Algorithms Illuminated

Part 1: The Basics

Tim Roughgarden

First Edition

Fourth Printing, 2021

Cover image: *Stanza*, by Andrea Belag

ISBN: 978-0-9992829-0-8 (Paperback)
ISBN: 978-0-9992829-1-5 (ebook)

Library of Congress Control Number: 2017914282

Soundlikeyourself Publishing, LLC
New York, NY
`soundlikeyourselfpublishing@gmail.com`
`www.algorithmsilluminated.org`

Contents

Preface

This book is the first in a series based on my online algorithms courses that have been running regularly since 2012, which in turn are based on an undergraduate course that I taught many times at Stanford University.

What We'll Cover in This Book

Algorithms Illuminated, Part 1 provides an introduction to and basic literacy in the following four topics.

Asymptotic analysis and big-O notation. Asymptotic notation provides the basic vocabulary for discussing the design and analysis of algorithms. The key concept here is "big-O" notation, which is a modeling choice about the granularity with which we measure the running time of an algorithm. We'll see that the sweet spot for clear high-level thinking about algorithm design is to ignore constant factors and lower-order terms, and to concentrate on how an algorithm's performance scales with the size of the input.

Divide-and-conquer algorithms and the master method. There's no silver bullet in algorithm design, no single problem-solving method that cracks all computational problems. However, there are a few general algorithm design techniques that find successful application across a range of different domains. In this part of the series, we'll cover the "divide-and-conquer" technique. The idea is to break a problem into smaller subproblems, solve the subproblems recursively, and then quickly combine their solutions into one for the original problem. We'll see fast divide-and-conquer algorithms for sorting, integer and matrix multiplication, and a basic problem in computational geometry. We'll also cover the master method, which is a powerful tool for analyzing the running time of divide-and-conquer algorithms.

Randomized algorithms. A randomized algorithm "flips coins" as it runs, and its behavior can depend on the outcomes of these coin flips. Surprisingly often, randomization leads to simple, elegant, and practical algorithms. The canonical example is randomized QuickSort, and we'll explain this algorithm and its running time analysis in detail.

Sorting and selection. As a byproduct of studying the first three topics, we'll learn several famous algorithms for sorting and selection, including MergeSort, QuickSort, and linear-time selection (both randomized and deterministic). These computational primitives are blazingly fast—almost as fast as simply reading the input. It's important to cultivate a collection of such "for-free primitives," both to apply directly to data and to use as building blocks for solutions to more difficult problems.

For a more detailed look into the book's contents, check out the "Upshot" sections that conclude each chapter and highlight the most important points. The starred sections are the most advanced ones, and a time-constrained reader can skip them on a first reading without any loss of continuity.

Topics covered in the other parts. *Part 2* of the series covers data structures (heaps, balanced search trees, hash tables, bloom filters), graph primitives (breadth- and depth-first search, connectivity, shortest paths), and their applications (ranging from deduplication to social network analysis). *Part 3* focuses on greedy algorithms (scheduling, minimum spanning trees, clustering, Huffman codes) and dynamic programming (knapsack, sequence alignment, shortest paths, optimal search trees). *Part 4* is all about NP-hard problems: how to recognize them in the wild (using reductions); ways to compromise on your algorithmic ambitions for them (through approximation or an exponential worst-case running time); and algorithmic tools to realize those revised ambitions (greedy heuristic algorithms, local search, advanced dynamic programming, MIP and SAT solvers).

Skills You'll Learn From This Book Series

Mastering algorithms takes time and effort. Why bother?

Become a better programmer. You'll learn several blazingly fast subroutines for processing data as well as several useful data structures for organizing data that you can deploy directly in your own programs. Implementing and using these algorithms will stretch and improve your programming skills. You'll also learn general algorithm design paradigms that are relevant to many different problems across different domains, as well as tools for predicting the performance of such algorithms. These "algorithmic design patterns" can help you come up with new algorithms for problems that arise in your own work.

Sharpen your analytical skills. You'll get lots of practice describing and reasoning about algorithms. Through mathematical analysis, you'll gain a deep understanding of the specific algorithms and data structures that these books cover. You'll acquire facility with several mathematical techniques that are broadly useful for analyzing algorithms.

Think algorithmically. After you learn about algorithms, you'll start seeing them everywhere, whether you're riding an elevator, watching a flock of birds, managing your investment portfolio, or even watching an infant learn. Algorithmic thinking is increasingly useful and prevalent in disciplines outside of computer science, including biology, statistics, and economics.

Literacy with computer science's greatest hits. Studying algorithms can feel like watching a highlight reel of many of the greatest hits from the last sixty years of computer science. No longer will you feel excluded at that computer science cocktail party when someone cracks a joke about Dijkstra's algorithm. After reading these books, you'll know exactly what they mean.

Ace your technical interviews. Over the years, countless students have regaled me with stories about how mastering the concepts in these books enabled them to ace every technical interview question they were ever asked.

How These Books Are Different

This series of books has only one goal: *to teach the basics of algorithms in the most accessible way possible.* Think of them as a transcript of what an expert algorithms tutor would say to you over a series of one-on-one lessons.

