Three Algorithmic Journeys

* * * DRAFT 0.2 * * *

Alexander A. Stepanov
Daniel E. Rose

December 21, 2012

Contents

Authors' Note

The book you are about to read is based on notes from a course taught by Alex Stepanov at A9.com during 2012. However, it is not a transcript of the lectures; rather, it is a re-imagining of the course material in prose. In some cases topics have been reorganized and details have been added or removed to better suit the needs of the medium and to make the material more accessible to a less mathematically advanced reader. 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 that a mathematician would, or using slightly different terminology, the fault is mine.

Those who have heard about the course or seen the lectures might notice that the course was originally called *Four* Algorithmic Journeys, while this book contains only three. As the course evolved, Alex realized that the three journeys presented here form a coherent whole: each journey has an associated branch of mathematics (Number Theory, Abstract Algebra, and Logic/Set Theory, respectively), which the originally imagined fourth journey did not have.

[The current work is only a draft. You will find many places where there are parenthetical notes like "More explanation here" or "Biography goes here." These notes, which are placeholders for changes and editions in the final version, are shown in square brackets.]

— D.E.R.

This book is about journeys — the algorithmic journeys of the title, but also my own journey through life, encountering towering figures of mathematics through their work; along the way, I met wonderful people like Pythagoras, Plato, Euclid, and Archimedes, Fermat, Descartes, Euler, Gauss, and Cauchy, and now I'm sharing what they told me with you. I will try to present their message within our setting of modern programming.

Although I have always enjoyed mathematics and was fortunate enough to study with some great mathematicians early in my life, the ideas really came alive when I started to learn the history of the field and some of its heroes. So that's the approach we're going to take here. I have always found that learning the connections and the context of an idea

helps me understand it better. Hopefully some of you will find this helpful as well.

Many have been conditioned to believe that mathematics is boring, but I hope to show you that it isn't. If at the end of the book, some readers decide to buy a copy of Euclid's *Elements*, I'll know that I've been successful.

— A.A.S.

Prologue

> God always does geometry.
>
> – Plato [cite]

> But thou hast arranged all things by measure and number and weight.
>
> – Wisdom of Solomon 11:20 [cite]

> It is impossible to know about this world without knowing mathematics.
>
> – Roger Bacon [cite]

This book might be described as "math that programmers should know." At first glance it might seem that programmers don't need to know much about math; it's certainly possible to be a programmer without it. In fact, you might argue that very few jobs actually require knowledge of mathematics, so there's not much point in studying it.

However, this attitude is relatively recent. Studying mathematics used to be considered an essential part of learning how to think clearly. Great leaders of the United States held this view. Abraham Lincoln, for example, who grew up in a rural setting without much education, decided as an adult to develop his mind by reading Euclid "to improve his logic and language." [cite] In fact, he even referred to one of Euclid's geometric proofs during the famous Lincoln-Douglas debates. (Just try to imagine any of our current political leaders doing this!) Thomas Jefferson was another advocate. He insisted that a "thorough knowledge of Euclid" be an admission requirement to the University of Virginia [cite]. We hope to show how understanding a mathematical approach to problems can help improve the way you think about the programs you write.

In particular, we're going to look at three areas of mathematics which most non-math majors have never encountered: number theory (which deals with properties of integers, especially divisibility), abstract algebra (which shows how to reason about objects only in terms of abstract properties of operations on them), and set theory and logic (which

considers collections of objects and axioms describing them).

At the same time, this book is really about algorithms. It's about the three specific algorithms mentioned in the title, but more generally it's also about how to implement and refine an algorithm, how to generalize it to the broadest possible set of applications, and how to define the programming interface for a piece of code. The algorithms we're going to focus on are:

- Journey 1: Multiplication
- Journey 2: Division with remainder
- Journey 3: Adding 1

You're probably thinking, "but I learned these things in grade school!" Yes. What we plan to show you is that these simple algorithms played a profound role in the development of mathematics, and that these algorithms are still essential to your work as a programmer today.

Rather than explaining mathematical theories or a general formulation of an algorithm, we're going to take a historical perspective, tracing the evolution of these ideas from ancient times to the present. We're also going to tell you about some of the remarkable people who came up with these ideas, providing brief biographies of many of the mathematicians we discuss. These aren't encyclopedia entries, but our own perspective on what made this person interesting. Hopefully you'll find that this historical context helps bring some of these ideas to life and helps you understand the thought process that led to the discoveries. (The biographies are in shaded boxes; you can skip them without missing any of the core material.)

So, is this a math book, a programming book, or a history book? The answer is "none of the above"; it's a historically informed mathematical book for programmers. As we shall see, we'll interweave important ideas in mathematics with a discussion of both specific algorithms and general programming techniques. These topics will be presented from a historical perspective, so you can see how each idea evolved and why it is the way it is today. Given the wide range of topics, we can only give the briefest sketches of each idea. We hope that interested readers will explore these ideas more deeply on their own. We provide further reading suggestions at the end of the book.

Prerequisites

Throughout this book, we'll show implementations of algorithms in working C++ code. You don't need to be a C++ programmer to understand these examples, but you should be able to read a few lines of code. We mostly use a subset of C++ that should be familiar and

understandable to anyone who has programmed in a typical imperative language like C or Java. In cases where we rely on some C++-specific features, we'll refer to an appendix explaining how they are used.

This book also contains mathematical proofs. They are written so that you don't need any previous knowledge to follow them. Nevertheless, you'll probably find them easier to understand if you have some experience with the idea of formal proofs (for example, in a theory of computation class).

Many of the topics in this book are also covered much more formally in *Elements of Programming* by Stepanov and McJones [cite]. Readers interested in this additional depth may find that book to be a useful complement to this one.

Journey One: Legacy of Ahmes

Chapter 1

Egyptian Multiplication

This book follows three seminal algorithmic ideas and shows how they led to discoveries in both computing and mathematics, and how they can be applied to a variety of practical problems.

In our first journey, "Legacy of Ahmes," we'll see how the elementary properties of commutativity and associativity led to fundamental algorithmic and mathematical discoveries. We'll also cover some important results in a branch of mathematics called number theory, which focuses divisibility of integers, and see how this led to advances in modern cryptography. This chapter starts our journey by looking at an efficient algorithm for multiplication discovered by ancient Egyptians.

1.1 Proofs

From time immemorial people knew that addition obeys:

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

and multiplcation obeys:

$$ab = ba$$
$$(ab)c = a(bc)$$

Nobody "proved" these statements; they were considered to be *self-evident* propositions. People have happily relied on these for centuries. In fact, the first rigorous proof of commutativity and associativity of plus and times wasn't published until 1861 — and only then in a book by an obscure high school teacher named Hermann Grassmann. Everyone

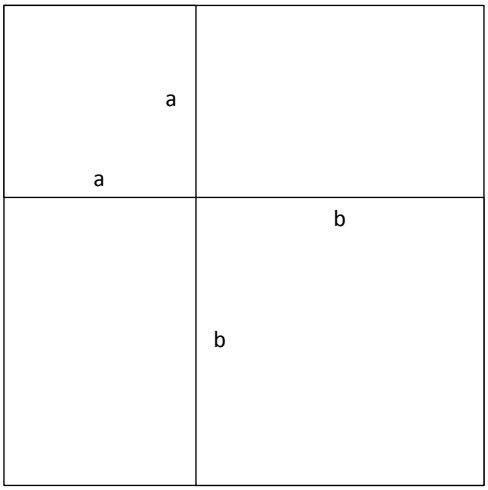
ignored it. After all (they thought), what could a high school teacher know about mathematics? It was almost 30 years before Richard Dedekind rediscovered the proof in 1888 which is still considered rigorous by today's mathematicians.[1]

While proving things is good, discovering things is better. It was the discovery of associativity that allowed people to successfully use it for thousands of years.

For centuries mathematicians relied on *geometric* proofs. The Greeks realized that they could use spatial reasoning to proof algebraic facts. They appeal to our innate intuition of space.

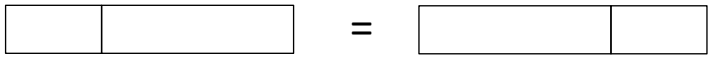
Here are some examples of some important geometric proofs.

$(a+b)^2 = a^2 + 2ab + b^2$:



It's clear just by looking that the rectangle on the lower left is the same area as the rectangle on the upper right — not only do they both have area ab, but you could literally cut one out, turn it sideways, and lay it on the other.

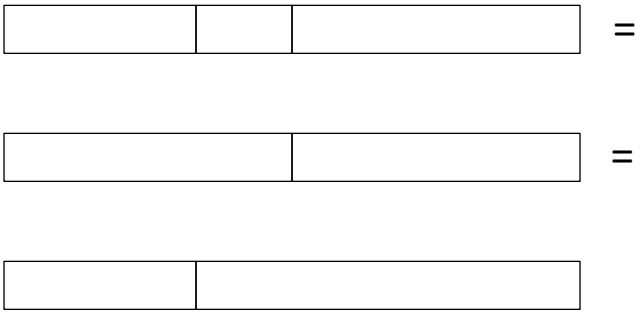
Commutativity of addition:



This isn't really a formal proof, but an appeal to our intuition.

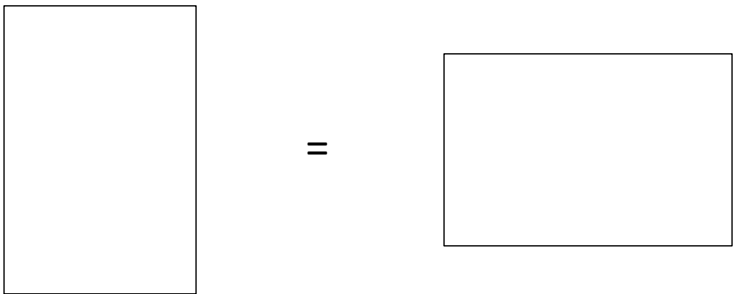
[1] We will examine the proof during our third journey.

Associativity of addition:



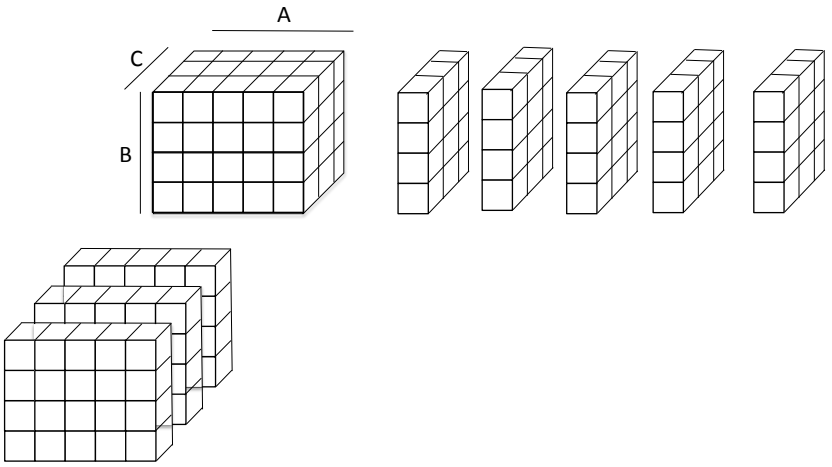
This one is also an appeal to our intuition.

Commutativity of multiplication:



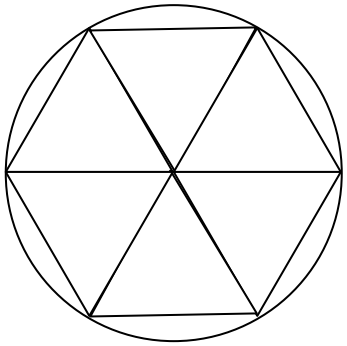
This one might really qualify as a proof. In fact, this essential argument appears in a book by Dirichlet, a great 19th-century mathematician, who says that whether you arrange soldiers in rows or columns, you still have the same number.

Associativity of multiplication:



Whether you slice this rectangular prism along one axis or along another, when you put the slices back together, you still have the same volume.

$\pi > 3$:



Here we've inscribed a regular unit hexagon (one whose sides are all of length 1) in the circle. It's evident that the perimeter of the hexagon is shorter than the circumference of the circle, because whenever we have two intersection points between the two figures, the shortest path from one point to the next is along the hexagon, not the circle. Since the triangles that make up the hexagon are equilateral, all their sides are length 1, so the diameter of the circle is length 2. So the ratio of the circle's circumference to its diameter (i.e. π) must be greater than the ratio of the hexagon's perimeter (6) to its diameter (2).

Exercise 1.1

Design geometric proofs for the following:

$$(a-b)^2 = a^2 - 2ab + b^2$$
$$a^2 + b^2 = (a+b)(a-b)$$
$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$
$$(a-b)^3 = a^3 - 3a^2b + 3ab^2 - b^3$$

Exercise 1.2

Using geometry, find formulas for:

- Sum of the first n positive integers.
- Sum of the first n positive odd integers.

We'll see the solution in Journey 2.

Exercise 1.3

Find a geometric upper bound for π.

Today, we use the following definition:

Definition 1.1. *A proof of a proposition is*

- *an argument*
- *accepted by the mathematical community*
- *to establish the proposition as valid.*

The second point is often overlooked: proof is a fundamentally social activity, and one that changes over time. Our confidence in a proof increases as more people understand and agree with it; proof is a social process. At the same time, what is considered a valid proof today might not be considered a valid proof three hundred years from now, just as many proofs that were viewed as valid by Euler — the greatest 18th century mathematician — are frowned upon today.

Some people believe that proof — at least in the case of program correctness — is a formal process than can be done mechanically. But if you told computer-literate people that the software controlling an airplane was proven correct by a verification program, they probably wouldn't want to fly on it, because they wouldn't trust the verification program.

[Further reading: DeMillo .]

1.2 Mathematical Traditions Around the World

It's a remarkable fact that there is no civilization without mathematics. All civilizations developed number systems, which were a fundamental requirement for two critical civic activities: collecting taxes, and computing calendars to determine cultivation dates.

Furthermore, all civilizations developed common mathematical concepts, such as Pythagorean triples.[2] Van der Waerden [cite] proposed that these ideas came from a common Neolithic source around 3000 BC and spread through Babylonia, China, and India. However, there is no evidence for this claim, and geography argues against it — how would Inca mathematics have been connected with Chinese mathematics? It seems more likely that this is simply the mathematical equivalent of convergent evolution, where the same characteristics evolve independently in unrelated species. The fact that these same mathematical ideas were rediscovered independently suggests their fundamental nature.

Many civilizations developed important mathematical traditions at some point in their history. For example, in China 3rd-century mathematician and poet Liu Hui wrote important commentaries on an earlier book, *Nine Chapters on the Mathematical Art*, and extended the work. Among other discoveries, he demonstrated that the value of π must be greater than 3, and provided several geometric techniques for surveying. In India, 5th-century mathematician and astronomer Aryabhata wrote a foundational text called the *Aryabhatiya*, which included algorithms for computing square and cube roots, as well as geometric techniques. These are people we should know about. However, Computer Science is rooted in the European mathematical tradition (which was heavily influenced by Arab, Persian, and Hebrew scholars, all writing in Arabic), so that is the one we will focus on. People like Von Neumann and Turing are part of this tradition, and so are we — whatever our national origin today might be. The other mathematical traditions did not survive until today.

1.3 Thales and Egyptian Mathematics

> ...the mathematical arts were founded in Egypt...
>
> Aristotle, *Metaphysics*, A 981b23-2 [cite]

> [Moses] speedily learned arithmetic, and geometry... This knowledge he derived from the Egyptians, who study mathematics above all things...
>
> Philo, *De Vita Mosis*, I, 23-4 [cite]

While much of our early mathematics comes from ancient Greece, the Greeks themselves believed that their knowledge was derived from that of the Egyptians. Greek civilization was just starting, while Egyptian civilization had already existed for thousands

[2] These are sets of three numbers a, b, c where $a^2 + b^2 = c^2$.

of years. As we will see, many leading thinkers of Greece would travel to Egypt to study with their priests and learn their wisdom.

The first such person known to us is Thales of Miletus (635BC-545BC). Thales was the founder of Western philosophy, and might be considered the first mathematician. He went to Egypt to learn the science of what the Egyptians called "rope stretchers," i.e. surveyors. Greeks called them geometers ("earth measurers").

The rope stretchers played an important part in Egyptian society, which was centered on the Nile and whose 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. So the rope stretchers — the priests who knew geometry — could go back to the records and reconstruct the boundaries.

While the Egyptians had algorithms, Thales had a theorem — in fact, he invented the very notion of a theorem.

Thales of Miletus (635BC–545 BC)

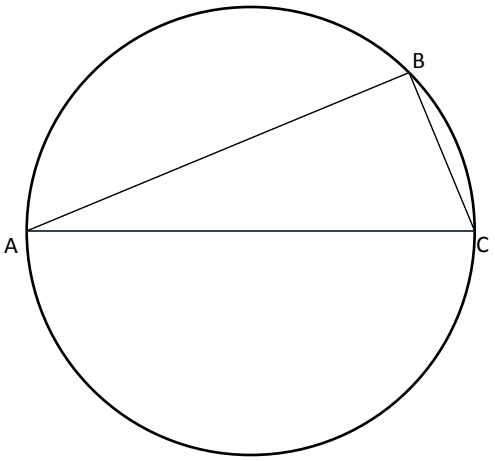
Some time around the year 750BC a new people started to appear in different coastal regions of the Mediterranean and even as far north as the Black Sea. They called themselves Hellenes — we call them Greeks. They came from a small mountainous country where the geography prevented the emergence of a large unified kingdom as was common elsewhere. Greeks lived in small, independent city-states unified not by a central government, but by a common language and a common mythology of gods and heroes as told in Homer's epics, *The Iliad* and *The Odyssey*. Whenever a city's population exceeded its resources, it would send a portion of its population to settle a colony, a new practically independent city on some conveniently located bay with a river. Within two hundred years they settled around the Mediterranean — according to Plato — "like frogs around a pond."

By somewhere around 600BC the colonies in Asia Minor (what is now Turkey) were getting wealthy. But instead of spending all the extra money on luxuries, some of them started supporting intellectual pursuits. For the first time in history they looked beyond mythology for the answers to eternal questions such as "what are we made of?". The first person to do this in a fundamental way was Thales of Miletus. Thales was the originator of what Greeks called philosophy and what we now call science. He wanted to find the natural, non-mythological explanation of reality. He proposed that all visible reality is made out of one single substance: water. Therefore, visible reality exists in one of three sates: gas, liquid, or solid and there are transitions between the states.

While in Egypt, Thales collected many geometric theorems and, probably, some Babylonian astronomical knowledge. Herodotus reports that Thales was able to predict a total solar eclipse a year in advance. Aristotle — usually a very reliable source — tells us that

Thales was able to predict an exceptionally large harvest of olives and, by buying options on the use of all the olive presses in the region, made a fortune. There are many other stories about his accomplishments, such as his discovery of static electricity. While we do not know exactly which stories are true, he clearly amassed a large body of scientific knowledge and was able to apply it to practical problems. His knowledge did not perish with him; his students carried the program forward. More importantly, all of modern science traces back to Thales.

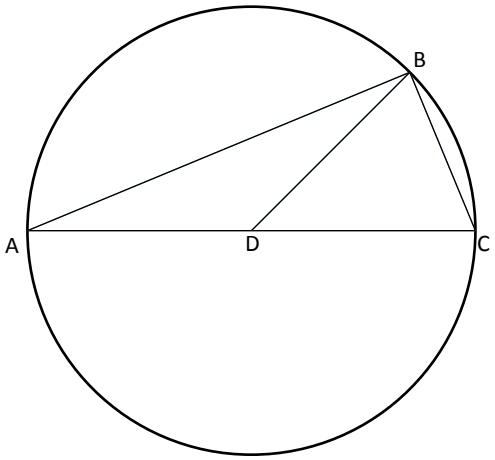
Thales' theorem states that for any triangle ABC formed by connecting the two ends of a circle's diameter (AC) with any other point B on the circle, $\angle ABC = 90^\circ$.



The proof is as follows:

$$\begin{aligned}
&\angle DAB = \angle DBA \\
&\angle DCB = \angle DBC \\
&\angle DAB + \angle DCB = \angle DBA + \angle DBC \\
&\angle DAB + \angle DCB + \angle DBA + \angle DBC = 180^\circ \\
&(\angle DBA + \angle DBC) + (\angle DBA + \angle DBC) = 180^\circ \\
&\angle DBA + \angle DBC = 90^\circ \\
&\angle CBA = 90^\circ
\end{aligned}$$

[Include explanations for each step.]



Although the Greeks tell us that ancient Egypt was the center of mathematical knowledge, we have very little written record of this knowledge. Only two mathematical documents survived from this period. The one we are concerned with is the Rhind Mathematical Papyrus, 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. Among these are two algorithms, one for fast multiplication and one for fast division. The first algorithm will be our topic for the rest of this chapter; we will consider the second in our next journey.

1.4 Ahmes' Multiplication Algorithm

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. (In fact, arithmetic computations were so complex that the Egyptians invented double-entry bookkeeping, to reduce the likelihood of errors.)

Every multiplication algorithm depends on *distributivity*, which the Egyptians knew about:

$$(n + m)a = na + ma \tag{1.1}$$

This allows us to *reduce* the problem. It starts with the inductive base:

$$1a = a \tag{1.2}$$

Can we prove this? No. As we shall see in the third journey, it is an axiom — it is part of the definition of multiplication. Next we have the inductive step:

$$(n + 1)a = na + a \tag{1.3}$$

It may be tempting to take equation 1.1 as an axiom and then do the induction as a theorem. But as we shall see later, modern mathematicians do not do this, because they view 1.1 as less self-evident than 1.2.

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 $O(n)$ additions are required. In C++, the algorithm looks like this:

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

The two lines of code correspond exactly to axioms 1.2 and 1.3 above. Both a and n must be positive.

The algorithm described by Ahmes — which the Greeks knew as "Egyptian multiplication" and which many modern authors refer to as the "Russian Peasant Algorithm,"[3] relies on the following insight:

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

By relying on associativity, we can compute $a + a$ only once and reduce the number of additions.

Our idea will be 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 x and y; 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 41 by 59:

1	✓	59
2		118
3		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 was relatively easy). The checked values

[3] Many computer scientists learned this name from Knuth's *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*), ..." [cite]

are what today we observe are 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 depends on knowing the difference between even and odd. Here's how the Greeks defined it:

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

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

We also rely on this requirement:

$$odd(n) \implies half(n) = half(n-1)$$

This is how we would express the algorithm today in C++:

```
// Fast Multiply Algorithm
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.

How many additions are we going to do? Every time we call the function, we'll need to do the addition indicated by the blue +. And some of the time, we'll need to do another addition indicated by the red +. Since we are halving the value as we recurse, we'll invoke the function $\log n$ times. 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 which is $O(\log n)$.

Is this algorithm optimal? Not always (despite the claim of some algorithms books). For example, if we want to multiply by 15, the above formula would give us:

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

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
}
```

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

Exercise 1.4

Find optimal addition chains for $n < 100$.

[Further reading on addition chains – Knuth vol 2.]

1.5 Improving the Algorithm

Our Fast Multiply algorithm is good as far as the number of additions are 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.

Rewriting Code

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

Now at some point during the process you be 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, it often seems that the code that lives the longest is the one you meant to be only a quick temporary hack.) Furthermore, as we'll see later, that inexpensive operation you're saving now may be replaced by a very costly one in some future version of the code.

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: $r + na = r_0 + n_0 a_0$

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 — which we'll take advantage of shortly.

We observe that:

- 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 two, 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 people think that compiler optimizations will do this kind of transformations for us, but that's rarely true. Compilers are not artificially intelligent.

This 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 1.2. *A* strict 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 — just replace the final `return` statement with a `goto` looping back to the beginning. However, since the abuse of `goto`s gave them a bad reputation [cite Dijkstra], we'll use a `while` construct and let the compiler put in the `goto` for us:

```
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;
  }
}
```

Now we can return to 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 r) 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 an 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);
}
```

[Concluding thoughts for chapter.]

Chapter 2

From Multiplication to Power

In this chapter we'll take the Egyptian multiplication algorithm from Chapter 1 and, by using some important mathematical abstractions, generalize it to apply to a wide variety of problems beyond simple arithmetic.

2.1 Untangling the Requirements

There are two steps required to write a good piece of code. The first step is to get the algorithm right. The second step is to figure out what sorts of things (types) it works for. Now, you might be thinking that you already know the type — it's `int` or `float` or whatever you started with. But that may not always be the case; things change. Next year someone may want it to work with `unsigned int`s or `double`s or something else entirely.

Let's take another look at the multiply-accumulate algorithm we developed in Chapter 1:

```
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;
  }
}
```

This time we've colored some of the code blue and some it red. Notice that the blue and red parts are disjoint; there are no places where a "blue variable" and a "red variable"

are combined or interact with each other in any way. This means that the requirements for being a blue variable don't have to be the same as the requirements for being a red variable — or to put it in programming language terms, they can be different types.

So what are the requirements for each color variable? Blue variables r and a must be some type that supports adding — we might say that they must be a "plusable" type. The red variable n must be able to be checked for oddness, compared with 1, and support division by 2 (it must be "halfable").

We've established that r and a are the same type, which we'll write using the template typename A. Similarly, we said that n is a different type, which we'll call N. So instead of insisting that all the arguments be of type `int`, we can now write the following more generic form of the program:

```
template <typename A, typename N>
A multiply_accumulate(A r, N n, A a) {
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}
```

This makes the problem easier — we can figure out the requirements for A separately from the requirements on N. Let's dig a bit deeper into these types, starting with the simpler one.

2.2 Requirements on A

What are the *syntactic requirements* on A? In other words, what operations can we do on things belonging to A? Just by looking at how variables of this type are used in the code, we can see that there are three:

- they can be added: they must have `operator+`
- they can be passed by value: they must have a copy constructor
- they can be assigned: they must have `operator=`

We also need to specify the *semantic requirements*; we need to say what these operations mean. Our main requirement is that + must be *associative*, which we express as follows:

$$\mathsf{A}(T) \implies \forall a, b, c \in T : a + (b + c) = (a + b) + c$$

(In English, we might read the part before the colon like this: "If type T is an A, then for any values a, b, and c in T, the following holds:...")

Even though + is associative in theory (and in math generally), things are not so simple on computers. There are real world cases where associativity of addition does not hold. For example, consider these two lines of code:

```
w = (x + y) + z;

w = x + (y + z)
```

Suppose x, y, and z are of type `float`, and z is negative. Then it is possible that for some very large values, $x + y$ will overflow, while this would not have happened if we added $y + z$ first. The problem arises because addition is not well-defined for all possible values of the type `float`; we say that + is a *partial* function.

To address this problem, we will be clarify our requirements. We will require that the axioms hold only inside the *domain of definition*.[1]

In fact, there are a couple more syntactic requirements that we missed. They are implied by copy construction and assignment. For example, copy construction means to make a copy that is equal. In order to do this, we need the ability to test things belonging to A for equality:

- they can be compared for equality: they must have `operator==`
- they can be compared for inequality: they must have `operator!=`

Accompanying these syntactic requirements are semantic requirements for what we call *equational reasoning*; equality on a type T should satisfy our expectations:

- inequality is the negation of equality

$$(a \neq b) \iff \neg(a = b)$$

- equality is reflexive, symmetric, and transitive

$$a = a$$

$$a = b \implies b = a$$

$$a = b \wedge b = c \implies a = c$$

- substitutability

$$\text{for any function } \mathbb{F} \text{ on T, } \quad a = b \implies \mathbb{F}(a) = \mathbb{F}(b)$$

[1] For a more rigorous treatment of this topic, see Section 3.1 of *Elements of Programming* [cite].

The three axioms in the middle provide what we call *equivalence*, but equational reasoning requires something stronger, so we add the substitutability requirement.

We have a special name for types that behave in the "usual way": *regular* types:

Definition 2.1. *A* regular type *is one that provides the following connections between construction, assignment, and equality:*

- `T a = b; assert(a == b);`
- `a = b; assert(a == b);`
- `T a = b;` $\iff$ `T a; a = b;`
- "normal" destructor

From now on, we'll assume that all types we use are regular, unless stated otherwise.

Exercise 2.1

It seems that all of the C/C++ built-in types are regular. Unfortunately, one of the most important logical axioms is sometimes violated, the *law of excluded middle* ($p \vee \neg p$), which implies that:

$$a = b \vee a \neq b$$

Sometimes this statement is false. Find the case; analyze it and propose a solution.

Exercise 2.2

There is a sizable body of work on constructive real numbers: the real numbers that can be computed with arbitrary precision. Apparently, the classes that implement them are not regular. Why?

[Further reading about constructive real numbers: Hans-Juergen Boehm, "Constructive Real Interpretation of Numerical Programs," *PLDI* 1987, pp. 214-221.]

Now we can formalize the requirements on `A`:

- Regular type
- Associative +

We say that `A` is an *additive semigroup*.

Definition 2.2. *A* semigroup *is a set with an associative binary operation.*

Definition 2.3. *An* additive semigroup *is a semigroup where the associative binary operation is* +.

Some examples of additive semigroups are: Positive even numbers, negative integers, real numbers, polynomials, planar vectors, boolean functions, line segments, and so on. As we shall see, $+$ may have different interpretations for these different types, but it will always be associative.

By convention, + is also commutative. Occasionally some programming language designers have used + to represent a noncommutative operation (such as string concatenation). This violates standard mathematical practice, which is a bad idea. The mathematical convention is:

- If a set has one binary operation and it is associative and commutative, call it $+$.
- If a set has one binary operation and it is associative and not commutative, call it $*$.

20th century logician Stephen Kleene introduced the notation *ab* to denote string concatenation (since in mathematics $*$ is elided).

So if we are coming up with a name for something, or overloading an existing name, we should follow the following principles:

1. If there is an established term, use it.
2. Do not use an established term inconsistently with its accepted meaning.
3. If there are conflicting usages, the much more established one wins.

The name *vector* in STL to represent random access containers was chosen to be consistent with the earlier programming languages Scheme and Common Lisp. Unfortunately, this was inconsistent with the much older meaning of the term in mathematics and violates Rule 3; these containers should have been called *array*. Sadly, if you make a mistake and violate these principles, the result might stay around forever.

2.3 Requirements on N

Now that we know that A must be a Noncommutative Additive Semigroup, we can specify that in our template:

