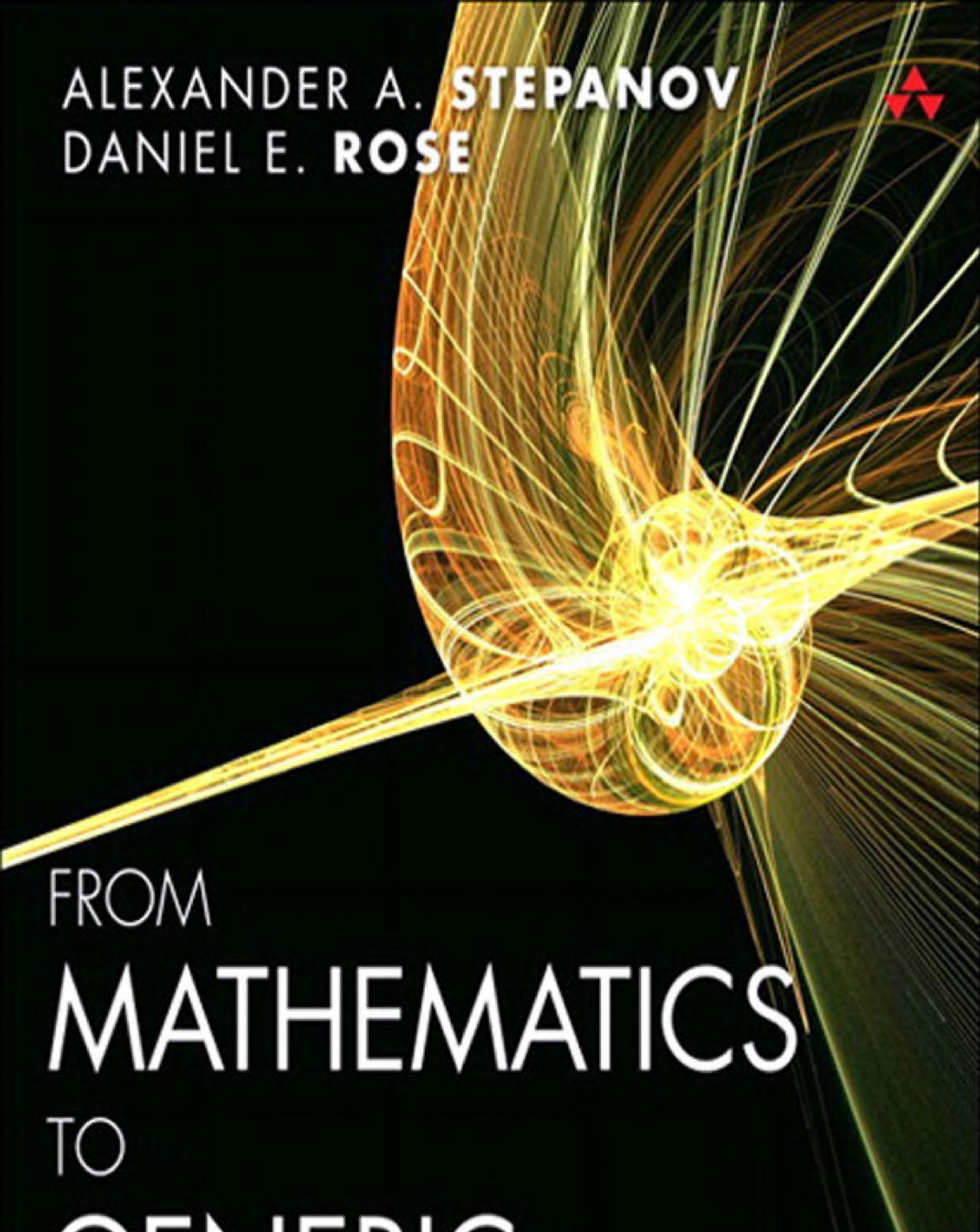
ALEXANDER A. STEPANOV
DANIEL E. ROSE



FROM MATHEMATICS TO GENERIC PROGRAMMING

From Mathematics to Generic Programming

Alexander A. Stepanov

Daniel E. Rose

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Stepanov, Alexander A.
From mathematics to generic programming / Alexander A. Stepanov, Daniel E. Rose.
pages cm
Includes bibliographical references and index.
ISBN 978-0-321-94204-3 (pbk. : alk. paper)
1. Generic programming (Computer science)—Mathematics. 2. Computer algorithms. I. Rose, Daniel E. II. Title.
QA76.6245.S74 2015
005.1'1—dc23

2014034539

ISBN-13: 978-0-321-94204-3
ISBN-10: 0-321-94204-3
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, November 2014

Contents

Acknowledgments

We would like to thank all the people who contributed to making this book a reality. Our management at A9.com actively supported this project from the beginning. Bill Stasior initiated the creation of the course this book is based on, and selected the topic from among several options we offered. Brian Pinkerton not only attended the course, but also strongly encouraged our idea of turning the material into a book. We also would like to thank Mat Marcus, who collaborated with Alex on a similar course at Adobe in 2004–2005.

