CAMBRIDGE

Cambridge University Press & Assessment
978-0-999-28298-4 — Algorithms Illuminated
Tim Roughgarden
Excerpt
More Information

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.

©2022, Soundlikeyourself Publishing, LLC

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-0-999-28298-4 — Algorithms Illuminated
Tim Roughgarden
Excerpt
More Information

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]

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

©2022, Soundlikeyourself Publishing, LLC

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-0-999-28298-4 — Algorithms Illuminated
Tim Roughgarden
Excerpt
More Information

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.

```
        5678
     ×  1234
     -------
       22712
      17034
     11356
     5678
     -------
     7006652
```


Figure 1.1: The grade-school integer multiplication algorithm.

[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, ...

©2022, Soundlikeyourself Publishing, LLC

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-0-999-28298-4 — Algorithms Illuminated
Tim Roughgarden
Excerpt
More Information

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. 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:

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

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.

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

©2022, Soundlikeyourself Publishing, LLC

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-0-999-28298-4 — Algorithms Illuminated
Tim Roughgarden
Excerpt
More Information

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 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).

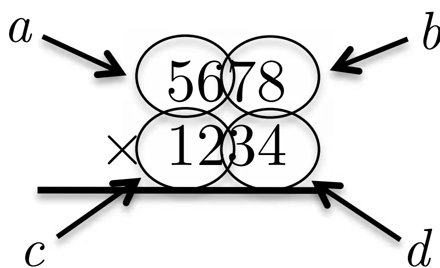

Figure 1.2: Thinking of four-digit numbers as pairs of two-digit numbers.

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.

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

©2022, Soundlikeyourself Publishing, LLC

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-0-999-28298-4 — Algorithms Illuminated
Tim Roughgarden
Excerpt
More Information

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]

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]

[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.

[8] Recursive algorithms also need one or more base cases, so that they don't keep calling themselves for 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.

©2022, Soundlikeyourself Publishing, LLC

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-0-999-28298-4 — Algorithms Illuminated
Tim Roughgarden
Excerpt
More Information

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.

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 $a \cdot d + b \cdot c$:

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

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

©2022, Soundlikeyourself Publishing, LLC

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-0-999-28298-4 — Algorithms Illuminated
Tim Roughgarden
Excerpt
More Information

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.

Karatsuba

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
  compute $p := a + b$ and $q := c + d$ using grade-school addition
  recursively compute $ac := a \cdot c$, $bd := b \cdot d$, and $pq := p \cdot q$
  compute $adbc := pq - ac - bd$ using grade-school addition
  compute $10^n \cdot ac + 10^{n/2} \cdot adbc + bd$ 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.

©2022, Soundlikeyourself Publishing, LLC

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE

Cambridge University Press & Assessment
978-0-999-28298-4 — Algorithms Illuminated
Tim Roughgarden
Excerpt
More Information

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.

©2022, Soundlikeyourself Publishing, LLC

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-0-999-28298-4 — Algorithms Illuminated
Tim Roughgarden
Excerpt
More Information

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 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.

[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.

©2022, Soundlikeyourself Publishing, LLC

© in this web service Cambridge University Press & Assessment www.cambridge.org