There are a number of excellent more traditional textbooks about algorithms, any of which usefully complement this book series with additional problems and topics. I encourage you to explore and find your own favorites. There are also several books that, unlike these books, cater to programmers looking for ready-made algorithm implementations in a specific programming language. Many such implementations are freely available on the Web as well.

Who Are You?

The whole point of these books and the online courses upon which they are based is to be as widely and easily accessible as possible. People of all ages, backgrounds, and walks of life are well represented in my online courses, and there are large numbers of students (high-school, college, etc.), software engineers (both current and aspiring), scientists, and professionals hailing from all corners of the world.

This book is not an introduction to programming. Ideally, you've already acquired basic programming skills, such as the use of arrays and recursion, in some standard programming language (be it Java, Python, C, Scala, Haskell, etc.). For a litmus test, check out Section 1.4—if it makes sense, you'll be fine for the rest of the book. If you need to beef up your programming skills, there are several outstanding free online courses that teach basic programming.

We also use mathematical analysis as needed to understand how and why algorithms really work. The freely available book *Mathematics for Computer Science*, by Eric Lehman, F. Thomson Leighton, and Albert R. Meyer is an excellent and entertaining refresher on mathematical notation (like $\sum$ and $\forall$), the basics of proofs (induction, contradiction, etc.), discrete probability, and much more. Appendices A and B also provide quick reviews of proofs by induction and discrete probability, respectively.

Additional Resources

These books are based on online courses that are currently running on the Coursera and edX platforms. I've made several resources available to help you replicate as much of the online course experience as you like.

Videos. If you're more in the mood to watch and listen than to read, check out the YouTube video playlists available at `www.algorithmsilluminated.org`. These videos cover all the topics in this book series, as well as additional advanced topics. I hope they exude a contagious enthusiasm for algorithms that, alas, is impossible to replicate fully on the printed page.

Quizzes. How can you know if you're truly absorbing the concepts in this book? Quizzes with solutions and explanations are scattered throughout the text; when you encounter one, I encourage you to pause and think about the answer before reading on.

End-of-chapter problems. At the end of each chapter, you'll find several relatively straightforward questions that test your understanding, followed by harder and more open-ended challenge problems. Hints or solutions to most of these problems (as indicated by an "*(H)*" or "*(S)*," respectively) are included at the end of the book. Readers can interact with me and each other about the end-of-chapter problems through the book's discussion forum (see below).

Programming problems. Many of the chapters conclude with a suggested programming project whose goal is to help you develop a detailed understanding of an algorithm by creating your own working implementation of it. Data sets, along with test cases and their solutions, can be found at `www.algorithmsilluminated.org`.

Discussion forums. A big reason for the success of online courses is the opportunities they provide for participants to help each other understand the course material and debug programs through discussion forums. Readers of these books have the same opportunity, via the forums available at `www.algorithmsilluminated.org`.

Changes Since the First Printing

The second printing of this book (July 2018) features a number of corrections, solutions to selected problems, and a complete rewrite of Section 5.6.3 on sorting lower bounds. The third printing (August 2018) incorporates a small number of additional corrections. The fourth printing (July 2021) includes numerous minor improvements throughout the text, several new end-of-chapter problems, hints or solutions to all end-of-chapter problems, and a complete rewrite of Section 3.4 on the closest pair problem.

Acknowledgments

These books would not exist without the passion and hunger supplied by the hundreds of thousands of participants in my algorithms courses over the years. I am particularly grateful to those who supplied detailed feedback on an earlier draft of this book: Tonya Blust, Yuan Cao, Jim Humelsine, Bayram Kuliyev, Patrick Monkelban, Kyle Schiller, Nissanka Wickremasinghe, and Daniel Zingaro.

I always appreciate suggestions and corrections from readers. These are best communicated through the discussion forums mentioned above.

Tim Roughgarden
Stanford, California
September 2017

Chapter 1

Introduction

The goal of this chapter is to get you excited about the study of algorithms. We begin by discussing algorithms in general and why they're so important. Then we use the problem of multiplying two integers to illustrate how algorithmic ingenuity can improve on more straightforward or naive solutions. We then discuss the `MergeSort` algorithm in detail, for several reasons: it's a practical and famous algorithm that you should know; it's a good warm-up to get you ready for more intricate algorithms; and it's the canonical introduction to the "divide-and-conquer" algorithm design paradigm. The chapter concludes by describing several guiding principles for how we'll analyze algorithms throughout the rest of the book.

1.1 Why Study Algorithms?

Let me begin by justifying this book's existence and giving you some reasons why you should be highly motivated to learn about algorithms. So what is an algorithm, anyway? It's a set of well-defined rules—a recipe, in effect—for solving some computational problem. Maybe you have a bunch of numbers and you want to rearrange them so that they're in sorted order. Maybe you have a road map and you want to compute the shortest path from some origin to some destination. Maybe you need to complete several tasks before certain deadlines, and you want to know in what order you should finish the tasks so that you complete them all by their respective deadlines.

So why study algorithms?