The other members of the Fundamental Data Structures and Algorithms for Search team played important roles throughout the process. Anil Gangolli helped shape the content of the course, Ryan Ernst provided much of the programming infrastructure, and Paramjit Oberoi gave invaluable feedback during the writing stage. We have enjoyed working with all of them and are grateful for their input.

We are grateful to our editors, Peter Gordon and Greg Doench, and to the team of experts assembled by Addison-Wesley, including managing editor John Fuller, production editor Mary Kesel Wilson, copyeditor Jill Hobbs, and compositor/LaTeX expert Lori Hughes for all their work in turning our rough manuscript into a polished book.

Finally, we'd like to thank the many friends, family members, and colleagues who read earlier drafts of the book and/or gave us comments, corrections, suggestions, advice, or other help: Gašper Ažman, John Banning, Cynthia Dwork, Hernan Epelman, Ryan Ernst, Anil Gangolli, Susan Gruber, Jon Kalb, Robert Lehr, Dmitry Leshchiner, Tom London, Mark Manasse, Paul McJones, Nicolas Nicolov, Gor Nishanov, Paramjit Oberoi, Sean Parent, Fernando Pelliccioni, John Reiser, Robert Rose, Stefan Vargyas, and Adam Young. The book is much better as a result of their contributions.

This page intentionally left blank

About the Authors

Alexander A. Stepanov studied mathematics at Moscow State University from 1967 to 1972. He has been programming since 1972: first in the Soviet Union and, after emigrating in 1977, in the United States. He has programmed operating systems, programming tools, compilers, and libraries. His work on foundations of programming has been supported by GE, Polytechnic University, Bell Labs, HP, SGI, Adobe, and, since 2009, A9.com, Amazon's search technology subsidiary. In 1995 he received the *Dr. Dobb's Journal* Excellence in Programming Award for the design of the C++ Standard Template Library.

Daniel E. Rose is a research scientist who has held management positions at Apple, AltaVista, Xigo, Yahoo, and A9.com. His research focuses on all aspects of search technology, ranging from low-level algorithms for index compression to human–computer interaction issues in web search. Rose led the team at Apple that created desktop search for the Macintosh. He holds a Ph.D. in cognitive science and computer science from University of California, San Diego, and a B.A. in philosophy from Harvard University.

This page intentionally left blank

Authors' Note

The separation of computer science from mathematics greatly impoverishes both. The lectures that this book is based on were my attempt to show how these two activities—an ancient one going back to the very beginnings of our civilization and the most modern one—can be brought together.

I was very fortunate that my friend Dan Rose, under whose management our team was applying principles of generic programming to search engine design, agreed to convert my rather meandering lectures into a coherent book. Both of us hope that our readers will enjoy the result of our collaboration.

—A.A.S.

The book you are about to read is based on notes from an "Algorithmic Journeys" course taught by Alex Stepanov at A9.com during 2012. But as Alex and I worked together to transform the material into book form, we realized that there was a stronger story we could tell, one that centered on generic programming and its mathematical foundations. This led to a major reorganization of the topics, and removal of the entire section on set theory and logic, which did not seem to be part of the same story. At the same time, we added and removed details to create a more coherent reading experience and to make the material more accessible to less mathematically advanced readers.

While Alex comes from a mathematical background, I do not. I've tried to learn from my own struggles to understand some of the material and to use this experience to identify ideas that require additional explanation. If in some cases we describe something in a slightly different way than a mathematician would, or using slightly different terminology, or using more simple steps, the fault is mine.

—D.E.R.

This page intentionally left blank

1 one

What This Book Is About

It is impossible to know things of this world
unless you know mathematics.
Roger Bacon, *Opus Majus*

This book is about programming, but it is different from most programming books. Along with algorithms and code, you'll find mathematical proofs and historical notes about mathematical discoveries from ancient times to the 20th century.

More specifically, the book is about *generic programming*, an approach to programming that was introduced in the 1980s and started to become popular following the creation of the C++ Standard Template Library (STL) in the 1990s. We might define it like this:

Definition 1.1. Generic programming is an approach to programming that focuses on designing algorithms and data structures so that they work in the most general setting without loss of efficiency.

If you've used STL, at this point you may be thinking, "Wait a minute, that's all there is to generic programming? What about all that stuff about templates and iterator traits?" Those are tools that enable the language to support generic programming, and it's important to know how to use them effectively. But generic programming itself is more of an *attitude* toward programming than a particular set of tools.

We believe that this attitude—trying to write code in this general way—is one that all programmers should embrace. The components of a well-written generic program are easier to use and modify than those of a program whose data structures, algorithms, and interfaces hardcode unnecessary assumptions about a specific application. Making a program more generic renders it simultaneously both simpler and more powerful.

1.1 Programming and Mathematics

So where does this generic programming attitude come from, and how do you learn it? It comes from mathematics, and especially from a branch of mathematics called *abstract algebra*. To help you understand the approach, this book will introduce you to a little bit of abstract algebra, which focuses on how to reason about objects in terms of abstract properties of operations on them. It's a topic normally studied only by university students majoring in math, but we believe it's critical in understanding generic programming.

