda não tiver encontrado um caminho mais curto desse vértice até o vértice inicial. Por isso, agora, verificará se a distância atual a partir do vértice inicial é maior do que uma distância que você já registrou no dicionário `distances`. Se for maior, não se preocupe com esse caminho porque você já registrou um caminho mais curto, então use a palavra-chave `continue` para voltar ao topo do loop `while` e examinar outro vértice (se houver um):

```
if current_distance > distances[current_vertex]:
continue
```

Se `current_distance` não for maior do que (em outras palavras, for mais curta ou igual a) `distances[current_vertex]`, percorra todos os vértices adjacentes ao vértice atual:

```
for neighbor, weight in graph[current_vertex].items():
```

Para cada vértice adjacente, calcule sua distância do vértice inicial adicionando `current_distance` ao seu peso. Esse cálculo funciona porque `current_distance` representa quanto o vértice atual está distante do inicial. A variável `weight` representa quanto o vértice adjacente está distante do vértice inicial, logo, quando você as somar, obterá a distância a partir do vértice inicial:

```
distance = current_distance + weight
```

Em seguida, verifique se o novo caminho que encontrou para esse vértice adjacente é mais curto do que o caminho que já tem para ele no dicionário `distances`. Se for, atualize o dicionário com o novo caminho. Depois, insira a nova distância e o vértice na fila de prioridade para que o algoritmo possa visitá-lo:

```
if distance < distances[neighbor]:
distances[neighbor] = distance
heapq.heappush(pq, (distance, neighbor))
```

Quando sair do loop `while`, significará que examinou todos os vértices e, agora, `distances` contém o caminho mais curto do vértice inicial até todos os outros vértices do grafo. Só falta retornar `distances`.

```
return distances
```

Vocabulário

adjacentes: dois ou mais vértices conectados em um grafo.

algoritmo de Dijkstra: algoritmo que podemos usar para encontrar o caminho mais curto do vértice de um grafo até todos os outros vértices.

aresta: conexão entre os vértices de um grafo.

caminho: sequência de vértices conectados por arestas.

ciclo: caminho em um grafo que começa e termina no mesmo vértice.

grafo: tipo de dado abstrato no qual um dado se conecta a um ou mais outros dados.

grafo acíclico: grafo que não contém um ciclo.

grafo completo: grafo no qual cada vértice se conecta a todos os outros vértices.

grafo direcionado: grafo no qual cada aresta tem uma direção associada a ela e só podemos nos mover entre dois vértices nessa direção.

grafo incompleto: grafo no qual alguns vértices, mas não todos, não estão conectados.

grafo não direcionado: tipo de grafo no qual as arestas são bidirecionais, o que significa que podemos avançar e recuar nas duas direções entre dois vértices conectados.

lista de adjacências: coleção de listas não ordenadas em que cada lista representa as conexões de um único vértice.

lista de arestas: estrutura de dados na qual representamos cada aresta de um grafo com dois vértices que se conectam.

matriz de adjacências: matriz bidimensional de linhas e colunas que contém os vértices de um grafo.

payload: dados adicionais do vértice de um grafo.

peso: custo do trajeto entre os vértices.

vértice: elemento de dados de um grafo.

Desafio

1. Modifique o algoritmo de Dijkstra para que só retorne o caminho de um vértice inicial até outro vértice que você passar.

CAPÍTULO 17

Inspiração autodidata: Elon Musk

Atualmente, Elon Musk é mais conhecido por ser o fundador da Tesla, SpaceX e PayPal, que revolucionaram seus setores industriais. No entanto, muito antes de se tornar um empreendedor e um dos homens mais ricos do mundo, Musk foi influenciado por uma ideia mais simples: queria projetar videogames. Como o programador autodidata Musk passou de um garoto jogador de games a bilionário? Neste capítulo, você conhecerá a trajetória educacional de Musk e como seu interesse por jogos o levou a aprender a programar.

