Algorithms Unlocked

Algorithms Unlocked

Thomas H. Cormen

The MIT Press
Cambridge, Massachusetts   London, England

© 2013 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

This book was set in Times Roman and Mathtime Pro 2 by the author and was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Cormen, Thomas H.
Algorithms Unlocked / Thomas H. Cormen.
p. cm
Includes bibliographical references and index.
ISBN 978-0-262-51880-2 (pbk. : alk. paper)
1. Computer algorithms. I. Title.
QA76.9.A43C685 2013
005.1—dc23

2012036810

10 9 8 7 6 5 4 3 2 1

Contents

Preface *ix*

1 What Are Algorithms and Why Should You Care? ***1***
- Correctness *2*
- Resource usage *4*
- Computer algorithms for non-computer people *6*
- Computer algorithms for computer people *6*
- Further reading *8*

2 How to Describe and Evaluate Computer Algorithms ***10***
- How to describe computer algorithms *10*
- How to characterize running times *17*
- Loop invariants *21*
- Recursion *22*
- Further reading *24*

3 Algorithms for Sorting and Searching ***25***
- Binary search *28*
- Selection sort *32*
- Insertion sort *35*
- Merge sort *40*
- Quicksort *49*
- Recap *57*
- Further reading *59*

4 A Lower Bound for Sorting and How to Beat It ***60***
- Rules for sorting *60*
- The lower bound on comparison sorting *61*
- Beating the lower bound with counting sort *62*
- Radix sort *68*
- Further reading *70*

5 Directed Acyclic Graphs *71*
Directed acyclic graphs *74*
Topological sorting *75*
How to represent a directed graph *78*
Running time of topological sorting *80*
Critical path in a PERT chart *80*
Shortest path in a directed acyclic graph *85*
Further reading *89*

6 Shortest Paths *90*
Dijkstra's algorithm *92*
The Bellman-Ford algorithm *101*
The Floyd-Warshall algorithm *106*
Further reading *114*

7 Algorithms on Strings *115*
Longest common subsequence *115*
Transforming one string to another *121*
String matching *129*
Further reading *136*

8 Foundations of Cryptography *138*
Simple substitution ciphers *139*
Symmetric-key cryptography *140*
Public-key cryptography *144*
The RSA cryptosystem *146*
Hybrid cryptosystems *155*
Computing random numbers *156*
Further reading *157*

9 Data Compression *158*
Huffman codes *160*
Fax machines *167*
LZW compression *168*
Further reading *178*

10 Hard? Problems ***179***
Brown trucks *179*
The classes P and NP and NP-completeness *183*
Decision problems and reductions *185*
A Mother Problem *188*
A sampler of NP-complete problems *190*
General strategies *205*
Perspective *208*
Undecidable problems *210*
Wrap-up *211*
Further reading *212*

Bibliography ***213***
Index ***215***

In loving memory of my mother, Renee Cormen.

Preface

How do computers solve problems? How can your little GPS find, out of the gazillions of possible routes, the fastest way to your destination, and do so in mere seconds? When you purchase something on the Internet, how is your credit-card number protected from someone who intercepts it? The answer to these, and a ton of other questions, is ***algorithms***. I wrote this book to unlock the mystery of algorithms for you.

I coauthored the textbook *Introduction to Algorithms*. It's a wonderful book (of course, I'm biased), but it gets pretty technical in spots.

This book is not *Introduction to Algorithms*. It's not even a textbook. It goes neither broadly nor deeply into the field of computer algorithms, it doesn't prescriptively teach techniques for designing computer algorithms, and it contains nary a problem or exercise for the reader to solve.

So just what is this book? It's a place for you to start, if

- you're interested in how computers solve problems,
- you want to know how to evaluate the quality of these solutions,
- you'd like to see how problems in computing and approaches to solving them relate to the non-computer world,
- you can handle a little mathematics, and
- you have not necessarily ever written a computer program (though it doesn't hurt to have programmed).

Some books about computer algorithms are conceptual, with little technical detail. Some are chock full of technical precision. Some are in between. Each type of book has its place. I'd place this book in the in-between category. Yes, it has some math, and it gets rather precise in some places, but I've avoided getting deep into details (except perhaps toward the end of the book, where I just couldn't control myself).

I think of this book as a bit like an antipasto. Suppose you go to an Italian restaurant and order an antipasto, holding off on deciding whether to order the rest of the meal until you've had the antipasto. It arrives, and you eat it. Maybe you don't like the antipasto, and you decide to not order anything else. Maybe you like it, but it fills you up,

so that you don't need to order anything else. Or maybe you like the antipasto, it does not fill you up, and you're looking forward to the rest of the meal. Thinking of this book as the antipasto, I'm hoping for one of the latter two outcomes: either you read this book, you're satisfied, and you feel no need to delve deeper into the world of algorithms; or you like what you read here so much that you want to learn more. Each chapter ends with a section titled "Further reading," which will guide you to books and articles that go deeper into the topics.

What will you learn from this book?

I can't tell you what you will learn from this book. Here's what I *intend* for you to learn from this book:

- What computer algorithms are, one way to describe them, and how to evaluate them.
- Simple ways to search for information in a computer.
- Methods to rearrange information in a computer so that it's in a prescribed order. (We call this task "sorting.")
- How to solve basic problems that we can model in a computer with a mathematical structure known as a "graph." Among many applications, graphs are great for modeling road networks (which intersections have direct roads to which other intersections, and how long are these roads?), dependencies among tasks (which task must precede which other tasks?), financial relationships (what are the exchange rates among all world currencies?), or interactions among people (who knows whom? who hates whom? which actor appeared in a movie with which other actor?).
- How to solve problems that ask questions about strings of textual characters. Some of these problems have applications in areas such as biology, where the characters represent base molecules and the strings of characters represent DNA structure.
- The basic principles behind cryptography. Even if you have never encrypted a message yourself, your computer probably has (such as when you purchase goods online).
- Fundamental ideas of data compression, going well beyond "f u cn rd ths u cn gt a gd jb n gd pay."

- That some problems are hard to solve on a computer in any reasonable amount of time, or at least that nobody has ever figured out how to do so.

What do you already need to know to understand the material in this book?

As I said earlier, there's some math in this book. If math scares you, then you can try skipping over it, or you can try a less technical book. But I've done my best to make the math accessible.

I don't assume that you've ever written or even read a computer program. If you can follow instructions in outline format, you should be able to understand how I express the steps that, together, form an algorithm. If you get the following joke, you're already part of the way there:

> Did you hear about the computer scientist who got stuck in the shower? He[1] was washing his hair and following the instructions on the shampoo bottle. They read "Lather. Rinse. Repeat."