In fact, it turns out that many of the fundamental ideas in programming came from mathematics. Learning how these ideas came into being and evolved over time can help you think about software design. For example, Euclid's *Elements*, a book written more than 2000 years ago, is still one of the best examples of how to build up a complex system from small, easily understood pieces.

Although the essence of generic programming is abstraction, abstractions do not spring into existence fully formed. To see how to make something more general, you need to start with something concrete. In particular, you need to understand the specifics of a particular domain to discover the right abstractions.

The abstractions that appear in abstract algebra largely come from concrete results in one of the oldest branches of mathematics, called *number theory*. For this reason, we will also introduce some key ideas from number theory, which deals with properties of integers, especially divisibility.

The thought process you'll go through in learning this math can improve your programming skills. But we'll also show how some of the mathematical results themselves turn out to be crucial to some modern software applications. In particular, by the end of the book we'll show how some of these results are used in cryptographic protocols underlying online privacy and online commerce.

The book will move back and forth between talking about math and talking about programming. In particular, we'll interweave important ideas in mathematics with a discussion of both specific algorithms and general programming techniques. We'll mention some algorithms only briefly, while others will be refined and generalized throughout the book. A couple of chapters will contain only mathematical material, and a couple will contain only programming material, but most have a mixture of both.

1.2 A Historical Perspective

We've always found that it's easier and more interesting to learn something if it's part of a story. What was going on at the time? Who were the people involved, and how did they come to have these ideas? Was one person's work an attempt to build on another's—or an attempt to reject what came before? So as we introduce the mathematical ideas in this book, we'll try to tell you the story of those ideas and of the people who came up with them. In many cases, we've provided short biographical sketches of the mathematicians who are the main characters in our story. These aren't comprehensive encyclopedia entries, but rather an attempt to give you some context for who these people were.

Although we take a historical perspective, that doesn't mean that the book is intended as a history of mathematics or even that all the ideas are presented in the order in which they were discovered. We'll jump around in space and time when necessary, but we'll try to give a historical context for each of the ideas.

1.3 Prerequisites

Since a lot of the book is about mathematics, you may be concerned that you need to have taken a lot of math classes to understand it. While you'll need to be able to think logically (something you should already be good at as a programmer), we don't assume any specific mathematical knowledge beyond high school algebra and geometry. In a couple of sections, we show some applications that use a little linear algebra (vectors and matrices), but you can safely skip these if you haven't been exposed to the background material before. If you're unfamiliar with any of the notation we use, it's explained in Appendix A.

An important part of mathematics is being able to prove something formally. This book contains quite a few proofs. You'll find the book easier to understand if you've done some proofs before, whether in high school geometry, in a computer science class on automata theory, or in logic. We've described some of the common proof techniques we use, along with examples, in Appendix B.

We assume that if you're reading this book, you're already a programmer. In particular, you should be reasonably proficient in a typical imperative programming language like C, C++, or Java. Our examples will use C++, but we expect you'll be able to understand them even if you've never programmed in that language before. When we make use of a construct unique to C++, we explain it in Appendix C. Irrespective of our use of C++, we believe that the principles discussed in this book apply to programming in general.

Many of the programming topics in this book are also covered from a different perspective, and more formally, in *Elements of Programming* by Stepanov and McJones. Readers interested in additional depth may find that book to be a useful companion to this one. Throughout this book, we occasionally refer interested readers to a relevant section of *Elements of Programming*.

1.4 Roadmap

Before diving into the details, it's useful to see a brief overview of where we're headed:

- Chapter 2 tells the story of an ancient algorithm for multiplication, and how to improve it.
- Chapter 3 looks at some early observations about properties of numbers, and an efficient implementation of an algorithm for finding primes.
- Chapter 4 introduces an algorithm for finding the greatest common divisor (GCD), which will be the basis for some of our abstractions and applications later on.
- Chapter 5 focuses on mathematical results, introducing a couple of important theorems that will play a critical role by the end of the book.
- Chapter 6 introduces the mathematical field of abstract algebra, which provides the core idea for generic programming.
- Chapter 7 shows how these mathematical ideas allow us to generalize our multiplication algorithm beyond simple arithmetic to a variety of practical programming applications.
- Chapter 8 introduces new abstract mathematical structures, and shows some new applications they enable.
- Chapter 9 talks about axiom systems, theories, and models, which are all building blocks of generic programming.
- Chapter 10 introduces concepts in generic programming, and examines the subtleties of some apparently simple programming tasks.
- Chapter 11 continues the exploration of some fundamental programming tasks, examining how different practical implementations can exploit theoretical knowledge of the problem.
- Chapter 12 looks at how hardware constraints can lead to a new approach for an old algorithm, and shows new applications of GCD.
- Chapter 13 puts the mathematical and algorithmic results together to build an important cryptography application.
- Chapter 14 is a summary of some of the principal ideas in the book.