Important for all other branches of computer science. First, understanding the basics of algorithms and the closely related field of data structures is essential for doing serious work in pretty much any branch of computer science. For example, during my years as a professor at Stanford University, every degree the computer science department offered (B.S., M.S., and Ph.D.) required an algorithms course. To name just a few examples:

1. Routing protocols in communication networks piggyback on classical shortest-path algorithms.

2. Public-key cryptography relies on efficient number-theoretic algorithms.

3. Computer graphics requires the computational primitives supplied by geometric algorithms.

4. Database indices rely on balanced search tree data structures.

5. Computational biologists use dynamic programming algorithms to measure genome similarity.

6. Clustering algorithms are one of the most ubiquitous tools in unsupervised machine learning.

And the list goes on.

Driver of technological innovation. Second, algorithms play a key role in modern technological innovation. To give just one obvious example, search engines use a tapestry of algorithms to efficiently compute the relevance of various Web pages to a given search query. The most famous such algorithm is the `PageRank` algorithm, which gave birth to Google. Indeed, in a December 2010 report to the United States White House, the President's council of advisers on science and technology wrote the following:

> "Everyone knows Moore's Law — a prediction made in 1965 by Intel co-founder Gordon Moore that the density of transistors in integrated circuits would continue to double every 1 to 2 years. . . in many areas, performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed."[1]

[1]Excerpt from Report to the President and Congress: Designing a Digital Future, December 2010 (page 71).

Lens on other sciences. Third, although this is beyond the scope of this book, algorithms are increasingly used to provide a novel "lens" on processes outside of computer science and technology. For example, the study of quantum computation has provided a new computational viewpoint on quantum mechanics. Price fluctuations in economic markets can be fruitfully viewed as an algorithmic process. Even evolution can be thought of as a surprisingly effective search algorithm.

Good for the brain. Back when I was a student, my favorite classes were always the challenging ones that, after I struggled through them, left me feeling a few IQ points smarter than when I started. I hope this material provides a similar experience for you.

Fun! Finally, I hope that by the end of the book you can see why the design and analysis of algorithms is simply fun. It's an endeavor that requires a rare blend of precision and creativity. It can certainly be frustrating at times, but it's also highly addictive. And let's not forget that you've been learning about algorithms since you were a little kid.

1.2 Integer Multiplication

1.2.1 Problems and Solutions

When you were in third grade or so, you probably learned an algorithm for multiplying two numbers—a well-defined set of rules for transforming an input (two numbers) into an output (their product). It's important to distinguish between two different things: the description of the *problem being solved*, and that of the *method of solution* (that is, the algorithm for the problem). In this book, we'll repeatedly follow the pattern of first introducing a computational problem (the inputs and desired output), and then describing one or more algorithms that solve the problem.

1.2.2 The Integer Multiplication Problem

In the integer multiplication problem, the input is two n-digit numbers, which we'll call x and y. The length n of x and y could be any positive integer, but I encourage you to think of n as large, in the thousands or even more.[2] (Perhaps you're implementing a cryptographic application that must manipulate very large numbers.) The desired output in the integer multiplication problem is the product $x \cdot y$.

Problem: Integer Multiplication

Input: Two n-digit nonnegative integers, x and y.

Output: The product $x \cdot y$.

1.2.3 The Grade-School Algorithm

Having defined the computational problem precisely, we describe an algorithm that solves it—the same algorithm you learned in third grade. We will assess the performance of this algorithm through the number of "primitive operations" it performs, as a function of the number of digits n in each input number. For now, let's think of a primitive operation as any of the following: (i) adding two single-digit numbers; (ii) multiplying two single-digit numbers; or (iii) adding a zero to the beginning or end of a number.

To jog your memory, consider the concrete example of multiplying $x = 5678$ and $y = 1234$ (so $n = 4$). See also Figure 1.1. The algorithm first computes the "partial product" of the first number and the last digit of the second number $5678 \cdot 4 = 22712$. Computing this partial product boils down to multiplying each of the digits of the first number by 4, and adding in "carries" as necessary.[3] When computing the next partial product ($5678 \cdot 3 = 17034$), we do the same thing, shifting the result one digit to the left, effectively adding a "0" at the end. And so on for the final two partial products. The final step is to add up all the partial products.

Back in third grade, you probably accepted that this algorithm is *correct*, meaning that no matter what numbers x and y you start with, provided that all intermediate computations are done properly, it eventually terminates with the product $x \cdot y$ of the two input numbers.

[2] If you want to multiply numbers with different lengths (like 1234 and 56), a simple hack is to just add some zeros to the beginning of the smaller number (for example, treat 56 as 0056). Alternatively, the algorithms we'll discuss can be modified to accommodate numbers with different lengths.

[3] $8 \cdot 4 = 32$, carry the 3, $7 \cdot 4 = 28$, plus 3 is 31, carry the 3, ...

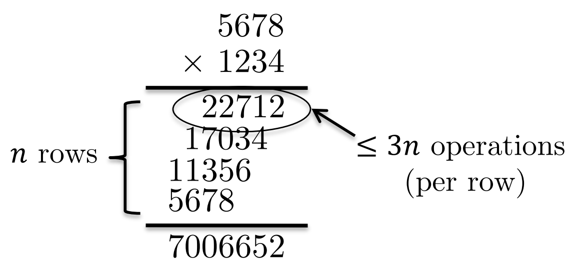

Figure 1.1: The grade-school integer multiplication algorithm.