A formação de Musk começou com viagens ao redor do planeta até ele chegar em sua casa, em Los Angeles, onde reside hoje. Nascido e criado na África do Sul, passou a se interessar por computadores quando tinha 10 anos. Foi uma criança determinada que, às vezes, passava dez horas diárias lendo. Também era obcecado por videogames. Musk disse que sua paixão por videogames o levou a aprender a programar. "Pensava que poderia criar meus próprios jogos. Queria ver como funcionam", explicou. "Foi isso que me levou a aprender a programar computadores."

Musk começou com um livro sobre a linguagem de programação BASIC, uma linguagem popular nos anos de 1960, que muitos computadores ainda usavam na década de 1980. O livro oferecia um programa para aprender a codificar em seis meses, mas Musk estudou o programa inteiro em três dias. Não demorou muito para ele programar seu primeiro videogame. Em 1984, quando tinha apenas 12 anos, criou o *Blastar*. O lançador de projéteis espacial foi inspirado em *Alien Invaders*. No jogo de Musk, os jogadores derrubavam espaçonaves que carregavam bombas de hidrogênio, desviando-se, ao mesmo tempo, de raios mortais.

Musk apresentou seu jogo para a revista *PC and Office Technology*, que se

ofereceu para comprar o *Blastar* por 500 dólares. Ele aprendeu a obter lucro com seu primeiro empreendimento de programação: uma reviravolta importante na sua educação. O *Blastar* ensinou-lhe muitas lições. Em primeiro lugar, percebeu que, após ler um livro e codificar, poderia criar seu próprio videogame. Converter essa aprendizagem em um produto final também trouxe resultados: com apenas 12 anos, Musk ganhou dinheiro proveniente de suas habilidades de programação.

Contudo, a formação de Musk não parou aí. A motivação que o levou a aprender a programar continuou durante a adolescência. Aos 17 anos, mudou-se da África do Sul para o Canadá, onde planejava viver com seu tio-avô em Montreal. Só havia um problema: seu tio tinha se mudado para Minnesota, algo que Musk só veio a saber quando chegou ao Canadá. Contudo, não desistiu. Musk tinha outros parentes no Canadá, então comprou uma passagem de ônibus e começou a procurá-los. Foi necessária uma viagem de ônibus de 3.200 quilômetros para encontrar um primo que lhe ofereceu um lugar para ficar. Ainda adolescente, Musk trabalhou em uma fazenda em Saskatchewan, cortou madeira em Vancouver e limpou caldeiras.

Musk descreveu a limpeza de caldeiras em *Elon Musk: como o CEO bilionário da SpaceX e da Tesla está moldando o nosso futuro*, Ecco Press, edição ilustrada (24 de janeiro de 2017), lançado no Brasil pela Editora Intrínseca: "É preciso colocar um traje de proteção e se esgueirar por um pequeno túnel no qual você quase não cabe", explicou. "Você leva uma pá e retira a areia, a gosma e outros resíduos, que ainda estarão muito quentes, e depois tem de empurrá-los com a pá pelo mesmo túnel por onde entrou. Não há outro jeito. Alguém do outro lado tem de recolher os resíduos com uma pá e colocar em um carrinho de mão. Se você ficar nesse local por mais de 30 minutos, o superaquecimento o mata."

Em 1989, Musk se matriculou na Queen's University, em Ontário. Na faculdade, disse a um amigo: "Se houvesse uma maneira de não precisar comer para trabalhar mais, não comeria. Gostaria que houvesse um meio de obter nutrientes sem precisar parar para fazer uma refeição". A motivação de Musk continuou a impulsioná-lo. Ele construía e vendia

computadores em seu dormitório. “Sempre conseguia criar algo para atender às necessidades dos consumidores, como uma máquina de jogos personalizada ou um simples processador de texto que custavam menos do que eles pagariam em uma loja”, Musk relatou.