The strands of programming and mathematics are interwoven throughout, though one or the other may lie hidden for a chapter or two. But every chapter plays a part in the overall chain of reasoning that summarizes the entire book:

To be a good programmer, you need to understand the principles of generic programming. To understand the principles of generic programming, you need to understand abstraction. To understand abstraction, you need to understand the mathematics on which it's based.

That's the story we're hoping to tell.

This page intentionally left blank

2
two

The First Algorithm

Moses speedily learned arithmetic, and geometry.
...This knowledge he derived from the Egyptians,
who study mathematics above all things.
Philo of Alexandria, *Life of Moses*

An algorithm is a terminating sequence of steps for accomplishing a computational task. Algorithms are so closely associated with the notion of computer programming that most people who know the term probably assume that the idea of algorithms comes from computer science. But algorithms have been around for literally thousands of years. Mathematics is full of algorithms, some of which we use every day. Even the method schoolchildren learn for long addition is an algorithm.

Despite its long history, the notion of an algorithm didn't always exist; it had to be invented. While we don't know when algorithms were first invented, we do know that some algorithms existed in Egypt at least as far back as 4000 years ago.

* * *

Ancient Egyptian civilization was centered on the Nile River, and its agriculture depended on the river's floods to enrich the soil. The problem was that every time the Nile flooded, all the markers showing the boundaries of property were washed away. The Egyptians used ropes to measure distances, and developed procedures so they could go back to their written records and reconstruct the property boundaries. A select group of priests who had studied these mathematical techniques were responsible for this task; they became known as "rope-stretchers." The Greeks would later call them *geometers*, meaning "Earth-measurers."

Unfortunately, we have little written record of the Egyptians' mathematical knowledge. Only two mathematical documents survived from this period. The one we are concerned with is called the Rhind Mathematical Papyrus, named after the 19th-century Scottish collector who bought it in Egypt. It is a document from about 1650 BC written by a scribe named Ahmes, which contains a series of arithmetic and geometry problems, together with some tables for computation. This scroll contains the first recorded algorithm, a technique for fast multiplication, along with a second one for fast division. Let's begin by looking at the fast multiplication algorithm, which (as we shall see later in the book) is still an important computational technique today.

2.1 Egyptian Multiplication

The Egyptians' number system, like that of all ancient civilizations, did not use positional notation and had no way to represent zero. As a result, multiplication was extremely difficult, and only a few trained experts knew how to do it. (Imagine doing multiplication on large numbers if you could only manipulate something like Roman numerals.)

How do we define multiplication? Informally, it's "adding something to itself a number of times." Formally, we can define multiplication by breaking it into two cases: multiplying by 1, and multiplying by a number larger than 1.

We define multiplication by 1 like this:

$$1a = a \tag{2.1}$$

Next we have the case where we want to compute a product of one more thing than we already computed. Some readers may recognize this as the process of induction; we'll use that technique more formally later on.

$$(n + 1)a = na + a \tag{2.2}$$

One way to multiply n by a is to add instances of a together n times. However, this could be extremely tedious for large numbers, since $n - 1$ additions are required. In C++, the algorithm looks like this:

```
int multiply0(int n, int a) {
    if (n == 1) return a;
    return multiply0(n - 1, a) + a;
}
```

The two lines of code correspond to equations 2.1 and 2.2. Both a and n must be positive, as they were for the ancient Egyptians.

The algorithm described by Ahmes—which the ancient Greeks knew as "Egyptian multiplication" and which many modern authors refer to as the "Russian Peasant Algorithm"[1]—relies on the following insight:

$$\begin{aligned} 4a &= ((a + a) + a) + a \\ &= (a + a) + (a + a) \end{aligned}$$

This optimization depends on the law of associativity of addition:

$$a + (b + c) = (a + b) + c$$

It allows us to compute $a + a$ only once and reduce the number of additions.

The idea is to keep halving n and doubling a, constructing a sum of power-of-2 multiples. At the time, algorithms were not described in terms of variables such as a and n; instead, the author would give an example and then say, "Now do the same thing for other numbers." Ahmes was no exception; he demonstrated the algorithm by showing the following table for multiplying $n = 41$ by $a = 59$:

1	✓	59
2		118
4		236
8	✓	472
16		944
32	✓	1888

Each entry on the left is a power of 2; each entry on the right is the result of doubling the previous entry (since adding something to itself is relatively easy). The checked values correspond to the 1-bits in the binary representation of 41. The table basically says that

$$41 \times 59 = (1 \times 59) + (8 \times 59) + (32 \times 59)$$

where each of the products on the right can be computed by doubling 59 the correct number of times.

The algorithm needs to check whether n is even and odd, so we can infer that the Egyptians knew of this distinction, although we do not have direct proof. But ancient Greeks, who claimed that they learned their mathematics from the

[1]Many computer scientists learned this name from Knuth's *The Art of Computer Programming*, which says that travelers in 19th-century Russia observed peasants using the algorithm. However, the first reference to this story comes from a 1911 book by Sir Thomas Heath, which actually says, "I have been told that there is a method in use today (some say in Russia, but *I have not been able to verify this*), ..."