I've used a fairly informal writing style in this book, hoping that a personal approach will help make the material accessible. Some chapters depend on material in previous chapters, but such dependencies are few. Some chapters start off in a nontechnical manner and become progressively more technical. Even if you find that you're getting in over your head in one chapter, you can probably benefit from reading at least the beginning of the next chapter.

Reporting errors

If you find an error in this book, please let me know about it by sending email to unlocked@mit.edu.

Acknowledgments

Much of the material in this book draws from *Introduction to Algorithms*, and so I owe a great deal to my coauthors on that book, Charles Leiserson, Ron Rivest, and Cliff Stein. You'll find that throughout this

---

[1] Or she. Given the unfortunate gender ratio in computer science, chances are it was he.

book, I shamelessly refer to (read: plug) *Introduction to Algorithms*, known far and wide by the initials CLRS of the four authors. Writing this book on my own makes me realize how much I miss collaborating with Charles, Ron, and Cliff. I also transitively thank everyone we thanked in the preface of CLRS.

I also drew on material from courses that I've taught at Dartmouth, especially Computer Science 1, 5, and 25. Thanks to my students for letting me know, by their insightful questions, which pedagogical approaches worked and, by their stony silence, which did not.

This book came to be at the suggestion of Ada Brunstein, who was our editor at the MIT Press when we prepared the third edition of CLRS. Ada has since moved on, and Jim DeWolf took her place. Originally, this book was slated to be part of the MIT Press "Essential Knowledge" series, but the MIT Press deemed it too technical for the series. (Imagine that—I wrote a book too technical for MIT!) Jim handled this potentially awkward situation smoothly, allowing me to write the book that I wanted to write rather than the book that the MIT Press originally thought I was writing. I also appreciate the support of Ellen Faran and Gita Devi Manaktala of the MIT Press.

Julie Sussman, P.P.A., was our technical copyeditor for the second and third editions of CLRS, and I am once again thrilled to have her copyedit this book. Best. Technical. Copyeditor. Ever. She let me get away with nothing. Here's evidence, in the form of part of an email that Julie sent me about an early draft of Chapter 5:

> Dear Mr. Cormen,
>
> Authorities have apprehended an escaped chapter, which has been found hiding in your book. We are unable to determine what book it has escaped from, but we cannot imagine how it could have been lodging in your book for these many months without your knowledge, so we have no option but to hold you responsible. We hope that you will take on the task of reforming this chapter and will give it an opportunity to become a productive citizen of your book. A report from the arresting officer, Julie Sussman, is appended.

In case you're wondering what "P.P.A." stands for, the first two letters are for "Professional Pain." You can probably guess what the "A" stands for, but I want to point out that Julie takes pride in this title, and rightly so. Thanks a googol, Julie!

I am no cryptographer, and the chapter on principles of cryptography benefited tremendously from comments and suggestions by Ron Rivest, Sean Smith, Rachel Miller, and Huijia Rachel Lin. That chapter has a footnote on baseball signs, and I thank Bob Whalen, the baseball coach at Dartmouth, for patiently explaining to me some of the signing systems in baseball. Ilana Arbisser verified that computational biologists align DNA sequences in the way that I explain in Chapter 7. Jim DeWolf and I went through several iterations of titles for this book, but it was an undergraduate student at Dartmouth, Chander Ramesh, who came up with *Algorithms Unlocked*.

The Dartmouth College Department of Computer Science is an awesome place to work. My colleagues are brilliant and collegial, and our professional staff is second to none. If you're looking for a computer science program at the undergraduate or graduate level, or if you seek a faculty position in computer science, I encourage you to apply to Dartmouth.

Finally, I thank my wife, Nicole Cormen; my parents, Renee and Perry Cormen; my sister, Jane Maslin; and Nicole's parents, Colette and Paul Sage, for their love and support. My father is sure that the figure on page 2 is a 5, not an S.

TOM CORMEN

Hanover, New Hampshire
November 2012

1 What Are Algorithms and Why Should You Care?

Let's start with the question that I'm often asked: "What is an algorithm?"[1]

A broad answer would be "a set of steps to accomplish a task." You have algorithms that you run in your everyday life. You have an algorithm to brush your teeth: open the toothpaste tube, pick up your toothbrush, squeeze toothpaste onto the brush until you have applied enough to the brush, close the tube, put the brush into one quadrant of your mouth, move the brush up and down for N seconds, etc. If you have to commute to a job, you have an algorithm for your commute. And so on.

But this book is about algorithms that run on computers or, more generally, computational devices. Just as algorithms that *you* run affect your everyday life, so do algorithms that run on computers. Do you use your GPS to find a route to travel? It runs what we call a "shortest-path" algorithm to find the route. Do you buy products on the Internet? Then you use (or should be using) a secure website that runs an encryption algorithm. When you buy products on the Internet, are they delivered by a private delivery service? It uses algorithms to assign packages to individual trucks and then to determine the order in which each driver should deliver packages. Algorithms run on computers all over the place—on your laptop, on servers, on your smartphone, on embedded systems (such as in your car, your microwave oven, or climate-control systems)—everywhere!

What distinguishes an algorithm that runs on a computer from an algorithm that you run? You might be able to tolerate it when an algorithm is imprecisely described, but a computer cannot. For example, if you drive to work, your drive-to-work algorithm might say "if traffic is bad, take an alternate route." Although you might know what you mean by "bad traffic," a computer does not.

So a computer algorithm is a set of steps to accomplish a task that is described precisely enough that a computer can run it. If you have

[1] Or, as a fellow with whom I used to play hockey would ask, "What's a nalgorithm?"

done even a little computer programming in Java, C, C++, Python, Fortran, Matlab, or the like, then you have some idea of what that level of precision means. If you have never written a computer program, then perhaps you will get a feel for that level of precision from seeing how I describe algorithms in this book.

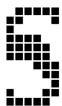
Let's go to the next question: "What do we want from a computer algorithm?"

Computer algorithms solve computational problems. We want two things from a computer algorithm: given an input to a problem, it should always produce a correct solution to the problem, and it should use computational resources efficiently while doing so. Let's examine these two desiderata in turn.

Correctness

What does it mean to produce a correct solution to a problem? We can usually specify precisely what a correct solution would entail. For example, if your GPS produces a correct solution to finding the best route to travel, it might be the route, out of all possible routes from where you are to your desired destination, that will get you there soonest. Or perhaps the route that has the shortest possible distance. Or the route that will get you there soonest but also avoids tolls. Of course, the information that your GPS uses to determine a route might not match reality. Unless your GPS can access real-time traffic information, it might assume that the time to traverse a road equals the road's distance divided by the road's speed limit. If the road is congested, however, the GPS might give you bad advice if you're looking for the fastest route. We can still say that the routing algorithm that the GPS runs is correct, however, even if the input to the algorithm is not; for the input given to the routing algorithm, the algorithm produces the fastest route.