That is, you're never going to get a wrong answer, and the algorithm can't loop forever.

1.2.4 Analysis of the Number of Operations

Your third-grade teacher might not have discussed the number of primitive operations needed to carry out this procedure to its conclusion. To compute the first partial product, we multiplied 4 times each of the digits $5, 6, 7, 8$ of the first number. This is four primitive operations. Each carry might force us to add a single-digit number to a double-digit one, for another two primitive operations. In general, computing a partial product involves n multiplications (one per digit) and at most $2n$ additions (at most two per carry), for a total of at most $3n$ primitive operations. There's nothing special about the first partial product: every partial product requires at most $3n$ operations. Because there are n partial products—one per digit of the second number—computing all of them requires at most $n \cdot 3n = 3n^2$ primitive operations. We still have to add them all up to compute the final answer, but this takes a comparable number of operations (at most another $3n^2$, as you should check). Summarizing:

$$\textbf{total number of operations} \leq \underbrace{\textbf{constant}}_{\mathbf{=6}} \cdot \mathbf{n^2}.$$

Thinking about how the amount of work the algorithm performs *scales* as the input numbers grow bigger and bigger, we see that the work required grows quadratically with the number of digits. If you double the length of the input numbers, the work required jumps by a factor of 4. Quadruple their length and it jumps by a factor of 16, and so on.

1.2.5 Can We Do Better?

Depending on what type of third-grader you were, you might well have accepted this procedure as the unique or at least optimal way to multiply two numbers. If you want to be a serious algorithm designer, you'll need to grow out of that kind of obedient timidity. The classic algorithms book by Aho, Hopcroft, and Ullman, after iterating through a number of algorithm design paradigms, has this to say:

> "Perhaps the most important principle for the good algorithm designer is to refuse to be content."[4]

Or as I like to put it, every algorithm designer should adopt the mantra:

> *Can we do better?*

This question is particularly apropos when you're faced with a naive or straightforward solution to a computational problem. In the third grade, you might not have asked if one could do better than the straightforward integer multiplication algorithm. Now is the time to ask, and answer, this question.

1.3 Karatsuba Multiplication

The algorithm design space is surprisingly rich, and there are certainly other interesting methods of multiplying two integers beyond what you learned in the third grade. This section describes a method called *Karatsuba multiplication*.[5]

1.3.1 A Concrete Example

To get a feel for Karatsuba multiplication, let's re-use our previous example with $x = 5678$ and $y = 1234$. We're going to execute a sequence of steps, quite different from the grade-school algorithm, culminating in the product $x \cdot y$. The sequence of steps should strike

---

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974, page 70.

[5] Discovered in 1960 by Anatoly Karatsuba, who at the time was a 23-year-old student.

you as very mysterious, like pulling a rabbit out of a hat; later in the section we'll explain exactly what Karatsuba multiplication is and why it works. The key point to appreciate now is that there's a dazzling array of options for solving computational problems like integer multiplication.

First, to regard the first and second halves of x as numbers in their own right, we give them the names a and b (so $a = 56$ and $b = 78$). Similarly, c and d denote 12 and 34, respectively (Figure 1.2).

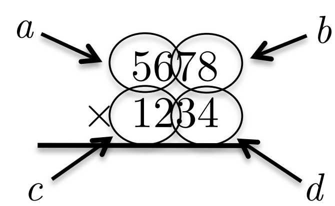

Figure 1.2: Thinking of four-digit numbers as pairs of two-digit numbers.

Next we'll perform a sequence of operations that involve only the double-digit numbers a, b, c, and d, and finally collect all the terms together in a magical way that results in the product of x and y.

Step 1: Compute $a \cdot c = 56 \cdot 12$, which is 672 (as you're welcome to check).

Step 2: Compute $b \cdot d = 78 \cdot 34 = 2652$.

The next two steps are still more inscrutable.

Step 3: Compute $(a + b) \cdot (c + d) = 134 \cdot 46 = 6164$.

Step 4: Subtract the results of the first two steps from the result of the third step: $6164 - 672 - 2652 = 2840$.

Finally, we add up the results of steps 1, 2, and 4, but only after adding four trailing zeroes to the answer in step 1 and two trailing zeroes to the answer in step 4.

Step 5: Compute $10^4 \cdot 672 + 10^2 \cdot 2840 + 2652 = 6720000 + 284000 + 2652 = 7006652$.

This is exactly the same (correct) result computed by the grade-school algorithm in Section 1.2!

You should not have any intuition about what just happened. Rather, I hope that you feel some mixture of bafflement and intrigue, and appreciate the fact that there seem to be fundamentally different algorithms for multiplying integers than the one you learned as a kid. Once you realize how rich the space of algorithms is, you have to wonder: can we do better than the third-grade algorithm? Does the algorithm above already do better?

1.3.2 A Recursive Algorithm

Before tackling full-blown Karatsuba multiplication, let's explore a simpler recursive approach to integer multiplication.[6] A recursive algorithm for integer multiplication presumably involves multiplications of numbers with fewer digits (like 12, 34, 56, and 78 in the computation above).