```
template <NonCommutativeAdditiveSemigroup A, typename N>
A multiply_accumulate(A r, N n, A a) {
  while (true) {
    if (odd(n)) {
      r = r + a;
```

```
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}
```

Here we're using "NonCommutativeAdditiveSemigroup" as a C++ *concept*, a set of requirements on types that we'll discuss in Journey 3. We are following the syntax of the proposed C++ concepts language feature [cite]. Since concepts are not yet supported in the language, we're doing a bit of preprocessor slight-of-hand:

```
#typedef typename NonCommutativeAdditiveSemigroup
```

As far as the compiler is concerned, `A` is just a typename, but for us, it's a NonCommutativeAdditiveSemigroup. We'll use this trick from now on when we want to specify the type requirements in templates.

What about the requirements on our other argument's type, `N`? Let's start with the syntactic requirements: `N` must be a regular type with:

- `half`
- `odd`
- `== 1`
- copy constructor
- assignment

Here are the semantic requirements on `N`:

- $even(n) \implies half(n) + half(n) = n$
- $odd(n) \implies even(n-1)$
- $odd(n) \implies half(n-1) = half(n)$
- Axiom: $n = 1 \ \vee \ half(n) = 1 \ \vee \ half(half(n)) = 1 \vee \ldots$

What C++ types satisfy these requirements? There are several: `uint8_t`, `int8_t`, `uint64_t`, and so on.

Rather than defining requirements through axioms as we did for `A`, mathematicians sometimes define a set consisting of the disjunction of *intended models* — types on which we want our algorithm to work. We'll do this for `N`:

$$\mathsf{N} = \{\texttt{uint8\_t}, \texttt{int8\_t}, \ldots, \texttt{uint64\_t}, \ldots\}$$

We call N Integer.

Now we can finally write a fully *generic* version of our multiply-accumulate function by specifiying the correct requirements:

```
template <NoncommutativeAdditiveSemigroup A, Integer N>
A multiply_accumulate_semigroup(A r, N n, A a) {
  precondition(n >= 0);
  if (n == 0) return r;
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}
```

Note that we've added an additional line of code, which returns r when n is zero. We're allowed to do this because this basically says "when n is zero, don't do anything." However, the same is not true for multiply, as we'll see in a moment.

Here's the multiply function with the same requirements:

```
template <NoncommutativeAdditiveSemigroup A, Integer N>
A multiply_semigroup(N n, A a) {
  precondition(n > 0);
  while (!odd(n)) {
    a = a + a;
    n = half(n);
  }
  if (n == 1) return a;
  return multiply_accumulate_semigroup
    (a, half(n - 1), a + a);
}
```

2.4 Semigroups, Monoids, and Groups

Notice that our precondition for multiply says that n must be strictly greater than zero. Why can't we allow n to be zero? Recall that we said that the distributive law must hold,

so the following must be true:

$$a = 1a = (1+0)a = 1a + 0a = a + 0a$$

So $0a$ must be something that, when added to a, doesn't change the result. We call $0a$ the *right additive identity*. Similarly:

$$a = 1a = (0+1)a = 0a + 1a = 0a + a$$

where $0a$ is the *left additive identity*. $0a$ is the *additive identity* — a zero. But an additive semigroup is not required to have an identity element, so we can't depend on this property. In other words, we can't rely on there being an equivalent of zero. (Remember, a no longer has to be an integer — it can be any NoncommutativeAdditiveSemigroup, such as positive integers or nonempty strings.) That's why n can't be zero.

But there is an alternative: Instead of requiring $n > 0$, we can relax the requirement on n from additive semigroup to *monoid*.

Definition 2.4. *A* monoid *is a semigroup that contains an identity element* id *such that for its operation* $\circ$

$$a \circ id = id \circ a = a$$

Definition 2.5. *An* additive monoid *is an additive semigroup which has an identity element called "0":*

$$a + 0 = 0 + a = a$$

Our new multiply function simply wraps our old multiply function, checking for zero before calling it:

```
template <NoncommutativeAdditiveMonoid A, Integer N>
A multiply_monoid(N n, A a) {
  precondition(n >= 0);
  if (n == 0) return A(0);
  return multiply_semigroup(n, a);
}
```

What if we want to allow negative numbers when we multiply? We need to insure that "multiplying by a negative" makes sense for any type we might have. This turns out to be equivalent to saying that the type must support an *inverse operation*. Again, we find that our current requirement — Noncommutative Additive Monoid — is not guaranteed to have this property. For this, we need a *group*:

Definition 2.6. *A* group *is a monoid with an inverse operation:*

$$a \circ inv(a) = inv(a) \circ a = id$$

This is called a *cancellation law*. (As above, *id* is the identity element.) We'll cover groups in more detail in Journey 2.

Definition 2.7. *An* additive group *is an additive monoid with unary minus such that*

$$a + -a = -a + a = 0$$

Having strengthened our type requirements, we can loosen our requirements on n to allow negative values. Again, we'll wrap the last version of our function with our new one:

```
template <AdditiveGroup A, Integer N>
A multiply_group(N n, A a) {
  if (n < 0) {
    n = -n;
    a = -a;
  }
  return multiply_monoid(n, a);
}
```

Évariste Galois (1811-1832)

The concept of groups started with the work of Évariste Galois, a young French college dropout involved in a revolutionary movement, and the most romantic figure in the history of mathematics.

In the early 19th century, a romantic spirit spread through Europe; young people idolized the English poet Byron, who died fighting for Greek independence, and others who were willing to give their lives for a cause. They remembered Napoleon not as a tyrant but as a young hero who abolished feudalism throughout Europe.

Paris in the early 1830s was aflame with revolutionary activity. Galois, who was a bohemian hothead, joined the revolutionary movement. As a romantic rebel, Galois did not follow the conventional path through a university education. After failing to be admitted to one school and being expelled from another, he studied mathematics on his own, becoming an expert on Lagrange's theory of polynomials. He served brief prison sentences for various protest activities, such as marching through the streets in the uniform of a banned national guard unit while carrying several loaded weapons — but kept doing mathematics while in prison.

At age 20, Galois, defending the honor of a woman he apparently barely knew, was challenged to a duel. The night before the duel, certain of his impending death, he wrote a long letter to a friend describing his mathematical ideas. This manuscript contained

the seeds of the theory of groups, fields, and their automorphisms (mappings onto themselves). As we shall see in Journey 2, this was the beginning of a major new field in mathematics, abstract algebra. According to mathematician Hermann Weyl, "This letter, if judged by the novelty and profundity of ideas it contains, is perhaps the most substantial piece of writing in the whole literature of mankind."

The next day, Galois fought the duel and died as a result of his wounds. It is ironic that while he only played at being a revolutionary in politics, he was a true revolutionary in mathematics.

2.5 Turning Multiply Into Power

Now that our code has been generalized to work for any additive semigroup (or monoid or group), we can make a remarkable observation:

If we replace $+$ with $*$ (thereby replacing doubling with squaring),
we can use our existing algorithm to compute a^n instead of $n \cdot a$.

Here's the C++ we get when we apply this transformation to our multiply_accumulate_semigroup function:

```
template <MultiplicativeSemigroup A, Integer N>
A power_accumulate_semigroup(A r, A a, N n) {
  precondition(n >= 0);
  if (n == 0) return r;
  while (true) {
    if (odd(n)) {
      r = r * a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a * a;
  }
}
```

The new function computes ra^n. The only things that have changed are highlighted in red. Note that we've changed the order of arguments a and n, in order to match the order of arguments in standard mathematical notation (i.e. we say na, but a^n).

Here's the function that computes power:

```
template <MultiplicativeSemigroup A, Integer N>
A power_semigroup(A a, N n) {
```

```
  precondition(n > 0);
  while (!odd(n)) {
    a = a * a;
    n = half(n);
  }
  if (n == 1) return a;
  return power_accumulate_semigroup(a, a * a, half(n - 1));
}
```

Here are the wrapped versions for multiplicative monoids and groups:

```
template <MultiplicativeMonoid A, Integer N>
A power_monoid(A a, N n) {
  precondition(n >= 0);
  if (n == 0) return A(1);
  return power_semigroup(a, n);
}

template <MultiplicativeGroup A, Integer N>
A power_group(A a, N n) {
  if (n < 0) {
    n = -n;
    a = multiplicative_inverse(a);
  }
  return power_monoid(a, n);
}
```

Just as we needed an additive identity (0) for our monoid multiply, we need a multiplicative identity (1) in our monoid power function. And just as we needed an additive inverse (unary minus) for our group multiply, we need a multiplicative inverse for our group power function. There's no built-in multiplicative inverse (reciprocal) operation in C++, but we can write one easily:

```
template <MultiplicativeGroup A>
A multiplicative_inverse(A a) {
  return A(1) / a;
}
```

2.6 Generalizing the Operation

We've seen examples of two semigroups — additive and multiplicative — each with their associated operation ($+$ and $*$ respectively). The fact that we could use the same algorithm

for both is wonderful, but it was annoying to have to go and write different versions of the same code for each case. And in reality, there could be many such semigroups, each with their associative operations (for example, bitwise `&` or `min`) that work on the same type T. Rather than having another version for every operation we want to use, we can generalize the *operation* itself, just as we generalized the types of the arguments before. In fact, there are many situations where we need to pass an operation to an algorithm; you may have seen examples of this in the C++ STL library.

Here's our accumulate function for an arbitrary semigroup:

```
template <Regular A, Integer N, SemigroupOperation Op>
requires (Domain<Op, A>)
A power_accumulate_semigroup(A r, A a, N n, Op op) {
  precondition(n >= 0);
  if (n == 0) return r;
  while (true) {
    if (odd(n)) {
      r = op(r, a);
      if (n == 1) return r;
    }
    n = half(n);
    a = op(a, a);
  }
}
```

Notice that we've added a clause that says that the domain of operation `Op` is A. And instead of saying that A is a semigroup, we say that the operation we're going to perform on A is a semigroup operation. [May need to explain this more clearly.]

We can use this to write a version of `power` for an arbitrary semigroup:

```
template <Regular A, Integer N, SemigroupOperation Op>
requires (Domain<Op, A>)
A power_semigroup(A a, N n, Op op) {
  precondition(n > 0);
  while (!odd(n)) {
    a = op(a, a);
    n = half(n);
  }
  if (n == 1) return a;
  return power_accumulate_semigroup(a, op(a, a), half(n - 1), op);
}
```

As before, we extend to monoids by adding an identity element. But since we don't

know in advance what operation we'll be passed, we have to obtain the identity from the operation:

```
template <Regular A, Integer N, MonoidOperation Op>
requires(Domain<Op, A>)
A power_monoid(A a, N n, Op op) {
  precondition(n >= 0);
  if (n == 0) return identity_element(op);
  return power_semigroup(a, n, op);
}
```

Here are examples of `identity_element` functions for $+$ and $*$:

```
template <NoncommutativeAdditiveMonoid T>
T identity_element(std::plus<T>) {
  return T(0);
}

template <MultiplicativeMonoid T>
T identity_element(std::multiplies<T>) {
  return T(1);
}
```

The first one says "the additive identity is 0." Of course, there will be different identity elements for different monoids, e.g. MAXINT for `min`.

For groups, we need an inverse operation, which is itself a function of the specified GroupOperation:

```
template <Regular A, Integer N, GroupOperation Op>
requires(Domain<Op, A>)
A power_group(A a, N n, Op op) {
  if (n < 0) {
    n = -n;
    a = inverse_operation(op)(a);
  }
  return power_monoid(a, n, op);
}
```

Examples of `inverse_operation` look like this:

```
template <AdditiveGroup T>
std::negate<T> inverse_operation(std::plus<T>) {
  return std::negate<T>();
}
```

```
template <MultiplicativeGroup T>
reciprocal<T> inverse_operation(std::multiplies<T>) {
  return reciprocal<T>();
}
```

STL already has a `negate` function, but (due to an oversight), has no `reciprocal`. So we'll write our own:

```
template <MultiplicativeGroup T>
struct reciprocal : public std::unary_function<T, T> {
  T operator()(const T& x) const {
    return T(1) / x;
  }
};
```

This is just a generalization of the `multiplicative_inverse` function we wrote in the previous section.

The Essential Insight: Algorithms Defined on Abstract Concepts

The process we have just gone through — taking an algorithm, generalizing it so that it works on abstract mathematical concepts, and then applying it to a myriad of different situations — is the basis for *generic programming*. While this idea was was originally applied in the language Tekton [cite] and in a library for Ada [cite], it was primarily popularized through the creation of the C++ Standard Template Library [cite].

2.7 Reduction

While the power algorithm is very important, there is another even more important algorithm defined on semigroups: *reduction*, in which a binary operation is applied successively to each element of a list and its previous result.

Two commonly seen examples of this in mathematics are the summation (Σ) function for additive semigroups, and the product (Π) function for multiplicative semigroups. We can generalize this to an arbitrary semigroup.

This generalized version of reduction was invented by Ken Iverson in his language APL [cite]. In APL terminology, the / represented the reduction operator, so for example, summation of a list is expressed as

```
+ / 1 2 3
```

Backus included a similar operator called *insert* in his language FP [cite]. (He called operators "functional forms.") Kapur et al. [cite] extended it to parallel reduction and clarified the relationship to associative operations. Dean's MapReduce [cite] re-uses many of the same ideas.

Exercise 2.3

Using our work on `multiply` and `power`, design a library version of the reduction algorithm.

Chapter 3

Applications of the Power Algorithm

In this chapter, we'll see how to apply the power algorithm developed in chapter 2 to a variety of applications beyond arithmetic.

3.1 Fibonacci Numbers

In Journey 3 we'll meet the early 13th-century mathematician Leonardo Pisano, often known today as Fibonacci. For now we're just going to look a famous problem he posed: If a pair of rabbits can produce one new pair of rabbits every month, how many pairs will there be after n months? The sequence looks like this:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34...$$

Each month's population is obtained by adding the populations of each of the previous two months. Today, elements of this sequence are called *Fibonacci numbers*, and it is defined formally like this:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2}$$

How long does it take to compute the nth Fibonacci number? The "obvious" answer is $n - 2$ — but the obvious answer is wrong. The naive way to implement this in C++ is something like this:

```
int fib0(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib0(n - 1) + fib0(n - 2);
}
```

However, this code does an awful lot of repeated work. Consider the calculations to compute `fib0(5)`:

$$
\begin{aligned}
&F_5 = \\
&F_4 + F_3 = \\
&(F_3 + F_2) + (F_2 + F_1) = \\
&((F_2 + F_1) + (F_1 + F_0)) + ((F_1 + F_0) + F_1) = \\
&(((F_1 + F_0) + F_1) + (F_1 + F_0)) + ((F_1 + F_0) + F_1)
\end{aligned}
$$

Even in this small example, the computation there are 17 additions, and just the quantity $F_1 + F_0$ is computed 5 times.

Exercise 3.1

How many additions are needed to compute fib0(n)?

Recomputing the same thing over and over is unacceptable, and there's no excuse for code like that. We can easily fix it by keeping a running state of the previous two results:

```
int fibonacci_iterative(int n) {
  if (n == 0) return 0;
  std::pair<int, int> v(0, 1);
  for (int i = 1; i < n; ++i)
    v = std::pair<int, int>(v.second, v.first + v.second);
  return v.second;
}
```

This is a perfectly respectable solution, which takes $O(n)$ operations. In fact, given that we want to find the nth element of a sequence, it might appear to be optimal. But the amazing thing is that we can actually compute the nth Fibonacci number in log n, which for most practical purposes, is less than 64.

Suppose we represent the computation of the next Fibonacci number from the previous two using the following matrix equation:

$$
\begin{bmatrix} v_{i+1} \\ v_i \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} v_i \\ v_{i-1} \end{bmatrix}
$$

Then the nth Fibonacci number may be obtained by:

$$
\begin{bmatrix} v_n \\ v_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}
$$

In other words, we can compute the nth Fibonacci number by raising a certain matrix to a power. [Cite Gries and Levin in 1980 (IPL).] As we will see, matrix multiplication can be used to solve many problems. Matrices are a multiplicative monoid, so *we already have a* $O(\log N)$ *algorithm* — our `power` algorithm from the last chapter.

Exercise 3.2

Implement computing Fibonacci numbers using power.

[Further reading: EoP pp. 45-46 shows how to reduce the number of operations needed to multiply two Fibonacci matrices.]

This is a very nice application of our power algorithm, but computing Fibonacci numbers isn't the only thing we can do. If we replace the $+$ with an arbitrary linear function, we can use the same technique to compute any *linear recurrence*.

Definition 3.1. *A linear recurrence of order k is a function f such that*

$$f(y_0, \ldots, y_{k-1}) = \sum_{i=0}^{k-1} a_i y_i$$

The Fibonacci sequence is a linear recurrence of order 2.

Definition 3.2. *A linear recurrence sequence is a sequence generated by such function from initial k values.*

For any linear recurrence, we can compute the nth step by doing matrix multiplication using our power algorithm:

$$\begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-k+1} \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & \ldots & a_{k-2} & a_{k-1} \\ 1 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & 0 \end{bmatrix}^{n-k+1} \begin{bmatrix} x_{k-1} \\ x_{k-2} \\ x_{k-3} \\ \vdots \\ x_0 \end{bmatrix}$$

The line of 1s just below the diagonal provide the "shifting" behavior, so that each value in the sequence depends on the previous k.

[Further reading: Reducing number of operations for linear recurrence computation: Fiduccia ref.]

3.2 Generalizing Matrix Multiplication

We combined power with matrix multiplication to compute linear recurrences. It turns out that we can use this technique for many other algorithms if we use a more general notion of matrix multiplication.

Let's quickly refresh how some basic vector and matrix operations are defined.

Inner product:

$$\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} x_i y_i$$

Matrix-vector product:

$$\vec{w} = \left[x_{ij}\right] \vec{v}$$
$$w_i = \sum_{j=1}^{n} x_{ij} v_j$$

Matrix-matrix product:

$$\left[z_{ij}\right] = \left[x_{ij}\right] \left[y_{ij}\right]$$
$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$$

Just as we generalized our power function to work with any operation, we can now generalize the notion of matrix multiplication. Normally we think of matrix multiplication as consisting of a series of sums of products, as shown in the above formula. But what's mathematically essential is actually that there be two operations, a "plus-like" one that is associative and commutative (denoted by $\oplus$) and a "times-like" one that is associative (denoted by $\otimes$), where the latter operation distributes over the first:

$$a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$$
$$(b \oplus c) \otimes a = b \otimes a \oplus c \otimes a$$

A type that includes these operations is called a *semiring*.

We can use other semirings to solve a variety of other problems. For example, we have a graph of friendship relations, as in a social network, and we want to find everyone I'm connected to through any path. In other words, we want to know who my friends are, the friends of my friends, the friends of the friends of my friends, and so on. Finding all such paths is known as finding the *transitive closure* of the graph.

To compute the transitive closure, we take an $n \times n$ Boolean matrix where entry x_{ij} is 1 if the relation holds between i and j (in this case, if person i is friends with person j), and 0 otherwise. Then we use our generalized matrix multiplication, where $\oplus$ is Boolean

OR ($\vee$) and $\otimes$ is Boolean AND ($\wedge$). We say this is the matrix multiplication generated by a *Boolean* or $\{\vee, \wedge\}$-*semiring*. Finally, we raise the matrix to the $n-1$ power, using our existing power algorithm. Of course, we can use this idea to compute the transitive closure of any relation.

Another example of a classic problem we can solve this way is finding the shortest path between any two nodes in a directed graph. We can represent the graph as an $n \times n$ matrix whose values a_{ij} represent the distance from node i to node j. This time, we use matrix multiplication generated by a *Tropical* or $\{\min, +\}$-*semiring*:

$$b_{ij} = \min_{k=1}^{n}(a_{ik} + a_{jk})$$

Again, we raise the resulting matrix to the $n-1$ power.

Note that the result is not saying that this is the shortest path of length $n-1$ steps; if there is a shorter path in fewer steps, the algorithm finds it.

Exercise 3.3

Implement the `power`-based shortest path algorithm.

Exercise 3.4

Modify the program from Exercise 3.3 to return not just the shortest distance but the shortest path (a sequence of edges).

3.3 Pythagoras and Some Geometric Properties of Integers

> He maintained that "the principles of mathematics are principles of all existing things."
>
> – Aristotle

Pythagoras, the Greek mathematician and philosopher who most of us only know 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. 570BC– ca. 490BC)

Pythagoras was born on the 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, who advised him to go to Egypt and learn their mathematical secrets.

At 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 was exposed to 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.

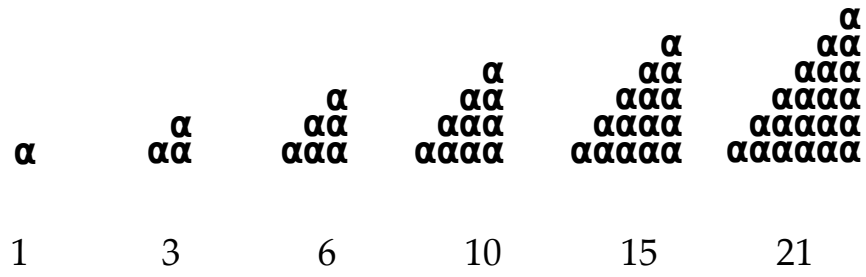
After returning to Greece he started a settlement in Croton, where he gathered followers — both men and women — who shared his ideas and followed his ascetic lifestyle. Their lives were centered on the study of four things: Astronomy, Geometry, Number Theory, and Music. Each of these disciplines was 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 spheres."

After the death of Pythagoras, the Pythagoreans spread to several other Greek colonies in southern Italy and developed a large body of mathematics. However, they kept their teachings secret; they also eliminated competition within their ranks by crediting all discoveries to Pythagoras himself.

Although the Pythagoreans were gone within a couple of hundred years, their work remains influential. As late as the 17th century, Leibniz (who with Newton was 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 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:



1 3 6 10 15 21

While oblong numbers are those that look like this:

2 6 12 20 30 42

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:

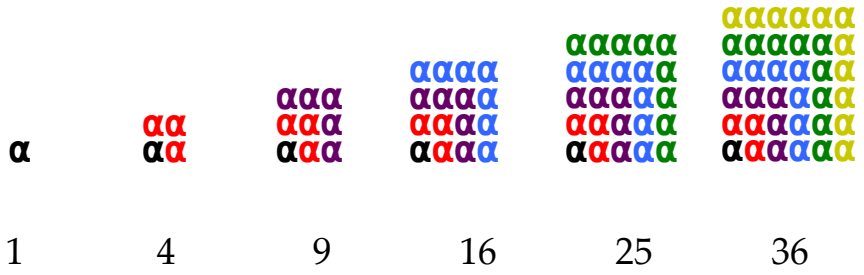
$$\square_n = 2\triangle_n = 2\sum_{i=1}^{n} i = n(n+1)$$

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 3.5

Find a geometric proof for the following: Take any triangular number, multiply it by 8, and add 1. The result is a square number. [Cite Plutarch, *Platonic Questions*.]

3.4 Sifting Primes

Pythagoreans also observed that some numbers could not be made into any shape. These are what we now call *prime numbers* — numbers that are not products of smaller numbers:

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

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:

> 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] Consider a number A. If it is prime, then we are done. If it composite, 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 divisible by a 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 either find a prime, or "an infinite sequence of numbers will divide the number, each of which is less than the other; and this is impossible." We will discuss this induction further in the third journey.

Another result, which some consider the most beautiful theorem in mathematics, is the fact that there are an infinite number of primes:

> 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

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

Because of the way we constructed q, we know it is not divisible by any p_i. Then either q is prime, in which case it is itself a prime not in the sequence, or q is divisible by some new prime, which by definition is not in the sequence. Therefore, there are infinitely many primes.

One of the best known techniques for finding primes is the *Sieve of Eratosthenes*. The idea is to start with all the candidates and then cross out the ones known not to be primes (since they are multiples of primes found so far). Today the Sieve is often shown starting with all positive integers up to a given number, but Eratosthenes knew not to bother with even numbers.

Eratosthenes (284BC-195BC)

[Biography of Eratosthenes goes here. Points to include:

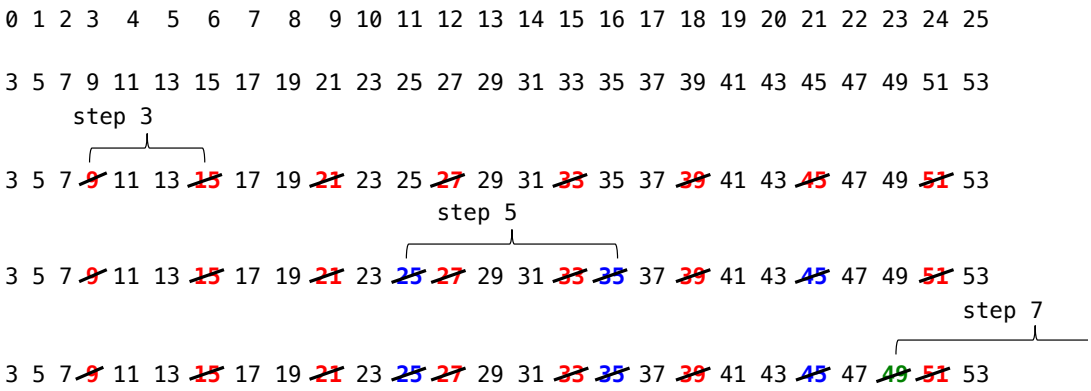
- Born in Cyrene in what is now Libya.
- Studied in Athens.
- Invented geography.

[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 above.

- Came up with a technique for measuring the circumference of the earth.

]

Here's one way to picture the sieve during its construction:



The top row is the index of the value; the following rows show all the candidate primes during the first few iterations of the algorithm. First we want to cross off multiples of 3, then multiples of 5 (the next prime), then multiples of 7, and so on. Observe that, since we are skipping even numbers, the next multiple of 3 in the table — one we get to by moving 3 steps forward — is actually 6 greater than the previous value; the next multiple of 5 is 10 greater, etc. Also, as we'll see below, the first number we cross off each time is actually the square of the prime we're working on, since earlier factors are already accounted for by earlier primes. At the end, all the numbers we haven't crossed off are primes.

Our implementation will use boolean values in the sieve; 1 means prime, 0 means nonprime. We call the process of "crossing out" nonprimes *marking* the sieve. Here's how we mark all the non primes for a given factor:

```
template <RandomAccessIterator I, Integer N>
void mark_sieve(I first, I last, N step) {
  // assert(first != last)
  *first = false;
  while (last - first > step) {
    first = first + step;
    *first = false;
  }
}
```

Before we see how to sift, we observe the following sifting lemmas:

- The square of the smallest prime factor of a composite number n is less than or equal to n.
- Any composite number less than p^2 is sifted by a prime less than p.
- When sifting by p, start marking at p^2.
- If the table is of size m, stop sifting when $p^2 \geq m$.

We will use the following formulas in our computation:

$$\begin{aligned}
\text{value at position } i:\ \text{val}(i) &= 3 + 2i = 2i + 3 \\
\text{position of value } v:\ \text{idx}(v) &= \frac{v-3}{2} \\
\text{step between multiples of } p:\ \text{step}(i) &= \text{idx}((n+2)p) - \text{idx}(np) \\
&= \frac{np + 2p - 3}{2} - \frac{np - 3}{2} \\
&= p = 2i + 3
\end{aligned}$$

$$\begin{aligned}
\text{position of square of value at } i:\ \text{idx}(\text{val}^2(i)) &= \frac{(2i+3)^2 - 3}{2} \\
&= \frac{4i^2 + 12i + 9 - 3}{2} \\
&= 2i^2 + 6i + 3
\end{aligned}$$

We can now make our first attempt at implementing the sieve:

```
template <RandomAccessIterator I, Integer N>
void sift0(I first, N n) {
   std::fill(first, first + n, true);
   N i(0);
   N square(3);
   while (square < n) {
     // invariant: square = 2i^2 + 6i + 3
     if (first[i]) {
       mark_sieve(first + square,
                  first + n,  // last
                  i + i + 3); // step
     }
     ++i;
     square =  3 + 2*i*(i + 3);
   }
}
```

One thing we notice is that we're computing step $(i + i + 3)$ and other quantities (shown in green) every time through the loop. We can hoist common subexpressions out of the loop; the changes are shown in red:

```
template <RandomAccessIterator I, Integer N>
void sift1(I first, N n) {
   I last = first + n;
   std::fill(first, last, true);
   N i(0);
   N square(3);
   N step(3);
   while (square < n) {
     // invariant: square = 2i^2 + 6i + 3, step = 2i + 3
     if (first[i]) mark_sieve(first + square, last, step);
     ++i;
     step   = i + i + 3;
     square =  3 + 2*i*(i + 3);
   }
}
```

The astute reader will notice that the `step` computation is actually slightly worse than before, since it happens every time through the loop, not just on iterations when the `if` test is true. However, we shall see below why making `step` a separate variable makes sense. A bigger issue is that we still have a relatively expensive operation — the computation of `square`, which involves two multiplications. So we will take a cue from compiler optimization and use a technique known as *strength reduction* in which more expensive operations like multiplication are replaced with equivalent code that uses less expensive operations like addition. If a compiler can do this automatically, we can certainly do it manually.

Let's look at these computations in more detail. Suppose we replaced

```
step = i + i + 3;
square = 3 + 2*i*(i + 3);
```

with

```
step += δ_step;
square += δ_square;
```

Where δ_{step} and δ_{square} are the differences between successive values of `step` and `square`:

$$\delta_{step}: \ (2(i+1)+3) - (2i+3) = 2$$

$$\begin{aligned}\delta_{square}: \ & (2(i+1)^2 + 6(i+1) + 3) - (2i^2 + 6i + 3) \\ & = 4i + 8 = (2i+3) + (2i+2+3) \\ & = (2i+3) + (2(i+1)+3) \\ & = \text{step}(i) + \text{step}(i+1)\end{aligned}$$

δ_{step} is easy; the variables cancel and we get the constant 2. But how did we simplify the expression for δ_{square}? We observe that by rearranging the terms, we can express it in terms of something we already have, namely `step(i)`, and something we need to compute anyway, `step(i + 1)`. (When you know you need to compute multiple quantities, it's useful to see if one can be computed in terms of another. This might allow you to do less work.)

With the above substitutions, we get our final version of `sift`; again, our improvements are shown in red:

```
template <RandomAccessIterator I, Integer N>
void sift(I first, N n) {
   I last = first + n;
   std::fill(first, last, true);
   N i(0);
   N square(3);
   N step(3);
   while (square < n) {
     // invariant: square = 2i^2 + 6i + 3, step = 2i + 3
     if (first[i]) mark_sieve(first + square, last, step);
     ++i;
     square += step;
     step   += N(2);
     square += step;
   }
}
```

Exercise 3.6

Time the sieve using different data sizes: `bool` (using `std::vector<bool>`), `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`.

Exercise 3.7

Using the sieve, graph the function

$$\pi(n) = \text{the number of primes} < n$$

for n up to 10^7 and find its analytic approximation.

We call primes that read the same backwards and forwards *palindromic primes*. Here we've highlighted the ones up to 1000:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107
109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223
227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337
347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457
461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593
599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719
727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857
859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997

Interestingly, there don't seem to be any palindromic primes between 1000 and 2000:

1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091 1093
1097 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187 1193 1201 1213
1217 1223 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291 1297 1301 1303
1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427 1429 1433 1439
1447 1451 1453 1459 1471 1481 1483 1487 1489 1493 1499 1511 1523 1531 1543
1549 1553 1559 1567 1571 1579 1583 1597 1601 1607 1609 1613 1619 1621 1627
1637 1657 1663 1667 1669 1693 1697 1699 1709 1721 1723 1733 1741 1747 1753
1759 1777 1783 1787 1789 1801 1811 1823 1831 1847 1861 1867 1871 1873 1877
1879 1889 1901 1907 1913 1931 1933 1949 1951 1973 1979 1987 1993 1997 1999

Exercise 3.8

Are there palindromic primes > 1000? What is the reason for the lack of them in the interval [1000, 2000]? What happens if we change our base to 16? To an arbitrary n?

Chapter 4

Some Properties of Primes

In this chapter, we are going to put programming aside for a bit and learn some of the mathematics we need for the rest of our journey. In particular, we will learn about some interesting properties of numbers, particularly prime numbers.

4.1 Perfect Numbers

As we saw in the last chapter, the ancient Greeks were interested in all sorts of properties of numbers. One idea they came up with was that of a *perfect* number — a number which is the sum of its proper divisors.[1] Formally:

- An *aliquot part* of a positive integer is a proper divisor of the integer.
- The *aliquot sum* of a positive integer is the sum of its aliquot parts.
- A positive integer is *perfect* if it is equal to its aliquot sum.

[Comment about ancient fascination with perfect numbers. Six days of creation, 28 days in lunar cycle, etc. Both the Hebrews and Greeks knew about perfect numbers; they may have learned from a common Babylonian source.]

The Greeks knew of four perfect numbers:

$$\begin{aligned} 6 &= 1+2+3 \\ 28 &= 1+2+4+7+14 \\ 496 &= 1+2+4+8+16+31+62+124+248 \\ 8128 &= 1+2+4+8+16+32+64+127+254+508+1016+2032+4064 \end{aligned}$$

[1] A proper divisor of a number n is a divisor of n other than n itself.

and their prime factorizations:

$$\begin{aligned} 6 &= 2 \cdot 3 \\ 28 &= 4 \cdot 7 \\ 496 &= 16 \cdot 31 \\ 8128 &= 64 \cdot 127 \end{aligned}$$

What the Greeks really wanted to know was whether there was a way to predict other perfect numbers. They observed the following pattern:

$$\begin{aligned} 6 = 2 \cdot 3 &= 2^1 \cdot (2^2 - 1) \\ 28 = 4 \cdot 7 &= 2^2 \cdot (2^3 - 1) \\ 120 = 8 \cdot 15 &= 2^3 \cdot (2^4 - 1) \text{ not perfect} \\ 496 = 16 \cdot 31 &= 2^4 \cdot (2^5 - 1) \\ 2016 = 32 \cdot 63 &= 2^5 \cdot (2^6 - 1) \text{ not perfect} \\ 8128 = 64 \cdot 127 &= 2^6 \cdot (2^7 - 1) \end{aligned}$$

Observe that the result of this expression is perfect when the second multiple is prime. It was Euclid who presented the proof of this fact around 300 BC:

Euclid IX, 36:

$$\text{If } \sum_{i=0}^{n} 2^i \text{ is prime then } 2^n \sum_{i=0}^{n} 2^i \text{ is perfect.}$$

Before we look at the proof, it is useful to remember a couple of useful algebraic formulas. The first is the *difference of powers*:

$$x^2 - y^2 = (x - y)(x + y) \tag{4.1}$$
$$x^3 - y^3 = (x - y)(x^2 + xy + y^2) \tag{4.2}$$
$$\vdots \tag{4.3}$$
$$x^{n+1} - y^{n+1} = (x - y)(x^n + x^{n-1}y + \cdots + xy^{n-1} + y^n) \tag{4.4}$$

This result can easily be derived using these two equations:

$$x(x^n + x^{n-1}y + \cdots + xy^{n-1} + y^n) = x^{n+1} + x^n y + x^{n-1}y^2 + \cdots + xy^n \tag{4.5}$$
$$y(x^n + x^{n-1}y + \cdots + xy^{n-1} + y^n) = \qquad x^n y + x^{n-1}y^2 + \cdots + xy^n + y^{n+1} \tag{4.6}$$

The left and right sides of 4.5 and 4.6 are equal by the distributive law. If we then subtract 4.6 from 4.5, we get 4.4.

The second useful formula is for the *sum of odd powers*:

$$x^{2n+1} + y^{2n+1} = (x+y)(x^{2n} - x^{2n-1}y + \cdots - xy^{2n-1} + y^{2n})$$

which we can derive by converting the sum to a difference and relying on our previous result:

$$\begin{aligned} x^{2n+1} + y^{2n+1} &= x^{2n+1} - -y^{2n+1} \\ &= x^{2n+1} - (-y)^{2n+1} \\ &= (x-(-y))(x^{2n} + x^{2n-1}(-y) + \cdots + (-y)^{2n}) \\ &= (x+y)(x^{2n} - x^{2n-1}y + \cdots - xy^{2n-1} + y^{2n}) \end{aligned}$$

We can get away with this because -1 to an odd power is still -1. We will rely heavily on both of these formulas in the proofs ahead.

Now since we know that

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

(think of the binary number you get when you add powers of 2), and since

$$2^n - 1 = (2-1)(2^{n-1} + 2^{n-2} + \cdots + 2 + 1)$$

by the difference of powers formula, we can restate Euclid's theorem as follows:

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

Now we will prove Euclid's theorem the way Gauss did. First, we define $\sigma(n)$ to be the sum of the divisors of n. If the prime factorization of n is:

$$n = p_1^{a_1} p_2^{a_2} \ldots p_m^{a_m},$$

then the set of all divisors consists of every possible combination of the prime divisors raised to every possible power up to a_i. For example, $24 = 2^3 \cdot 3^1$, so the divisors are $\{2^0 \cdot 3^0,\ 2^1 \cdot 3^0,\ 2^2 \cdot 3^0,\ 2^0 \cdot 3^1,\ 2^1 \cdot 3^1,\ 2^2 \cdot 3^1,\ 2^3 \cdot 3^1\}$. Their sum is

$$2^0 \cdot 3^0 + 2^1 \cdot 3^0 + 2^2 \cdot 3^0 + 2^0 \cdot 3^1 + 2^1 \cdot 3^1 + 2^2 \cdot 3^1 + 2^3 \cdot 3^1 = (2^0 + 2^1 + 2^2 + 2^3)(3^0 + 3^1)$$

That is, we can write the sum of the divisors for any number n as a product of sums:

$$\begin{aligned}\sigma(n) &= \prod_{i=1}^{m}(1 + p_i + p_i^2 + \cdots + p_i^{a_i}) \\ &= \prod_{i=1}^{m}\frac{p_i - 1}{p_i - 1}(1 + p_i + p_i^2 + \cdots + p_i^{a_i}) \\ &= \prod_{i=1}^{m}\frac{(p_i - 1)(1 + p_i + p_i^2 + \cdots + p_i^{a_i})}{p_i - 1} \\ &= \prod_{i=1}^{m}\frac{p_i^{a_i+1} - 1}{p_i - 1}\end{aligned}$$

where the last line relies on using the difference of powers formula to simplify the numerator.

Exercise 4.1

Prove that if n and m are *coprime* (have no common prime factors) then

$$\sigma(nm) = \sigma(n)\sigma(m)$$

(In number theory, another way to say this is that σ is a multiplicative function.)

We now define $\alpha(n)$, the *aliquot sum*, as:

$$\alpha(n) = \sigma(n) - n$$

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

Proof of Euclid IX, 36:

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

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 $n = p_1^{a_1}p_2^{a_2}\ldots p_m^{a_m}$, 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:

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

Then

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

i.e. 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. We don't know if any odd perfect numbers exist.

Exercise 4.2

Prove that every even perfect number is a triangular number.

Exercise 4.3

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

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

4.2 A Millennium Without Mathematics

As we have seen, ancient Greece was a source of several centuries of astonishing mathematical developments. By the 3rd century BC, mathematics was a flourishing field of study, with Archimedes (best known today for a story about discovering the principle of buoyancy in his bathtub) its most dominant figure. Unfortunately, the rise of Roman power led to a stagnation in Western mathematics that would last for a thousand years.

While the Romans built great works of engineering, they were generally uninterested in advancing the mathematics that made these possible. As the great Roman orator put it:

> Among the Greeks geometry was held in highest honor; nothing could outshine mathematics. But we have limited the usefulness of this art to measuring and calculating.
>
> – Cicero, *Tusculan Disputations* [cite]

While there were Greek mathematicians working in Roman times, it is a remarkable fact that there is no record of any original mathematical text written in Latin at that time.

Archimedes (287-212 BC)

[Biography of Archimedes goes here. Points to include:

- Lived in Syracuse, a Greek city on the island of Sicily.
- A great inventor and scientist as well as a great mathematician.
- Was in charge of engineering the defenses of Syracuse, and kept the Romans at bay for two years.
- Killed by Roman soldier sent to fetch him, after he said *Noli turbare circulos meos!* ("Don't disturb my circles").

]

In our next journey, we will see how the study of mathematics returns in Europe around 1200 AD. But for now, we will jump ahead to 17th-century France, in the age of the three musketeers, Cardinal Richelieu, and the creation of some important aspects of science as we know it today. Although Spain was a far greater power and cultural center at the time, it was France that became the center of mathematical progress for several generations.

4.3 Mersenne Primes and Fermat Primes

When is $2^n - 1$ prime? The Greeks knew that it was true for $n = 2, 3, 5$, and 7, and possibly 13. In 1536, Hudalricus Regius showed that it was false for $n = 11$, by finding

$$2^{11} - 1 = 2047 = 23 \times 89.$$

Pietro Cataldi added several more to the list in 1603 — 17, 19, (23), (29), 31, and (37) — but half of these, shown in parentheses, were incorrect. Pierre de Fermat discovered that:

$$2^{23} - 1 = 8388607 = 47 \times 178481$$
$$2^{37} - 1 = 137438953471 = 223 \times 61631877$$

In his 1644 book *Cogitata Physico Mathematica*, the French mathematician Mersenne states that if $n \leq 257$ then $2^n - 1$ is prime if an only if $n = 2, 3, 5, 7, 13, 17, 19, 31, (67), 117$, and (257). Two of these were wrong (shown in parentheses), and he missed 89 and 107. In any case, primes of this form became known as *Mersenne primes*. We still do not know many things about Mersenne primes, for example, whether there is an infinite number of them.

Marin Mersenne (1588-1648)

[Biography of Mersenne goes here. Some points to include:

- Mersenne was a French monk and polymath whose greatest contribution was the creation of the idea of sharing scientific information.
- Scientific journals did not yet exist, but Mersenne served as a "walking scientific journal," by exchanging letters with friends and sharing their ideas with each other. Fortunately, his friends were people like Galileo, Descartes, Fermat, and Pascal.
- When his letters were published after his death, they were in essence the world's first scientific proceedings.

In a letter to Mersenne in June 1640, Fermat wrote that his factorization of $2^{37} - 1$ depends on the following three discoveries:

1. If n is not a prime, $2^n - 1$ is not a prime.
2. If n is a prime, $2^n - 2$ is a multiple of $2n$.
3. If n is a prime, and p is a prime divisor of $2^n - 1$, then $p - 1$ is a multiple of n.

We'll look at the proof of #1 in a bit, but for now let's assume they are all true.

Fermat reasoned that if $2^{37} - 1$ is not prime, it must have a prime factor p, and by the observation 3 above, $p - 1$ is divisible by 37, which is equivalent to saying that

$$p = 37u + 1$$

Also, since p is odd, then u is even, i.e. $u = 2v$, so

$$p = 74v + 1$$

He therefore narrowed the factoring task from trying all possible numbers to just those primes produced by the above formula. Testing these in sequence:

What about $v = 1$? No, 75 is not a prime.

What about $v = 2$? No, 149 is prime, but is not a divisor of $2^{37} - 1$.

What about $v = 3$? Yes! 223 is prime, and is a divisor of $2^{37} - 1$.

Now let's look at Fermat's proof of discovery #1 above:

$$\text{If } 2^n - 1 \text{ is prime, then } n \text{ is prime}$$

Proof: Suppose n is not prime. Then there must be u and v such that

$$n = uv, \quad u > 1, \ \ v > 1$$

Then

$$\begin{aligned} 2^n - 1 &= 2^{uv} - 1 && (4.7)\\ &= (2^u)^v - 1 && (4.8)\\ &= (2^u - 1)((2^u)^{v-1} + (2^u)^{v-2} + \cdots + (2^u) + 1) && (4.9) \end{aligned}$$

where the last step again uses the difference of powers formula. Since $u > 1$, we know that both of the following are true:

$$\begin{aligned} 1 &< 2^u - 1\\ 1 &< (2^u)^{v-1} + (2^u)^{v-2} + \cdots + (2^u) + 1 \end{aligned}$$

So 4.9 shows that we have factored $2^n - 1$ into two numbers each greater than 1. But this contradicts the condition of the theorem is that $2^n - 1$ is prime! So our initial assumption must be incorrect, and n must be prime.

As for discoveries #2 and #3, Fermat never shared the proofs. In a letter to his friend Frenicle, Fermat wrote that "…he would send [the proof] if he did not fear being too long." We shall return to them soon.

Pierre de Fermat (1601-1665)

[Biography of Fermat goes here. Points to include:

- Fermat was a lawyer who lived in Toulouse.
- His friend Mersenne repeatedly invited him to visit Paris, but as far as we know, Fermat never went.
- He followed the convention of many of the older generation in not sharing his results. He would often say that he had a proof of something, yet come up with an excuse not to provide it.

]

Fermat made a lot of conjectures for which he left no proofs, but today every one has been proven true except one:

$$2^n + 1 \text{ is prime } \iff n = 2^i$$

Since then, numbers of this form $(2^{2^i} + 1)$ have been known as *Fermat primes*. It's easy to prove a part of his conjecture:

$$2^n + 1 \text{ is prime } \implies n = 2^i$$

Proof: Suppose $n \neq 2^i$. Then n must have an odd factor of the form $2q + 1$ which is > 1, so we can express n as

$$n = m(2q + 1)$$

Substituting $m(2q + 1)$ for n and then using the formula for sum of odd powers, we factor $2^n + 1$:

$$\begin{aligned} 2^n + 1 &= 2^{m(2q+1)} + 1 \\ &= 2^{m(2q+1)} + 1^{m(2q+1)} \\ &= (2^m)^{2q+1} + 1^{2q+1} \\ &= (2^m + 1)((2^m)^{2q} - (2^m)^{2q-1} + \cdots + 1) \end{aligned}$$

But factoring $2^n + 1$ contradicts the premise of the conjecture. So our assumption is incorrect, and $n = 2^i$.

What about other primes of the form $2^{2^i} + 1$? Fermat states that 3, 5, 17, 257, 65537, 4294967297, and 18446744073709551617 are prime, and so are all the rest of this form. Unfortunately, he was wrong about two of his examples — neither 4294967297 nor 18446744073709551617

are prime — and about his conjecture. In 1732 Euler showed that

$$2^{32} + 1 = 4294967297 = 641 \times 6700417$$

In fact, we now know that for $5 \leq i \leq 32$, the numbers are composite. Are there any more Fermat primes besides these five? No one knows.

4.4 The Little Fermat Theorem

We now come to the most important mathematical result of the first journey, which is known as the *Little Fermat Theorem*:

$$\text{If } p \text{ is prime, } a^{p-1} - 1 \text{ is divisible by } p \text{ for any } 0 < a < p.$$

It is important not only because of its practical utility (which we will soon describe), but also because it is the basis of a fundamental generalization in group theory, which we will see in Journey 2.

Fermat claimed to have the proof in 1640, but did not publish it. Leibniz discovered the proof some time between 1676 and 1680, but did not publish it either. Finally, Euler published two different proofs in 1742 and 1750. We will prove the theorem here, but first we need to derive several other results. While these may at first seem to be unrelated, we will see shortly how they come together.

Leonhard Euler (1707-1783)

[Biography of Euler goes here. Points to include:

- Euler (pronounced "OILer") born in Switzerland.
- Studied mathematics with the Bernoulli brothers, then got a job in St. Petersburg at age 23 doing mathematical research.
- After about 10 years, he moves to Berlin, at which point all the great leaders in Europe want him.
- Moves back to St. Petersburg for the rest of his career.
- After his death, the Russian Academy of Science takes 60 years to publish all the work he submitted.
- The Lutheran church made him a saint.

- Wrote the first book on popular science, *Letters to a German Princess.*
- Laplace said "*c'est notre maître à tous*" (he is the master of us all).

]

Our first step is another proposition from Euclid:

Euclid VII, 30:

The product of two integers smaller than a prime p is not divisible by p.

To not be divisible is to have a remainder, so we can restate the proposition like this:

$$p \text{ is prime } \ \wedge \ 0 < a, b < p \ \implies \ ab = mp + r \ \wedge 0 < r < p$$

Proof:
Assume the contrary, that ab is a multiple of p. Then for a given a, let b be the smallest integer such that $ab = mp$. Then since p is prime, we know dividing p by b leaves a remainder $v < b$:

$$p = bu + v \ \wedge \ 0 < v < b$$

Multiplying both sides by a and then substituting with $ab = mp$ gives:

$$\begin{aligned} ap &= abu + av \\ ap - abu &= av \\ ap - mpu &= av \\ av &= (a - mu)p \ \wedge \ v < b \end{aligned}$$

But this means that v is an integer smaller than b that satisfies $ab = np$. That's a contradiction, since we chose b to be the smallest such number. So our assumption is false, and ab is not divisible by p.

This is actually a common pattern for Greek proofs — choose the smallest of something, and then show that certain assumptions would lead to a smaller one.

Next, we prove a result about remainders:
Permutation of Remainders Lemma:

If p is prime, then for any $0 < a < p$,

$$\begin{aligned} a \cdot \{1, \ldots, p-1\} &= \{a, \ldots, a(p-1)\} \\ &= \{q_1 p + r_1, \ldots, q_{p-1} p + r_{p-1}\} \end{aligned}$$

where

$$0 < r_i < p \ \wedge \ i \neq j \implies r_i \neq r_j.$$

Basically, this says that if we take all the multiples of a from 1 to p, and express each multiple in the form $qp + r$, every remainder r will be unique and the set of remainders will be a permutation of $\{1, \ldots, p-1\}$. (We know each remainder is $< p$, so we have $p-1$ unique numbers in the range $[1, p-1]$.)
Proof: Suppose $r_i = r_j$ and $i < j$, i.e. two of the remainders were equal. Then we could take the difference of two elements in the set:

$$\begin{aligned}(q_j p + r_j) - (q_i p + r_i) &= q_j p - q_i p \\ &= (q_j - q_i)p\end{aligned}$$

Since the ith and jth elements of the set are the products ai and aj, we could equivalently write the difference of these two elements as $aj - ai$. That is:

$$\begin{aligned}aj - ai &= (q_j - q_i)p \\ a(j-i) &= (q_j - q_i)p\end{aligned}$$

But this is of the form $ab = mp$, i.e. it implies that the product of two integers smaller than p is divisible by p. Since this contradicts Euclid VII, 30, which we just proved above, our assumption must be false, and the remainders must be unique.

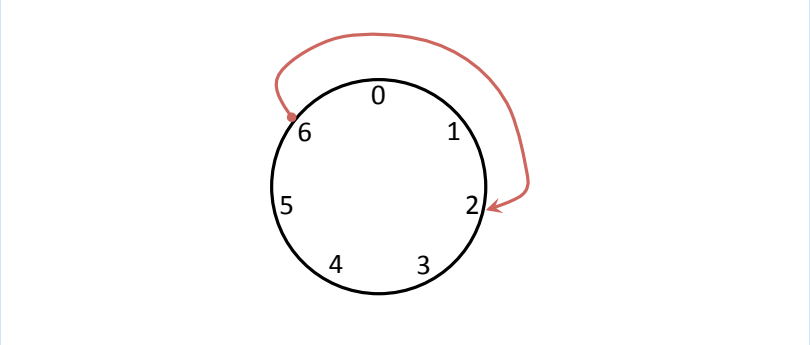
Now we will look at some results that deal with the notion of *cancellation*. If we are multiplying two numbers x and y, they cancel (i.e. their product is 1) when one is the multiplicative inverse of the other.

Cancellation and Modular Arithmetic

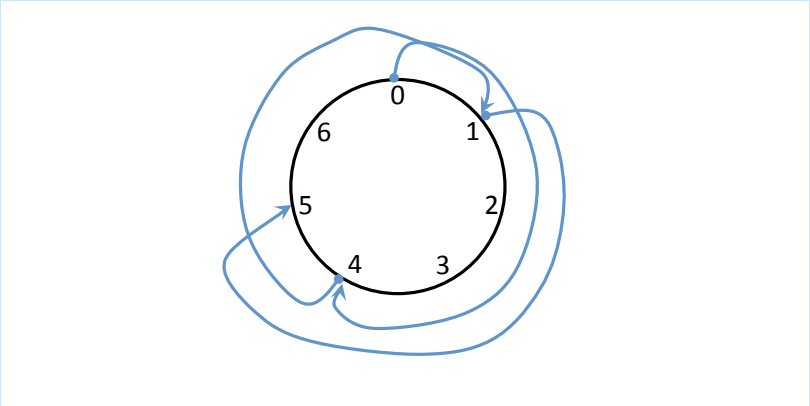
One way to view cancellation is in the context of *modular arithmetic*, which was introduced by Carl Gauss whom we shall meet in Journey 2. Although Euler did not use this technique in his proof, you may find it useful to understand the logic.

A good analogy for modular arithmetic is a standard 12-hour clock. If it's 10 o'clock, and you have to do something that's going to take 5 hours, you'll be done at 3 o'clock. In a sense, you're saying that 10 + 5 = 3. More precisely, you're saying that (10 + 5) mod 12 = 3. Of course, we can do modular arithmetic in any base. Here are a couple of examples using 7:

$$(6 + 3) \bmod 7 = 2$$



$$(3 \times 4) \bmod 7 = (4 + 4 + 4) \bmod 7 = 5$$



Notice in the later case that we could also express this product as:

$$(3 \times 4) = 12 = (1 \times 7) + 5$$

In other words, a product modulo n is equivalent to a remainder after using n as a divisor.

In regular arithmetic (arithmetic of natural numbers), if the product of two terms x and y is 1, then they are said to *cancel*, and x and y are called *inverses* of each other. The same is true in modular arithmetic, only the inverses will both be integers. For example

$$2 \times 4 \bmod 7 = 1$$

so 2 and 4 cancel, and are each other's inverse.

Formally, for integer $n > 1$ and integer $u > 0$, we call v a *multiplicative inverse modulo n* if there is an integer q such that $uv = 1 + qn$. In other words, *u and v are inverses if their product divided by n yields a remainder of 1*. We will rely heavily on this in the following

proofs.

The next result relies on this generalized notion of cancellation:
Cancellation Law:

If p is prime, then for any $0 < a < p$ there is a $0 < b < p$ such that $ab = mp + 1$.

In other words, a and b cancel with respect to p.
Proof: By the Permutation of Remainders Lemma we know that one of the possible products in the set

$$a \cdot \{1, \ldots, p-1\}$$

will have a remainder of 1. (It's the same idea as the pigeonhole principle.)

Note that 1 and $p-1$ are *self-canceling* elements — that is, if you multiply each by itself, the result is 1 mod p, or (equivalently), the result is of the form $mp+1$. It's obvious that $1 \cdot 1$ is of this form, since it's $0p + 1$. What about $p - 1$?

$$(p-1)^2 = p^2 - 2p + 1 = (p-2)p + 1 = mp + 1$$

It may help to view $p-1$ as -1, i.e. it's the position you'd get if you "turned the clock back" by 1.

In fact, 1 and $p-1$ are the *only* self-canceling elements,[2] which we'll now demonstrate:
Self-Canceling Lemma

$$\text{For any } 0 < a < p, \quad a^2 = mp + 1 \implies a = 1 \ \vee \ a = p - 1$$

Proof:
Assume there were a self-cancelling a that's neither 1 nor $p-1$:

$$a \neq 1 \ \wedge \ a \neq p - 1 \implies 1 < a < p - 1$$

Rearranging the condition of the proof, we have:

$$a^2 - 1 = mp$$

Factoring the expression on the left, we have:

$$(a-1)(a+1) = mp$$

But since by our assumption $0 < a - 1, a + 1 < p$, that means we have a product of two integers smaller than p that are divisible by p, a contradiction with Euclid VII, 30. So our

[2] For the mathematically inclined, what we are proving is that a quadratic equation over any field has two roots.

assumption is false, and the only self-cancellng elements are 1 and $p-1$.

We are almost ready to prove Fermat's Little Theorem, but we still need one more result: Wilson's theorem, announced by Edward Waring in 1770, who incorrectly attributed it to his student, John Wilson. At the time, Waring stated that he was unable to prove the theorem since he did not have the right notation — to which Gauss later remarked, "One needs notion, not notation."

Wilson's Theorem
If p is prime,

$$(p-1)! = mp + (p-1)$$

In other words, $(p-1)!$ gives the same remainder as $p-1$.

Proof: By definition,

$$(p-1)! = 1 \cdot 2 \cdot 3 \ldots (p-1).$$

By the Cancellation Law, every number a between 1 and $p-1$ has a number b in that range that's its inverse; by the Self-Canceling Lemma only 1 and $p-1$ are their own inverses. So every other number number in the product except 1 and $p-1$ is cancelled by its inverse, i.e. their product divided by p has remainder 1. In other words, we could express all the cancelled terms together — all the terms between 1 and $p-1$ — as $np+1$ for some n. We still have our uncanceled terms 1 and $p-1$, so our product now becomes:

$$\begin{aligned}(p-1)! &= 1 \cdot (np+1) \cdot (p-1) \\ &= np \cdot p - np + p - 1 \\ &= (np-n)p + (p-1) \\ &= mp + (p-1)\end{aligned}$$

since m can be any multiple of p.

Exercise 4.4

Prove that if $n > 4$ is composite then $(n-1)!$ is a multiple of n.

Finally, we can prove the Little Fermat Theorem:

If p is prime, $a^{p-1} - 1$ is divisible by p for any $0 < a < p$.

The proof essentially says that if we take the product of all the remainders, it's equal to the product of a permutation of all the remainders, because of commutativity.
Proof: Wilson's Theorem can be written as

$$\prod_{i=1}^{p-1} i = (p-1) + up$$

Therefore we can make the above substitution in the 2nd step below:

$$\begin{aligned}\prod_{i=1}^{p-1} ai &= a^{p-1}\prod_{i=1}^{p-1} i \\ &= a^{p-1}((p-1)+up) \\ &= a^{p-1}p - a^{p-1} + a^{p-1}up \\ &= (a^{p-1}+a^{p-1}u)p - a^{p-1}\end{aligned}$$

By the Permutation of Remainders Lemma, we can also express our original product in a different way:

$$\begin{aligned}\prod_{i=1}^{p-1} ai &= \prod_{i=1}^{p-1}(q_i p + r_i) \\ &= vp + \prod_{i=1}^{p-1} r_i \\ &= (p-1) + vp \\ &= -1 + (v+1)p\end{aligned}$$

We know our two expressions are equal, and need only some simple rearrangement:

$$\begin{aligned}-1 + (v+1)p &= (a^{p-1}+u)p - a^{p-1} \\ a^{p-1} - 1 + (v+1)p &= (a^{p-1}+u)p \\ a^{p-1} - 1 &= (a^{p-1}+u)p - (v+1)p \\ a^{p-1} - 1 &= (a^{p-1}+u-(v+1))p \\ a^{p-1} - 1 &= np\end{aligned}$$

So $a^{p-1} - 1$ is divisible by p.

We also observe that a^{p-2} is an inverse of a, since $a^{p-2} \cdot a = a^{p-1}$, which the Little Fermat Theorem tells us is $mp + 1$. (Remember that being an inverse with respect to p means having a remainder of 1 after dividing by p.)

What about the converse of the Little Fermat Theorem? To prove that, we need one more intermediate result:
Non-invertibility Lemma:

$$\text{If } n = uv \ \wedge \ 1 < u, v, \text{ then } u \text{ is not invertible modulo } n.$$

Proof: Let $n = uv$ and w be an inverse of u (i.e. $uv = mn + 1$). Then

$$\begin{aligned} wn &= wuv \\ &= (mn+1)v \\ &= mvn + v \\ wn - mvn &= v \end{aligned}$$

So

$$(w - mv)n = zn = v$$

Since $n > v$, then $zn > v$, which is a contradiction. So u cannot have an inverse.

This lemma tells us that when we are dealing with numbers modulo n, where n is not prime, there are invertible elements and noninvertible elements; elements that are not coprime are not invertible.

Converse of Little Fermat Theorem

$$\text{If for all } a,\ 0 < a < n, \quad a^{n-1} = 1 + q_a n \quad \text{then } n \text{ is prime.}$$

Proof: Suppose n is not prime, i.e. $n = uv$. Then by the non-invertibility lemma, u is not invertible. But we observed that $u^{n-2}u = u^{n-1} = 1 + q_u n$. In other words, u has an inverse, which is a contradiction. So n must be prime.

We started this chapter by seeing how the Greeks were interested in perfect numbers. There wasn't any practical value to this work; they were simply interested in exploring properties of certain kinds of numbers for their own sake. Yet over time, the search for these "useless" perfect numbers led to the discovery of the Little Fermat Theorem, one of the most practically useful theorems in all of mathematics. In particular, this theorem helps us to find large primes used daily in e-commerce, as we shall see in the next chapter.

Chapter 5

Public-Key Cryptography

In this chapter, we'll take the results we learned about primes and generalize them. This will provide the foundation needed to implement the well-known RSA algorithm for public-key cryptography.

5.1 Rethinking Our Results Using Modular Arithmetic

In chapter 4, we saw how modular multiplication was related to remainders. Let's take a closer look at some examples. Here's a multiplication table modulo 7:

$\times$	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

First we express the product as a multiple of 7 and a remainder; the modular product is then just the remainder. For example, $5 \times 4 = 20 = (2 \times 7) + 6 = 6 \bmod 7$, so there is a 6 in the table at the intersection of row 5 and column 4. Observe that every row is a permutation of every other row, and that every row contains a 1. Recall that if the product is 1, the two factors are inverses. In the table above, we see, for example, that 2 and 4 are inverses since $2 \times 4 = 1$. Here's a version of the table with the inverse of each factor on the left shown in the column on the right:

×	1	2	3	4	5	6	
1	**1**	2	3	4	5	6	1
2	2	4	6	**1**	3	5	4
3	3	6	2	5	**1**	4	5
4	4	**1**	5	2	6	3	2
5	5	3	**1**	6	4	2	3
6	6	5	4	3	2	**1**	6

We also note that only 1 and 6 (also known as -1) are their own inverses — that is, they are self-canceling — which is exactly what the Self-Canceling lemma told us. Furthermore, there are no zeros, because 7 is prime.

Let's take a look at a couple of our important results from the last chapter and see what some examples look like if we do them modulo 7.

Wilson's Theorem states that for a prime p,

$$(p-1)! = (p-1) + mp$$

Let's see if we can confirm that result if p is 7. We start by expanding 6! into its factors and rearranging them to put inverses together. With a bit of algebraic manipulation, we show that the result is some multiple of 7, with a remainder of 6:

$$\begin{aligned}
\mathbf{6!} &= 1 \times (2 \times 4) \times (3 \times 5) \times 6 \\
&= 1 \times (1 + 1 \times 7) \times (1 + 2 \times 7) \times 6 \\
&= (1 + 1 \times 7) \times (1 + 2 \times 7) \times 6 \\
&= (1 + 2 \times 7 + 1 \times 7 + 2 \times 7 \times 7) \times 6 \\
&= (1 + 17 \times 7) \times 6 \\
&= 6 + 17 \times 7 \times 6 \\
&= 6 + (17 \times 6) \times 7 \\
&= \mathbf{6} + 102 \times 7
\end{aligned}$$

Now we'll do the same for the Little Fermat Theorem, which says:

$$\text{If } p \text{ is prime, } a^{p-1} - 1 \text{ is divisible by } p \text{ for any } 0 < a < p.$$

Again, let's use $p = 7$, and try $a = 2$. First, we observe that:

$$\begin{aligned}
2^6 \times 6! &= (2 \times 1) \times (2 \times 2) \times (2 \times 3) \times (2 \times 4) \times (2 \times 5) \times (2 \times 6) \\
&= 2 \times 4 \times 6 \times (1 + 1 \times 7) \times (3 + 1 \times 7) \times (5 + 1 \times 7) \\
&= 2 \times 4 \times 6 \times ((1 \times 3 \times 5) + (\textit{lots of terms with } 1 \times 7 \textit{ in them})) \\
&= 6! + 7m
\end{aligned}$$

Then we use Wilson's Theorem to replace 6! with 6 on both sides of the equation. Wilson's theorem actually says we can replace 6! with $6 + 7m$ (a different m), but that just means we end up with a different multiple of 7 on the right, which we'll call u:

$$2^6 \times 6 = 6 + 7u$$

Then we multiply both sides by 6 (modulo 7). Since 6 is its own inverse, the 6s cancel and we can rearrange to get our desired result:

$$\begin{aligned} 2^6 \times 6 \times 6 &= (6 + 7u) \times 6 \\ 2^6 &= 1 + 7v \\ 2^6 - 1 &= 7v \end{aligned}$$

5.2 Euler's Theorem

Like any great mathematician, Euler was not satisfied with just proving the Little Fermat Theorem; he wanted to see if it could be generalized. Since the Little Fermat Theorem was only for primes, Euler wondered whether there was a similar result that would include composite numbers. But composite numbers do strange things in modular arithmetic. To illustrate this, let's take a look at the multiplication table modulo 10, again showing inverses of the left-hand factor in the right-hand column:

×	1	2	3	4	5	6	7	8	9	
1	1	2	3	4	5	6	7	8	9	1
2	2	4	6	8	0	2	4	6	8	
3	3	6	9	2	5	8	1	4	7	7
4	4	8	2	6	0	4	8	2	6	
5	5	0	5	0	5	0	5	0	5	
6	6	2	8	4	0	6	2	8	4	
7	7	4	1	8	5	2	9	6	3	3
8	8	6	4	2	0	8	6	4	2	
9	9	8	7	6	5	4	3	2	1	9

The table should look familiar, because it's just like the traditional 10×10 multiplication table, if you keep only the last digit of each product. Immediately we can see differences from the table we did for 7, which was prime. For one thing, the rows are no longer permutations of each other. More importantly, some rows now contain 0. That's a problem for multiplication — how can the product of two things be 0? That would mean

that we get into a situation where we can never escape zero — any product of the result will be zero.

The other property we noted earlier about primes — that only 1 and -1 are self-canceling — happens to be true for 10 as well, but is not always true for composite numbers. (8, for example, has four self-canceling elements: 1, 3, 5, and 7.)

Let's look at the multiplication table for 10 again, focusing on certain entries:

×	1	2	3	4	5	6	7	8	9	
1	1	2	3	4	5	6	7	8	9	1
2	2	4	6	8	0	2	4	6	8	
3	3	6	9	2	5	8	1	4	7	7
4	4	8	2	6	0	4	8	2	6	
5	5	0	5	0	5	0	5	0	5	
6	6	2	8	4	0	6	2	8	4	
7	7	4	1	8	5	2	9	6	3	3
8	8	6	4	2	0	8	6	4	2	
9	9	8	7	6	5	4	3	2	1	9

The rows that contain "good" products (i.e. no zeros) are the ones whose first factor is shown in red — which also happen to be the rows that have inverses. Which rows have this property? Those that represent numbers that are coprime with 10. (Remember, being coprime means having no common prime factors.)

So could we just use the good rows and leave out the rest? Not quite, because some of the results in good rows would themselves lead to bad rows if used in a successive product. (For example, 3 is a good row, but $(3 \times 5) \times 2 = 0$.) Euler's idea was basically to use only the entries in good *columns* as well as good rows — the numbers highlighted in red. Notice that those numbers have all the nice properties we saw for primes: the red numbers in each row are permutations of each other, each set of red numbers contains a 1, and so on.

Essentially, what Euler does is extend the Little Fermat Theorem by using only these red values. He starts by defining something called the *Euler Totient function*, the size of the set of coprimes:

$$\phi(n) = |\{0 < i < n \ \wedge \ \text{coprime}(i, n)\}|$$

which is equivalent to saying that $\phi(n)$ is the number of rows containing red entries in the multiplication table modulo n. For example, $\phi(10) = 4$, and $\phi(7) = 6$, as we can see from the multiplication tables above.

Since primes by definition don't share any prime factors with smaller numbers, the totient of a prime number is:

$$\phi(p) = p - 1$$

In other words, all numbers less than a given prime are coprime with it.

What Euler realized was that the $p-1$ in Fermat's theorem is just a special case; it's what ϕ happens to be for primes. Now we can state Euler's generalization of the Little Fermat Theorem.

Euler's Theorem:

$$\text{For } 0 < a < n,\ \text{coprime}(a, n) \iff a^{\phi(n)} = 1 + mn$$

Exercise 5.1

Prove Euler's theorem by modifying the proof of the Little Fermat Theorem. Steps:

- Replace Permutation of Remainders Lemma with Permutation of Coprime Remainders Lemma. (Essentially, use the same proof but look only at "red" elements.)
- Prove that every coprime remainder has a multiplicative inverse. (We just proved that the remainders form a permutation, so 1 has to be somewhere in the permutation.)
- Use the product of all coprime remainders where that proof of Little Fermat uses the product of all non-zero remainders. (Instead of using factorial, use product over all coprime numbers.)

Abstraction

Generalizing code is like generalizing theorems and their proofs. With algorithms, I might say, "well, I wrote this function to work for vectors, but could I make it also work for linked lists?" Euler did a similar thing: "I just proved something interesting for primes; is there a way to make it work for other numbers as well?"

Now let's look at the *totient of a power of a prime* p. We want to know the number of coprimes of p^m. We know there are at most $p^m - 1$ of them, because that's all the possible numbers less than p^m. But we also know that those divisible by p (i.e. multiples of p) are not coprime, so we need to subtract however many of these there are from our total:

$$\begin{aligned}
\phi(p^m) &= (p^m - 1) - |\{p, 2p, \ldots, p^m - p\}| \\
&= (p^m - 1) - |\{1, 2, \ldots, p^{m-1} - 1\}| \\
&= (p^m - 1) - (p^{m-1} - 1) \\
&= p^m - p^{m-1} \\
&= p^m(1 - \frac{1}{p})
\end{aligned}$$

The next step is to figure out what happens if we have two primes, i.e. the totient of $n = p^u q^v$. Again, we start with the maximum possible and then subtract off all the multiples. So we'll subtract the number of multiples of p and also the number of multiples of q, but then we have add back multiples of both p and q, because otherwise they'd be subtracted twice. (This general technique, known as the *inclusion-exclusion* principle, is often used in combinatorics.)

$$\begin{aligned}
\phi(n) &= (n-1) - (\frac{n}{p} - 1) - (\frac{n}{q} - 1) + (\frac{n}{pq} - 1) \\
&= n - \frac{n}{p} - \frac{n}{q} + \frac{n}{pq} \\
&= n(1 - \frac{1}{p} - \frac{1}{q} + \frac{1}{pq}) \\
&= n((1 - \frac{1}{p}) - \frac{1}{q}(1 - \frac{1}{p})) \\
&= n(1 - \frac{1}{p})(1 - \frac{1}{q}) \\
&= p^u(1 - \frac{1}{p})q^v(1 - \frac{1}{q}) = \phi(p^u)\phi(q^v)
\end{aligned}$$

So as a special case when we have a simple product of two primes, p_1 and p_2, we now know that

$$\phi(p_1 p_2) = \phi(p_1)\phi(p_2)$$

We'll be using this result later in the chapter.

Although the case we're going to care most about is the one above, we can generalize the formula to handle a product of any number of primes raised to powers, not just two. For example, if we had three factors p, q, and r, we'd subtract all the multiples of each, then add back the double-counted multiples of pq, pr, and qr, and then compensate for our overcompensation by again subtracting multiples of pqr. Extending this to m primes gives this formula:

$$\begin{aligned}
\phi(n) &= \phi(\prod_{i=1}^{m} p_i^{e_i}) \\
&= n \prod_{i=1}^{m} (1 - \frac{1}{p_i}) \\
&= \prod_{i=1}^{m} \phi(p_i^{e_i})
\end{aligned}$$

5.3 Primality Testing

> The problem of distinguishing prime number from composite . . . is known to be one of the most important and useful in arithmetic.
>
> C. F. Gauss, *Disquisitiones Arithmeticae*

We now have all the mathematical results we'll need for doing public-key cryptography. But we also need to know something about how to tell whether a number is prime. Gauss believed that (1) deciding whether a number is prime or composite is a very hard problem, and so is (2) factoring a number. He was wrong about #1, as we shall see. So far, he seems to be right about #2, which is a good thing for us, since modern cryptosystems are based on this assumption.

To find out if a number is prime, it helps to have a function that tells us whether it's divisible by a given number:

```
template <Integer I>
bool divides(const I& i, const I& n) {
  return n % i == I(0);
}
```

We can call this repeatedly to find the smallest divisor of a given number n. Notice that, just as we did with the Sieve of Eratosthenes, our loop for testing divisors will start at 3, advance by 2, and stop when the square of the current candidate reaches n:

```
template <Integer I>
I smallest_divisor(I n) {
  // precondition: n > 0
  if (even(n)) return I(2);
  for (I i(3); n >= i * i; i += I(2)) {
    if (divides(i, n)) return i;
  }
  return n;
}
```

Now we can create a simple function to determine whether n is prime:

```
template <Integer I>
I is_prime(const I& n) {
  return n > I(1) &&
         smallest_divisor(n) == n;
}
```

This is mathematically correct, but it's not going to be fast enough. Its complexity is $O(\sqrt{n}) = O(2^{(\log n)/2})$. That is, it's exponential in the *number of digits*. If we want to test a 200-digit number, we may be waiting for more time than the life of the universe.

To overcome this problem, we're going to need a different approach, which will rely on the ability to do modular multiplication. Here's a function object that provides what we need:

```
template <Integer I>
struct modulo_multiply {
  I modulus;
  modulo_multiply(const I& i) : modulus(i) {}
  I operator() const (const I& n, const I& m) {
    return (n * m) % modulus;
  }
};
```

We'll also need an identity element:

```
template <Integer I>
I identity_element(const modulo_multiply<I>&) {
  return I(1);
}
```

Now we can compute a multiplicative inverse modulo p. It uses the result we showed in the last chapter that the inverse of a is a^{p-2}:

```
template <Integer I>
I multiplicative_inverse_fermat(I a, I p) {
   // precondition: p is prime & a > 0
   modulo_multiply<I> op;
   return power_monoid(a, p - 2, op);
}
```

With these pieces, we can now use the Little Fermat Theorem to test if a number n is prime. Remember, the Little Fermat Theorem says that if p is prime, then $a^{p-1} - 1$ is divisible by p for any $0 < a < p$. We want to know if n is prime. So we take *some* number a smaller than n, raise it to the $n - 1$ power, and check if the result is 1. (We call the number a we're using a *witness*.) If the result is *not* equal to 1, we know definitely by the contrapositive of the theorem that n is not prime. If the result *is* equal to 1, we know that there's a good chance that n is prime, and if we do this for lots of random witnesses, there's a very good chance:

```
template <Integer I>
bool fermat_test(I n, I witness) {
```

```
  // precondition: 0 < witness < n
  modulo_multiply<I> op(n);
  I exp(power_semigroup(witness, n - I(1), op);
  return exp == I(1);
}
```

The Fermat test is very fast, because we have a fast way to raise a number to a power — our $O(\log n)$ generalized Egyptian Multiplication algorithm from Chapter 2.

While the Fermat test works the vast majority of the time, it turns out that there are some pathological cases of numbers that will fool it for all coprime witnesses; they produce remainders of 1 even though they're composite. These are called *Carmichael numbers*.

Definition 5.1. *A composite number $n > 1$ is a* Carmichael number *if and only if*

$$\forall b > 1,\ \ coprime(b, n) \implies \exists m,\ b^{n-1} = 1 + mn$$

or equivalently

$$\forall b > 1,\ \ coprime(b, n) \implies b^{n-1} = 1 \bmod n$$

Exercise 5.2

Implement the function: `bool is_carmichael(n)`

Exercise 5.3

Using your function from Exercise 5.2, find the first seven Carmichael numbers.

To avoid worrying about Carmichael numbers, we're going to use an improved version of our primality tester, called the *Miller-Rabin* test [cite]; it will again rely on the speed of our power algorithm.

We know that $n - 1$ is even (it would be awfully silly to run a primality test on an even n), so we can represent $n - 1$ as the product $2^k \cdot q$. The Miller-Rabin test explores a sequence of squares $w^{2^0 q}, w^{2^1 q}, \ldots, w^{2^k q}$. We're going to rely on the self-canceling lemma from the last chapter, except we'll write it with new variable names and assuming modular multiplication:

$$\text{For any } 0 < x < n \wedge \text{ prime}(n),\ \ x^2 = 1 \bmod n \implies x = 1 \vee -1$$

If we find some $x^2 = 1 \bmod n$ where x is neither 1 nor -1, then n is not prime. Now we can make two observations: (1) If $x^2 = 1 \bmod n$, then there's no point in squaring x again, because the result won't change; if we reach 1, we're done. (2) If $x^2 = 1 \bmod n$, and x is not -1, then we know n is not prime. Here's the code:

```
template <Integer I>
bool miller_rabin_test(I n, I q, I k, I w) {

  // precondition n > 1  ∧  n − 1 = 2^k q  ∧  q is odd

  modulo_multiply<I> op(n);
  I x = power_semigroup(w, q, op);
  if (x == I(1) || x == n - I(1)) return true;
  for (I i(1); i < k; ++i) {

    // invariant x = w^(2^(i−1) q)

    x = op(x, x);
    if (x == n - I(1)) return true;
    if (x == I(1))     return false;
  }
  return false;
}
```

Note that we pass in q and k as arguments. Since we're going to call the function many times with different witnesses, we don't want to refactor $n-1$ every time.

Let's look at an example. Suppose we want to know if $n = 2793$ is prime. We choose a random witness $w = 150$. We factor $n - 1 = 2792$ into $2^2 \cdot 349$, so $q = 349$ and $k = 2$. We compute

$$x = w^q \bmod n = 150^{349} \bmod 2792 = 2019$$

Since the result is neither 1 nor -1, we start squaring x

$$i = 1;\ \ x^2 = 150^{2^1 \cdot 349} = 1374$$
$$i = 2;\ \ x^2 = 150^{2^2 \cdot 349} = 2601$$

Since we haven't reached 1 or -1 yet, and $i = k$, we can stop and return false; 2793 is not prime.

Like the Fermat test, the Miller-Rabin test is right most of the time. Unlike the Fermat test, the Miller-Rabin test has a provable guarantee: it is right at least 75% of the time for a random witness w. (In practice, it's even more often.) Randomly choosing, say, 100 witnesses makes the probability of error less than 1 in 2^{200}. As Knuth remarked, "It is much more likely that our computer has dropped a bit, due ... to cosmic radiations." [cite]

AKS: A New Test for Primality

In 2002 Neeraj Kayal and Nitin Saxena, two undergraduate students at the Indian Institute of Technology at Kanpur, together with their advisor, Manindra Agrawal, came up with a *deterministic* polynomial-time algorithm for primality testing, and published their result. [cite] This is a problem that people in number theory had been working on for centuries.

[More about the technique, known as "AKS" after the initials of the authors...]

There is a very clear paper by Andrew Granville describing the technique [cite]. Although it is a dense mathematical paper, it is understandable to a surprisingly wide audience. (This is unusual; most important mathematical results being published in recent decades require years of prior mathematical study to be understood.) Determined readers who are willing to put in serious effort are encouraged to read it.

Despite the great accomplishment of the AKS algorithm, we're not going to use it here, because our probabilistic algorithm is still considerably faster.

5.4 Cryptology

Cryptology is the science of secret communication. *Cryptography* is concerned with developing codes and ciphers; *cryptanalysis* with breaking them.[1]

Cryptanalysis and Computer Science

[Perhaps include very brief historical notes about cryptology – Julius Caesar, Thomas Jefferson,"The Gold Bug," etc. Then jump to Bletchley Park, Turing. A lot of computer science came out of the work in British intelligence breaking codes.]

A *cryptosystem* consists of algorithms for encrypting and decrypting data. The original data is called the *plaintext*, and the encrypted data is called the *ciphertext*. A set of *keys* determine the behavior of the encryption and decryption algorithms:

$$\text{ciphertext} = \text{encryption}(key_0, \text{plaintext})$$
$$\text{plaintext} = \text{decryption}(key_1, \text{ciphertext})$$

The system is *symmetric* if $key_0 = key_1$; otherwise it is *asymmetric*.

[1] Technically, a *code* is a system for secret communication where a meaningful concept such as the name of a person, place, or event is replaced with some other text, while a *cipher* is a system for modifying text at the level of its representation (letters or bits). But we'll use the terms interchangeably; in particular we'll use *encode* and *decode* informally to mean *encipher* and *decipher*.

Many early cryptosystems were symmetric and used secret keys. The problem with this is that the sender and receiver of the message both must have the keys in advance. If the key is compromised and the sender wants to switch to a new one, he has the additional problem of figuring out how to secretly convey the new key to the receiver.

A *public-key cryptosystem* is an encryption scheme in which users have a pair of keys, a public key *pub* for encrypting, and a private key *prv* for decrypting. If Alice wants to send a message to Bob, she encrypts the message with Bob's public key. The ciphertext is then unreadable to anyone but Bob, who uses his private key to decrypt the message.

In order to have a public-key cryptosystem, the following requirements must be satisfied:

1. The encryption function needs to be a *one-way* function: easy to compute, with an inverse that is hard to compute. Here, "hard" has its technical computer science meaning of taking exponential time — in this case, exponential in the size of the key.

2. The inverse function has to be easy to compute when you have access to a certain additional piece of information, known as the *trapdoor*.

3. Both encryption and decryption algorithms are publicly known. This insures the confidence of all parties in the technique being used.

A function meeting the first two requirements is known as a *trapdoor one-way function*.

Who Invented Public-Key Cryptography?

For years, it was believed that Stanford professor Martin Hellman, together with two graduate students, Whitfield Diffie and Ralph Merkle, invented public-key cryptography in 1976. They proposed how such a system would work, and realized it would require a trapdoor one-way function. Unfortunately, they didn't have one — it was just a theoretical construct for them. In 1977, Ron Rivest, Adi Shamir, and Len Adleman came up with a procedure for creating a trapdoor one-way function, which became known as the *RSA* algorithm after the inventors' initials. In 1997, the British government disclosed that one of their intelligence researchers, Clifford Cocks, had actually invented a special case of RSA in 1973 — but it took 20 years after the publication of the RSA algorithm before they declassified Cocks' memo. After that, Admiral Bobby Ray Inman, the former head of the U.S. National Security Agency, claimed that his agency had invented some sort of public-key cryptographic technique even earlier, in the 1960s, although no evidence was given. Who knows which country's intelligence agency will come forward next with an earlier claim?

5.5 The RSA Algorithm: How and Why It Works

The RSA cryptosystem, named after its creators (Rivest, Shamir, and Adleman), uses the mathematical results for primality testing. RSA requires two steps: key-generation, which needs to be done only rarely, and encoding/decoding, which is done every time a message is sent or received.

Key generation works as follows. First, the following values are computed:

- two random large primes p_1 and p_2 (the Miller-Rabin test makes this feasible)
- their product $n = p_1 p_2$
- $\phi(p_1 p_2) = (p_1 - 1)(p_2 - 1)$
- a random public key *pub*, coprime with $\phi(p_1 p_2)$
- a private key *prv*, the multiplicative inverse of *pub* modulo $\phi(p_1 p_2)$. (We will derive the algorithm in our second journey.)

When these computations are complete, p_1 and p_2 are destroyed; *pub* and n are published, and *prv* is kept secret. At this point, there is no feasible way to factor n, since it is such a large number with such large factors.

The encoding and decoding process is simpler. The text is divided into equal-size blocks, say 256 bytes long, which are interpreted as large integers. The message block size s must be chosen so that $n > 2^s$. To encode a plaintext block, we use our familiar power algorithm:

```
power_semigroup(plaintext_block, pub, modulo_multiply<I>(n));
```

Decoding looks like this:

```
power_semigroup(ciphertext_block, prv, modulo_multiply<I>(n));
```

Note that *we do exactly the same operation to encode and decode.* The only difference is which text and which key we pass in.

How does RSA work? Encryption consists of raising a message m to a power *pub*; decryption consists of raising the result to the power *prv*. We need to show that the result of applying these two operations is the original message m (modulo n):

$$(m^{\text{pub}})^{\text{prv}} = m \bmod n$$

Proof: Recall that we specifically created *prv* to be the multiplicative inverse of *pub* modulo $\phi(p_1 p_2)$, so by definition, the product of *pub* and *prv* is some multiple q of $\phi(p_1 p_2)$ with a remainder of 1, and we can make that substitution in the exponent on the right.

After a bit of algebraic manipulation, we can then apply Euler's theorem and replace $m^{\phi(p_1 p_2)}$ with 1 plus some multiple of n.

$$\begin{aligned}(m^{\text{pub}})^{\text{prv}} &= m^{\text{pub}\times\text{prv}} \\ &= m^{1+q\phi(p_1p_2)} \\ &= mm^{q\phi(p_1p_2)} \\ &= m(m^{\phi(p_1p_2)})^q \\ &= m(1+vn)^q \\ &= m+wn\end{aligned}$$

Note that when we expand $(1+vn)^q$, every term will be a multiple of n except 1, so we can collapse all of these and just say we have 1 plus some other multiple of n.

The Euler theorem step depends on m being coprime with n, aka $p_1 p_2$. Since the message m could be anything, how do we know that it will be coprime? In practice, it turns out to be true with probability approaching 1, and people normally do not worry about it. However, if you want to address this, you can add one extra byte to the end of m. The extra byte is not actually part of the message, but is there only to insure that we have a coprime. When we create m, we check if it is coprime, and if it isn't, we simply add 1 to this extra byte.

Why does RSA work? In other words, why do we believe it's secure? The reason is that factoring is hard, and therefore, computing ϕ is not feasible. However, if quantum computers prove to be realizable in the future, it will be possible to run an exponential number of divisor tests in parallel, making factoring a polynomial time problem.

Journey One Project

1. Implement an RSA key generation library.
2. Implement an RSA message encoder/decoder that takes a string and the key as its arguments.

Hints:

- You'll need to download and install a package for handling arbitrary-precision integers ("bignums") in C++.
- Two numbers are coprime if their greatest common divisor (GCD) is 1. This will come in handy for one of the key generation steps.
- You'll need some results that we're going to derive in the second journey; for now, we'll just provide the code below.