Now, for some problems, it might be difficult or even impossible to say whether an algorithm produces a correct solution. Take optical character recognition for example. Is this 11×6 pixel image a 5 or an S?



Some people might call it a 5, whereas others might call it an S, so how could we declare that a computer's decision is either correct or incor-

rect? We won't. In this book, we will focus on computer algorithms that have knowable solutions.

Sometimes, however, we can accept that a computer algorithm might produce an incorrect answer, as long as we can control how often it does so. Encryption provides a good example. The commonly used RSA cryptosystem relies on determining whether large numbers—really large, as in hundreds of digits long—are prime. If you have ever written a computer program, you could probably write one that determines whether a number n is prime. It would test all candidate divisors from 2 through $n - 1$, and if any of these candidates is indeed a divisor of n, then n is composite. If no number between 2 and $n - 1$ is a divisor of n, then n is prime. But if n is hundreds of digits long, that's a lot of candidate divisors, more than even a really fast computer could check in any reasonable amount of time. Of course, you could make some optimizations, such as eliminating all even candidates once you find that 2 is not a divisor, or stopping once you get to $\sqrt{n}$ (since if d is greater than $\sqrt{n}$ and d is a divisor of n, then n/d is less than $\sqrt{n}$ and is also a divisor of n; therefore, if n has a divisor, you will find it by the time you get to $\sqrt{n}$). If n is hundreds of digits long, then although $\sqrt{n}$ has only about half as many digits as n does, it's still a really large number. The good news is that we know of an algorithm that tests quickly whether a number is prime. The bad news is that it can make errors. In particular, if it declares that n is composite, then n is definitely composite, but if it declares that n is prime, then there's a chance that n is actually composite. But the bad news is not all that bad: we can control the error rate to be really low, such as one error in every 2^{50} times. That's rare enough—one error in about every million billion times—for most of us to be comfortable with using this method to determine whether a number is prime for RSA.

Correctness is a tricky issue with another class of algorithms, called approximation algorithms. Approximation algorithms apply to optimization problems, in which we want to find the best solution according to some quantitative measure. Finding the fastest route, as a GPS does, is one example, where the quantitative measure is travel time. For some problems, we have no algorithm that finds an optimal solution in any reasonable amount of time, but we know of an approximation algorithm that, in a reasonable amount of time, can find a solution that is almost optimal. By "almost optimal," we typically mean that the quantitative measure of the solution found by the approximation algorithm is within

some known factor of the optimal solution's quantitative measure. As long as we specify what the desired factor is, we can say that a correct solution from an approximation algorithm is any solution that is within that factor of the optimal solution.

Resource usage

What does it mean for an algorithm to use computational resources efficiently? We alluded to one measure of efficiency in the discussion of approximation algorithms: time. An algorithm that gives a correct solution but takes a long time to produce that correct solution might be of little or no value. If your GPS took an hour to determine which driving route it recommends, would you even bother to turn it on? Indeed, time is the primary measure of efficiency that we use to evaluate an algorithm, once we have shown that the algorithm gives a correct solution. But it is not the only measure. We might be concerned with how much computer memory the algorithm requires (its "memory footprint"), since an algorithm has to run within the available memory. Other possible resources that an algorithm might use: network communication, random bits (because algorithms that make random choices need a source of random numbers), or disk operations (for algorithms that are designed to work with disk-resident data).

In this book, as in most of the algorithms literature, we will focus on just one resource: time. How do we judge the time required by an algorithm? Unlike correctness, which does not depend on the particular computer that the algorithm runs on, the actual running time of an algorithm depends on several factors extrinsic to the algorithm itself: the speed of the computer, the programming language in which the algorithm is implemented, the compiler or interpreter that translates the program into code that runs on the computer, the skill of the programmer who writes the program, and other activity taking place on the computer concurrently with the running program. And that all assumes that the algorithm runs on just one computer with all its data in memory.

If we were to evaluate the speed of an algorithm by implementing it in a real programming language, running it on a particular computer with a given input, and measuring the time the algorithm takes, we would know nothing about how fast the algorithm ran on an input of a different size, or possibly even on a different input of the same size. And if we wanted to compare the relative speed of the algorithm with some other algorithm for the same problem, we would have to implement them both

and run both of them on various inputs of various sizes. How, then, can we evaluate an algorithm's speed?

The answer is that we do so by a combination of two ideas. First, we determine how long the algorithm takes as a function of the size of its input. In our route-finding example, the input would be some representation of a roadmap, and its size would depend on the number of intersections and the number of roads connecting intersections in the map. (The physical size of the road network would not matter, since we can characterize all distances by numbers and all numbers occupy the same size in the input; the length of a road has no bearing on the input size.) In a simpler example, searching a given list of items to determine whether a particular item is present in the list, the size of the input would be the number of items in the list.

Second, we focus on how fast the function that characterizes the running time grows with the input size—the *rate of growth* of the running time. In Chapter 2, we'll see the notations that we use to characterize an algorithm's running time, but what's most interesting about our approach is that we look at only the dominant term in the running time, and we don't consider coefficients. That is, we focus on the ***order of growth*** of the running time. For example, suppose we could determine that a specific implementation of a particular algorithm to search a list of n items takes $50n + 125$ machine cycles. The $50n$ term dominates the 125 term once n gets large enough, starting at $n \geq 3$ and increasing in dominance for even larger list sizes. Thus, we don't consider the low-order term 125 when we describe the running time of this hypothetical algorithm. What might surprise you is that we also drop the coefficient 50, thereby characterizing the running time as growing linearly with the input size n. As another example, if an algorithm took $20n^3 + 100n^2 + 300n + 200$ machine cycles, we would say that its running time grows as n^3. Again, the low-order terms—$100n^2$, $300n$, and 200—become less and less significant as the input size n increases.

In practice, the coefficients that we ignore do matter. But they depend so heavily on the extrinsic factors that it's entirely possible that if we were comparing two algorithms, A and B, that have the same order of growth and are run on the same input, then A might run faster than B with a particular combination of machine, programming language, compiler/interpreter, and programmer, while B runs faster than A with some other combination. Of course, if algorithms A and B both produce correct solutions and A always runs twice as fast as B, then, all other things

being equal, we prefer to always run A instead of B. From the point of view of comparing algorithms in the abstract, however, we focus on the order of growth, unadorned by coefficients or low-order terms.

The final question that we ask in this chapter: "Why should I care about computer algorithms?" The answer to this question depends on who you are.

Computer algorithms for non-computer people

Even if you don't consider yourself a computer insider, computer algorithms matter to you. After all, unless you're on a wilderness expedition without a GPS, you probably use them every day. Did you search for something on the Internet today? The search engine you used—whether it was Google, Bing, or any other search engine—employed sophisticated algorithms to search the Web and to decide in which order to present its results. Did you drive your car today? Unless you're driving a classic vehicle, its on-board computers made millions of decisions, all based on algorithms, during your trip. I could go on and on.