In general, a number x with an even number n of digits can be expressed in terms of two $n/2$-digit numbers, its first half a and second half b:

$$x = 10^{n/2} \cdot a + b.$$

Similarly, we can write

$$y = 10^{n/2} \cdot c + d.$$

To compute the product of x and y, let's use the two expressions above and multiply out:

$$\begin{aligned} x \cdot y &= (10^{n/2} \cdot a + b) \cdot (10^{n/2} \cdot c + d) \\ &= 10^n \cdot (a \cdot c) + 10^{n/2} \cdot (a \cdot d + b \cdot c) + b \cdot d. \end{aligned} \tag{1.1}$$

Note that all of the multiplications in (1.1) are either between pairs of $n/2$-digit numbers or involve a power of 10.[7]

[6] I'm assuming you've heard of recursion as part of your programming background. A recursive procedure is one that invokes itself as a subroutine with a smaller input, until a base case is reached.

[7] For simplicity, we are assuming that n is a power of 2. A simple hack for enforcing this assumption is to add an appropriate number of leading zeroes to x and y, which at most doubles their lengths. Alternatively, when n is odd, it's also fine to break x and y into two numbers with almost equal lengths.

The expression (1.1) suggests a recursive approach to multiplying two numbers. To compute the product $x \cdot y$, we compute the expression (1.1). The four relevant products ($a \cdot c$, $a \cdot d$, $b \cdot c$, and $b \cdot d$) all concern numbers with fewer than n digits, so we can compute each of them recursively. Once our four recursive calls come back to us with their answers, we can compute the expression (1.1) in the obvious way: tack on n trailing zeroes to $a \cdot c$, add $a \cdot d$ and $b \cdot c$ (using grade-school addition) and tack on $n/2$ trailing zeroes to the result, and finally add these two expressions to $b \cdot d$.[8] We summarize this algorithm, which we'll call `RecIntMult`, in the following pseudocode.[9]

`RecIntMult`

Input: two n-digit positive integers x and y.
Output: the product $x \cdot y$.
Assumption: n is a power of 2.

---

if $n = 1$ **then** `// base case`
  compute $x \cdot y$ in one step and return the result
else `// recursive case`
  $a, b :=$ first and second halves of x
  $c, d :=$ first and second halves of y
  recursively compute $ac := a \cdot c$, $ad := a \cdot d$,
   $bc := b \cdot c$, and $bd := b \cdot d$
  compute $10^n \cdot ac + 10^{n/2} \cdot (ad + bc) + bd$ using
   grade-school addition and return the result

Is the `RecIntMult` algorithm faster or slower than the grade-school algorithm? You shouldn't necessarily have any intuition about this question, and the answer will have to wait until Chapter 4.

---

[8] Recursive algorithms also need one or more base cases, so that they don't keep calling themselves until the rest of time. Here, the base case is: if x and y are 1-digit numbers, multiply them in one primitive operation and return the result.

[9] In pseudocode, we use "=" to denote an equality test, and ":=" to denote a variable assignment.

1.3.3 Karatsuba Multiplication

Karatsuba multiplication is an optimized version of the `RecIntMult` algorithm. We again start from the expansion (1.1) of $x \cdot y$ in terms of a, b, c, and d. The `RecIntMult` algorithm uses four recursive calls, one for each of the products in (1.1) between $n/2$-digit numbers. But *we don't care about $a \cdot d$ or $b \cdot c$,* except inasmuch as we care about their sum $a \cdot d + b \cdot c$. With only three quantities that we care about—$a \cdot c$, $a \cdot d + b \cdot c$, and $b \cdot d$—can we get away with only three recursive calls? To see that we can, first use two recursive calls to compute $a \cdot c$ and $b \cdot d$, as before.

Step 1: Recursively compute $a \cdot c$.

Step 2: Recursively compute $b \cdot d$.

Instead of recursively computing $a \cdot d$ or $b \cdot c$, we recursively compute the product of $a + b$ and $c + d$.[10]

Step 3: Compute $a + b$ and $c + d$ using grade-school addition, and recursively compute $(a + b) \cdot (c + d)$.

The key trick in Karatsuba multiplication goes back to the early 19th-century mathematician Carl Friedrich Gauss, who was thinking about multiplying complex numbers. Subtracting the results of the first two steps from the result of the third step gives exactly what we want, the middle coefficient in (1.1) of $a \cdot d + b \cdot c$:

$$\underbrace{(a + b) \cdot (c + d)}_{=a \cdot c + a \cdot d + b \cdot c + b \cdot d} - a \cdot c - b \cdot d = a \cdot d + b \cdot c.$$

Step 4: Subtract the results of the first two steps from the result of the third step to obtain $a \cdot d + b \cdot c$.

The final step computes (1.1), as in the `RecIntMult` algorithm.

Step 5: Compute (1.1) by adding up the results of steps 1, 2, and 4, after adding n trailing zeroes to the answer in step 1 and $n/2$ trailing zeroes to the answer in step 4.