```
template <EuclideanDomain I>
std::pair <I, I> extended_gcd(I a , I b) {
  if (b == 0) return std::make_pair(I(1), a);
  I u(1);
  I v(0);
  while (true) {
    I q = a / b;
    a -= q * b;  // a = a % b;
    if (a == 0) return std::make_pair(v, b);
    u -= q * v;

    q = b / a;
    b -= q * a;  // b = b % a;
    if (b == 0) return std::make_pair(u, a);
    v -= q * u;
  }
}
```

This function returns the multiplicative inverse of a modulo n if it exists, or 0 if it does not:

```
template <Integer I>
I multiplicative_inverse(I a, I n) {
  std::pair<I, I> p = extended_gcd(a, n);
  if (p.second != I(1)) return I(0);
  if (p.first < I(0)) return p.first + n;
  return p.first;
}
```

You'll need this to get the private key from the public key.

5.6 Lessons of the Journey

Our first journey led us from surveyors in ancient Egypt to modern cryptographic algorithms, introducing some aspects of number theory along the way. We also saw the first examples of abstraction, letting us take an algorithm designed for multiplication and use it to solve many other problems. We'll spend more time looking at abstraction in the next journey.

Just as the ancient Greeks harvested the knowledge of the Egyptians, so programmers today can harvest the knowledge of mathematicians. While some ideas in mathematics may not make sense to translate, others have immediate impact on our work.

Journey Two: Heirs of Pythagoras

Chapter 6

Euclid and the Greatest Common Measure

Our second journey is about abstraction. How can you take an algorithm and find the most general setting in which it applies? Although we saw some examples of this in the first journey, we're going to go deeper here, into an area of mathematics called abstract algebra. Although abstract algebra is relatively new by math standards, our story will begin in ancient Greece.

In Chapter 3, we met Pythagoras and the secret order he founded to study astronomy, geometry, number theory, and music. In this chapter, we'll take a closer look at the challenge the Pythagorean school took on.

6.1 The Original 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 two fundamental perceptible aspects of our reality. In addition to their focus on understanding "figurate" numbers (such as square, oblong, and triangular), they believed that there was discrete structure to space. Their challenge, then, was to provide a way to ground geometry in numbers — essentially, to have a unified theory of mathematics based on natural numbers.

To do this, they came up with the idea that one segment could be "measured" by another:

Definition 6.1. *A segment V is a* measure *of a segment A if and only if A can be represented as a finite concatenation of copies of V.*

Note that a measure must be small enough that an exact integral number of copies produces the desired segment; there are no "fractional" measures. Of course, different

measures might be used for different segments. If one wanted to use the same measure for two segments, it had to be a *common measure*:

Definition 6.2. *A segment V is a* common measure *of segments A and B if and only if it is a measure of both.*

For any given situation, there is a common measure for all the objects of interest. Therefore, space could be represented discretely.

Since there could be many common measures, they also came up with the idea of the *greatest common measure*:

Definition 6.3. *A segment V is the* greatest common measure *of A and B if it is greater than any other common measure of A and B.*

The Pythagoreans also recognized several properties of greatest common measure (gcm), which we represent in modern notation as:

$$gcm(a, a) = a \tag{6.1}$$
$$gcm(a, b) = gcm(a, a + b) \tag{6.2}$$
$$b < a \implies gcm(a, b) = gcm(a - b, b) \tag{6.3}$$
$$gcm(a, b) = gcm(b, a) \tag{6.4}$$

Using these properties, they came up with the most important procedure in Greek mathematics — perhaps in all mathematics — a way to compute the greatest common measure of two segments. In C++, it looks like this:

```
line_segment gcm(line_segment a, line_segment b) {
    if (a == b) return a;
    if (b < a)  return gcm(a - b, b);
    if (a < b)  return gcm(a, b - a);
}
```

This code makes use of the *trichotomy law*, a common pattern based on the fact that if you have two things a and b of the same ordered type, then either $a = b$, $a < b$, or $a > b$.

Let's look at an example. What's `gcm(196, 42)`?

196 > 42, so gcm(196, 42) = gcm(196 − 42, 42) = gcm(154, 42)
154 > 42, so gcm(154, 42) = gcm(154 − 42, 42) = gcm(112, 42)
112 > 42, so gcm(112, 42) = gcm(112 − 42, 42) = gcm(70, 42)
70 > 42, so gcm(70, 42) = gcm(70 − 42, 42) = gcm(28, 42)
28 < 42, so gcm(28, 42) = gcm(28, 42 − 28) = gcm(28, 14)
28 > 14, so gcm(28, 14) = gcm(28 − 14, 14) = gcm(14, 14)

$14 = 14$, so we're done!

Of course, when we say `gcm(196, 42)`, we really mean gcm of segments with length 196 and 42, but for the examples in this chapter, we'll just use the integers as shorthand.

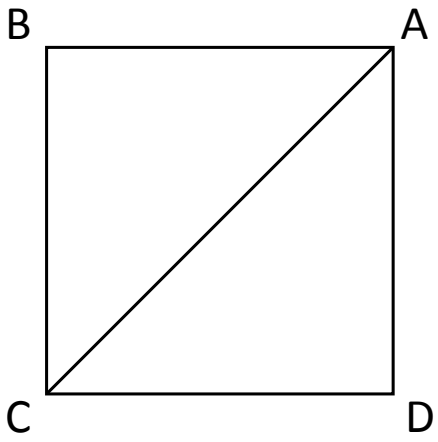
Note: We're going to use versions of this algorithm throughout this journey, so it's important to understand it and have a good feel for how it works. You may want to try computing a few more examples by hand to convince yourself.

6.2 A Fatal Flaw in the Program

Greek mathematicians found that the *well-ordering principle* — the fact that any set of natural numbers has a smallest element — provided a powerful proof technique. In order to prove that something does not exist, prove that if it did exist, a smaller one would also exist.

Using this logic, the Pythagoreans discovered a proof that undermined their entire program.[1] We're going to use a 19th century reconstruction of the proof by George Chrystal [cite].

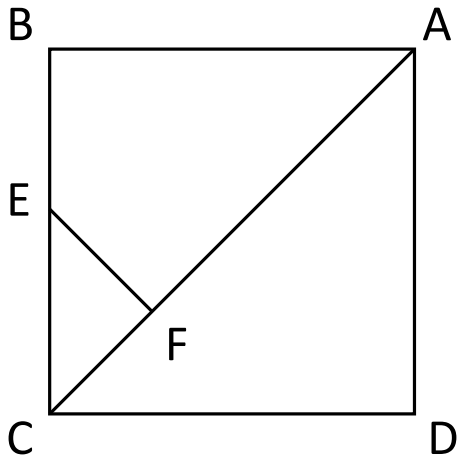
Assume that there is a segment that can measure both the side and diagonal of some square. Let us take the smallest such square for this segment:



Using a "ruler" and "compass"[2], we can construct a segment $\overline{AF}$ with the same length as $\overline{AB}$, and then create a segment starting at F and perpendicular to $\overline{AC}$.

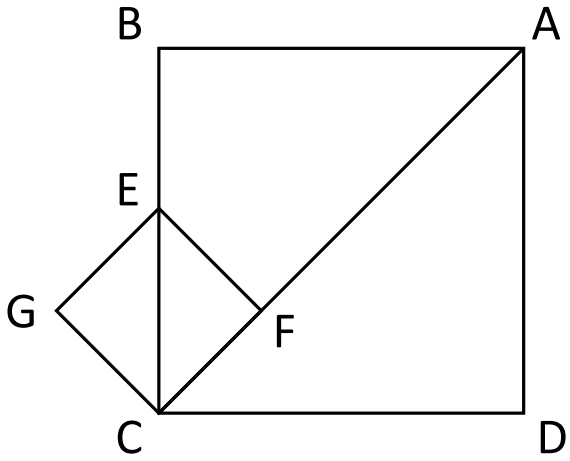
[1] We don't know if Pythagoras himself made this discovery, or one of his early followers.

[2] Although modern readers may think of a ruler as being used to measure distances, for Euclid it was only a way to draw straight lines. Similarly, although a modern compass can be fixed to measure equal distances, Euclid's compass was only used to draw circles with a given radius.



$$\overline{AB} = \overline{AF} \ \wedge \ \overline{AC} \perp \overline{EF}$$

Now we construct two more perpendicular segments, $\overline{CG}$ and $\overline{EG}$:

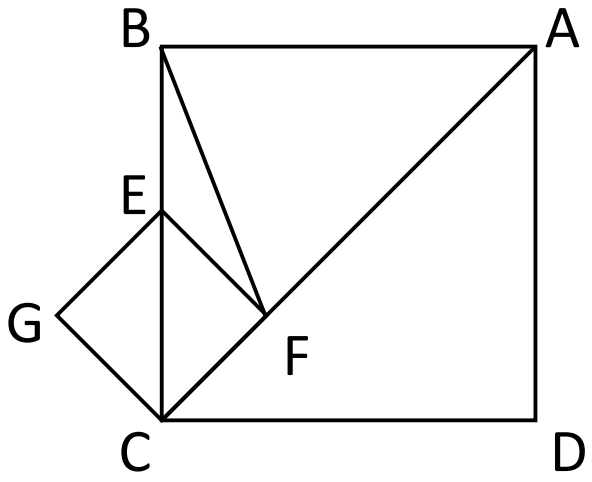


$$\overline{AC} \perp \overline{CG} \ \wedge \ \overline{EG} \perp \overline{EF}$$

We know that $\angle CFE = 90^\circ$ (by construction) and that $\angle ECF = 45^\circ$ (since it's the same as $\angle BCA$, which is the angle formed by the diagonal of a square, and therefore is half of 90°). We also know that the three angles of a triangle sum to 180°. Therefore

$$\angle CEF = 180^\circ - \angle CFE - \angle ECF = 180^\circ - 90^\circ - 45^\circ = 45^\circ$$

So $\angle CEF = \angle ECF$, which means CEF is an isosceles triangle, so the sides opposite equal angles are equal, i.e. $\overline{FC} = \overline{FE}$. Finally, we add one more segment $\overline{BF}$:



Triangle ABF is also isosceles, with $\angle ABF = \angle AFB$, since we constructed $\overline{AB} = \overline{AF}$. And $\angle ABC = \angle AFE$, since both were constructed with perpendiculars. So

$$\begin{aligned} \angle ABC - \angle ABF &= \angle AFE - \angle AFB \\ \angle EBF &= \angle EFB \\ \implies \overline{BE} &= \overline{EF} \end{aligned}$$

Now, we know $\overline{AC}$ is measurable since that's part of our premise, and we know $\overline{AF}$ is measurable, since it's the same as $\overline{AB}$, which is also measurable by our premise. So by equation 6.3, $\overline{FC} = \overline{AC} - \overline{AF}$ is measurable. Since we just showed that CEF and BEF are both isosceles:

$$\overline{FC} = \overline{EF} = \overline{BE}$$

We know $\overline{BC}$ is measurable, again by our premise, and we've just shown that $\overline{FC}$, and therefore $\overline{BE}$ is measurable. So $\overline{EC} = \overline{BC} - \overline{BE}$ is measurable.

So we now have a smaller square whose side ($\overline{EF}$) and diagonal ($\overline{EC}$) are both measurable by our common unit. But our original square was chosen to be the smallest for which the relationship held — a contradiction. So our original assumption was wrong: *There is no segment that can measure both the side and diagonal of a square.* And if you try to find one, you'll be at it forever — our `line_segment_gcm(a, b)` procedure will not terminate.

To put it another way, the ratio of the diagonal and the side of a square cannot be expressed as a rational number (the ratio of two integers). Today we would say that with this proof, the Pythagoreans had discovered irrational numbers, and specifically, that $\sqrt{2}$ is irrational.

The discovery of irrational numbers was unbelievably shocking. It undermined the Pythagoreans' entire program; it meant that geometry could not be grounded in numbers. So they did what many organizations do when faced with bad news: they swore everyone to secrecy. When one of the order leaked the story, legend has it that the gods punished him by sinking the ship carrying him, drowning all on board.

Eventually, Pythagoras' followers came up with a new strategy. If they couldn't unify mathematics on a foundation of numbers, they would unify it on a foundation of geometry. This was the origin of the straightedge-and-compass constructions still used today to teach geometry; no numbers are used, or needed.

Later mathematicians came up with an alternate, number-theoretic proof of the irrationality of $\sqrt{2}$. One version was included as proposition 117 in some editions of Book X of Euclid's *Elements*. However, it is no longer believed to be from Euclid's time, and was more likely added later. In any case, it is an important proof:

Proof that $\sqrt{2}$ is irrational: Assume $\sqrt{2}$ were rational. Then it could be expressed as the ratio of two integers m and n, where m/n is irreducible:

$$\begin{aligned} m/n &= \sqrt{2} \\ (m/n)^2 &= 2 \\ m^2 &= 2n^2 \end{aligned}$$

So m^2 is even, which means that m is also even,[3] so we can write it as 2 times some number u and do a bit more algebraic manipulation:

$$\begin{aligned} m &= 2u \\ (2u)^2 &= 2n^2 \\ 4u^2 &= 2n^2 \\ 2u^2 &= n^2 \end{aligned}$$

So n^2 is even, which means that n is also even. But if m and n are both even, then m/n is not irreducible — a contradiction. So our assumption is false — there is no way to represent $\sqrt{2}$ as the ratio of two integers.

The fact that we have two completely different proofs of the same result is good. It's quite common for mathematicians to look for multiple proofs of the same mathematical fact, since it increases their confidence in the result. (Gauss spent his much of his career coming up with multiple proofs for one important theorem, the quadratic reciprocity law.)

While at first glance we might think the Pythagoreans were naive to believe that they could represent continuous reality with discrete numbers, computer scientists do the same thing today — we approximate the real world with binary numbers, often with a fixed

[3] This is easily shown: The product of two odd numbers is an odd number, so if m were not even, m^2 could not be even. Euclid proved this and many other results about odd and even numbers earlier in *Elements*.

resolution. In fact, the tension between continuous and discrete has remained a central theme in mathematics through the present day, and will probably be with us forever. However, it has been the source of great progress and revolutionary insights for ages.

6.3 Athens and Alexandria

> At the center of scientific life stood the personality of Plato. He guided and inspired scientific work in his Academia and outside.
>
> B.L. van der Waerden
> *Science Awakening*

We now turn to one of the most amazing times and places in history: Athens in the 5th century BC. For 150 years following the miraculous defeat of the invading Persians in the battles of Marathon, Salamis, and Platea, Athens became the center of culture, learning, and science, laying the foundations for much of western civilization.

It was in the middle of this period of Athenian cultural dominance that Plato invented what is essentially the world's first university, the Academy (named after the mythical hero, Academus). Although we think of Plato today as a philosopher, the center of the Academy's program was the study of mathematics, and he had the inscription Μηδείς αγεωμέτρητος εισίτω — "Let no one ignorant of geometry enter" — written over the entrance. Among his discoveries were what we now call the five *Platonic solids* — the only physical shapes in which every side is an identical regular polygon.

Plato (427BC–347BC)

[Biography of Plato goes here. Points to include:

- Born Aristocles, the son of a prominent family. He was given the nickname Πλάτων (Pláton) — Greek for "the broad one" — by his wrestling coach.
- Became a follower of Socrates, one of the founders of Philosophy, who taught and learned by questioning everything.
- Came up with the idea of education as a public activity (as opposed to going to learn the secrets of the priests).
- ...

]

Athenian culture spread throughout the Mediterranean, especially during the reign of Alexander the Great. Among his achievements, he founded the Egyptian city of Alexandria (named for himself) which became the new center of research and learning. Over a thousand scholars worked in what we would now think of as a research institute, the *Mouseion* — the "Institution of the Muses" — from which we get our word "museum." Their patrons were the Greek kings of Egypt, the Ptolemys, who paid their salaries and provided free room and board. Part of the Mousein was the Library of Alexandria, which was given the task of collecting all the world's knowledge. Supposedly containing 500,000 scrolls, the library kept a large staff of scribes busy copying, translating, and editing scrolls.

It was during this period that Euclid, one of the scholars at the Mouseion, wrote his *Elements*, one of the most important books in the history of mathematics. Euclid incorporated mathematical results and proofs from many existing texts, but he wove them together to form a carefully crafted coherent story. In Book I, he starts with the fundamental tools for geometric construction with straightedge and compass and ends with what we now call the Pythagorean Theorem. In the thirteenth and final book, he shows how to construct the five Platonic solids, and proves that they are the only regular polyhedra (bodies whose faces are congruent, regular polygons) that exist.

Euclid (325BC-265BC)

We know very little about Euclid, although we know he took his Geometry very seriously. According to a story told by the philosopher Proclus:

> Ptolemy [the king of Egypt] once asked Euclid whether there was any shorter way to a knowledge of geometry than by study of the Elements, whereupon Euclid answered that there was no royal road to geometry.

It appears that Euclid studied at the Academy some time after Plato's death, and brought the mathematics he learned to Alexandria. [Nothing else is known about Euclid!]

6.4 Greatest Common Measure, Revisited

Book X of Euclid's *Elements* contained a concise summary of the existence of incommensurable quantities, i.e. irrational numbers:

Proposition 2. *If, when the less of two unequal magnitudes is continually subtracted in turn from the greater that which is left never measures the one before it, then the two magnitudes are incommensurable.*

Essentially, Euclid is saying what we observed earlier in the chapter: If our procedure for computing greatest common measure never terminates, then there is no common measure.

Euclid then goes on to explicitly describe the algorithm and prove that it computes the gcm. Since this is the first algorithm termination proof in history, we're including the entire text:

Proposition 3. *Given two commensurable magnitudes, to find their greatest common measure.*

> Let the two given commensurable magnitudes be AB, CD of which AB is the less; thus it is required to find the greatest common measure of AB, CD.
>
> Now the magnitude AB either measures CD or it does not.
>
> If then it measures it—and it measures itself also—AB is a common measure of AB, CD.
>
> And it is manifest that it is also the greatest; for a greater magnitude than the magnitude AB will not measure AB.
>
> Next, let AB not measure CD.
>
> **Then, if the less be continually subtracted in turn from the greater, that which is left over will sometime measure the one before it**, because AB, CD are not incommensurable; [cf. X. 2] let AB, measuring ED, leave EC less than itself, let EC, measuring FB, leave AF less than itself, and let AF measure CE.
>
> Since, then, AF measures CE, while CE measures FB, therefore AF will also measure FB.
>
> But it measures itself also; therefore AF will also measure the whole AB.
>
> But AB measures DE; therefore AF will also measure ED.
>
> But it measures CE also; therefore it also measures the whole CD.
>
> Therefore AF is a common measure of AB, CD.
>
> I say next that it is also the greatest.
>
> For, if not, there will be some magnitude greater than AF which will measure AB, CD.
>
> Let it be G.
>
> Since then G measures AB, while AB measures ED, therefore G will also measure ED.
>
> But it measures the whole CD also; therefore G will also measure the remainder CE.
>
> But CE measures FB; therefore G will also measure FB.
>
> But it measures the whole AB also, and it will therefore measure the remainder AF, the greater the less: which is impossible.
>
> Therefore no magnitude greater than AF will measure AB, CD; therefore AF is the greatest common measure of AB, CD.

> Therefore the greatest common measure of the two given commensurable magnitudes AB, CD has been found. Q. E. D. [cite]

This *anthyphairesis* ("continual subtraction") approach to gcm is known as *Euclid's algorithm* (or sometimes *the Euclidean algorithm*). It's basically an iterative version of the one we saw earlier. We write it like this:

```
line_segment gcm0(line_segment a, line_segment b) {
  while (a != b) {
    if (b < a) a = a - b;
    else       b = b - a;
  }
  return a;
}
```

Exercise 6.1

`gcm0` is inefficient when one segment is much longer than the other. Come up with a more efficient implementation. (Remember that the arguments are segments; the code must "run" with straightedge and compass, i.e. you can't introduce operations that couldn't be done by geometric construction.)

Exercise 6.2

Prove that if a segment measures two other segments, then it measures their greatest common measure.

To work toward a more efficient version of `line_segment_gcm`, we'll start by rearranging, checking for $b < a$ as long as we can:

```
line_segment gcm1(line_segment a,  line_segment b) {
while (a != b) {
    while (b < a) a = a - b;
    std::swap(a, b);
  }
  return a;
}
```

We could avoid a swap in the case where $a = b$, but that would require an extra test, and we're not quite ready to optimize the code anyway. Instead, we observe that the inner `while` loop is computing the *remainder* of a and b. Let's factor out that piece of functionality:

```
line_segment
segment_remainder(line_segment a, line_segment b) {
    while (b < a) a = a - b;
    return a;
}
```

How do we know the loop will terminate? It's not as obvious as it might appear. For example, if our definition of `line_segment` included the half line starting at a point and continuing infinitely in one direction, the code would not terminate. The required assumptions are encapsulated in what is known as the *Axiom of Archimedes*:

> For any quantities a and b, there is a natural number n such that $a \leq nb$.

Essentially, what this says is that there are no infinite quantities. (Remember that the Greeks did not allow for the possibility of a zero-length segment.)

Now we can rewrite our gcm function with a call to `segment_remainder`:

```
line_segment gcm(line_segment a, line_segment b) {
while (a != b) {
    a = segment_remainder(a, b);
    std::swap(a, b);
  }
  return a;
}
```

So far we have refactored our code but not improved its performance. Most of the work is done in `segment_remainder`. To speed up that function, we will use the same idea we used in Journey 1 — doubling and halving our quantities. This requires knowing something about the relationship of doubled segments to remainder:

Recursive Remainder Lemma. If $r = \text{segment\_remainder}(a, 2b)$ then

$$\text{segment\_remainder}(a, b) = \begin{cases} r & \text{if } r \leq b \\ r - b & \text{if } r > b \end{cases}$$

Suppose, for example, that we wanted to find the remainder of something when divided by 10. We'll try to take the remainder divided by 20. If the result is less than 10, we're done. If the result is between 11 and 20, we'll take away 10 from the initial result and get the remainder that way.

Using this strategy, we can write our faster function:

```
line_segment
fast_segment_remainder(line_segment a, line_segment b)
{
```

```
  if (a <= b) return a;
  if (a - b <= b) return a - b;
  a = fast_segment_remainder(a, b + b);
  if (a <= b) return a;
  return a - b;
}
```

It's recursive, but it's a less intuitive form of *upward* recursion. In most recursive programs, we go down from n to $n-1$ when we recurse; here, we're making our argument *bigger* every time, going from n to $2n$. It's not obvious where the work is done, but it works.

Let's look at an example. Suppose we want to find the remainder of 35 divided by 6:

$a = 35, b = 6.$
$a \leq b?$ $(35 \leq 6?)$ No.
$a - b \leq b?$ $(29 \leq 6?)$ No.
Recurse :
  $a = 35, b = 12$
  $a \leq b?$ $(35 \leq 12?)$ No.
  $a - b \leq b?$ $(23 \leq 12?)$ No.
  Recurse :
    $a = 35, b = 24$
    $a \leq b?$ $(35 \leq 24?)$ No.
    $a - b \leq b?$ $(11 \leq 24?)$ Yes, return $a - b = 11$
  $a = 11$
  $a \leq b?$ $(11 \leq 12?)$ Yes, return $a = 11$
$a = 11$
$a \leq b?$ $(11 \leq 6?)$ No.
return $a - b = 11 - 6 = 5$

This peculiar algorithm was actually known long before the Greeks. In fact, it's a variant of another of the algorithms Ahmes included in the Rhind Papyrus. Essentially, it's the inverse of the Egyptian Multiplication algorithm from Journey 1, and is known as the *Egyptian division* algorithm. Remember that since the Greeks had no notion of a zero-length segment, their remainders were in the range [1, n].

Of course, we still have the overhead of recursion, so we'll eventually want to come up with an iterative solution, but we'll put that aside for now.

Finally, we can plug it in to our gcm function, providing a solution to Exercise 6.1:

```
line_segment
fast_segment_gcm(line_segment a, line_segment b) {
  while (a != b) {
    a = fast_segment_remainder(a, b);
    std::swap(a, b);
  }
  return a;
}
```

Of course, no matter how fast it is, it still will never terminate if a and b do not have a common measure.

6.5 The Strange History of Zero

The next development in the history of GCD required something the Greeks didn't have: zero. Many people have heard that ancient societies had no notion of zero, and that it was invented by Indians or Arabs, but this is only partially correct. In fact, Babylonian astronomers were using zero as early as 1500 BC, together with a positional number system. However, their number system used base 60. Despite this, the rest of their society used base 10 — for example, in commerce — without either zero or positional notation. Amazingly, this state of affairs persisted for *centuries*. Greek astronomers eventually learned the Babylonian system and used it (still in base 60) for their trigonometric computations, but again, this approach was used only for this one application and was unknown to the rest of society. (They also started using the Greek letter omicron, which looks just like our letter "o," to represent zero.)

What is particularly surprising about the lack of zero outside of astronomy is that it persisted despite the fact that the *abacus* was well known and commonly used for commerce in nearly every ancient civilization. Abaci consist of stones or beads arranged in columns; the columns corresponded to 1s, 10s, 100s, etc., and each bead represented one unit of a given power of 10. In other words, ancient societies used a device that represented numbers in base-10 positional notation, yet there was no commonly used written representation of zero for another 1000 years.

The unification of a written form of zero with a decimal positional notation is due to early Indian mathematicians some time around the 6th century AD. It then spread to Persia between the 6th and 9th century AD. Arab scholars learned the technique and spread it across their empire, from Cordoba in the west to Baghdad in the east. There is no evidence that zero was known anywhere in Europe outside this empire (even in the rest of Spain); 300 years would pass before this innovation crossed from one culture to the other.

This period of history was not kind to the formerly great societies of Europe. In Byzantium, the Greek-speaking Eastern remnant of the former Roman Empire, mathematics was

still studied, but innovation declined. By the 6th-7th century, scholars still read Euclid, but usually just the first book of *Elements*. And as we noted in Chapter 4, Latin translations didn't even bother to include the proofs. By the end of the first millennium, if you were a European who wanted to study mathematics, you had to go to Cairo or Baghdad.

The breakthrough came in 1203 when Leonardo Pisano, also known as Fibonacci, published *Liber Abaci* (The Book of Calculating). In addition to introducing zero and positional decimal notation, this astonishing book described for Europeans, for the first time, the standard algorithms for doing arithmetic that we are now taught in elementary school: long addition, long subtraction, long multiplication, long division. We might consider it the first book of computer science.

Leonardo Pisano (1170-1250)

The city of Pisa, which today is landlocked, was a major port, trading center, and naval power in the 12th century. 1000 Pisan traders crisscrossed the Mediterranean and Constantinople [Byzantium], and the Pisan government sent trade representatives to major cities to insure their success. One of these representatives, a certain Bonaccio, was posted to Algeria. He decided to bring his son Leonardo along, a decision that changed the course of history. Leonardo learned "Hindu digits" from the Arabs, and continued his studies during business trips to Egypt, Syria, Sicily, Greece, and Provence. He would go on to introduce their innovations (including zero) to Europe, and contribute some of the most important mathematical developments in centuries. He called himself Leonardo Pisano (Leonardo the Pisan), although since the 19th century he is usually known as *filius Bonacci* ("son of Bonaccio"), or Fibonacci for short. [More biography of Leonardo here. Frederick II, etc.]

Leonardo would go on to write several more books an various branches of mathematics. His *Liber Quadratorum* (The Book of Squaring) is probably the greatest book on number theory between Diophantus and Fermat.

Exercise 6.3 (easy)

Prove that $\sqrt[3]{16} + \sqrt[3]{54} = \sqrt[3]{250}$.

Why was a problem like this difficult for the Greeks? They had no terminating procedure for computing cube roots (in fact, it was later proven that no such process exists). So from their perspective, the problem starts out: "First, execute a nonterminating procedure..."

Leonardo's insight will be familiar to any middle-school algebra student, but it was

revolutionary in the 13th century. Basically, what he said was,"Even though I don't know how to compute $\sqrt[3]{2}$, I'll just pretend I do and assign it an arbitrary symbol."

Here's another example of the kind of problem Leonardo solved:

Exercise 6.4

Prove the following proposition from *Liber Quadratorum*: For any odd square number x there is an even square number y, such that $x + y$ is a square number.

Exercise 6.5 (hard)

Prove the following proposition from *Liber Quadratorum*: If x and y are both sums of two squares, then so is their product xy. (This is an important result that Fermat builds on.)

Once zero was widely used in mathematics, it actually took centuries longer before it occurred to anyone that a segment could have zero length — specifically, the segment $\overline{AA}$.

6.6 Remainder, Quotient, and GCD

Zero-length segments force us to rethink our gcm and remainder procedures, because Archimedes' axiom no longer holds — we can add a zero-length segment forever, and we'll never exceed a nonzero segment. So we'll allow a to be zero, but we need a precondition to insure that b is not zero. On the other hand, having zero lets us shift our remainders to the range $[0, n-1]$, which will be crucial for modular arithmetic and other developments. Here's the code:

```
line_segment
fast_segment_remainder1(line_segment a, line_segment b)
{
  // precondition: b != 0
  if (a < b) return a;
  if (a - b < b) return a - b;
  a = fast_segment_remainder1(a, b + b);
  if (a < b) return a;
  return a - b;
}
```

Notice that the only thing we've changed are the conditions; everywhere we used to say `a <= b`, we now check `a < b`.

Let's see if we can get rid of the recursion. Every time we recurse down, we double b, so in the iterative version, we'd like to precompute the maximum amount of doubling

we'll need. We can define a function that finds the first repeated doubling of b that exceeds the difference $a - b$:

```
line_segment
largest_doubling(line_segment a, line_segment b) {
  // precondition: b != 0
  while (a - b >= b) b = b + b;
  return b;
}
```

Of course, on the way out of the recursion we'll need the "undoubled" version of b, so we'll need a way to compute half. Remember, we're still "computing" with straightedge and compass. Fortunately, there is a Euclidean procedure for "halving" a segment[4], so we can use a `half` function. Now we can write an iterative version of remainder:

```
line_segment
remainder(line_segment a, line_segment b) {
  // precondition: b != 0
  if (a < b) return a;
  line_segment c = largest_doubling(a, b);
  a = a - c;
  while (c != b) {
    c = half(c);
    if (c <= a) a = a - c;
  }
  return a;
}
```

The first part of the function, which finds the largest doubling value, does what the "downward" recursion does, while the last part does what happens on the way back

[4]Draw a circle with the center at one end of the segment and radius equal to the segment; repeat for the other end. Use ruler to connect the two points where the circles intersect. The resulting line will bisect the original segment.

up out of the recursive calls. Let's look at our example again:

$a = 35, b = 6$
$a < b$? $(35 < 6?)$ No.
$c =$ `largest_doubling`$(35, 6) = 24$
$a = a - c = 35 - 24 = 11$
loop :
  $c \neq b$? $(24 \neq 6)$? Yes, keep going.
    $c =$ `half`$(c) =$ `half`$(24) = 12$
    $c \leq a$? $(12 \leq 11)$? No.
  $c \neq b$? $(12 \neq 6)$? Yes, keep going.
    $c =$ `half`$(c) =$ `half`$(12) = 6$
    $c \leq a$? $(6 \leq 11)$? Yes. $a = a - c = 11 - 6 = 5$
  $c \neq b$? $(6 \neq 6)$? No, done with loop.
return $a = 5$

Notice that the successive values of c in the iterative implementation are the same as the values of b following each recursive call in the recursive implementation.

This is an extremely efficient algorithm, nearly as fast as the hardware-implemented remainder operation in modern processors.

Suppose we wanted to compute *quotient* instead of remainder? It turns out that the code is almost the same. All we need are a couple of minor modifications, shown in red:

```
integer quotient(line_segment a, line_segment b) {
  // Precondition: b > 0
  if (a < b) return integer(0);
  line_segment c = largest_doubling(a, b);
  integer n(1);
  a = a - c;
  while (c != b) {
      c = half(c); n = n + n;
      if (c <= a) { a = a - c; n = n + 1; }
  }
  return n;
}
```

Basically, we are going to count multiples of b. If $a < b$, then we don't have any multiples of b and we return 0. But if $a \geq b$, we initialize the counter to 1, then double it each time we halve c, adding one more multiple for each iteration when it fits. Again, let's work

through an example — this time instead of finding the remainder of 35 divided by 6, we'll find the quotient of 35 divided by 6.

$a = 35, b = 6$
$a < b$? $(35 < 6?)$ No.
$c =$ `largest_doubling`$(35, 6) = 24$
$n = 1$
$a = a - c = 35 - 24 = 11$
loop :
  $c \neq b$? $(24 \neq 6)$? Yes, keep going.
    $c =$ `half`$(c) =$ `half`$(24) = 12$; $n = n + n = 1 + 1 = 2$
    $c \leq a$? $(12 \leq 11)$? No.
  $c \neq b$? $(12 \neq 6)$? Yes, keep going.
    $c =$ `half`$(c) =$ `half`$(12) = 6$; $n = n + n = 2 + 2 = 4$
    $c \leq a$? $(6 \leq 11)$? Yes. $a = a - c = 11 - 6 = 5$; $n = n + 1 = 4 + 1 = 5$
  $c \neq b$? $(6 \neq 6)$? No, done with loop.
return $n = 5$

Essentially, this is the Egyptian multiplication algorithm in reverse. And as we mentioned earlier, Ahmes knew it — a primitive variant of this, known to the Greeks as *Egyptian division*, appears in the Rhind papyrus.

Since the majority of the code is shared between quotient and remainder, it would make much more sense to combine them into a single function; the complexity of the combined function is the same as either individual function:

```
std::pair<integer, line_segment>
quotient_remainder(line_segment a, line_segment b) {
  // Precondition: b > 0
  if (a < b) return std::make_pair(integer(0), a);
  line_segment c = largest_doubling(a, b);
  integer n(1);
  a = a - c;
  while (c != b) {
      c = half(c); n = n + n;
      if (c <= a) { a = a - c; n = n + 1; }
  }
  return std::make_pair(n, a);
}
```

In fact, any quotient or remainder function does nearly all the work of the other.

Programming Principle: The Law of Useful Return

Our `quotient_remainder` function illustrates an important programming principle, which we call *The Law of Useful Return*. If you've already done a bunch of work to get a result, don't throw it away. Return it to the caller. This may allow the caller to get some extra work done "for free" (as in the `quotient_remainder` case), or return data that can be used in future invocations of the function.

Unfortunately, this principle is not always followed. For example, while early processors had a combined quotient and remainder instruction, the Intel instruction set implements quotient and remainder in two separate instructions, doubling the cost when both results are needed. This decision was based on the fact that the C programming language has separate quotient and remainder operators; the instruction set designers felt that there was no point in implementing the unified instruction if no one would use it. This led to a chicken-and-egg problem, where language designers don't want to add the unified function because there is no instruction supporting it.

Most computing architectures, whether straightedge-and-compass or modern CPUs, provide an easy way to compute `half` — for us, it's just a 1-bit right shift. However, if you should happen to be working with an architecture that doesn't support this, there is a remarkable version of the remainder function by Floyd and Knuth that does not require halving. It's based on the idea of the Fibonacci sequence — another of Pisano's inventions. Instead of the next number being double of the previous one, we'll make the next number be the sum of the two previous ones:[5]

```
line_segment
remainder_fibonacci(line_segment a, line_segment b) {
  // Precondition: b > 0
  if (a < b) return a;
  line_segment c = b;
  do {
    line_segment tmp = c; c = b + c; b = tmp;
  } while (a >= c);
  do {
    if (a >= b) a = a - b;
    line_segment tmp = c - b; c = b; b = tmp;
  } while (b < c);
  return a;
}
```

[5]Note that this sequence starts at b, so the values will not be the same as the traditional Fibonacci sequence.

The first loop is equivalent to computing `largest_doubling` in our previous algorithm. The second loop corresponds to the "halving" part of the code. But instead of halving, we use subtraction to get back the earlier number in the Fibonacci sequence. This works because we always keep one previous value around in a temporary variable.

Exercise 6.6

Design `quotient_fibonacci` and `quotient_remainder_fibonacci`.

Now that we have an efficient implementation of remainder, we can now return to our original problem, greatest common measure. Using our new remainder function, we can rewrite Euclid's algorithm like this:

```
line_segment
gcm_remainder(line_segment a, line_segment b) {
  while (b != line_segment(0)) {
    a = remainder(a, b);
    std::swap(a, b);
  }
  return a;
}
```

Note that since we now allow remainder to return zero, the termination condition for the main loop is when b (the previous iteration's remainder) is zero, instead of comparing a and b as we did originally.

We will use this algorithm for the rest of our journey, leaving its structure intact but exploring how it applies to different types. For example, for integers, the equivalent function is *greatest common divisor* (GCD):

```
integer gcd(integer a, integer b) {
  while (b != integer(0)) {
    a = a % b;
    std::swap(a, b);
  }
  return a;
}
```

Note that the code is identical, except that we have replaced `line_segment` with `integer` and used the modulus operator `%` to compute remainder. In practice, a remainder function like this — rather than the idea of computing remainders by doubling and halving — is used to compute GCD. This first became possible when Leonardo Pisano introduced positional decimal notation and long division.

[A more general treatment of the material in this chapter is in EoP Chapter 5.]

[Some final words for chapter go here.]

Chapter 7

From Line Segments to Concepts

[Intro to chapter here.]

7.1 Polynomials and GCD

> ... with one stroke, the classical restrictions of "numbers" to integers or to rational fractions was eliminated. [Stevin's] general notion of a real number was accepted by all later scientists.
>
> B.L. van der Waerden
> *History of Algebra*

Some of the most important contributions to mathematics were due to one its least-known figures, the 16th century Flemish mathematician Simon Stevin. In addition to his contributions to engineering, physics, and music, Stevin revolutionized the way we think about and operate on numbers. In his 1585 book *De Thiende* ("The Tenth"), published in English as *Disme: The Art of Tenths, or, Decimall Arithmetike*, Steven invents and explains the use of decimal fractions. This was the first time anyone had proposed using positional notation in the other direction — for tenths, hundredths, etc. *Disme* (pronounced "dime") was one of the most widely read books in the history of mathematics. It was one of Thomas Jefferson's favorites, and is the reason why U.S. currency has a coin called a "dime," and uses decimal fractions rather than the British pounds, shillings, and pence in use at the time.

Simon Stevin (1548-1620)

[Biography of Stevin goes here. Points to include:

- Stevin lived during the Dutch Golden Age. Prior to this period, the Netherlands provinces were part of the Spanish empire, led by its Hapsburg kings. In 1568, the Dutch began a war of independence, united by a common culture and language, ultimately creating a republic and then an empire of their own.
- Stevin, a Dutch patriot and military engineer, developing a system of sluices that could be used to flood invading Spanish troops.
- Things Stevin invented or studied: Parallelogram of forces, hydrostatics, dropped things and measured acceleration before Galileo, relationship of frequency and music intervals etc.
- Proponent of Dutch language, used word frequency and length to "prove" that it was the best language.

]

In *Disme*, Stevin expands the notion of number from integers and fractions to "that which expresseth the quantitie of each thing." Essentially, Stevin invented the entire concept of real numbers and the number line. Any quantity could go on the number line, including negative numbers, irrational numbers, and what he called "inexplicable" numbers (by which he may have meant transcendental numbers). Of course, Stevin's decimal representations had their own drawbacks, particularly the need to write an infinite number of digits to express a simple value, such as 0.142857142857142857142857142857... for 1/7.

Stevin's representation enabled the solution of previously unsolvable problems. For example, he showed how to compute cube roots, which had given the Greeks so much trouble. He used the *Intermediate Value Theorem*, which essentially says that if a continuous function is negative at one point and positive at another, then there must be an intermediate point where its value is zero. Stevin's idea was to find the number (initially, an integer) where the function goes from negative to positive, then divide the interval between that number and the next into tenths, and repeat the process with the tenths, hundredths, and so on. He realized that by "zooming in," any such problem could be solved to whatever degree of accuracy was needed, or as he put it, "one may obtain as many decimals of [the true value] as one may wish and come indefinitely near to it."

Although Stevin saw how to represent any number as a point along a line, he did not make the leap to showing pairs of numbers as points on a plane. That invention — what we now call Cartesian coordinates — came from the great French philosopher and mathematician Rene Descartes (Renatus Cartesius in Latin).

Rene Descartes (1596-1650)

[Biography of Descartes here.]

Stevin's next great achievement was the invention of polynomials, also introduced in 1585, in a book called *Arithmétique*. Consider the expression:

$$4x^4 + 7x^3 - x^2 + 27x - 3$$

Prior to Stevin's work, the only way to construct such a number was by performing an algorithm: Take a number, raise it to the 4th power, multiply it by 4, and so on. In, fact, they would need a different algorithm for every polynomial. Stevin realized that a polynomial is simply a finite sequence of numbers — $\{4, 7, -1, 27, -3\}$ for the example above. In modern computer science terms, we might say that Stevin was the first to realize that *code could be treated as data*.

With Stevin's insight, we can pass polynomials as data to a generic evaluation function. We'll write one that takes advantage of *Horner's rule*, which uses associativity to insure that we never have to multiply powers of x higher than 1:

$$4x^4 + 7x^3 - x^2 + 27x - 3 = (((4x + 7)x - 1)x + 27)x - 3$$

For a polynomial with n terms, we need $n - 1$ multiplications and $n - 1$ additions. Using this rule, we can implement a polynomial evaluation function like this:

```
template <InputIterator I, Semiring R>
R polynomial_value(I first, I last, R x) {
  if (first == last) return R(0);
  R sum(*first);
  while (++first != last) {
    sum = sum * x;
    sum = sum + *first;
  }
  return sum;
}
```

What's interesting about this code are the requirements on the types I and R. I is an iterator, because we want to iterate over the sequence of coefficients.[1] But the value type of the iterator (the type of the coefficients of the polynomial) does not have to be equal to

[1] We'll explain iterators more formally in the next Journey, but for now we can think of them as generalized pointers.

the semiring R (the argument type of the polynomial and its result type).[2] For example, if we have a polynomial like $ax^2 + b$ where the coefficients are real numbers, that doesn't mean x has to be a real number — in fact, it could be something completely different, like a matrix.

Exercise 7.1

What are the requirements on R and the value type of the iterator? In other words, what are the requirements on coefficients of polynomials and on their values?

Stevin's breakthrough allowed polynomials to be treated as numbers and to participate in normal arithmetic operations. To add or subtract polynomials, we simply add or subtract common elements. To multiply, we multiply every combination of elements, i.e. if a_i and b_i are the ith coefficients of the polynomials being multiplied (starting from the lowest-order term), and c_i is the ith coefficient of the result, then:

$$\begin{aligned} c_0 &= a_0 b_0 \\ c_1 &= a_0 b_1 + a_1 b_0 \\ c_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\ &\vdots \\ c_k &= \sum_{k=i+j} a_i b_j \\ &\vdots \end{aligned}$$

To divide polynomials, we need the notion of *degree*.

Definition 7.1. *The degree of a polynomial is the index of the highest coefficient.*

For example:

$$\begin{aligned} \deg(5) &= 0 \\ \deg(x+3) &= 1 \\ \deg(x^3 + x - 7) &= 3 \end{aligned}$$

Now we can define division with remainder:

Definition 7.2. *Polynomial a is divisible by polynomial b with remainder if there are polynomials q and r such that*

$$a = bq + r \ \wedge \ \deg(r) < \deg(b)$$

[2] A semiring is something whose elements can be added and multiplied and has distributivity.

Doing polynomial division with remainder is just like doing long division of numbers:

$$
\begin{array}{r|l}
 & 3x^2 + 2x - 2 \\
\hline
x-2 & 3x^3 - 4x^2 - 6x + 10 \\
 & \underline{3x^3 - 6x^2} \\
 & \quad 2x^2 - 6x \\
 & \quad \underline{2x^2 - 4x} \\
 & \quad\quad -2x + 10 \\
 & \quad\quad \underline{-2x + 4} \\
 & \quad\quad\quad 6
\end{array}
$$

Exercise 7.2

Prove that for any two polynomials $p(x)$ and $q(x)$

1. $p(x) = q(x) \cdot (x - x_0) + r \implies p(x_0) = r$
2. $p(x_0) = 0 \implies p(x) = q(x) \cdot (x - x_0)$

Since polynomials are just numbers, Stevin realized that he could use the same Euclidean algorithm (the one we looked at in the previous chapter) to compute their gcd:

```
polynomial<real> gcd(polynomial<real> a, polynomial<real> b) {
   while (b != polynomial<real>(0)) {
     a = remainder(a, b);
     std::swap(a, b);
   }
   return a;
}
```

Stevin's realization is the essence of what we now call generic programming: *an algorithm in one domain can be applied in another domain*.

How do we know this algorithm works? By "works," we mean two things: first that it actually computes gcd, and second, that the number of steps is finite:

1. $a = qb + r \implies \gcd(a, b) = \gcd(b, r)$
2. $\deg(r) < \deg(\mathrm{b})$

Obviously, if a divisor divides a and it divides b, then it must divide r, because $r = a - qb$; it's the same proof we had for gcd of integers. r is also divisible by gcd, so we can just reduce. And how do we know the algorithm terminates? Because at every step, the degree of r is reduced.

Exercise 7.3 (from Chrystal, *Algebra*)

Find gcd of:

1. $16x^4 - 56x^3 - 88x^2 + 278x + 105$,
   $16x^4 - 64x^3 - 44x^2 + 232x + 70$

2. $7x^4 + 6x^3 - 8x^2 - 6x + 1$,
   $11x^4 + 15x^3 - 2x^2 - 5x + 1$

3. $nx^{n+1} - (n+1)x^n + 1$,
   $x^n - nx + (n-1)$

7.2 Göttingen and German Mathematics

Wir müssen wissen — wir werden wissen!
We must know — we will know!

– David Hilbert [cite]

In the 18th and 19th centuries, starting long before Germany existed as a unified country, German culture flourished. Composers like Bach, Mozart, and Beethoven, poets like Goethe and Schiller, and philosophers like Kant, Hegel, and Marx were creating timeless works of depth and beauty. German universities created a unique role for German professors as civil servants bound by their commitment to the truth. Eventually this system would produce the greatest mathematicians and physicists of their age, many of them teaching or studying at the University of Götttingen.

The University of Göttingen

The center of German mathematics was a seemingly unlikely place: The University of Göttingen. Unlike many great European universities which started hundreds of years earlier in medieval times, Göttingen was relatively young, founded in 1734. And the city of Göttingen was not a major population center. Despite this, the University of Göttingen was home to an astonishing series of top mathematicians, including Gauss, Riemann, Dirichlet, Dedekind, Klein, Minkowski, and Hilbert, many of whom we will discuss later in this journey. By the late 20th century its community of physicists was equally impressive, including quantum theorists Max Born and Werner Heisenberg.

Göttingen's greatness was destroyed in 1933 by the Nazis, who expelled all Jews from the faculty and student body — including many of the top physicists and mathematicians. Some years later, the Minister of Education asked the great German mathematician David Hilbert, "How is mathematics in Göttingen now that it has been freed of Jewish influence?" Hilbert replied, "Mathematics in Göttingen? There is none any more."

Perhaps the most important mathematician to come out of Göttingen was Carl Friedrich Gauss, who was the founder of German mathematics in the modern sense. Among his many accomplishments was his seminal work on number theory, described in his 1801 book *Disquisitiones Arithmeticae* ("Investigations of Arithmetic"). Gauss's book is to number theory what Euclid's *Elements* is to geometry — the foundation on which all work in the field is based. Among other results, it includes the *Fundamental Theorem of Arithmetic*, which states that every integer has a unique prime factorization.

Carl Friedrich Gauss (1777-1855)

[Biography of Gauss goes here. Some points to include:

- How he became a mathematician – construction of 17-gon.
- Called *princeps mathematicorum* ("the prince of mathematics").
- Didn't publish, but claimed priority anyway.
- Worked on electromagnetism and co-invented a version of the telegraph.
- ...

]

Another of Gauss's innovations was the notion of *complex numbers*. Mathematicians had used imaginary numbers (xi where $i^2 = -1$) for over 200 years, but they were not well understood and were usually avoided. The same was true for the first 30 years of Gauss's career; we have evidence from his notebooks that he used imaginary numbers to derive some of his results, but then he reconstructed the proofs so the published versions would not mention i. ("The metaphysics of i is very complicated," he wrote in a letter.)

But in 1831, Gauss had an unbelievable insight: he realized that numbers of the form $z = a + bi$ could be viewed as points (a, b) on a Cartesian plane. These *complex numbers*, he saw, were just as legitimate and self-consistent as any other numbers.

Here are a few definitions and properties we'll use for complex numbers:

complex number:	$z = x + yi$
complex conjugate:	$\overline{z} = x - yi$
real part:	$\mathrm{Re}(z) = \frac{1}{2}(z + \overline{z}) = x$
imaginary part:	$\mathrm{Im}(z) = \frac{1}{2i}(z - \overline{z}) = y$
norm:	$z\overline{z} = \|z\| = x^2 + y^2$
absolute value:	$\lvert z\rvert = \sqrt{\|z\|} = \sqrt{x^2 + y^2}$
argument:	$\arg(z) = \arccos(\frac{x}{\lvert z\rvert})$

The absolute value is the length of the vector z on the complex plane, while the argument is the angle between the real axis and the vector z. For example, $|i| = 1$ and $\arg(i) = 90^\circ$.

Just as Stevin did for polynomials, Gauss demonstrated that complex numbers were in fact full-fledged numbers capable of supporting ordinary arithmetic operations:

addition:	$z_1 + z_2 = (x_1 + x_2) + (y_1 + y_2)i$
subtraction:	$z_1 - z_2 = (x_1 - x_2) + (y_1 - y_2)i$
multiplication:	$z_1 z_2 = (x_1 x_2 - y_1 y_2) + (x_2 y_1 + x_1 y_2)i$
reciprocal:	$\frac{1}{z} = \frac{\overline{z}}{\|z\|} = \frac{x}{x^2+y^2} - \frac{y}{x^2+y^2}i$

Multiplying two complex numbers can also be done by adding the arguments and multiplying the absolute values. For example, if we want to find $\sqrt{i}$ we know it will also have an absolute value of 1 and an argument of 45°(since $1 \cdot 1 = 1$ and $45 + 45 = 90$).

Gauss also defined the notion of what are now called *Gaussian integers*, which are complex numbers with integer coefficients. Gaussian integers have some interesting properties. For example, the Gaussian integer 2 is not prime, since it can be expressed as the product of two other Gaussian integers, $1 + i$ and $1 - i$.

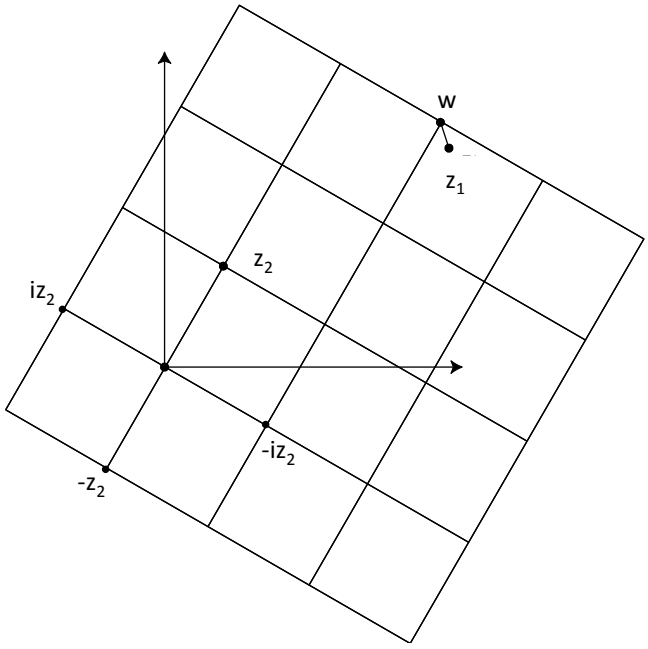
We can't do full division with Gaussian integers, but we can do division with remainder. To compute the remainder $z_1 \% z_2$, Gauss proposed the following procedure:

1. Construct a grid on the complex plan generated by $z_2, iz_2, -iz_2$, and $-z_2$.

2. Find a square in the grid containing z_1.

3. Find a vertex w of the square closest to z_1.

4. $z_1 - w$ is the remainder.



Gauss realized that with this remainder function, he could apply Euclid's gcd algorithm to complex integers, as we've done here:

```
complex<integer> gcd(complex<integer> a, complex<integer> b) {
   while (b != complex<integer>(0)) {
     a = remainder(a, b);
     std::swap(a, b);
   }
   return a;
}
```

The only thing we've changed are the types.

Gauss's work was extended by another Göttingen professor, Dirichlet. While Gauss's complex numbers were of the form (in Dirichlet's terminology) $t + n\sqrt{-1}$, Dirichlet realized that this was a special case of $t + n\sqrt{-a}$ where a did not have to be 1, and that different properties followed from the use of different values. For example, the standard gcd algorithm works on numbers of this form when $a = 1$, but it fails when $a = 5$ since there end up being numbers that don't have a unique factorization. For example:

$$21 = 3 \cdot 7 = (1 + 2\sqrt{-5}) \cdot (1 - 2\sqrt{-5})$$

It turns out that if Euclid's algorithm works, then there is a unique factorization; since we have no unique factorization here, then Euclid's algorithm doesn't work in this case.

Dirichlet's greatest result was his proof that if a and b are coprime (that is, if $\gcd(a, b) = 1$) then there are infinitely many primes of the form $an + b$.

Most of Dirichlet's results were described in the second great book on number theory, appropriately called *Vorlesungen über Zahlentheorie* ("Lectures on Number Theory"). The book contains the following insight:

> [T]he whole structure of number theory rests on a single foundation, namely the algorithm for finding the greatest common divisor of two numbers.
>
> All the subsequent theorems ... are still only simple consequences of the result of this initial investigation....

The book was actually written and published after Dirichlet's death by his Göttingen colleague, Richard Dedekind, based on Dedekind's notes from Dirichlet's lectures. Dedekind was so modest that he published the book under Dirichlet's name, even after adding many additional results of his own in later editions. Unfortunately, Dedekind's modesty hurt his career; he failed to get tenure at Göttingen and ended up on the faculty of a minor technical university.

Richard Dedekind (1831-1916)

[Bio of Dedekind goes here. Some points to include:

- Last student of Gauss, etc.
- ...

]

Dedekind observed that Gaussian integers and Dirichlet's extensions of them were special cases of a more general concept of *algebraic integers,* which are linear integral combinations of roots of *monic* polynomials (polynomials where the coefficient of the highest-order term is 1) with integer coefficients. For example:

$x^2 + 1$ generates Gaussian integers $ai + b$
$x^3 - 1$ generates *Eisenstein integers* $a + b\frac{-1+i\sqrt{3}}{2}$
$x^2 + 5$ generates integers $\mathbb{Z}[\sqrt{-5}]$

Dedekind's work on algebraic integers contained almost all the fundamental building blocks of modern abstract algebra. But it would take another great Göttingen mathematician, Emmy Noether, to make the breakthrough to full abstraction.

7.3 Noether and the Birth of Abstract Algebra

> It was [Noether] who taught us to think in terms of simple and general algebraic concepts — homomorphic mappings, groups and rings with operators, ideals...
>
> P.S. Alexandrov [cite]

> For Emmy Noether, relationships among numbers, functions, and operations became transparent, amenable to generalization, and productive only after they have been dissociated from any particular objects and have been reduced to general conceptual relationships...
>
> B. L. van der Waerden [cite]

Emmy Noether's revolutionary insight was that *it is possible to derive results about certain kinds of mathematical entities without knowing anything about the entities themselves*. In programming terms, we would say that she realized that we could use *concepts* in our algorithms and data structures, without knowing anything about which specific *types* would be used. In a very real sense, Noether provided the theory for what we now call generic programming.

[More here about how abstraction allows you to simplify and organize knowledge. Complex results become trivial.]

Emmy Noether (1882-1935)

[Bio of Noether goes here. Some points to include:

- Had to audit classes at Erlangen because she was female.
- Taught there for 7 years without pay.
- Invited by Hilbert to teach at Gottingen, but other faculty objected, so she initially had to advertise her lectures as Hilbert's with her "assisting."
- Contributed to physics as well as math.
- Expelled by Nazis, went to Bryn Mawr.

]

Noether was known for willingness to help students and give them ideas to publish, but she published relatively little herself. Fortunately, a young Dutch mathematician, Bartel van der Waerden, audited her course and wrote a book based on her lectures. Called *Modern Algebra*, it was the first book to describe the abstract approach she had developed.

7.4 Groups

Noether showed how important mathematical properties were associated with sets for which certain operations are defined and which obey certain axioms. We have already encountered two such sets (which are today known as *algebraic structures*), additive semigroup and monoid, in our first journey. Now we will look at them more formally.

Definition 7.3. *A* group *is a set on which the following are defined:*

$$\begin{aligned} \text{operations}: &\quad x \circ y, \;\; x^{-1} \\ \text{constant}: &\quad e \end{aligned}$$

and on which the following axioms hold:

$$x \circ (y \circ z) = (x \circ y) \circ z$$
$$x \circ e = e \circ x = x$$
$$x \circ x^{-1} = x^{-1} \circ x = e$$

The constant e is the *identity element*, which we will refer to as 1 in multiplicative contexts. The operation x^{-1} is the *inverse* operation; applying the operator to an item and its inverse results in the identity, as the last axiom shows. Note that $\circ$ (sometimes written $*$) can represent *any* binary operation, as long as it follows the axioms.

The group operation is not necessarily commutative. We often treat it as multiplication, and even refer to "multiplying" two elements of a group, although what we really mean is applying the group operation, whatever it might be. Just as with multiplication, the symbol for the operation is often dropped, i.e. $x \circ y$ may be written xy, and $x \circ x = xx = x^2$. (Exception: When the group operation is commutative, the group is is known as commutative or *Abelian*; normally the symbol $+$ is used to represent its operation and 0 its identity element.)

Some examples of groups are:

- Additive group of integers (i.e. the group where the elements of the set are integers and the operator is addition)
- Multiplicative group of remainders modulo 7
- Permutation group of 52 cards

- Multiplicative group of invertible matrices with real coefficients
- Group of rotations of the plane

Note that there is no multiplicative group of integers, because the multiplicative inverse of most integers is not an integer. To be a group, the set must be *closed* under its operations, i.e. applying the operations to any elements of the group must result in another element that is also in the group.

An important observation is that all groups are *transformation groups*. In other words, every element a of the group G defines a transformation of G onto itself:

$$x \to ax$$

For example, for the additive group of integers if we choose $a = 5$, then this acts as a "+5" operation, transforming the set of elements x to the set $x + 5$. These transformations are one-to-one because of our invertibility axiom:

$$y \to a^{-1}x$$

In our example, we can "undo" our $+5$ transformation by applying the inverse, -5.

Let's prove that group transformations are one-to-one. This is equivalent to saying that for any finite set S of elements of group G and any element a of G, a set of elements aS has the same number of elements as S.

Proof: If $S = \{s_1, \ldots, s_n\}$ then $aS = \{as_1, \ldots, as_n\}$. We know that the set aS can't contain more unique elements than S, but could it contain fewer? (That would be the case if two of the elements in S were mapped to the same element in aS.) Suppose two elements of aS were the same, i.e. $as_i = as_j$. Then

$$\begin{aligned} a^{-1}(as_i) &= a^{-1}(as_j) & \\ (a^{-1}a)s_i &= (a^{-1}a)s_j & \text{(by associativity)} \\ es_i &= es_j & \text{(by cancellation)} \\ s_i &= s_j & \text{(by identity)} \end{aligned}$$

So if the results of the transformation are equal, then the arguments must be equal. By the contrapositive, if the arguments are not equal, then the results of the transformation are not equal. Since we started with n distinct arguments, we will have n distinct results.

Here are a few more simple results about groups:

There is a unique inverse for every element.

$$ab = e \implies b = a^{-1} \tag{7.1}$$

Proof: Suppose $ab = e$. Then we can multiply both sides by a^{-1} on the left, like this:

$$\begin{aligned} ab &= e \\ a^{-1}(ab) &= a^{-1}e \\ (a^{-1}a)b &= a^{-1} \\ eb &= a^{-1} \\ b &= a^{-1} \end{aligned}$$

The inverse of a product is the product of the inverses.

$$(ab)^{-1} = b^{-1}a^{-1} \tag{7.2}$$

Proof: The two expressions are equal if and only if multiplying one by the inverse of the other yields the identity element. We'll use the inverse of $(ab)^{-1}$, which by definition is (ab), and multiply it by $b^{-1}a^{-1}$:

$$\begin{aligned} (ab)(b^{-1}a^{-1}) &= a(bb^{-1})a^{-1} \\ &= aa^{-1} \\ &= e \end{aligned}$$

The power of an inverse is the inverse of the power.

$$(x^{-1})^n = (x^n)^{-1} \tag{7.3}$$

Proof by induction:
Case 1:

$$(x^{-1})^1 = x^{-1} = (x^1)^{-1}$$

Case n: Assume that $(x^{-1})^{n-1} = (x^{n-1})^{-1}$.
We want to show that $(x^{-1})^n = (x^n)^{-1}$, i.e. the inverse of x^n is $(x^{-1})^n$. If that's true, then when we multiply them together, we should get the identity element. We'll rewrite x^n as $x \cdot x^{n-1}$ and $(x^{-1})^n$ as $x^{-1} \cdot (x^{-1})^{n-1}$, regroup the terms to get some to cancel, then substitute using our inductive assumption:

$$\begin{aligned} x^n(x^{-1})^n &= (x^{n-1}x)(x^{-1}(x^{-1})^{n-1}) \\ x^n(x^{-1})^n &= x^{n-1}(xx^{-1})(x^{-1})^{n-1} \\ x^n(x^{-1})^n &= x^{n-1}(x^{-1})^{n-1} \\ x^n(x^{-1})^n &= x^{n-1}(x^{n-1})^{-1} \\ &= e \end{aligned}$$

Therefore $(x^n)^{-1} = (x^{-1})^n$.

Exercise 7.4 (very easy)

Prove that any group has at least one element.

If a group has $n > 0$ elements, n is called the group's *order*. If n has infinitely many elements, it is of *infinite* order.

There is also the notion of the *order of an element* of a group: An element a has an order $n > 0$ if $a^n = e$ and for any $0 < k < n, a^k \neq e$. If such n does not exist, a has an infinite order.

Exercise 7.5 (very easy)

What is the order of e? Prove that e is the only element of such order.

We now come to an important theorem about groups:

Every element of a finite group has finite order.

Proof: If n is an order of the group, then for any element a, $\{a, a^2, a^3, \ldots, a^{n+1}\}$ has at least one repetition a_i and a_j. Let us assume that $i < j$ and a_i is the first repeated element. Then

$$\begin{aligned} a^j &= a^i \\ a^j a^{-i} &= a^i a^{-i} = e \\ a^{j-i} &= e \end{aligned}$$

and $j - i > 0$ is the order of a. This proof uses a version of the *pigeonhole principle*, introduced by Dirichlet[3]. Basically, the principle says if there are more things than places to put them, then at least one place must contain more than one thing. Note that the above result guarantees that this simple algorithm for computing the order of an element will terminate: Just keep multiplying by itself until you get e.

Exercise 7.6

Prove that if a is an element of order n, then $a^{-1} = a^{n-1}$.

7.5 Subgroups, Cyclic Groups, and Cosets

A *subgroup* is a group that is a subset of another group. For example, the additive group of even numbers is a subgroup of the additive group of integers; so is the additive group of numbers divisible by 5. Some groups have many subgroups, but almost all groups

[3]Though he called it the "drawer principle."

have at least two: the group itself, and the group consisting of just the element e. (The only group that doesn't have at least two subgroups is the group that consists of only the identity element.)

In the first journey, we looked at the multiplication table for integers modulo 7:

$\times$	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

The set of all nonzero remainders modulo 7 form a multiplicative group. We can see from the multiplication table above that it has four multiplicative subgroups:

$$\{1\}, \{1,6\}, \{1,2,4\}, \{1,2,3,4,5,6\}$$

For example, consider the set {1, 2, 4}: if we multiply any element of the set by itself or any other member of the set any number of times (mod 7), we will still get a result in the set.

Exercise 7.7

- Find orders of every element of the multiplicative group of remainders mod 7.
- Find orders of every element of the multiplicative group of remainders mod 11.

The simplest group is the *trivial* group, which contains only the identity element. The simplest nontrivial finite group is a *cyclic* group. A finite group is called *cyclic* if it has an element a such that for any element b there is an integer n where

$$b = a^n$$

In other words, every element in the group can be generated by raising an element to different powers. Such an element is called a *generator* of the group; a group may have multiple generators. The additive group of reminders modulo n is an example of a cyclic group.

Exercise 7.X

Prove that a trivial group is a cyclic group.

Exercise 7.Y

Prove that every nontrivial cyclic group contains at least one element that is not a generator.

Exercise 7.8

Prove that any subgroup of a cyclic group is cyclic.

Exercise 7.9

Prove that a cyclic group is Abelian.

Powers of a given element in a finite group form a subgroup. In other words, every element of a finite group is contained in a cyclic subgroup generated by this element.
Proof: We know they have an identity element, since we showed earlier that every element of a finite group has finite order.[4] And we know that they have an inverse, because we proved this in Exercise 7.6.

7.6 Lagrange's Theorem

One of the remarkable things about abstract algebra is that we can prove results for groups without knowing anything about the specific items or the operation. To see an important example of this, we will start by proving a few simple results about cosets.

Cosets. If G is a group and $H \subset G$ its subgroup, then for any $a \in G$ the *(left) coset* of a by H is a set

$$aH = \{g \in G \mid \exists h \in H \ : \ g = ah\}$$

In other words, a coset aH is a set of all elements in G obtainable by multiplying elements of H by a. As an example, consider the additive group[5] of integers $\mathbb{Z}$ and its subgroup, integers divisible by 4, $4\mathbb{Z}$. It has four distinct cosets: $4n, 4n+1, 4n+2$, and $4n+3$.

Size of cosets. The number of elements in a coset aH is the same as the number of elements in the subgroup H.
Proof: We already proved the one-to-oneness of transformations aS when S is a subset of G. Since a subgroup is by definition also a subset, then we know the mapping from H to aH is also one-to-one. If there is a one-to-one mapping between two sets, they are the

[4] Recall that part of the definition of an element a having a finite order is that $a^n = e$.

[5] Remember, in an additive group, the role of group "multiplication" is played by addition. So the coset aH consists of elements of G obtainable by *adding* a to elements of H.

same size.

Complete coverage by cosets. Every element a of a group G belongs to some coset of subgroup H.
Proof: $a = ae \implies a \in aH$. That is, every element a belongs at least to the coset generated by itself (that is, aH), because H, being a subgroup and therefore a group, contains the identity element.

Cosets are either disjoint or identical. If two cosets aH and bH in a group G have a common element c, then $aH = bH$.
Proof: Suppose the common element c is ah_a in one coset and bh_b in the other. By definition, they are equal:

$$ah_a = bh_b$$

Multiplying both sides on the right by h_a^{-1}, we get:

$$\begin{aligned} ah_a h_a^{-1} &= bh_b h_a^{-1} \\ a &= bh_b h_a^{-1} \end{aligned}$$

The term on the right is b times something from H (we know it's from H because h_b is from H and h_a^{-1} is from H, and since H is a subgroup, it is closed under multiplication). If we take any element x from H, ax is (by definition) in the coset aH. But if we substitute our result on the previous line for a, the term on the right is b times something from H, so it's also in the coset bH:

$$\forall x \in H, ax = bh_b h_a^{-1} x \in bH$$

Therefore $aH \subseteq bH$. We can then repeat the process from the beginning, this time using h_b^{-1} instead of h_a^{-1}, to show that $bH \subseteq aH$. So $bH = aH$.

With these results, we can state an important theorem in group theory, which illustrates the power of abstract reasoning. If you want to learn one theorem in group theory, this is the one:

Lagrange's Theorem. The order of a subgroup H of a finite group G divides the order of the group.
Proof:

1. The group G is covered by cosets of H. (Proved above.)

2. Different cosets are disjoint. (Proved above.)

3. They are of the same size n, where n is the order of H. (Proved above.)

Therefore the order of G is nm where m is the number of distinct cosets, which means that the order of G is a multiple of the order of H, i.e. the order of H divides the order of G.

Corollary 1. The order of any element in a finite group divides the order of the group.
Proof: The powers of an element of G form a subgroup of G. Since the order of an element is the order of the subgroup, and since the order of the subgroup must divide the order of the group, then the order of the element must divide the order of the group. The order of an element is equal to the order of the cyclic group of its powers.

Corollary 2. Given a group G of order n, if a is an element of G, then $a^n = e$.
Proof: If a has an order m, then m divides n (by the previous corollary), so $n = qm$. $a^m = e$ (by definition of order of an element), therefore $(a^m)^q = e$ and $a^n = e$.
Note that this doesn't say that the order of a is n; it could be smaller.

Lagrange's theorem lets us easily prove some important results from the first journey in much simpler fashion than we did the first time:

Little Fermat Theorem: If p is prime, $a^{p-1} - 1$ is divisible by p for any $0 < a < p$.
Proof: Let us take the multiplicative group of remainders modulo p. It contains $p - 1$ nonzero remainders. Since $p - 1$ is the order of the group, it follows immediately from the second corollary of Lagrange's Theorem that

$$a^{p-1} = e$$

Since the identity element for a multiplicative group is 1 (specifically, 1 mod p in our group of remainders), we have

$$a^{p-1} = 1 \bmod p$$
$$a^{p-1} - 1 = 0 \bmod p$$

which is what it means to be divisible by p.

Euler's Theorem: For $0 < a < n$, where a and n are coprime, $a^{\phi(n)} - 1$ is divisible by n.
Proof: Let us take the multiplicative group of coprime remainders modulo n. Since $\phi(n)$ is by definition the order of the group, it follows immediately from the second corollary of Lagrange's Theorem that

$$a^{\phi(n)} = e$$
$$a^{\phi(n)} = 1 \bmod n$$

or equivalently

$$a^{\phi(n)} - 1 = 0 \bmod n$$

Note that the logic is exactly the same as the previous proof.

Exercise 7.10 (very easy)

What are subgroups of a group of order 101?

Exercise 7.11

Prove that every group of prime order is cyclic.

Interestingly, the converse of Lagrange Theorem is not true: If I have a group of order n, it's not necessarily the case that I'll have a subgroup for any particular number that divides it.

[Closing thoughts to the chapter about Noether, abstraction, and concepts...]

Chapter 8

More Abstractions and the Domain of GCD

Now that Noether has given us the idea of abstract algebraic structures, we will finally be able to say what kinds of things Euclid's GCD algorithm applies to. But first, we need to introduce a few more concepts.

8.1 Rings

We now move on to another algebraic structure, *rings*, which are needed to express some familiar mathematical sets.[1]

Definition 8.1. *A* ring *is a set on which the following are defined:*

$$\begin{aligned} \text{operations}: &\quad x+y, -x, xy \\ \text{constants}: &\quad 0_R, 1_R \end{aligned}$$

[1] The term "ring," coined by Hilbert, was intended to use the metaphor of a bunch of people involved in a common enterprise, like a criminal ring. It has nothing to do with jewelry rings.

and on which the following axioms hold:

$$x + (y + z) = (x + y) + z$$
$$x + 0 = 0 + x = x$$
$$x + -x = -x + x = 0$$
$$x + y = y + x$$
$$x(yz) = (xy)z$$
$$1 \neq 0$$
$$1x = x1 = x$$
$$0x = x0 = 0$$
$$x(y + z) = xy + xz \qquad (y + z)x = yx + zx$$

Note that rings[2] have the properties we associate with integer arithmetic — operators that act like addition and multiplication, where addition is commutative and multiplication distributes over addition. Indeed, rings may be thought of as an abstraction of integers (just as iterators in programming are an abstraction of pointers), and the canonical example of a ring is the set of integers, $\mathbb{Z}$. Also observe that every ring is an additive group and therefore an Abelian group. The "addition" operator is required to have an inverse, but the "multiplication" operator is not.

In practice, mathematicians write the constants without their subscripts, just as we've done in the axioms. For example, in discussing a ring of matrices, "0" refers not to the single integer zero but to the additive identity matrix.

Some other examples of rings are:

- $n \times n$ matrices with real coefficients
- Gaussian integers
- Univariate polynomials with integer coefficients.

In addition, we say that a ring is commutative if $xy = yx$. Noncommutative rings usually come from the realm of linear algebra where matrix multiplication, for example, does not commute. On the other hand, polynomial rings and rings of algebraic integers do commute. These two types of rings lead to two branches of abstract algebra, known as *commutative algebra* and *noncommutative algebra*. Rings are often not explicitly labeled as "commutative" or "noncommutative"; instead, one type of ring or the other is assumed from the branch of algebra. For the rest of this journey, we will be concerned with commutative algebra — the kind that Dedekind, Hilbert, and Noether worked on — so from now on we will assume our rings are commutative.

[2]Some mathematicians define rings without the multiplicative identity 1 and its axioms, and call rings that include them *unitary rings*; we do not make that distinction here.

Invertible elements and units. An element of a ring is called *invertible* if there is an element x^{-1} such that

$$xx^{-1} = x^{-1}x = 1$$

Every ring contains at least one invertible element: 1 and -1. An invertible element of a ring is called a *unit* of that ring.

Exercise 8.1 (very easy)

Which ring contains exactly one invertible element?
What are units in the ring $\mathbb{Z}$ of integers?
What are units in the ring $\mathbb{Z}[\sqrt{-1}]$ of Gaussian integers?

Units are closed under multiplication, i.e. a product of units is a unit.
Proof: Suppose a is a unit and b is a unit. Then (by definition of units) $aa^{-1} = 1$ and $bb^{-1} = 1$. So

$$1 = aa^{-1} = a \cdot 1 \cdot a^{-1} = a(bb^{-1})a^{-1} = (ab)(b^{-1}a^{-1})$$

Similarly, $a^{-1}a = 1$ and $b^{-1}b = 1$, so

$$1 = b^{-1}b = b^{-1} \cdot 1 \cdot b = b^{-1}(a^{-1}a)b = (b^{-1}a^{-1})(ab)$$

Since we now have a term which, multiplied by ab from either side, gives 1, that term is the inverse of ab, i.e.

$$(ab)^{-1} = b^{-1}a^{-1}$$

So ab is a unit.

Exercise 8.X

Prove that:

- 1 is a unit.
- The inverse of a unit is a unit.

An element x of a ring is called a *zero divisor* if:

1. $x \neq 0$
2. There exists a $y \neq 0$, $xy = 0$

A commutative ring is called an *integral domain* if it has no zero divisors. It's called "integral" because its elements act like integers — you don't get zero when you multiply two nonzero things. Some examples of integral domains are:

- integers
- Gaussian integers
- Polynomials over integers
- Rational functions over integers, e.g. $\frac{x^2+1}{x^3-1}$. (A rational function is the ratio of two polynomials.)

The ring of remainders modulo 6 is not an integral domain. (Whether a ring of remainders is integral depends on whether the modulus is prime.)

Exercise 8.2 (very easy)

Prove that a zero divisor is not a unit.

8.2 Euclidean Domains

With the abstractions she had developed, Noether was finally able to say what sorts of things Euclid's gcd algorithm works on (the *setting* for the algorithm). She called it the *Euclidean domain*; it is also sometimes known as a *Euclidean ring*.

Definition 8.2. *E is a Euclidean domain if:*

- *E is an integral domain*
- *E has operations* quot *and* rem *(quotient and remainder) satisfying*

$$b \neq 0 \implies a = \text{quot}(a,b) \cdot b + \text{rem}(a,b)$$

- *E has a nonnegative norm* $\|x\| : E \to \mathbb{N}$ *satisfying*

$$\begin{aligned} \|a\| = 0 &\iff a = 0 \\ b \neq 0 &\implies \|ab\| \geq \|a\| \\ \|\text{rem}(a,b)\| &< \|b\| \end{aligned}$$

The term "norm" here is a measure of magnitude, but it should not be confused with the Euclidean norm you may be familiar with from linear algebra. For integers, the norm is absolute value; for polynomials, it is the degree of the polynomial; for Gaussian integers, it's the complex norm. The important idea is that when you compute the remainder, the norm decreases, and that it eventually goes to zero, since it maps into natural numbers. We need this property to guarantee that Euclid's algorithm terminates.

Now we can write the fully generic version of the gcd algorithm:

```
template <EuclideanDomain R>
R gcd(R m, R n) {
   while (n != R(0)) {
     R t = m % n;          //     ‖t‖  <  ‖n‖
     m = n;
     n = t;
   }
   return m;
}
```

To make something generic, you don't add extra mechanisms. Rather, you remove constraints and strip down the algorithm to its essentials.

8.3 Fields

The next important abstraction is the *field*[3].

Definition 8.3. *An integral domain where every non-zero element is invertible is called a* field.

Just as integers are the canonical example of rings, rational numbers ($\mathbb{Q}$) are the canonical example of fields. Other important examples of fields are:

- Real numbers $\mathbb{R}$
- Prime remainder fields $\mathbb{Z}_p$
- Complex numbers $\mathbb{C}$

A *prime field* is a field which does not have a proper subfield (a subfield different from itself). It turns out that every field has one of two kinds of prime subfields: $\mathbb{Q}$ or $\mathbb{Z}_p$. A field is said to have *characteristic* p if its prime subfield is $\mathbb{Z}_p$, and characteristic 0 if its prime subfield is $\mathbb{Q}$.

We can extend a field by adding elements that still satisfy the field properties.

In particular, we can extend a field *algebraically* by adding an extra element which is a root of a polynomial. For example, we can extend $\mathbb{Q}$ with $\sqrt{2}$, which is not a rational number, since it is the root of the polynomial $x^2 - 2$.

We can also extend a field *topologically* by "filling in the holes." Rational numbers leave gaps in the number line, but real numbers have no gaps, so the field of real numbers is a topological extension of the field of rational numbers. We can also extend the field to two dimensions with complex numbers. Surprisingly, there are no other finite-dimensional fields.[4]

[3] The term "field" relies on the metaphor of a field of study, not fields of wheat, etc.

[4] There are 4- and 8-dimensional field-like structures called *quaternions* and *octonions*. These are not quite

8.4 A Faster GCD

Programmers often assume that since a data structure or algorithm is in a textbook or has been used for years, it represents the best solution to a problem. Surprisingly, this is often not the case — even if the algorithm has been used for *thousands* of years and has been worked on by everyone from Euclid to Gauss.

In 1961, a young Israeli Ph.D. student, Josef "Yossi" Stein, was working on something called Racah Algebra (named after his advisor) for his thesis. He needed to do rational arithmetic, which required reducing fractions, which uses gcd. But because he had only limited time on a slow computer, he was motivated to find a better way. As he explains:

> Using "Racah Algebra" meant doing calculations with numbers of the form $a/b \cdot \sqrt{c}$, where, a, b, c were integers. I wrote a program for the only available computer in Israel at that time — the WEIZAC at the Weizmann institute. Addition time was 57 microseconds, division took about 900 microseconds. Shift took less than addition. ...[W]e had neither compiler nor assembler, and no floating-point numbers, but used hexadecimal code for the programming, and ...[had] only 2 hours of computer-time per week for Racah and his students, and you see that I had the right conditions for finding that algorithm. Fast GCD meant survival. [cite]

What Stein observed was that there were certain situations where gcd could be easily computed, or easily expressed in terms of another gcd expression. He looked at special cases like taking the gcd of an even number and an odd number, or a number and itself. Eventually, he came up with the following exhaustive list of cases:

$$\begin{aligned}
&\gcd(n, 0) = \gcd(0, n) = n \\
&\gcd(n, n) = n \\
&\gcd(2n, 2m) = 2 \cdot \gcd(n, m) \\
&\gcd(2n, 2m + 1) = \gcd(n, 2m + 1) \\
&\gcd(2n + 1, 2m) = \gcd(2n + 1, m) \\
&\gcd(2n + 1, 2(n + k) + 1) = \gcd(2n + 1, k) \\
&\gcd(2(n + k) + 1, 2n + 1) = \gcd(2n + 1, k)
\end{aligned}$$

Using these facts, he wrote the following algorithm:

```
template <BinaryInteger I>
I stein_gcd(I m, I n) {
```

fields, because they are missing certain axioms; both quaternions and octonions lack commutativity of multiplication, and octonions also lack associativity of multiplication. There are no other finite-dimensional extensions of real numbers.