As an end user of algorithms, you owe it to yourself to learn a little bit about how we design, characterize, and evaluate algorithms. I assume that you have at least a mild interest, since you have picked up this book and read this far. Good for you! Let's see if we can get you up to speed so that you can hold your own at your next cocktail party in which the subject of algorithms comes up.[2]

Computer algorithms for computer people

If you're a computer person, then you had better care about computer algorithms! Not only are they at the heart of, well, everything that goes on inside your computer, but algorithms are just as much a technology as everything else that goes on inside your computer. You can pay a premium for a computer with the latest and greatest processor, but you

---

[2] Yes, I realize that unless you live in Silicon Valley, the subject of algorithms rarely comes up at cocktail parties that you attend, but for some reason, we computer science professors think it important that our students not embarrass us at cocktail parties with their lack of knowledge in particular areas of computer science.

need to run implementations of good algorithms on that computer in order for your money to be well spent.

Here's an example that illustrates how algorithms are indeed a technology. In Chapter 3, we are going to see a few different algorithms that sort a list of n values into ascending order. Some of these algorithms will have running times that grow like n^2, but some will have running times that grow like only $n \lg n$. What is $\lg n$? It is the base-2 logarithm of n, or $\log_2 n$. Computer scientists use base-2 logarithms so frequently that just like mathematicians and scientists who use the shorthand $\ln n$ for the natural logarithm—$\log_e n$—computer scientists use their own shorthand for base-2 logarithms. Now, because the function $\lg n$ is the inverse of an exponential function, it grows very slowly with n. If $n = 2^x$, then $x = \lg n$. For example, $2^{10} = 1024$, and therefore $\lg 1024$ is only 10; similarly $2^{20} = 1{,}048{,}576$ and so $\lg 1{,}048{,}576$ is only 20; and $2^{30} = 1{,}073{,}741{,}824$ means that $\lg 1{,}073{,}741{,}824$ is only 30. So a growth of $n \lg n$ vs. n^2 trades a factor of n for a factor of only $\lg n$, and that's a deal you should take any day.

Let's make this example more concrete by pitting a faster computer (computer A) running a sorting algorithm whose running time on n values grows like n^2 against a slower computer (computer B) running a sorting algorithm whose running time grows like $n \lg n$. They each must sort an array of 10 million numbers. (Although 10 million numbers might seem like a lot, if the numbers are eight-byte integers, then the input occupies about 80 megabytes, which fits in the memory of even an inexpensive laptop computer many times over.) Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose further that just an average programmer writes for computer B, using a high-level language with an inefficient compiler, with the resulting code taking $50n \lg n$ instructions. To sort 10 million numbers, computer A takes

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ seconds},$$

which is more than 5.5 hours, while computer B takes

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds} ,$$

which is under 20 minutes. By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs more than 17 times faster than computer A! The advantage of the $n \lg n$ algorithm is even more pronounced when we sort 100 million numbers: where the n^2 algorithm on computer A takes more than 23 days, the $n \lg n$ algorithm on computer B takes under four hours. In general, as the problem size increases, so does the relative advantage of the $n \lg n$ algorithm.

Even with the impressive advances we continually see in computer hardware, total system performance depends on choosing efficient algorithms as much as on choosing fast hardware or efficient operating systems. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well.

Further reading

In my highly biased opinion, the clearest and most useful source on computer algorithms is *Introduction to Algorithms* [CLRS09] by four devilishly handsome fellows. The book is commonly called "CLRS," after the initials of the authors. I've drawn on it for much of the material in this book. It's far more complete than this book, but it assumes that you've done at least a little computer programming, and it pulls no punches on the math. If you find that you're comfortable with the level of mathematics in this book, and you're ready to go deeper into the subject, then you can't do better than CLRS. (In my humble opinion, of course.)

John MacCormick's book *Nine Algorithms That Changed the Future* [Mac12] describes several algorithms and related aspects of computing that affect our everyday lives. MacCormick's treatment is less technical than this book. If you find that my approach in this book is too mathematical, then I recommend that you try reading MacCormick's book. You should be able to follow much of it even if you have a meager mathematical background.

In the unlikely event that you find CLRS too watered down, you can try Donald Knuth's multi-volume set *The Art of Computer Programming* [Knu97, Knu98a, Knu98b, Knu11]. Although the title of the series makes it sound like it might focus on details of writing code, these books

contain brilliant, in-depth analyses of algorithms. Be warned, however: the material in *TAOCP* is intense. By the way, if you're wondering where the word "algorithm" comes from, Knuth says that it derives from the name "al-Khowârizmî," a ninth-century Persian mathematician.

In addition to CLRS, several other excellent texts on computer algorithms have been published over the years. The chapter notes for Chapter 1 of CLRS list many such texts. Rather than replicate that list here, I refer you to CLRS.

2 How to Describe and Evaluate Computer Algorithms

In the previous chapter, you got a taste of how we couch the running time of a computer algorithm: by focusing on the running time as a function of the input size, and specifically on the order of growth of the running time. In this chapter, we'll back up a bit and see how we describe computer algorithms. Then we'll see the notations that we use to characterize the running times of algorithms. We'll wrap up this chapter by examining some techniques that we use to design and understand algorithms.

How to describe computer algorithms

We always have the option of describing a computer algorithm as a runnable program in a commonly used programming language, such as Java, C, C++, Python, or Fortran. Indeed, several algorithms textbooks do just that. The problem with using real programming languages to specify algorithms is that you can get bogged down in the details of the language, obscuring the ideas behind the algorithms. Another approach, which we took in *Introduction to Algorithms*, uses "pseudocode," which looks like a mashup of various programming languages with English mixed in. If you've ever used a real programming language, you can figure out pseudocode easily. But if you have not ever programmed, then pseudocode might seem a bit mysterious.

The approach I'm taking in this book is that I'm not trying to describe algorithms to software or hardware, but to "wetware": the gray matter between your ears. I am also going to assume that you have never written a computer program, and so I won't express algorithms using any real programming language or even pseudocode. Instead, I'll describe them in English, using analogies to real-world scenarios whenever I can. In order to indicate what happens when (what we call "flow of control" in programming), I'll use lists and lists within lists. If you want to implement an algorithm in a real programming language, I'll give you credit for being able to translate my description into runnable code.