Ele também passava horas em jogos como *Civilization* e pensava em seguir carreira na área de games. Após se transferir para a Universidade da Pensilvânia, Musk começou a se interessar por negócios e tecnologia. Embora os jogos tivessem sido sua paixão desde garoto, queria produzir mais impacto sobre o planeta. “Gosto muito de jogos de computador, mas se criasse jogos realmente bons, que efeito isso teria sobre o restante do mundo?”, questionava-se. “Não teria um grande efeito. Mesmo tendo grande paixão por videogames, não poderia adotar isso como carreira”.

Na faculdade, Musk sabia que aprendia rápido. Já estava interessado em energia solar, espaço, internet e carros elétricos. Após obter diplomas de graduação em Economia e Física, mudou-se para a Califórnia para obter Ph.D. em Física de Energia, em Stanford. O Vale do Silício chamou rapidamente sua atenção e ele abandonou o programa de doutorado após apenas dois dias.

Musk lançou, então, a Zip2 Corporation, que criava guias de cidades online. Vendeu a empresa em 1999 por mais de 300 milhões de dólares. Desde então, envolveu-se com muitas outras empresas de sucesso, incluindo PayPal, SpaceX, Tesla Motors e The Boring Company. A motivação que o levou a aprender por conta própria a programar o ajudou a tornar-se um dos empreendedores mais bem-sucedidos de todos os tempos.

CAPÍTULO 18

Próximos passos

Para a maioria dos habitantes do nosso planeta, a revolução digital ainda não começou. Nos próximos dez anos, tudo isso mudará. Façamos todo mundo começar a codificar!

– Eric Schmidt

Bom trabalho! Você percorreu todas as partes técnicas deste livro. Seu trabalho árduo está valendo a pena e você está no caminho certo para tornar-se um engenheiro de software. Gostaria de lhe agradecer por ler meu livro e decidir fazer parte da comunidade de autodidatas. É difícil acreditar no tamanho ao qual nossa comunidade chegou. É maravilhoso ter a chance de encontrar tantas pessoas inspiradoras e não posso esperar para ler sua história de sucesso. Neste último capítulo, abordarei o que você deve fazer para seguir em frente e fornecerei alguns recursos que poderão ajudá-lo.

O que fazer a seguir?

Primeiro, faremos uma pausa para celebrar aonde você chegou para tornar-se um programador autodidata. Além de saber como programar, você também conhece muitos conceitos básicos da Ciência da Computação. Sabe como escrever algoritmos para resolver vários problemas e pode examinar dois algoritmos e decidir rapidamente qual deseja usar. Pode escrever algoritmos recursivos para resolver problemas elegantemente e buscar e ordenar dados de diversas maneiras. Está familiarizado com uma grande variedade de estruturas de dados e não sabe apenas o que são: também sabe quando as usar. De modo geral, você se tornou um programador muito mais bem informado com tantas

ferramentas novas em seu kit de ferramentas.

Além de ter aumentado significativamente seu conhecimento de programação, com um pouco de prática, conseguirá passar em uma entrevista técnica, o que significa que está prestes a obter um emprego como engenheiro de software. Então, o que deve fazer a seguir? Depende da sua experiência profissional em programação. Se já tiver experiência, poderá pular a próxima seção. Se não tiver nenhuma experiência e quiser melhorar as chances de ser contratado, continue lendo.

Subindo a escada dos freelancers

Algo que os novos programadores autodidatas enfrentam é procurar vagas sem nenhuma experiência. Eles enfrentam um problema clássico: precisam de experiência para conseguir um emprego e de um emprego para ganhar experiência. Descobri uma solução para esse problema com a qual meus alunos têm obtido muito sucesso: eu a chamo de subir a escada dos freelancers. Usei o mesmo método para ser contratado como engenheiro de software no eBay, sem nenhuma experiência em uma empresa tradicional.