Egyptians, certainly did. Here's how they defined[2] even and odd, expressed in modern notation:[3]

$$n = \frac{n}{2} + \frac{n}{2} \implies \text{even}(n)$$

$$n = \frac{n-1}{2} + \frac{n-1}{2} + 1 \implies \text{odd}(n)$$

We will also rely on this requirement:

$$\text{odd}(n) \implies \text{half}(n) = \text{half}(n-1)$$

This is how we express the Egyptian multiplication algorithm in C++:

```
int multiply1(int n, int a) {
    if (n == 1) return a;
    int result = multiply1(half(n), a + a);
    if (odd(n)) result = result + a;
    return result;
}
```

We can easily implement `odd(x)` by testing the least significant bit of x, and `half(x)` by a single right shift of x:

```
bool odd(int n) { return n & 0x1; }
int half(int n) { return n >> 1; }
```

How many additions is `multiply1` going to do? Every time we call the function, we'll need to do the addition indicated by the `+` in `a + a`. Since we are halving the value as we recurse, we'll invoke the function $\log n$ times.[4] And some of the time, we'll need to do another addition indicated by the `+` in `result + a`. So the total number of additions will be

$$\#_+(n) = \lfloor \log n \rfloor + (\nu(n) - 1)$$

where $\nu(n)$ is the number of 1s in the binary representation of n (the *population count* or *pop count*). So we have reduced an $O(n)$ algorithm to one that is $O(\log n)$.

[2]The definition appears in the 1st-century work *Introduction to Arithmetic*, Book I, Chapter VII, by Nicomachus of Gerasa. He writes, "The even is that which can be divided into two equal parts without a unit intervening in the middle; and the odd is that which cannot be divided into two equal parts because of the aforesaid intervention of a unit."

[3]The arrow symbol "$\implies$" is read "implies." See Appendix A for a summary of the mathematical notation used in this book.

[4]Throughout this book, when we write "log," we mean the base 2 logarithm, unless specified otherwise.

Is this algorithm optimal? Not always. For example, if we want to multiply by 15, the preceding formula would give this result:

$$\#_+(15) = 3 + 4 - 1 = 6$$

But we can actually multiply by 15 with only 5 additions, using the following procedure:

```
int multiply_by_15(int a) {
    int b = (a + a) + a;      // b == 3*a
    int c = b + b;            // c == 6*a
    return (c + c) + b;       // 12*a + 3*a
}
```

Such a sequence of additions is called an *addition chain*. Here we have discovered an optimal addition chain for 15. Nevertheless, Ahmes's algorithm is good enough for most purposes.

Exercise 2.1. Find optimal addition chains for $n < 100$.

At some point the reader may have observed that some of these computations would be even faster if we first reversed the order of the arguments when the first is greater than the second (for example, we could compute 3×15 more easily than 15×3). That's true, and the Egyptians knew this. But we're not going to add that optimization here, because as we'll see in Chapter 7, we're eventually going to want to generalize our algorithm to cases where the arguments have different types and the order of the arguments matters.

2.2 Improving the Algorithm

Our `multiply1` function works well as far as the number of additions is concerned, but it also does $\lfloor \log n \rfloor$ recursive calls. Since function calls are expensive, we want to transform the program to avoid this expense.

One principle we're going to take advantage of is this: *It is often easier to do more work rather than less.* Specifically, we're going to compute

$$r + na$$

where r is a running result that accumulates the partial products na. In other words, we're going to perform *multiply-accumulate* rather than just multiply. This principle turns out to be true not only in programming but also in hardware design and in mathematics, where it's often easier to prove a general result than a specific one.

Here's our multiply-accumulate function:

```
int mult_acc0(int r, int n, int a) {
    if (n == 1) return r + a;
    if (odd(n)) {
        return mult_acc0(r + a, half(n), a + a);
    } else {
        return mult_acc0(r, half(n), a + a);
    }
}
```

It obeys the invariant: $r + na = r_0 + n_0 a_0$, where r_0, n_0 and a_0 are the initial values of those variables.

We can improve this further by simplifying the recursion. Notice that the two recursive calls differ only in their first argument. Instead of having two recursive calls for the odd and even cases, we'll just modify the value of r before we recurse, like this:

```
int mult_acc1(int r, int n, int a) {
    if (n == 1) return r + a;
    if (odd(n)) r = r + a;
    return mult_acc1(r, half(n), a + a);
}
```

Now our function is *tail-recursive*—that is, the recursion occurs only in the return value. We'll take advantage of this fact shortly.

We make two observations:

- n is rarely 1.
- If n is even, there's no point checking to see if it's 1.

So we can reduce the number of times we have to compare with 1 by a factor of 2, simply by checking for oddness first:

```
int mult_acc2(int r, int n, int a) {
    if (odd(n)) {
        r = r + a;
        if (n == 1) return r;
    }
    return mult_acc2(r, half(n), a + a);
}
```

Some programmers think that compiler optimizations will do these kinds of transformations for us, but that's rarely true; they do not transform one algorithm into another.