Although I will try to keep descriptions as nontechnical as possible, this book is about algorithms for computers, and so I will have to use computing terminology. For example, computer programs contain ***procedures*** (also known as functions or methods in real programming languages), which specify how to do something. In order to actually get the procedure to do what it's supposed to do, we ***call*** it. When we call a procedure, we supply it with inputs (usually at least one, but some procedures take no inputs). We specify the inputs as ***parameters*** within parentheses after the name of the procedure. For example, to compute the square root of a number, we might define a procedure SQUARE-ROOT(x); here, the input to the procedure is referred to by the parameter x. The call of a procedure may or may not produce output, depending on how we specified the procedure. If the procedure produces output, we usually consider the output to be something passed back to its caller. In computing parlance we say that the procedure ***returns*** a value.

Many programs and algorithms work with arrays of data. An ***array*** aggregates data of the same type into one entity. You can think of an array as being like a table, where given the ***index*** of an ***entry***, we can talk about the array ***element*** at that index. For example, here is a table of the first five U.S. presidents:

Index	President
1	George Washington
2	John Adams
3	Thomas Jefferson
4	James Madison
5	James Monroe

For example, the element at index 4 in this table is James Madison. We think of this table not as five separate entities, but as one table with five entries. An array is similar. The indices into an array are consecutive numbers that can start anywhere, but we will usually start them at 1.[1] Given the name of an array and an index into the array, we combine them with square brackets to indicate a particular array element. For example, we denote the ith element of an array A by $A[i]$.

[1] If you program in Java, C, or C++, you are used to arrays that start at 0. Starting arrays at 0 is nice for computers, but for wetware it's often more intuitive to start at 1.

Arrays in computers have one other important characteristic: it takes equally long to access any element of an array. Once you give the computer an index i into an array, it can access the ith element as quickly as it can access the first element, regardless of the value of i.

Let's see our first algorithm: searching an array for a particular value. That is, we are given an array, and we want to know which entry in the array, if any, holds a given value. To see how we can search an array, let's think of the array as a long bookshelf full of books, and suppose that you want to know where on the shelf you can find a book by Jonathan Swift. Now, the books on the shelf might be organized in some way, perhaps alphabetically by author, alphabetically by title, or, in a library, by call number. Or perhaps the bookshelf is like my bookshelf at home, where I have not organized my books in any particular way.

If you couldn't assume that the books were organized on the shelf, how would you find a book by Jonathan Swift? Here's the algorithm I would follow. I would start at the left end of the shelf and look at the leftmost book. If it's by Swift, I have located the book. Otherwise, I would look at the next book to the right, and if that book is by Swift, I have located the book. If not, I would keep going to the right, examining book after book, until either I find a book by Swift or I run off the right-hand end of the shelf, in which case I can conclude that the bookshelf does not contain any book by Jonathan Swift. (In Chapter 3, we'll see how to search for a book when the books *are* organized on the shelf.)

Here is how we can describe this searching problem in terms of computing. Let's think of the books on the bookshelf as an array of books. The leftmost book is in position 1, the next book to its right is in position 2, and so on. If we have n books on the shelf, then the rightmost book is in position n. We want to find the position number on the shelf of any book by Jonathan Swift.

As a general computing problem, we are given an array A (the entire shelf full of books to search through) of n elements (the individual books), and we want to find whether a value x (a book by Jonathan Swift) is present in the array A. If it is, then we want to determine an index i such that $A[i] = x$ (the ith position on the shelf contains a book by Jonathan Swift). We also need some way to indicate that array A does not contain x (the bookshelf contains no books by Jonathan Swift). We do not assume that x appears at most once in the array (perhaps you have multiple copies of some book), and so if x is present in array A, it may appear multiple times. All we want from a searching algorithm

is *any* index at which we'll find x in the array. We'll assume that the indices of this array start at 1, so that its elements are $A[1]$ through $A[n]$.

If we search for a book by Jonathan Swift by starting at the left end of the shelf, checking book by book as we move to the right, we call that technique ***linear search***. In terms of an array in a computer, we start at the beginning of the array, examine each array element in turn ($A[1]$, then $A[2]$, then $A[3]$, and so on, up through $A[n]$) and record where we find x, if we find it at all.

The following procedure, LINEAR-SEARCH, takes three parameters, which we separate by commas in the specification.

Procedure LINEAR-SEARCH(A, n, x)

Inputs:

- A: an array.
- n: the number of elements in A to search through.
- x: the value being searched for.

Output: Either an index i for which $A[i] = x$, or the special value NOT-FOUND, which could be any invalid index into the array, such as 0 or any negative integer.

1. Set *answer* to NOT-FOUND.
2. For each index i, going from 1 to n, in order:
   A. If $A[i] = x$, then set *answer* to the value of i.
3. Return the value of *answer* as the output.

In addition to the parameters A, n, and x, the LINEAR-SEARCH procedure uses a ***variable*** named *answer*. The procedure ***assigns*** an initial value of NOT-FOUND to *answer* in step 1. Step 2 checks each array entry $A[1]$ through $A[n]$ to see if the entry contains the value x. Whenever entry $A[i]$ equals x, step 2A assigns the current value of i to *answer*. If x appears in the array, then the output value returned in step 3 is the last index in which x appeared. If x does not appear in the array, then the equality test in step 2A never evaluates to true, and the output value returned is NOT-FOUND, as assigned to *answer* back in step 1.

Before we continue discussing linear search, a word about how to specify repeated actions, such as in step 2. It is quite common in algorithms to perform some action for a variable taking values in some range. When we perform repeated actions, we call that a ***loop***, and we call each time through the loop an ***iteration*** of the loop. For the loop of

step 2, I wrote "For each index i, going from 1 to n, in order." Instead, from now on, I'll write "For $i = 1$ to n," which is shorter, yet conveys the same structure. Notice that when I write a loop in this way, we have to give the ***loop variable*** (here, i) an initial value (here, 1), and in each iteration of the loop, we have to test the current value of the loop variable against a limit (here, n). If the current value of the loop variable is less than or equal to the limit, then we do everything in the loop's ***body*** (here, step 2A). After an iteration executes the loop body, we ***increment*** the loop variable—adding 1 to it—and go back and compare the loop variable, now with its new value, with the limit. We repeatedly test the loop variable against the limit, execute the loop body, and increment the loop variable, until the loop variable exceeds the limit. Execution then continues from the step immediately following the loop body (here, step 3). A loop of the form "For $i = 1$ to n" performs n iterations and $n + 1$ tests against the limit (because the loop variable exceeds the limit in the $(n + 1)$st test).

I hope that you find it obvious that the LINEAR-SEARCH procedure always returns a correct answer. You might have noticed, however, that this procedure is inefficient: it continues to search the array even after it has found an index i for which $A[i] = x$. Normally, you wouldn't continue searching for a book once you have found it on your bookshelf, would you? Instead, we can design our linear search procedure to stop searching once it finds the value x in the array. We assume that when we say to return a value, the procedure immediately returns the value to its caller, which then takes control.