Antes de conseguir meu emprego no eBay, trabalhei como programador freelancer. O segredo do meu sucesso foi não ter começado tentando conseguir ótimos empregos. Comecei com pequenos trabalhos em uma plataforma chamada Elance (que, agora, chama-se Upwork). Se você não conhece a Upwork, é um site para freelancers. Clientes que procuram trabalho por contrato postam um projeto e os interessados informam seu preço para fazer o serviço. Existem outros sites semelhantes à Upwork, como o `Freelancer.com` e o Fiverr. Recebi cerca de 25 dólares por meu primeiro trabalho, o qual demandou várias horas: não foi um grande preço por hora. Felizmente, o projeto compensou muito mais em termos de experiência. O cliente ficou satisfeito com meu trabalho no projeto e deixou uma avaliação de cinco estrelas. Isso me ajudou a conseguir o projeto seguinte. Novamente, trabalhei muito e ganhei mais uma avaliação de cinco estrelas. Fui conseguindo trabalhos cada vez maiores e acabei fazendo projetos de milhares de dólares.

Quando fui entrevistado no eBay, não tinha nenhuma experiência anterior em programação em uma empresa tradicional, no entanto, durante a entrevista, concentrei-me em falar sobre todos os projetos como freelancer nos quais trabalhei. Meus entrevistadores no eBay acharam que minha experiência como freelancer me tornava um ótimo candidato e acabaram me oferecendo o emprego. Se tivesse tentado fazer a entrevista sem experiência anterior em programação, em uma empresa tradicional, e sem experiência como freelancer, duvido que conseguiria o emprego que alavancou minha carreira.

Se quiser trabalhar como engenheiro de software, mas sem nenhuma experiência profissional, não comece a fazer entrevistas. Inscreva-se em uma plataforma como a Upwork e tente conseguir qualquer trabalho, mesmo que receba apenas 25 dólares. Em seguida, suba a escada de freelancers ganhando avaliações de cinco estrelas e faça propostas para projetos que remunerem mais. Quando tiver acumulado bastante experiência, estará pronto para se candidatar ao emprego de engenheiro de software com o qual tem sonhado em sua empresa favorita.

Como conseguir uma entrevista

Ingressei em meu primeiro emprego como engenheiro de software por meio do LinkedIn, um excelente recurso para quem quer ser contratado. Após obter alguma experiência subindo a escada dos freelancers, recomendo reservar algum tempo para atualizar seu perfil no LinkedIn. Certifique-se de incluir sua experiência como freelancer como seu emprego mais recente e insira engenheiro de software como título do cargo. Considere procurar algumas empresas para as quais trabalhou e pergunte se elas se importariam em confirmar suas habilidades de programação no LinkedIn.

Quando seu perfil estiver atualizado (e seu currículo também), chegará o momento de começar a desenvolver sua networking. Recomendo selecionar de cinco a dez empresas nas quais estiver interessado em trabalhar e entrar em contato com os recrutadores ou membros de outras equipes dessas empresas. Em geral, faltam engenheiros nas empresas e

estas oferecem bônus por indicação a funcionários que apresentem engenheiros. Portanto, se estiver qualificado para o cargo, provavelmente vão recebê-lo.

Você também pode usar um recurso como o Meetup.com para encontrar grupos que se reúnem para formar redes e conhecer novas pessoas ou candidatar-se diretamente a vagas usando sites como o Angel.co ou o Indeed.com.

Como se preparar para uma entrevista técnica

Quando chegar a hora de candidatar-se às vagas, você terá que passar por uma entrevista técnica. É preciso reservar bastante tempo para se preparar. Não existe uma regra fixa, mas recomendo que passe de dois a três meses se preparando. Isso também vai depender do nível de concorrência das vagas nas empresas às quais estiver se candidatando. Se estiver se candidatando a uma vaga em empresas FAANG (Facebook, Amazon, Apple, Netflix ou Google/Alphabet), não será incomum ouvir falar de engenheiros determinados que passaram seis meses ou mais se preparando para as avaliações técnicas. Por outro lado, se estiver se candidatando a uma vaga em uma startup, poderá se preparar em apenas algumas semanas.

Recomendo dedicar pelo menos algumas horas do dia para resolver problemas no LeetCode, um dos meus recursos favoritos de preparação para entrevistas técnicas que apresenta centenas de problemas práticos de estruturas de dados e algoritmos e também suas soluções.