What we have so far is pretty good, but we're eventually going to want to eliminate the recursion to avoid the function call overhead. This is easier if the function is strictly tail-recursive.

Definition 2.1. A **strictly tail-recursive** procedure is one in which all the tail-recursive calls are done with the formal parameters of the procedure being the corresponding arguments.

Again, we can achieve this simply by assigning the desired values to the variables we'll be passing before we do the recursion:

```
int mult_acc3(int r, int n, int a) {
    if (odd(n)) {
        r = r + a;
        if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
    return mult_acc3(r, n, a);
}
```

Now it is easy to convert this to an iterative program by replacing the tail recursion with a `while(true)` construct:

```
int mult_acc4(int r, int n, int a) {
    while (true) {
        if (odd(n)) {
            r = r + a;
            if (n == 1) return r;
        }
        n = half(n);
        a = a + a;
    }
}
```

With our newly optimized multiply-accumulate function, we can write a new version of multiply. Our new version will invoke our multiply-accumulate helper function:

```
int multiply2(int n, int a) {
    if (n == 1) return a;
    return mult_acc4(a, n - 1, a);
}
```

Notice that we skip one iteration of `mult_acc4` by calling it with result already set to a.

This is pretty good, except when *n* is a power of 2. The first thing we do is subtract 1, which means that `mult_acc4` will be called with a number whose binary representation is all 1s, the worst case for our algorithm. So we'll avoid this by doing some of the work in advance when *n* is even, halving it (and doubling *a*) until *n* becomes odd:

```
 int multiply3(int n, int a) {
    while (!odd(n)) {
        a = a + a;
        n = half(n);
    }
    if (n == 1) return a;
    return mult_acc4(a, n - 1, a);
}
```

But now we notice that we're making `mult_acc4` do one unnecessary test for $n = 1$, because we're calling it with an even number. So we'll do one halving and doubling on the arguments before we call it, giving us our final version:

```
int multiply4(int n, int a) {
    while (!odd(n)) {
        a = a + a;
        n = half(n);
    }
    if (n == 1) return a;
    // even(n − 1) ⟹ n − 1 ≠ 1
    return mult_acc4(a, half(n - 1), a + a);
}
```

Rewriting Code

As we have seen with our transformations of the multiply algorithm, rewriting code is important. No one writes good code the first time; it takes many iterations to find the most efficient or general way to do something. No programmer should have a single-pass mindset.

At some point during the process you may have been thinking, "One more operation isn't going to make a big difference." But it may turn out that your code will be reused many times for many years. (In fact, a temporary hack often becomes the code that lives the longest.) Furthermore, that inexpensive operation you're saving now may be replaced by a very costly one in some future version of the code.

Another benefit of striving for efficiency is that the process forces you to understand the problem in more depth. At the same time, this increased depth of understanding leads to more efficient implementations; it's a virtuous circle.

2.3 Thoughts on the Chapter

Students of elementary algebra learn how to keep transforming expressions until they can be simplified. In our successive implementations of the Egyptian multiplication algorithm, we've gone through an analogous process, rearranging the code to make it clearer and more efficient. Every programmer needs to get in the habit of trying code transformations until the final form is obtained.

We've seen how mathematics emerged in ancient Egypt, and how it gave us the first known algorithm. We're going to return to that algorithm and expand on it quite a bit later in the book. But for now we're going to move ahead more than a thousand years and take a look at some mathematical discoveries from ancient Greece.

This page intentionally left blank

3

three

Ancient Greek Number Theory

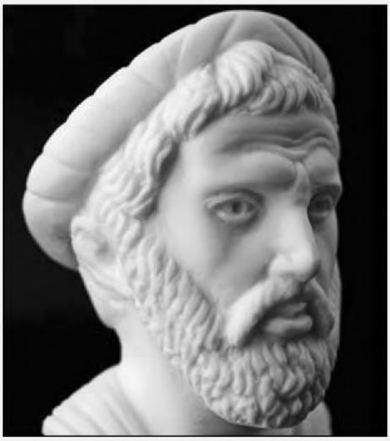
Pythagoreans applied themselves to the study of mathematics. . . .
They thought that its principles must be the principles of all existing things.
Aristotle, *Metaphysics*

In this chapter, we're going to look at some of the problems studied by ancient Greek mathematicians. Their work on patterns and "shapes" of numbers led to the discovery of prime numbers and the beginnings of a field of mathematics called *number theory*. They also discovered paradoxes that ultimately produced some mathematical breakthroughs. Along the way, we'll examine an ancient algorithm for finding primes, and see how to optimize it.

3.1 Geometric Properties of Integers

Pythagoras, the Greek mathematician and philosopher who most of us know only for his theorem, was actually the person who came up with the idea that understanding mathematics is necessary to understand the world. He also discovered many interesting properties of numbers; he considered this understanding to be of great value in its own right, independent of any practical application. According to Aristotle's pupil Aristoxenus, "He attached supreme importance to the study of arithmetic, which he advanced and took out of the region of commercial utility."