[10]The numbers $a + b$ and $c + d$ might have as many as $(n/2) + 1$ digits, but the algorithm still works fine.

```
Karatsuba

Input: two n-digit positive integers x and y.
Output: the product x · y.
Assumption: n is a power of 2.
────────────────────────────────────────────
if n = 1 then                          // base case
    compute x · y in one step and return the result
else                                   // recursive case
    a, b := first and second halves of x
    c, d := first and second halves of y
    compute p := a + b and q := c + d using
      grade-school addition
    recursively compute ac := a · c, bd := b · d, and
      pq := p · q
    compute adbc := pq − ac − bd using grade-school
      addition
    compute 10^n · ac + 10^(n/2) · adbc + bd using
      grade-school addition and return the result
```

Thus Karatsuba multiplication makes only three recursive calls! Saving a recursive call should save on the overall running time, but by how much? Is the `Karatsuba` algorithm faster than the grade-school multiplication algorithm? The answer is far from obvious, but it is an easy application of the tools you'll acquire in Chapter 4 for analyzing the running time of such "divide-and-conquer" algorithms.

On Pseudocode

This book explains algorithms using a mixture of high-level pseudocode and English (as in this section). I'm assuming that you have the skills to translate such high-level descriptions into working code in your favorite programming language. Several other books and resources on the Web offer concrete implementations of various algorithms in specific programming languages.

The first benefit of emphasizing high-level descriptions over language-specific implementations is flexibility: while I assume familiarity with *some* programming language, I don't care which one. Second, this approach promotes the understanding of algorithms at a deep and conceptual level, unencumbered by low-level details. Seasoned programmers and computer scientists generally think and communicate about algorithms at a similarly high level.

Still, there is no substitute for the detailed understanding of an algorithm that comes from providing your own working implementation of it. I strongly encourage you to implement as many of the algorithms in this book as you have time for. (It's also a great excuse to pick up a new programming language!) For guidance, see the end-of-chapter Programming Problems and supporting test cases.

1.4 `MergeSort`: The Algorithm

This section and the next provide our first taste of analyzing the running time of a non-trivial algorithm—the famous `MergeSort` algorithm.

1.4.1 Motivation

`MergeSort` is a relatively ancient algorithm, and was certainly known to John von Neumann as early as 1945. Why begin a modern course on algorithms with such an old example?

Oldie but a goodie. Despite being over 70 years old, `MergeSort` is still one of the methods of choice for sorting. It's used all the time in practice, and is the default sorting algorithm in a number of programming libraries.

Canonical divide-and-conquer algorithm. The "divide-and-conquer" algorithm design paradigm is a general approach to solving problems, with applications in many different domains. The basic idea is to break your problem into smaller subproblems, solve the subproblems recursively, and finally combine the solutions to the subproblems into one for the original problem. `MergeSort` is an ideal introduction to the divide-and-conquer paradigm, the benefits it offers, and the analysis challenges it presents.

Calibrate your preparation. Our `MergeSort` discussion will give you a good indication of whether your current skill set is a good match for this book. My assumption is that you have the programming and mathematical backgrounds to (with some work) translate the high-level idea of `MergeSort` into a working program in your favorite programming language and to follow our running time analysis of the algorithm. If this and the next section make sense, you're in good shape for the rest of the book.

Motivates guiding principles for algorithm analysis. Our running time analysis of `MergeSort` exposes a number of more general guiding principles, such as the quest for running time bounds that hold for every input of a given size, and the importance of the rate of growth of an algorithm's running time (as a function of the input size).

Warm-up for the master method. We'll analyze `MergeSort` using the "recursion tree method," which is a way of tallying up the operations performed by a recursive algorithm. Chapter 4 builds on these ideas and culminates with the "master method," a powerful and easy-to-use tool for bounding the running time of many different divide-and-conquer algorithms, including the `RecIntMult` and `Karatsuba` algorithms of Section 1.3.

1.4.2 Sorting

Look around and you'll notice that computers seem pretty good at sorting things instantaneously. Think of the list of contacts on your phone. Are they listed in the order that you entered them? Of course not—that would be annoying, so they're instead sorted alphabetically for easy reference. Meanwhile, your favorite spreadsheet program has no trouble sorting a large file according to whatever criterion you want. How do they do it?

Here's a canonical version of the sorting problem:

Problem: Sorting

Input: An array of n numbers, in arbitrary order.

Output: An array of the same numbers, sorted from smallest to largest.

For example, given the input array

5	4	1	8	7	2	6	3

the desired output array is

1	2	3	4	5	6	7	8

In the example above, the eight numbers in the input array are distinct. Sorting isn't really any harder when there are duplicates, and it can even be easier. But to keep the discussion as simple as possible, let's assume—among friends—that the numbers in the input array are always distinct. I strongly encourage you to think about how our sorting algorithms need to be modified (if at all) to handle duplicates.[11]

If you don't care about optimizing the running time, it's not too difficult to come up with a correct sorting algorithm. Perhaps the simplest approach is to first scan through the input array to identify the minimum element and copy it over to the first element of the output array; then do another scan to identify and copy over the second-smallest element; and so on. This algorithm is called `SelectionSort`. You may have heard of `InsertionSort`, which can be viewed as a slicker implementation of the same idea of iteratively growing a prefix of the sorted output array. You might also know `BubbleSort`, in which you identify adjacent pairs of elements that are out of order, and

[11]In practice, there is often data (called the *value*) associated with each number (which is called the *key*). For example, you might want to sort employee records (with the name, salary, etc.), using social security numbers as keys. We focus on sorting the keys, with the understanding that each key retains its associated data.

perform repeated swaps until the entire array is sorted. All of these algorithms have quadratic running times, meaning that—analogous to the grade-school multiplication algorithm in Section 1.2.3—the number of operations performed on arrays of length n scales with n^2, the square of the input length.