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

Inputs and Output: Same as LINEAR-SEARCH.

1. For $i = 1$ to n:
   A. If $A[i] = x$, then return the value of i as the output.
2. Return NOT-FOUND as the output.

Believe it or not, we can make linear search even more efficient. Observe that each time through the loop of step 1, the BETTER-LINEAR-SEARCH procedure makes two tests: a test in step 1 to determine whether $i \leq n$ (and if so, perform another iteration of the loop) and the equality test in step 1A. In terms of searching a bookshelf, these tests correspond to you having to check two things for each book: have you

gone past the end of the shelf and, if not, is the next book by Jonathan Swift? Of course, you don't incur much of a penalty for going past the end of the shelf (unless you keep your face really close to the books as you examine them, there's a wall at the end of the shelf, and you smack your face into the wall), but in a computer program it's usually very bad to try to access array elements past the end of the array. Your program could crash, or it could corrupt data.

You can make it so that you have to perform only one check for every book you examine. What if you knew for sure that your bookshelf contained a book by Jonathan Swift? Then you'd be assured of finding it, and so you'd never have to check for running off the end of the shelf. You could just check each book in turn to see whether it's by Swift.

But perhaps you lent out all your books by Jonathan Swift, or maybe you thought you had books by him but you never did, so you might not be sure that your bookshelf contains any books by him. Here's what you can do. Take an empty box the size of a book and write on its narrow side (where the spine of a book would be) "*Gulliver's Travels* by Jonathan Swift." Replace the rightmost book with this box. Then, as you search from left to right along the bookshelf, you need to check only whether you're looking at something that is by Swift; you can forget about having to check whether you're going past the end of the bookshelf because you *know* that you'll find something by Swift. The only question is whether you really found a book by Swift, or did you find the empty box that you had labeled as though it were by him? If you found the empty box, then you didn't really have a book by Swift. That's easy to check, however, and you need to do that only once, at the end of your search, rather than once for every book on the shelf.

There's one more detail you have to be aware of: what if the only book by Jonathan Swift that you had on your bookshelf was the rightmost book? If you replace it by the empty box, your search will terminate at the empty box, and you might conclude that you didn't have the book. So you have to perform one more check for that possibility, but it's just one check, rather than one check for every book on the shelf.

In terms of a computer algorithm, we'll put the value x that we're searching for into the last position, $A[n]$, after saving the contents of $A[n]$ into another variable. Once we find x, we test to see whether we *really* found it. We call the value that we put into the array a ***sentinel***, but you can think of it as the empty box.

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

Inputs and Output: Same as LINEAR-SEARCH.

1. Save $A[n]$ into *last* and then put x into $A[n]$.
2. Set i to 1.
3. While $A[i] \neq x$, do the following:
   A. Increment i.
4. Restore $A[n]$ from *last*.
5. If $i < n$ or $A[n] = x$, then return the value of i as the output.
6. Otherwise, return NOT-FOUND as the output.

Step 3 is a loop, but not one that counts through some loop variable. Instead, the loop iterates as long as a condition holds; here, the condition is that $A[i] \neq x$. The way to interpret such a loop is to perform the test (here, $A[i] \neq x$), and if the test is true, then do everything in the loop's body (here, step 3A, which increments i). Then go back and perform the test, and if the test is true, execute the body. Keep going, performing the test then executing the body, until the test comes up false. Then continue from the next step after the loop body (here, continue from step 4).

The SENTINEL-LINEAR-SEARCH procedure is a bit more complicated than the first two linear search procedures. Because it places x into $A[n]$ in step 1, we are guaranteed that $A[i]$ will equal x for some test in step 3. Once that happens, we drop out of the step-3 loop, and the index i won't change thereafter. Before we do anything else, step 4 restores the original value in $A[n]$. (My mother taught me to put things back when I was done with them.) Then we have to determine whether we really found x in the array. Because we put x into the last element, $A[n]$, we know that if we found x in $A[i]$ where $i < n$, then we really did find x and we want to return the index i. What if we found x in $A[n]$? That means we didn't find x before $A[n]$, and so we need to determine whether $A[n]$ equals x. If it does, then we want to return the index n, which equals i at this point, but if it does not, we want to return NOT-FOUND. Step 5 does these tests and returns the correct index if x was originally in the array. If x was found only because step 1 put it into the array, then step 6 returns NOT-FOUND. Although SENTINEL-LINEAR-SEARCH has to perform two tests after its loop terminates, it performs only one test in each loop iteration, thereby making it more efficient than either LINEAR-SEARCH or BETTER-LINEAR-SEARCH.

How to characterize running times

Let's return to the LINEAR-SEARCH procedure from page 13 and understand its running time. Recall that we want to characterize the running time as a function of the input size. Here, our input is an array A of n elements, along with the number n and the value x that we're searching for. The sizes of n and x are insignificant as the array gets large—after all, n is just a single integer and x is only as large as one of the n array elements—and so we'll say that the input size is n, the number of elements in A.

We have to make some simple assumptions about how long things take. We will assume that each individual operation—whether it's an arithmetic operation (such as addition, subtraction, multiplication, or division), a comparison, assigning to a variable, indexing into an array, or calling or returning from a procedure—takes some fixed amount of time that is independent of the input size.[2] The time might vary from operation to operation, so that division might take longer than addition, but when a step comprises just simple operations, each individual execution of that step takes some constant amount of time. Because the operations executed differ from step to step, and because of the extrinsic factors listed back on page 4, the time to execute a step might vary from step to step. Let's say that each execution of step i takes t_i time, where t_i is some constant that does not depend on n.

Of course, we have to take into account that some steps execute multiple times. Steps 1 and 3 execute just once, but what about step 2? We have to test i against n a total of $n + 1$ times: n times in which $i \le n$, and once when i equals $n + 1$ so that we drop out of the loop. Step 2A executes exactly n times, once for each value of i from 1 to n. We don't know in advance how many times we set *answer* to the value of i; it could be anywhere from 0 times (if x is not present in the array) to n times (if every value in the array equals x). If we're going to be precise in our accounting—and we won't normally be this precise—we need to

---

[2] If you know a bit about actual computer architecture, you might know that the time to access a given variable or array element is not necessarily fixed, for it could depend on whether the variable or array element is in the cache, in main memory, or out on disk in a virtual-memory system. Some sophisticated models of computers take these issues into account, but it's often good enough to just assume that all variables and array entries are in main memory and that they all take the same amount of time to access.

recognize that step 2 does two different things that execute a different number of times: the test of i against n happens $n + 1$ times, but incrementing i happens only n times. Let's separate the time for line 2 into t'_2 for the test and t''_2 for incrementing. Similarly, we'll separate the time for step 2A into t'_{2A} for testing whether $A[i] = x$ and t''_{2A} for setting *answer* to i. Therefore, the running time of LINEAR-SEARCH is somewhere between

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

and

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

Now we rewrite these bounds, collecting terms that multiply by n together, and collecting the rest of the terms, and we see that the running time is somewhere between the ***lower bound***

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

and the ***upper bound***

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

Notice that both of these bounds are of the form $c \cdot n + d$, where c and d are constants that do not depend on n. That is, they are both *linear functions* of n. The running time of LINEAR-SEARCH is bounded from below by a linear function of n, and it is bounded from above by a linear function of n.