Pythagoras (ca. 570 BC–ca. 490 BC)



Pythagoras was born on the Greek island of Samos, which was a major naval power at the time. He came from a prominent family, but chose to pursue wisdom rather than wealth. At some point in his youth he traveled to Miletus to study with Thales, the founder of philosophy (see Section 9.2), who advised him to go to Egypt and learn the Egyptians' mathematical secrets.

During the time Pythagoras was studying abroad, the Persian empire conquered Egypt. Pythagoras followed the Persian army eastward to Babylon (in what is now Iraq), where he learned Babylonian mathematics and astronomy. While there, he may have met travelers from India; what we know is that he was exposed to and began espousing ideas we typically associate with Indian religions, including the transmigration of souls, vegetarianism, and asceticism. Prior to Pythagoras, these ideas were completely unknown to the Greeks.

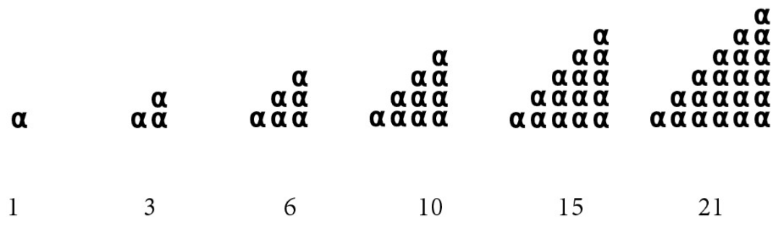
After returning to Greece, Pythagoras started a settlement in Croton, a Greek colony in southern Italy, where he gathered followers—both men and women—who shared his ideas and followed his ascetic lifestyle. Their lives were centered on the study of four things: astronomy, geometry, number theory, and music. These four subjects, later known as the *quadrivium*, remained a focus of European education for 2000 years. Each of these disciplines was related: the motion of the stars could be mapped geometrically, geometry could be grounded in numbers, and numbers generated music. In fact, Pythagoras was the first to discover the numerical structure of frequencies in musical octaves. His followers said that he could "hear the music of the celestial spheres."

After the death of Pythagoras, the Pythagoreans spread to several other Greek colonies in the area and developed a large body of mathematics. However, they kept their teachings secret, so many of their results may have been lost. They also eliminated competition within their ranks by crediting all discoveries to Pythagoras himself, so we don't actually know which individuals did what.

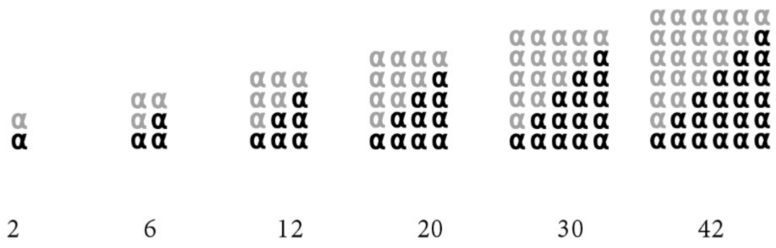
Although the Pythagorean communities were gone after a couple of hundred years, their work remains influential. As late as the 17th century, Leibniz (one of the inventors of calculus) described himself as a Pythagorean.

Unfortunately, Pythagoras and his followers kept their work secret, so none of their writings survive. However, we know from contemporaries what some of his discoveries were. Some of these come from a first-century book called *Introduction to Arithmetic* by Nicomachus of Gerasa. These included observations about geometric properties of numbers; they associated numbers with particular shapes.

Triangular numbers, for example, which are formed by stacking rows representing the first n integers, are those that formed the following geometric pattern:



Oblong numbers are those that look like this:



It is easy to see that the nth oblong number is represented by an $n \times (n+1)$ rectangle:

$$\square_n = n(n+1)$$

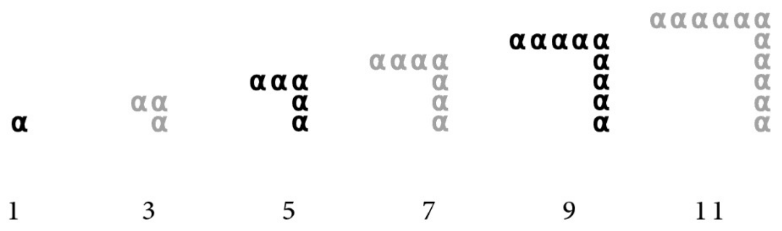
It's also clear geometrically that each oblong number is twice its corresponding triangular number. Since we already know that triangular numbers are the sum of the first n integers, we have

$$\square_n = 2\triangle_n = 2\sum_{i=1}^{n} i = n(n+1)$$

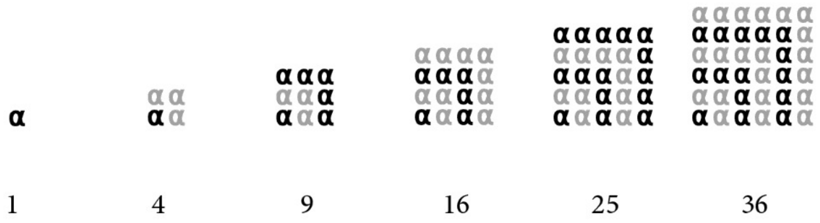
So the geometric representation gives us the formula for the sum of the first n integers:

$$\triangle_n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Another geometric observation is that the sequence of odd numbers forms the shape of what the Greeks called *gnomons* (the Greek word for a carpenter's square; a gnomon is also the part of a sundial that casts the shadow):