Can we do better? By using the divide-and-conquer algorithm design paradigm, the MergeSort algorithm improves dramatically over these more straightforward sorting algorithms.[12]

1.4.3 An Example

The easiest way to understand MergeSort is through a picture of a concrete example (Figure 1.3). We'll use the input array from Section 1.4.2.

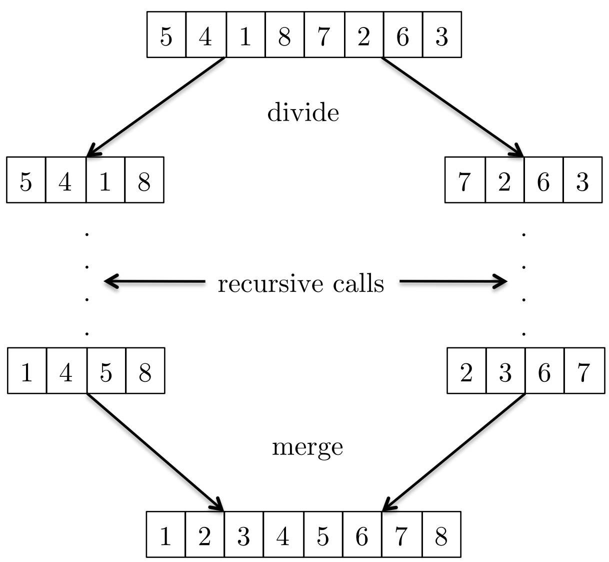

Figure 1.3: A bird's-eye view of MergeSort on a concrete example.

As a recursive divide-and-conquer algorithm, MergeSort calls itself on smaller arrays. The simplest way to decompose a sorting problem into smaller sorting problems is to break the input array in half. The

[12]While generally dominated by MergeSort, InsertionSort is still useful in practice in certain cases, especially for small input sizes.

first and second halves are each sorted recursively. For example, in Figure 1.3, the first and second halves of the input array are $[5, 4, 1, 8]$ and $[7, 2, 6, 3]$. By the magic of recursion (or induction, if you prefer), the first recursive call correctly sorts the first half, returning the array $[1, 4, 5, 8]$. The second recursive call returns the array $[2, 3, 6, 7]$. The final "merge" step combines these two sorted arrays of length 4 into a single sorted array of all 8 numbers. Details of this step are given below; the basic idea is to walk indices down each of the sorted subarrays, populating the output array from left to right in sorted order.

1.4.4 Pseudocode

The picture in Figure 1.3 suggests the following pseudocode, with two recursive calls and a merge step, for the general problem. As usual, our description cannot necessarily be translated line by line into working code (though it's pretty close).

`MergeSort`

Input: array A of n distinct integers.
Output: array with the same integers, sorted from smallest to largest.

```
// ignoring base cases
C := recursively sort first half of A
D := recursively sort second half of A
return Merge (C,D)
```

There are several deliberate omissions from the pseudocode that deserve comment. As a recursive algorithm, there should also be one or more base cases, in which there is no further recursion and the answer is returned directly. So if the input array A contains only 0 or 1 elements, `MergeSort` returns it (it is already sorted). The pseudocode does not detail what "first half" and "second half" mean when n is odd, but the obvious interpretation (with one "half" having one more element than the other) works fine. Finally, the pseudocode ignores the implementation details of how to actually pass the two subarrays to their respective recursive calls. These details depend somewhat on the programming language. The point of high-level pseudocode is to ignore such details and focus on the concepts that transcend any particular programming language.

1.4.5 The Merge Subroutine

How should we implement the Merge step? At this point, the two recursive calls have done their work and we have in our possession two sorted subarrays C and D of length $n/2$. The idea is to traverse both the sorted subarrays in order and populate the output array from left to right in sorted order.[13]

Merge

Input: sorted arrays C and D (length $n/2$ each).
Output: sorted array B (length n).
Simplifying assumption: n is even.

```
1 i := 1
2 j := 1
3 for k := 1 to n do
4     if C[i] < D[j] then
5         B[k] := C[i]          // populate output array
6         i := i + 1                       // increment i
7     else                                // D[j] < C[i]
8         B[k] := D[j]
9         j := j + 1
```

We traverse the output array using the index k, and the sorted subarrays with the indices i and j. All three arrays are traversed from left to right. The for loop in line 3 implements the pass over the output array. In the first iteration, the subroutine identifies the minimum element in either C or D and copies it over to the first position of the output array B. The minimum element overall is either in C (in which case it's $C[1]$, because C is sorted) or in D (in

[13]Unless otherwise noted, we number our array entries beginning with 1 (rather than 0), and use the syntax "$A[i]$" for the ith entry of an array A. These details vary across programming languages.

which case it's $D[1]$, because D is sorted). Advancing the appropriate index (i or j) effectively removes from further consideration the element just copied, and the process is then repeated to identify the smallest element remaining in C or D (the second-smallest overall). In general, the smallest element not yet copied over to B is either $C[i]$ or $D[j]$; the subroutine explicitly checks to see which one is smaller and proceeds accordingly. Because every iteration copies over the smallest element still under consideration in C or D, the output array is indeed populated in sorted order.