We use a special notation to indicate that a running time is bounded from above by some linear function of n and from below by some (possibly different) linear function of n. We write that the running time is $\Theta(n)$. That's the Greek letter theta, and we say "theta of n" or just "theta n." As promised in Chapter 1, this notation discards the low-order term $(t_1 + t'_2 + t_3)$ and the coefficients of n ($t'_2 + t''_2 + t'_{2A}$ for the lower bound and $t'_2 + t''_2 + t'_{2A} + t''_{2A}$ for the upper bound). Although we lose precision by characterizing the running time as $\Theta(n)$, we gain the advantages of highlighting the order of growth of the running time and suppressing tedious detail.

This Θ-notation applies to functions in general, not just those that describe running times of algorithms, and it applies to functions other than linear ones. The idea is that if we have two functions, $f(n)$ and $g(n)$, we say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is within a constant factor of $g(n)$

for sufficiently large n. So we can say that the running time of LINEAR-SEARCH is within a constant factor of n once n gets large enough.

There's an intimidating technical definition of Θ-notation, but fortunately we rarely have to resort to it in order to use Θ-notation. We simply focus on the dominant term, dropping low-order terms and constant factors. For example, the function $n^2/4 + 100n + 50$ is $\Theta(n^2)$; here we drop the low-order terms $100n$ and 50, and we drop the constant factor $1/4$. Although the low-order terms will dominate $n^2/4$ for small values of n, once n goes above 400, the $n^2/4$ term exceeds $100n + 50$. When $n = 1000$, the dominant term $n^2/4$ equals 250,000, while the low-order terms $100n + 50$ amount to only 100,050; for $n = 2000$ the difference becomes 1,000,000 vs. 200,050. In the world of algorithms, we abuse notation a little bit and write $f(n) = \Theta(g(n))$, so that we can write $n^2/4 + 100n + 50 = \Theta(n^2)$.

Now let's look at the running time of BETTER-LINEAR-SEARCH from page 14. This one is a little trickier than LINEAR-SEARCH because we don't know in advance how many times the loop will iterate. If $A[1]$ equals x, then it will iterate just once. If x is not present in the array, then the loop will iterate all n times, which is the maximum possible. Each loop iteration takes some constant amount of time, and so we can say that *in the worst case*, BETTER-LINEAR-SEARCH takes $\Theta(n)$ time to search an array of n elements. Why "worst case"? Because we want algorithms to have low running times, the worst case occurs when an algorithm takes the maximum time over any possible input.

In the best case, when $A[1]$ equals x, BETTER-LINEAR-SEARCH takes just a constant amount of time: it sets i to 1, checks that $i \leq n$, the test $A[i] = x$ comes up true, and the procedure returns the value of i, which is 1. This amount of time does not depend on n. We write that the *best-case running time* of BETTER-LINEAR-SEARCH is $\Theta(1)$, because in the best case, its running time is within a constant factor of 1. In other words, the best-case running time is a constant that does not depend on n.

So we see that we cannot use Θ-notation for a blanket statement that covers all cases of the running time of BETTER-LINEAR-SEARCH. We cannot say that the running time is always $\Theta(n)$, because in the best case it's $\Theta(1)$. And we cannot say that the running time is always $\Theta(1)$, because in the worst case it's $\Theta(n)$. We can say that a linear function of n is an *upper bound* in all cases, however, and we have a notation for that: $O(n)$. When we speak this notation, we say "big-oh of n" or just

"oh of n." A function $f(n)$ is $O(g(n))$ if, once n becomes sufficiently large, $f(n)$ is bounded from above by some constant times $g(n)$. Again, we abuse notation a little and write $f(n) = O(g(n))$. For BETTER-LINEAR-SEARCH, we can make the blanket statement that its running time in all cases is $O(n)$; although the running time might be better than a linear function of n, it's never worse.

We use O-notation to indicate that a running time is never *worse* than a constant times some function of n, but how about indicating that a running time is never *better* than a constant times some function of n? That's a lower bound, and we use Ω-notation, which mirrors O-notation: a function $f(n)$ is $\Omega(g(n))$ if, once n becomes sufficiently large, $f(n)$ is bounded from below by some constant times $g(n)$. We say that "$f(n)$ is big-omega of $g(n)$" or just "$f(n)$ is omega of $g(n)$," and we can write $f(n) = \Omega(g(n))$. Since O-notation gives an upper bound, Ω-notation gives a lower bound, and Θ-notation gives both upper and lower bounds, we can conclude that a function $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$.

We can make a blanket statement about a lower bound for the running time of BETTER-LINEAR-SEARCH: in all cases it's $\Omega(1)$. Of course, that's a pathetically weak statement, since we'd expect any algorithm on any input to take at least constant time. We won't use Ω-notation much, but it will occasionally come in handy.

The catch-all term for Θ-notation, O-notation, and Ω-notation is ***asymptotic notation***. That's because these notations capture the growth of a function as its argument asymptotically approaches infinity. All of these asymptotic notations give us the luxury of dropping low-order terms and constant factors so that we can ignore tedious details and focus on what's important: how the function grows with n.

Now let's turn to SENTINEL-LINEAR-SEARCH from page 16. Just like BETTER-LINEAR-SEARCH, each iteration of its loop takes a constant amount of time, and there may be anywhere from 1 to n iterations. The key difference between SENTINEL-LINEAR-SEARCH and BETTER-LINEAR-SEARCH is that the time per iteration of SENTINEL-LINEAR-SEARCH is less than the time per iteration of BETTER-LINEAR-SEARCH. Both take a linear amount of time in the worst case, but the constant factor for SENTINEL-LINEAR-SEARCH is better. Although we'd expect SENTINEL-LINEAR-SEARCH to be faster in practice, it would be by only a constant factor. When we express the running times of BETTER-LINEAR-SEARCH and SENTINEL-LINEAR-SEARCH

using asymptotic notation, they are equivalent: $\Theta(n)$ in the worst case, $\Theta(1)$ in the best case, and $O(n)$ in all cases.

Loop invariants

For our three flavors of linear search, it was easy to see that each one gives a correct answer. Sometimes it's a bit harder. There's a wide range of techniques, more than I can cover here.