Uma das partes mais difíceis das entrevistas técnicas é o ambiente pouco natural no qual ocorrem. Em geral, quando estamos programando, não temos ninguém olhando por cima de nossos ombros e avaliando-nos. Programadores também não estão acostumados a resolver problemas em curtos períodos. No entanto, essas restrições artificiais são o que você encontrará em uma entrevista técnica. A programação competitiva é a melhor solução que encontrei para me preparar para codificar nesse tipo de ambiente. Programação competitiva é codificar por esporte. Você compete com outros programadores para resolver problemas de Ciência

da Computação. É a melhor maneira de se preparar para uma entrevista técnica porque o prepara para as condições únicas que enfrentará: resolver problemas pressionado pelo tempo. Quando programei competitivamente para me preparar para um conjunto de entrevistas técnicas, tive um desempenho melhor do que em épocas anteriores, em que só me preparei para resolver problemas por conta própria. Você pode tentar usar sites como o Codeforces quando estiver pronto para programar competitivamente.

Após ter usado a programação competitiva para se acostumar a resolver com rapidez problemas técnicos desafiadores, tente fazer algumas entrevistas fictícias com um engenheiro de software, de preferência alguém que já tenha conduzido entrevistas. Se não conseguir encontrar um amigo para ajudá-lo, tente contratar um engenheiro de software em uma plataforma de freelancers, como Upwork ou Codementor. Você pode contratar um engenheiro de software experiente nessas plataformas, por aproximadamente 30 a 60 dólares, e praticar, mesmo que sejam apenas algumas horas, como se estivesse em uma entrevista fictícia. Será um ótimo investimento.

Recursos adicionais

Como sabe, a Ciência da Computação é um tópico extenso. Meu objetivo era escrever um livro que você pudesse ler até o final e abordar assuntos que pudessem ajudá-lo em sua carreira de programador. Infelizmente, isso significa que precisei deixar de fora muitos tópicos. Nesta seção, abordarei brevemente alguns desses itens e indicarei recursos para conhecê-los melhor.

Embora você tenha conhecido as árvores binárias no livro, não abordei outras árvores com detalhes. Você pode querer dedicar mais tempo estudando diferentes tipos de árvores, como as de busca binária, as árvores AVL e as árvores de parsing. Também não abordei todos os algoritmos de ordenação mais comuns. Se estiver se preparando para uma entrevista técnica ou quiser aprender ainda mais sobre algoritmos de ordenação, deverá considerar estudar a ordenação heapsort, a ordenação

por seleção, a ordenação quicksort, a ordenação por contagem e a ordenação radix sort.

Conheça todos esses tópicos e muitos outros em *Introduction to Algorithms*, de Thomas H. Cormen (MIT Press, 2010). No entanto, não se trata de um livro fácil de ler. Agora que conhece os aspectos básicos da Ciência da Computação, provavelmente achará muito mais fácil entender o conteúdo. *Computer Science Illuminated*, de Nell Dale e John Lewis (Jones & Bartlett Learning, 2012), é uma ótima opção para uma aprendizagem mais profunda de assuntos da Ciência da Computação que não sejam estruturas de dados nem algoritmos.

Considerações finais

Obrigado por ter decidido ler meu livro. Espero que tenha apreciado a leitura tanto quanto gostei de escrevê-lo. Se tiver dúvidas ou comentários, fique à vontade para entrar em contato comigo no grupo Self-Taught Programmers, no Facebook, em: `https://facebook.com/groups/selftaughtprogrammers`. Para saber mais sobre a comunidade de autodidatas, inscreva-se em minha newsletter, em `https://selftaught.blog`. Para concluir, mantenha contato comigo nas mídias sociais. Meu handle é `@coryalthoff` no Instagram, Twitter e Facebook. Se quiser deixar uma avaliação na Amazon, serei eternamente grato. Cada avaliação ajuda muito nas vendas, o que permitirá que eu continue criando novos materiais educativos para programadores autodidatas.

Cuide-se!