As usual, our pseudocode is intentionally a bit sloppy, to emphasize the forest over the trees. A full implementation should also keep track of when the traversal of C or D falls off the end of the array, at which point the remaining elements of the other array are copied into the final entries of B (in order). Now is a good time to work through your own implementation of the `MergeSort` algorithm.

1.5 `MergeSort`: The Analysis

What's the running time of the `MergeSort` algorithm, as a function of the length n of the input array? Is it faster than more straightforward methods of sorting, such as `SelectionSort`, `InsertionSort`, and `BubbleSort`? By "running time," we mean the number of lines of code executed in a concrete implementation of the algorithm. Think of walking line by line through this implementation using a debugger, one "primitive operation" at a time. We're interested in the number of steps the debugger takes before the program completes.

1.5.1 Running Time of `Merge`

Analyzing the running time of the `MergeSort` algorithm is an intimidating task, as it's a recursive algorithm that calls itself over and over. So let's warm up with the simpler task of understanding the number of operations performed by a single invocation of the `Merge` subroutine when called on two sorted arrays of length $\ell/2$ each. We can do this directly, by inspecting the code in Section 1.4.5 (where n corresponds to ℓ). First, lines 1 and 2 each perform an initialization, and we'll count this as two operations. Then, we have a for loop that executes a total of ℓ times. Each iteration of the loop performs a comparison in line 4, an assignment in either line 5 or line 8, and an increment in either line 6 or line 9. The loop index k also needs to get incremented each loop iteration. This means that 4 primitive operations are performed for each of the ℓ iterations of the loop.[14] Totaling up, we conclude that the `Merge` subroutine performs at most $4\ell + 2$ operations to merge two sorted arrays of length $\ell/2$ each. Let me abuse our friendship further with a true but sloppy inequality that will make our lives easier: for every positive integer $\ell \geq 1$, we have $4\ell + 2 \leq 4\ell + 2\ell = 6\ell$. That is, 6ℓ is also a valid upper bound on the number of operations performed by the `Merge` subroutine.

Lemma 1.1 (Running Time of `Merge`) *For every pair of sorted input arrays C, D of length $\ell/2$, the `Merge` subroutine performs at most 6ℓ operations.*

On Lemmas, Theorems, and the Like

In mathematical writing, the most important technical statements are labeled *theorems*. A *lemma* is a technical statement that assists with the proof of a theorem (much as `Merge` assists with the implementation of `MergeSort`). A *corollary* is a statement that follows immediately from an already-proven result, such as a special case of a theorem. We use the term *proposition* for stand-alone technical statements that are not particularly important in their own right.

1.5.2 Running Time of `MergeSort`

How can we go from the straightforward analysis of the `Merge` subroutine to an analysis of `MergeSort`, a recursive algorithm that spawns further invocations of itself? Especially terrifying is the rapid proliferation of recursive calls, the number of which is blowing up exponentially with the depth of the recursion. The one thing we have going for us is the fact that every recursive call is passed an input substantially

[14] One could quibble with the choice of 4. Does comparing the loop index k to its upper bound also count as an additional operation each iteration, for a total of 5? Section 1.6 explains why such differences in accounting don't really matter. So let's agree, among friends, that it's 4 primitive operations per iteration.

smaller than the one we started with. There's a tension between two competing forces: on the one hand, the explosion of different subproblems that need to be solved; and on the other, the ever-shrinking inputs for which these subproblems are responsible. Reconciling these two forces will drive our analysis of `MergeSort`. In the end, we'll prove the following concrete and useful upper bound on the number of operations performed by `MergeSort` (across all its recursive calls).

Theorem 1.2 (Running Time of `MergeSort`) *For every input array of length $n \geq 1$, the `MergeSort` algorithm performs at most*

$$6n \log_2 n + 6n$$

operations, where $\log_2$ *denotes the base-2 logarithm.*

On Logarithms

Some people are unnecessarily frightened by the appearance of a logarithm, which is actually a very down-to-earth concept. For a positive integer n, $\log_2 n$ just means the following: type n into a calculator, and count the number of times you need to divide it by 2 before the result is 1 or less.[a] For example, it takes five divide-by-twos to bring 32 down to 1, so $\log_2 32 = 5$. Ten divide-by-twos bring 1024 down to 1, so $\log_2 1024 = 10$. These examples make it intuitively clear that $\log_2 n$ is much less than n (compare 10 vs. 1024), especially as n grows large. A plot confirms this intuition (Figure 1.4).

[a]To be pedantic, $\log_2 n$ is not an integer if n is not a power of 2, and what we have described is really $\log_2 n$ rounded up to the nearest integer. We can ignore this minor distinction.

Theorem 1.2 is a win for the `MergeSort` algorithm and showcases the benefits of the divide-and-conquer algorithm design paradigm. We mentioned that the running times of simpler sorting algorithms, like `SelectionSort`, `InsertionSort`, and `BubbleSort`, depend *quadratically* on the input size n, meaning that the number of operations