One common method of showing correctness uses a ***loop invariant***: an assertion that we demonstrate to be true each time we start a loop iteration. For a loop invariant to help us argue correctness, we have to show three things about it:

Initialization: It is true before the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: The loop terminates, and when it does, the loop invariant, along with the reason that the loop terminated, gives us a useful property.

As an example, here's a loop invariant for BETTER-LINEAR-SEARCH:

> At the start of each iteration of step 1, if x is present in the array A, then it is present in the ***subarray*** (a contiguous portion of an array) from $A[i]$ through $A[n]$.

We don't even need this loop invariant to show that if the procedure returns an index other than NOT-FOUND, then the index returned is correct: the only way that the procedure can return an index i in step 1A is because x equals $A[i]$. Instead, we will use this loop invariant to show that if the procedure returns NOT-FOUND in step 2, then x is not anywhere in the array:

Initialization: Initially, $i = 1$ so that the subarray in the loop invariant is $A[1]$ through $A[n]$, which is the entire array.

Maintenance: Assume that at the start of an iteration for a value of i, if x is present in the array A, then it is present in the subarray from $A[i]$ through $A[n]$. If we get through this iteration without returning, we know what $A[i] \neq x$, and therefore we can safely say that if x is present in the array A, then it is present in the subarray from $A[i+1]$ through $A[n]$. Because i is incremented before the next iteration, the loop invariant will hold before the next iteration.

Termination: This loop must terminate, either because the procedure returns in step 1A or because $i > n$. We have already handled the case where the loop terminates because the procedure returns in step 1A.

To handle the case where the loop terminates because $i > n$, we rely on the contrapositive of the loop invariant. The ***contrapositive*** of the statement "if A then B" is "if not B then not A." The contrapositive of a statement is true if and only if the statement is true. The contrapositive of the loop invariant is "if x is not present in the subarray from $A[i]$ through $A[n]$, then it is not present in the array A."

Now, when $i > n$, the subarray from $A[i]$ through $A[n]$ is empty, and so this subarray cannot hold x. By the contrapositive of the loop invariant, therefore, x is not present anywhere in the array A, and so it is appropriate to return NOT-FOUND in step 2.

Wow, that's a lot of reasoning for what's really just a simple loop! Do we have to go through all that every time we write a loop? I don't, but there are a few computer scientists who insist on such rigorous reasoning for every single loop. When I'm writing real code, I find that most of the time that I write a loop, I have a loop invariant somewhere in the back of my mind. It might be so far back in my mind that I don't even realize that I have it, but I could state it if I had to. Although most of us would agree that a loop invariant is overkill for understanding the simple loop in BETTER-LINEAR-SEARCH, loop invariants can be quite handy when we want to understand why more complex loops do the right thing.

Recursion

With the technique of ***recursion***, we solve a problem by solving smaller instances of the same problem. Here's my favorite canonical example of recursion: computing $n!$ ("n-factorial"), which is defined for nonnegative values of n as $n! = 1$ if $n = 0$, and

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

if $n \geq 1$. For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Observe that

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

and so

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

for $n \geq 1$. We have defined $n!$ in terms of a "smaller" problem, namely $(n-1)!$. We could write a recursive procedure to compute $n!$ as follows:

Procedure FACTORIAL(n)

Input: An integer $n \geq 0$.

Output: The value of $n!$.

1. If $n = 0$, then return 1 as the output.
2. Otherwise, return n times the value returned by recursively calling FACTORIAL($n - 1$).

The way I wrote step 2 is pretty cumbersome. I could instead just write "Otherwise, return $n \cdot$ FACTORIAL($n - 1$)," using the recursive call's return value within a larger arithmetic expression.

For recursion to work, two properties must hold. First, there must be one or more ***base cases***, where we compute the solution directly without recursion. Second, each recursive call of the procedure must be on a *smaller instance* of *the same problem* that will eventually reach a base case. For the FACTORIAL procedure, the base case occurs when n equals 0, and each recursive call is on an instance in which the value of n is reduced by 1. As long as the original value of n is nonnegative, the recursive calls will eventually get down to the base case.

Arguing that a recursive algorithm works might feel overly simple at first. The key is to believe that each recursive call produces the correct result. As long as we are willing to believe that recursive calls do the right thing, arguing correctness is often easy. Here is how we could argue that the FACTORIAL procedure returns the correct answer. Clearly, when $n = 0$, the value returned, 1, equals $n!$. We assume that when $n \geq 1$, the recursive call FACTORIAL($n - 1$) does the right thing: it returns the value of $(n - 1)!$. The procedure then multiplies this value by n, thereby computing the value of $n!$, which it returns.

Here's an example where the recursive calls are not on smaller instances of the same problem, even though the mathematics is correct. It is indeed true that if $n \geq 0$, then $n! = (n + 1)!/(n + 1)$. But the following recursive procedure, which takes advantage of this formula, would fail to ever give an answer when $n \geq 1$:

Procedure BAD-FACTORIAL(n)

Input and Output: Same as FACTORIAL.

1. If $n = 0$, then return 1 as the output.
2. Otherwise, return BAD-FACTORIAL$(n + 1)/(n + 1)$.

If we were to call BAD-FACTORIAL(1), it would generate a recursive call of BAD-FACTORIAL(2), which would generate a recursive call of BAD-FACTORIAL(3), and so on, never getting down to the base case when n equals 0. If you were to implement this procedure in a real programming language and run it on an actual computer, you would quickly see something like a "stack overflow error."

We can often rewrite algorithms that use a loop in a recursive style. Here is linear search, without a sentinel, written recursively:

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

Inputs: Same as LINEAR-SEARCH, but with an added parameter i.

Output: The index of an element equaling x in the subarray from $A[i]$ through $A[n]$, or NOT-FOUND if x does not appear in this subarray.

1. If $i > n$, then return NOT-FOUND.
2. Otherwise ($i \leq n$), if $A[i] = x$, then return i.
3. Otherwise ($i \leq n$ and $A[i] \neq x$), return RECURSIVE-LINEAR-SEARCH$(A, n, i + 1, x)$.

Here, the subproblem is to search for x in the subarray going from $A[i]$ through $A[n]$. The base case occurs in step 1 when this subarray is empty, that is, when $i > n$. Because the value of i increases in each of step 3's recursive calls, if no recursive call ever returns a value of i in step 2, then eventually i becomes greater than n and we reach the base case.

Further reading

Chapters 2 and 3 of CLRS [CLRS09] cover much of the material in this chapter. An early algorithms textbook by Aho, Hopcroft, and Ullman [AHU74] influenced the field to use asymptotic notation to analyze algorithms. There has been quite a bit of work in proving programs correct; if you would like to delve into this area, try the books by Gries [Gri81] and Mitchell [Mit96].