Combining the first n gnomons creates a familiar shape—a square:



This picture also gives us a formula for the sum of the first n odd numbers:

$$\square_n = \sum_{i=1}^{n} (2i - 1) = n^2$$

Exercise 3.1. Find a geometric proof for the following: take any triangular number, multiply it by 8, and add 1. The result is a square number. (This problem comes from Plutarch's *Platonic Questions*.)

3.2 Sifting Primes

Pythagoreans also observed that some numbers could not be made into any nontrivial rectangular shape (a shape where both sides of the rectangle are greater than 1). These are what we now call *prime numbers*—numbers that are not products of smaller numbers:

$$2, 3, 5, 7, 11, 13, \ldots$$

("Numbers" for the Greeks were always whole numbers.) Some of the earliest observations about primes come from Euclid. While he is usually associated with geometry, several books of Euclid's *Elements* actually discuss what we now call number theory. One of his results is this theorem:

Theorem 3.1 (Euclid VII, 32): *Any number is either prime or divisible by some prime.*

The proof, which uses a technique called "impossibility of infinite descent," goes like this:[1]

Proof. Consider a number A. If it is prime, then we are done. If it is composite (i.e., nonprime), then it must be divisible by some smaller number B. If B is prime, we are done (because if A is divisible by B and B is prime, then A is divisible by a prime). If B is composite, then it must be divisible by some smaller number C, and so on. Eventually, we will find a prime or, as Euclid remarks in his proof of the previous proposition, "an infinite sequence of numbers will divide the number, each of which is less than the other; and this is impossible." □

This Euclidean principle that *any descending sequence of natural numbers terminates* is equivalent to the induction axiom of natural numbers, which we will encounter in Chapter 9.

* * *

Another result, which some consider the most beautiful theorem in mathematics, is the fact that there are infinitely many primes:

Theorem 3.2 (Euclid IX, 20): *For any sequence of primes $\{p_1, \ldots, p_n\}$, there is a prime p not in the sequence.*

Proof. Consider the number

$$q = 1 + \prod_{i=1}^{n} p_i$$

[1] Euclid's proof of VII, 32 actually relies on his proposition VII, 31 (any composite number is divisible by some prime), which contains the reasoning shown here.

In other words, the aliquot sum is the sum of all *proper* divisors of n—all the divisors except n itself.

Now we're ready for the proof of Theorem 3.3, also known as Euclid IX, 36:

If $2^n - 1$ *is prime, then* $2^{n-1}(2^n - 1)$ *is perfect.*

Proof. Let $q = 2^{n-1}(2^n - 1)$. We know 2 is prime, and the theorem's condition is that $2^n - 1$ is prime, so $2^{n-1}(2^n - 1)$ is already a prime factorization of the form $n = p_1^{a_1} p_2^{a_2} \dots p_m^{a_m}$, where $m = 2, p_1 = 2, a_1 = n - 1, p_2 = 2^n - 1$, and $a_2 = 1$. Using the sum of divisors formula (Equation 3.6):

$$\begin{aligned}
\sigma(q) &= \frac{2^{(n-1)+1} - 1}{1} \cdot \frac{(2^n - 1)^2 - 1}{(2^n - 1) - 1} \\
&= (2^n - 1) \cdot \frac{(2^n - 1)^2 - 1}{(2^n - 1) - 1} \cdot \frac{(2^n - 1) + 1}{(2^n - 1) + 1} \\
&= (2^n - 1) \cdot \frac{((2^n - 1)(2^n - 1) - 1)((2^n - 1) + 1)}{((2^n - 1)(2^n - 1) - 1)} \\
&= (2^n - 1)((2^n - 1) + 1) \\
&= 2^n(2^n - 1) = 2 \cdot 2^{n-1}(2^n - 1) = 2q
\end{aligned}$$

Then

$$\alpha(q) = \sigma(q) - q = 2q - q = q$$

That is, q is perfect. □

We can think of Euclid's theorem as saying that if a number has a certain form, then it is perfect. An interesting question is whether the converse is true: if a number is perfect, does it have the form $2^{n-1}(2^n - 1)$? In the 18th century, Euler proved that if a perfect number is even, then it has this form. He was not able to prove the more general result that *every* perfect number is of that form. Even today, this is an unsolved problem; we don't know if any odd perfect numbers exist.

Exercise 3.7. Prove that every even perfect number is a triangular number.

Exercise 3.8. Prove that the sum of the reciprocals of the divisors of a perfect number is always 2. Example:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{6} = 2$$

3.5 The Pythagorean Program

For Pythagoreans, mathematics was not about abstract symbol manipulation, as it is often viewed today. Instead, it was the science of numbers and space—the