```
  if (m < I(0)) m = -m;
  if (n < I(0)) n = -n;
  if (m == I(0)) return n;
  if (n == I(0)) return m;

  // m > 0 && n > 0

  int d_m = 0;
  while (even(m)) { m >>= 1; ++d_m;}

  int d_n = 0;
  while (even(n)) { n >>= 1; ++d_n;}

  // odd(m) && odd(n)

  while (m != n) {
    if (n > m) swap(n, m);
    m -= n;
    do m >>= 1; while (even(m));
  }

  // m == n

  return m << min(d_m, d_n);
}
```

Let's look at what the code is doing. The function takes two BinaryIntegers — that is, an integer representation that supports fast shift and even/odd testing, like typical computer integers. First, it eliminates the easy cases where one of the arguments is zero, and inverts the sign if an argument is negative, so that we are dealing with two positive integers.

Next, it removes factors of 2 from each argument (by shifting), keeping track of how many there were. Note that we can use a simple `int` for the counts, since what we're counting is at most the total number of bits in the original arguments. After this part, we are operating on two odd numbers.

Now comes the main loop. We repeatedly subtract the smaller from the larger, each time (since we know the difference of two odd numbers is even) using shift to remove additional powers of 2 from the result.[5] When we're done, our two numbers will be equal. Since we're halving at least once each time through the loop, we know we'll iterate no more than $\log_2 n$ times; the algorithm is bounded by the number of 1 bits we encounter.

[5]Note that we use `do-while` rather than `while` because we don't need to run the the test the first time; we know we're starting with an even number so we know we have to do at least one shift.

Finally, we return our result, using a shift to multiply our number by 2 for each of the minimum number of 2s we factored out at the beginning. Note that we don't need to worry about 2s in the main loop, because we've reduced the problem to gcd of two odd numbers; this gcd does not have 2 as a factor.

Here's an example of the algorithm in operation. Suppose we want to compute gcd(196, 42). The computation looks like this:

m	n	d_m	d_n
factor out 2s :			
196	42	0	0
98	42	1	0
49	42	2	0
49	21	2	1
main loop :			
28	21	2	1
14	21	2	1
7	21	2	1
14	7	2	1
7	7	2	1
result :			

$$7 \times 2^{\min(2,1)} = 7 \times 2 = 14$$

As we saw, Stein took some observations about special cases and turned them into a faster algorithm. The special cases had to do with even and odd numbers, and places where we could factor out 2, which is easy on a computer; that's why Stein's algorithm is faster in practice. But is this just a clever hack, or is there more here than meets the eye? Does it only make sense because computers use binary arithmetic?

As a historical note, computers didn't always use binary arithmetic. Some electromechanical computers in the mid-20th century used decimal arithmetic. Even electronic computers didn't always use binary; knowing that the theoretical minimal number of gates can be achieved with a base of e, Russians built a base 3 (ternary) computer in the 1950s [cite], reasoning that 3 is closer to e than 2 is.

In any case, we'll soon see the importance of the "2" in Stein's algorithm.

8.5 Generalizing Stein's Algorithm

Does Stein's algorithm only make sense for integers, or can we generalize it just as we did with Euclid's algorithm? Let's review some of the historical milestones for Euclid's gcd:

- Integers: Greeks (5th century BC)
- Polynomials: Stevin (ca. 1600)
- Gaussian Integers: Gauss (ca. 1830)
- Algebraic Integers: Dirichlet, Dedekind (ca. 1860)
- Generic Version: Noether, van der Waerden (ca. 1930)

It took 2000 years to extend Euclid's algorithm from integers to polynomials. Fortunately, it took much less time for Stein's algorithm. In fact, just two years after its publication, Knuth already knew of a version for single-variable polynomials over a field $\mathbb{F}[x]$. [cite]

The surprising insight was that we can have x play the role for polynomials that 2 plays for integers. That is, we can factor out powers of x, etc. Carrying the analogy further, we see that $x^2 + x$ (or anything else divisible by x) is "even," $x^2 + x + 1$ (or anything else with a zero-order coefficient) is "odd," and $x^2 + x$ "shifts" to $x + 1$. Just as division by 2 is easier than general division for binary integers, division by x is easier than general division for polynomials — in both cases, all we need is a shift. (Remember that a polynomial is really a sequence of coefficients, so division by x is literally a shift of the sequence.)

Stein's "special cases" for polynomials look like this:

$$\gcd(p, 0) = \gcd(0, p) = p \tag{8.1}$$

$$\gcd(p, p) = p \tag{8.2}$$

$$\gcd(xp, xq) = x \cdot \gcd(p, q) \tag{8.3}$$

$$\gcd(xp, xq + c) = \gcd(p, xq + c) \tag{8.4}$$

$$\gcd(xp + c, xq) = \gcd(xp + c, q) \tag{8.5}$$

$$\deg(p) \geq \deg(q) \implies \gcd(xp + c, xq + d) = \gcd(p - \frac{c}{d}q, xq + d) \tag{8.6}$$

$$\deg(p) < \deg(q) \implies \gcd(xp + c, xq + d) = \gcd(xp + c, q - \frac{d}{c}q) \tag{8.7}$$

Notice how each of the last two rules cancels one of the zero-order coefficients, so we convert the "odd, odd" case to an "even, odd" case.

In order to get the equivalence expressed by rule 8.6, we rely on two facts. First, we can multiply a polynomial by any coefficient and it's still the same polynomial, so it will have the same GCD. Second, gcd(a, b) = gcd(a, b - a), which we proved earlier in the journey. Using this, we can transform the first argument on the left side of 8.6, $xp + c$, to the first argument on the right side, $p - \frac{c}{d}q$, using the following steps: First, multiply the second argument by $\frac{c}{d}$. Next, subtract it from the first argument. This gives $xp + c - \frac{c}{d}(xq + d)$. The c cancels, giving $xp - \frac{c}{d}xq$. Then we use the fact that if one side is divisible by x and

the other is not, we can drop x because GCD will not contain it. So we "shift" out the x, which gives $p - \frac{c}{d}q$.

We also see that in each transformation, the Euclidean algebraic norm — in this case, the degree of the polynomial — gets reduced. Here's how the algorithm would compute $\gcd(x^3 - 3x - 2, x^2 - 4)$

m	n	$operation$
$x^2 - 3x - 2$	$x^2 - 4$	$m - (0.5x^2 - 2)$
$x^3 - 0.5x^2 - 3x$	$x^2 - 4$	shift(m)
$x^2 - 0.5x - 3$	$x^2 - 4$	$m - (0.75x^2 - 3)$
$0.25x^2 - 0.5x$	$x^2 - 4$	normalize(m)
$x^2 - 2x$	$x^2 - 4$	shift(m)
$x - 2$	$x^2 - 4$	$m - (2x - 4)$
$x - 2$	$x^2 - 2x$	shift(n)
$x - 2$	$x - 2$	GCD : $x - 2$

First we see that the ratio of their free coefficients is $1/2$, so we will multiply n by $1/2$ and subtract it from m (shown in the first line above), resulting in the new m shown on the second line. Then we "shift" m by factoring out x, resulting in the 3rd line, and so on.

In 2000, Andre Weilert further generalized Stein's algorithm for Gaussian integers. [cite] This time, $1 + i$ plays the role of 2; the "shift" operation — in this case, division by $1 + i$ — looks like this:

$$\frac{a + bi}{1 + i} = \frac{(a + bi)(1 - i)}{(1 + i)(1 - i)} = \frac{(a + b) - (a - b)i}{2}$$

All we have to do is $+, -$ and "shift." This also gives us a simple test for divisibility: a Gaussian integer $a + bi$ is divisible by $1 + i$ if and only if $a = b \pmod 2$. In other words, one is divisible by the other if they're both "even" or both "odd."

In 2003, Damgård and Frandsen [cite] extended the algorithm to Eisenstein integers $\mathbb{Z}[\zeta]$, i.e. the integers extended with a complex primitive cubic root of 1:

$$\zeta = \frac{-1 + \sqrt{-3}}{2}$$

In their version, $1 - \zeta$ plays the role of 2.

In 2004 Agarwal and Frandsen [cite] demonstrated that there is a ring that is not a Euclidean domain (ring of integers in $\mathbb{Q}[\sqrt{-19}]$)) where Stein algorithm works. In other words, there are cases where Stein's algorithm works but Euclid's does not. If the domain of the Stein algorithm is not the Euclidean domain, then what is it? As of this writing, it's an unsolved problem.

What we do know is that there are some fundamental operations on elements of rings that must be valid for Stein's algorithm to work:

- `is_unit(u)`: there is a v such that $vu = 1$
- `are_associates(m,n)`: $m = un$ where u is a unit

  Two elements of a ring are *associates* if one is the product of the other and a unit. (If x is an associate of y, then y is an associate of x, because the unit has an inverse.)

- `is_smallest_prime(p)`: $q \neq 0 \wedge \text{norm}(q) < \text{norm}(p) \implies q$ is a unit

  p is a smallest prime if there is no non-unit with a smaller norm.

It's the notion of *smallest prime* that gives us the key to Stein's algorithm. Why do we factor out 2 when we're computing the gcd of integers? Because when we repeatedly divide by 2, we eventually get 1 as a remainder; i.e. we have an odd number. Once we have two odd numbers, i.e. two numbers whose remainders are both units, we can use subtraction to keep our GCD algorithm going. This works because 2 is the smallest integer prime. Similarly, x is the smallest prime for polynomials, and $i + 1$ for Gaussian integers.[6] Division by the smallest prime always gives remainder of zero or a unit, because a unit is the number with the smallest nonzero norm. So 2 works for integers because it's the smallest prime, not because computers use binary arithmetic. The algorithm is possible because of fundamental properties of integers, not because of the hardware implementation (although the algorithm is *efficient* because computers use binary arithmetic, making shifts fast).

We can now state three lemmas about the Stein domain:

- If a Euclidean ring is not a field, it has a smallest prime.
- u is a unit $\implies \|a\| = \|ua\|$.
- p is a smallest prime $\implies \text{rem}(q, p)$ is either a unit or 0.

Conjecture: Every Euclidean domain is a Stein domain. We can't quite prove it, but we know the opposite is not true, since we saw examples that the Stein algorithm could handle and the Euclidean algorithm could not.

Even though we haven't been able formally define the Stein domain, we can still write most of the generalized algorithm. Notice that instead of the main loop running while $m \neq n$, we run as long as m and n are not associates.

The part that we cannot define is isolated in the function `reduce_associate_remainders`; even though we have a way to do cancelation, we can't guarantee that the norm will decrease.

[6] Note that 2 is not prime in the ring of Gaussian integers, since it can be factored into $(1 + i)(1 - i)$.

```
template <SteinDomain R>
R stein_gcd(R m, R n) {

  if (m == R(0)) return n;
  if (n == R(0)) return m;

  int d_m = 0;
  while (divisible_by_smallest_prime(m)) {
     m = divide_by_smallest_prime(m);
     ++d_m;
  }
  int d_n = 0;
  while (divisible_by_smallest_prime(n)) {
     n = divide_by_smallest_prime(n);
     ++d_n;
  }

  while (!is_associate(m, n)) {
    if (norm(n) > norm(m)) swap(n, m);
    m = reduce_associate_remainders(n, m);
    do {
      m = divide_by_smallest_prime(m);
    } while (divisible_by_smallest_prime(m));
  }

  R p = smallest_prime<R>();
  return m * power(p, min(d_m, d_n));
}
```

At the end, we use the `power` function from our first journey to play the role of shift.

8.6 Lessons of Stein

The discovery of the Stein algorithm illustrates a few important programming principles:

1. Every useful algorithm is based on some fundamental mathematical truth.

When Yossi Stein noticed some useful patterns in computing gcd of odd and even numbers, he wasn't thinking about smallest primes. Indeed, it's very common that the discoverer of an algorithm might not see this fundamental truth. There is often a long time between the first discovery of the algorithm and its understanding. Nevertheless, its

underlying mathematical truth is there. For this reason, every useful program is a worthy object of study, and behind every optimization there is solid mathematics.

2. Even a classical problem studied by great mathematicians may have a new solution.

When someone tells you, for example, that sorting can't be done faster than $n \log n$, don't believe them. It might not be true for your particular problem.

3. Performance constraints are good for creativity.

The limitations of the WEIZAC computer that that Stein faced in 1961 are what drove him to look for alternatives to the traditional approach. The same is true in many situations; necessity really is the mother of invention.

Project 8.3

Compare the performance of the Stein and Euclid algorithms on random integers from the ranges $[0, 2^{16})$, $[0, 2^{32})$, and $[0, 2^{64})$.

Chapter 9

Extensions of GCD

In this chapter, we'll finish our survey of algebraic structures, and then see how the GCD algorithm can be extended and applied to a variety of problems. [More here.]

9.1 Bézout's Identity

An interesting theorem in abstract algebra is *Bézout's identity*, which says that for any two values in the Euclidean domain, there exist coefficients such that the linear combination gives the gcd of the original values:

$$\forall a, b \; \exists x, y \; : \; xa + yb = \gcd(a, b)$$

Like many results in mathematics, this one is named after someone other than its discoverer. Although 18th-century French mathematician Étienne Bézout did prove the result for polynomials, it was actually shown first for integers a hundred years earlier, by Bachet.

Claude Gaspard Bachet de Mérizac (1581–1638)

Claude Gaspard Bachet de Mérizac, generally known as Bachet, was a French mathematician during the Renaissance. Although he was a scholar in many fields, he is best known for two things. First, he translated Diophantus' *Arithmetic* from Greek to Latin. It is his 1621 translation that most mathematicians relied on, and it was in a copy of his translation that Fermat famously wrote the marginal note describing his last theorem. Second, Bachet wrote the first book on recreational mathematics, *Problèmes Plaisants*, originally published in 1612. This book introduced magic squares, as well as what is now known as Bézout's identity. [More bio of Bachet...]

[Further reading: *Mathematical Recreations and Essays* by W. W. Rouse Ball]

To prove Bézout's identity, we need to show that the coefficients x and y exist. To do this, we need to introduce a new algebraic structure, the *ideal*.

Definition 9.1. *An* ideal I *is a non-empty subset of a ring* R *such that:*

$$1.\ \forall x, y \in I\ :\ x + y \in I$$
$$2.\ \forall x \in I\ \forall a \in R\ :\ ax \in I$$

The first property says that the ideal is closed under addition, i.e. if you add any two elements of the ideal, the result is in the ideal. But the second property is a bit more subtle; it says that the ideal is closed under multiplication with *any* element of the ring, not necessarily an element of the ideal.

Some examples of ideals are even numbers, univariate polynomials with root 5, and polynomials with x and y and free coefficient 0 (e.g. $x^2 + 3y^2 + xy + x$); we'll see shortly why this last case is important. Note that just because something is a subring doesn't mean it's an ideal. Integers are a subring of Gaussian integers, but they aren't an ideal, because multiplying an integer by i does not produce an integer.

Exercise 9.1

1. Prove that an ideal I is closed under subtraction.
2. Prove that I contains 0.

Linear combination ideal. In a ring, for any two elements a and b, the set of all elements $\{xa + yb\}$ forms an ideal.
Proof: First, this set is closed under addition:

$$(x_1a + y_1b) + (x_2a + y_2b) = (x_1 + x_2)a + (y_1 + y_2)b$$

Next, it is closed under multiplication by an arbitrary element:

$$z(xa + yb) = (zx)a + (zy)b$$

Therefore, it is an ideal.

Exercise 9.2

Prove that all the elements of a linear combination ideal are divisible by any of the common divisors of a and b.

Ideals in Euclidean domains. Any ideal in a Euclidean domain is closed under the remainder operation and under Euclidean gcd. **Proof:**

1. Closed under remainder: $\text{rem}(a, b) = a - \text{quot}(a, b) \cdot b$.
   If b is in the ideal, then by the second ideal axiom, anything times b is in the ideal, so the term on the right of the minus sign is. By Exercise 9.1, the difference of two elements of an ideal is in the ideal.

2. Closed under gcd: Since gcd consists of repeatedly applying remainder, this immediately follows from 1.

Definition 9.2. *An ideal I of the ring R is called a* principal ideal *if there is an element $a \in R$ called the* principal element *such that*

$$x \in I \iff \exists y \in R \ : \ x = ay$$

In other words, a principal ideal is an ideal that can be generated from one element. An example of a principal ideal is the set of even numbers (2 is the principal element). Polynomials with root 5 are another principal ideal. On the other hand, polynomials with x and y and free coefficient 0 are ideals, but not principal ideals. Remember the polynomial $x^2 + 3y^2 + xy + x$, which we gave as an example of an ideal? There's no way to generate it starting with just x (it would never contain y), and vice versa.

Exercise 9.3

Prove that any element in a principal ideal is divisible by the principal element.

Definition 9.3. *An integral domain is called a* principal ideal domain (PID) *if every ideal in it is a principal ideal.*

$ED \implies PID$. Every Euclidean domain is a principal ideal domain.
Proof:
Any ideal I in a Euclidean domain contains contains an element m with a minimal positive norm (a "smallest element"). Consider an arbitrary element $a \in I$; it's either a multiple of m, or it has a remainder r:

$$a = qm + r \quad \text{where } 0 < \|r\| < \|m\|$$

But we chose m as the smallest element, so we cannot have a smaller remainder — contradiction. Therefore our element a can't have a remainder; $a = qm$. So we can obtain every element from one element, which is the definition of a PID.

Now we can prove Bézout's identity, which we'll restate as follows:

A linear combination ideal $I = \{xa + yb\}$ of a Euclidean domain contains gcd(a, b).

Proof: We already proved (#2 on p. 147) that any ideal in a Euclidean domain is closed under gcd, i.e.:

$$u \in I \ \wedge \ v \in I \implies \gcd(u, v) \in I$$

And $a = 1 \cdot a + 0 \cdot b$ and is therefore in I, and similarly for b. Therefore $\gcd(a, b)$ is in I.

We can also use Bézout's identity to prove the Invertibility Lemma, which we encountered in the first journey:

$$\forall a, n \in \mathbb{Z} \ : \ \gcd(a, n) = 1 \implies \exists x \in \mathbb{Z}_n \ : \ ax = xa = 1 \mod n$$

Proof:
By Bézout's identity,

$$\exists x, y \in \mathbb{Z} \ : \ xa + yn = gcd(a, n) = 1$$

Therefore, $xa = -yn + 1$, i.e.

$$xa = 1 \mod n$$

Exercise 9.4

Using Bézout's identity, prove that if p is prime, then any $0 < a < p$ has a multiplicative inverse modulo p.

9.2 A Constructive Proof

The proof of Bézout's identity that we saw in the last section is interesting; it shows why the result must be true, but it doesn't actually tell us how to find the coefficients. This is an example of a *non-constructive* proof. For a long time, there was a debate in mathematics about whether non-constructive proofs were as valid as constructive proofs. Those who opposed the use of non-constructive proofs were known as *constructivists* or *intuitionists*. At the turn of the 20th century, the tide turned against the constructivists, with David Hilbert and the Göttingen school leading the charge. The lone major defender of constructivism, Henri Poincaré, lost the battle, and today non-constructive proofs are routinely used.

Henri Poincaré (1854–1912)

[Biography of Poincare goes here. Some points to include:]
Jules Henri Poincaré was a French mathematician and physicist. He came from a deeply patriotic family, and his cousin Raymond was prime minister of France. He published over 500 papers on a variety of subjects, including several on special relativity developed independently of, and in some cases prior to, Einstein's work on the subject. Although he worked on many practical problems, such as establishing time zones, he also valued science for its own sake:

> Science has had marvelous applications, but a science that would only have applications in mind would not be science anymore, it would be only cookery.

[cite]. He also wrote several important books about the philosophy of science, and was elected a member of the *Académie Française*.

Poincaré contributed to almost every branch of mathematics, originating several subfields, such as algebraic topology. However, his criticism of set theory and the formalist agenda of Hilbert put him on the wrong side of the rivalry between France and a recently-unified Germany. The rejection of Poincaré's work by the formalists was a great loss for 20th century mathematics.

Despite the view of formalist mathematicians, from a programming perspective, it is clearly more satisfactory to be given a program than simply to know that one exists. So in this section, we will derive a constructive proof of Bézout's identity — in other words, an algorithm for finding x and y such that $xa + yb = \gcd(a, b)$.

To understand the process, it's helpful to recall what happens in several iterations of Euclid's algorithm to compute the GCD of a and b. Each time through the main loop, we replace a by the remainder of a and b, then swap a and b; our final remainder will be the GCD. So we are computing this sequence of remainders:

$$\begin{aligned} r_1 &= remainder(a, b) \\ r_2 &= remainder(b, r_1) \\ r_3 &= remainder(r_1, r_2) \\ &\dots \\ r_n &= remainder(r_{n-2}, r_{n-1}) \end{aligned}$$

Since the remainder of a and b is what's left over from a after dividing a by b, we can write

the terms like this:

$$\begin{aligned} r_1 &= a - b \cdot q_1 \\ r_2 &= b - r_1 \cdot q_2 \\ &\dots \\ r_{i+2} &= r_i - r_{i+1} \cdot q_{i+2} \\ &\dots \end{aligned}$$

Bachet's algorithm begins by expressing a and b as the following linear combinations:

$$\begin{aligned} a &= 1 \cdot a + 0 \cdot b \\ b &= 0 \cdot a + 1 \cdot b \end{aligned}$$

Then we see that we can also write the first two remainders as linear combinations, plugging in the previous values at each step:

$$\begin{aligned} r_1 &= 1 \cdot a - q_1 \cdot b \\ r_2 &= b - (a - b \cdot q_1) \cdot q_2 \\ &= -q_2 \cdot a + (1 + q_1 \cdot q_2) \cdot b \end{aligned}$$

Next, we have an iterative recurrence. Assume that we already figured out how to represent two successive remainders as linear combinations:

$$\begin{aligned} r_i &= x_i \cdot a + y_i \cdot b \\ r_{i+1} &= x_{i+1} \cdot a + y_{i+1} \cdot b \end{aligned}$$

Then we can use our previous observation about how to define an arbitrary remainder in the sequence using the two previous ones, substituting those previous values, and distributing and rearranging to group all the coefficients with a and all the coefficients with b:

$$\begin{aligned} r_{i+2} &= r_i - r_{i+1} \cdot q_{i+2} \\ &= x_i \cdot a + y_i \cdot b - (x_{i+1} \cdot a + y_{i+1} \cdot b) \cdot q_{i+2} \\ &= (x_i - x_{i+1} \cdot q_{i+2}) \cdot a + (y_i - y_{i+1} \cdot q_{i+2}) \cdot b \end{aligned}$$

Now we observe that the coefficients on a are defined in terms of the previous coefficients on a, and the coefficients on b are defined in terms of the previous coefficients on b. In particular, the coefficients on the $i+2$nd iteration are:

$$\begin{aligned} x_{i+2} &= x_i - x_{i+1} \cdot q_{i+2} \\ y_{i+2} &= y_i - y_{i+1} \cdot q_{i+2} \end{aligned}$$

So, we now have a way to compute the coefficients x and y such that $xa + yb = gcd(a, b)$, which was what we wanted all along.

In fact, we can make the further observation that the ys do not depend on the xs and the xs do not depend on the ys. Since we know $xa + yb = gcd(a, b)$, then as long as $b \neq 0$, we can rearrange these terms to define y as

$$y = \frac{\gcd(a, b) - ax}{b}$$

This means we don't need to go through the trouble of computing all the intermediate values of y.

Exercise 9.5

What are x and y if $b = 0$?

Now we can write the algorithm to compute the xs, which we'll call *extended gcd*. We don't need to keep every value in the recurrence; we just need the two previous ones, which we'll call x_0 and x_1. We'll initialize these to 1 and 0, which are the coefficients of a in the first two linear combinations. We can use the x_{i+2} formula we have just derived to compute the new value each time. We'll also need the new remainder, which we can get from the two previous remainders. Recall that our function `quotient_remainder` returns a pair whose first element is the quotient and second is the remainder; we'll need both of these for our computation — the quotient to compute the coefficient x, and the remainder to compute gcd. Here's the code:

```
template <EuclideanDomain R>
std::pair<R, R> extended_gcd(R a, R b) {
  R x0(1);
  R x1(0);
  while (b != R(0)) {
    // compute new r and x
    std::pair<R, R> qr = quotient_remainder(a, b);
    R x2 = x1 - qr.first * x0;
    // shift r and x
    x0 = x1; x1 = x2;
    a = b; b = qr.second;
  }
  return std::pair<R, R>(x0, a);
}
```

At the end, when b is zero, we'll return a pair consisting of the value of x that we wanted for Bézout's identity, and the gcd of a and b.

Observant readers may remember the `extended_gcd` function from Journey 1, where we used it for the RSA project, although the original version (on p. 84) was slightly different.

Project 9.6

Develop a version of extended gcd based on Stein's algorithm.

9.3 Applications of GCD

Now we can see some of the applications of gcd.

Cryptography. The extended Euclid algorithm allows us to find the multiplicative inverse of a number. We know that

$$xa + yb = gcd(a, b) \quad \text{so}$$

$$xa = gcd(a, b) - yb$$

If $gcd(a, b) = 1$, then

$$xa = 1 - yb,$$

i.e. multiplying x and a give 1 plus some multiple of b, or to put it another way,

$$xa = 1 \bmod b$$

As we learned in journey 1, two numbers whose product is 1 are multiplicative inverses. Since the extended_gcd algorithm returns x and $gcd(a, b)$, if the gcd is 1, then x is the multiplicative inverse of a mod b; we don't even need y.

As you may recall from Journey 1, the RSA algorithm depends on being able to find the multiplicative inverse of large numbers, and in fact, the function we provided (on p. original-multiplicative-inverse uses the extended_gcd algorithm in the way we've just described. Most e-commerce transactions rely on the RSA algorithm, so it's fair to say that e-commerce depends on gcd.

Rational Arithmetic. Rational arithmetic is very useful in many areas, and it can't be done without reducing fractions to their canonical form, which needs gcd.

Symbolic Integration. One of the primary tasks in symbolic integration is decomposing a rational fraction into primitive fractions, which uses GCD of polynomials over real numbers.

Rotation algorithms, such as `std::rotate`. We'll return to this in the next chapter.

9.4 Permutations and Transpositions

A *permutation* is a function from a sequence of n objects onto itself. The formal notation for permutations[1] looks like this:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 1 & 3 \end{pmatrix}$$

The first row represents the indexes (positions) of a sequence of objects; mathematicians start numbering from 1. The second row represents where the items in those positions end up after applying the permutation. In this example, the item that used to be at position 1 will end up in position 2, the item that was in position 2 will end up in position 4, and so on.

In practice, permutations are usually written using a shorthand format which omits the first row:

$$\begin{pmatrix} 2 & 4 & 1 & 3 \end{pmatrix}$$

In other words, at position i, you write where the ith original element will end up. An example of applying a permutation is:

$$(2\,4\,1\,3) : \{a, b, c, d\} = \{c, a, d, b\}$$

We can use the notion of permutation to define a *symmetric group*:

Definition 9.4. *A set of all permutations on n elements constitutes a group called the* symmetric group S_n.

A symmetric group has the following group properties:

operation :	composition (associative)
identity element :	identity permutation
inverse :	inverse permutation

This is perhaps the most important group to know about, since every finite group is a subgroup of a symmetric group.

[1] This is the same notation mathematicians use for matrices; hopefully the context will make it clear which interpretation is intended.

Exercise 9.7

What is the order of S_n?

Now we can look at a special case of permutation, *transposition*.

Definition 9.5. *A* transposition(i, j) *is a permutation that exchanges the ith and jth elements* $(i \neq j)$*, leaving the rest in place.*

The notation for transpositions indicates which two positions should get exchanged:

$$(2\ 3) : \{a, b, c, d\} = \{a, c, b, d\}$$

In C++, we call it `swap`.

The *transposition lemma* demonstrates how fundamental this `swap` operation is:

Any permutation is a product of transpositions.

Proof: One transposition can put one element into its final destination. Therefore, $n - 1$ transpositions will put all elements into their final destinations.

Why do we need only $n - 1$ transpositions? Because once $n - 1$ items are in the right place, the nth item must also be in the right place; there's no place else for it to go.

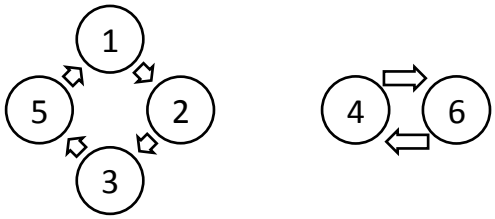
We can also prove that every permutation can be represented as a product of *adjacent* transpositions. One way to see this is to follow this procedure: starting from the first position, we swap an element with its neighbor until the desired item is in position n; then we repeat for the item we want in position $n - 1$, and so on.

An permutation is said to be *even* if it is the product of an even number of transpositions; otherwise it is *odd*. You can have a subgroup of all the even permutations, since the product of two even permutations is even. This is another very important group, called the *alternating* group A_n.

Exercise 9.8

Prove that if $n > 2$, S_n is not Abelian.

Every permutation defines a directed graph of n elements. After applying the permutation enough times, a given element will eventually be put back in its original position, representing a *cycle* in the graph. Every permutation can be decomposed into cycles. For example, consider the permutation (2 3 5 6 1 4). Note that the element in position 4 moves to position 6, and the element in position 6 moves to position 4, so after applying the permutation twice, both of those elements will end up where they started. We see a similar pattern for the elements at positions 1, 2, 3, and 5, although this time it takes four applications before they get back to the beginning. We say that the permutation (2 3 5 6 1 4) can be decomposed into two cycles, and we show them graphically like this:



We write this decomposition as: (2 3 5 6 1 4) = (1 2 3 5)(4 6).

The cycle notation, used on the right, may be thought of as an extension of the transposition notation. Although the notation is ambiguous (is this a cycle or a permutation?), it is usually clear from the context. Also, permutations always contain all the integers from 1 to n, while cycles might not.

Cycles are disjoint. If you are at a position in a cycle, you can get to all other positions in that cycle. Therefore, if two cycles share one position, they share all positions, i.e. they are the same cycle. So the only way to have separate cycles is if they don't share any positions.

A cycle containing one element is called a *trivial cycle*.

Exercise 9.9

How many non-trivial cycles could a permutation of n elements contain?

Number of assignments. The number of assignments needed to perform an arbitrary permutation in place is $n - u + v$, where n is the number of elements, u is the number of trivial cycles, and v is the number of non-trivial cycles.

Proof:
Every non-trivial cycle of length k requires $k + 1$ assignments, since each element needs to be moved, plus we need to save the first value being overwritten; since every non-trivial cycle requires one extra move, moving all v cycles requires v extra moves. Elements in trivial cycles don't need to be moved at all, and there are u of those. So we need to move $n - u$ elements, plus v additional moves for the cycles.

A common permutation that has exactly $n/2$ cycles is *reverse*. In some sense this is the "hardest" permutation because it requires the largest number of assignments. We'll look at reverse in greater detail in the next chapter.

Exercise 9.10

Design an in-place[2] reverse algorithm for forward iterators; that is, it should work for singly-linked lists without modifying the links.

[Final thoughts for chapter.]

[2] An algorithm is *in-place* if for an input of length n it uses $O(p(\log n))$ additional space, where p is a polynomial.

Chapter 10

Rotate and Reverse

Over the past few chapters, we've seen how mathematicians were able to obtain very general results though abstraction; we've also seen how quite complex theorems can be understood by breaking the reasoning into a sequence of small self-contained lemmas. These ideas carries through to programming. We should write the most general form of an algorithm, rather than writing code that only works for specific types. And we should break our code into small independent pieces, making the result more understandable. In this chapter, we'll look at some of these small building blocks of code that are useful in many applications.

10.1 Swapping Ranges

A common operation in programming is to swap values in one range of data with values in another (possibly overlapping) range. We do it in a loop, one swap at a time:

```
while (condition) swap(*f0++, *f1++);
```

In programming, a range from i to j can be either *semi-open* or *closed*; a closed range includes items i and j, while a semi-open range includes i but ends just before j. As we shall see, semi-open ranges are the most convenient for defining interfaces; this is because algorithms that operate on sequences of n elements need to be able to refer to $n + 1$ positions. For example, there are $n + 1$ places to insert a new item.

A range can also be specified one of two ways: a *bounded* range has two iterators (one pointing to the beginning and one pointing to the end), while a *counted* range has an iterator pointing to the beginning and an integer n indicating how many items are included. This gives us four kinds of ranges altogether:

	semi-open	closed
Bounded: two iterators	$[i, j)$	$[i, j]$
Counted: iterator and integer	$[i, n)$	$[i, n]$

(Note that a closed counted range must have $n > 0$.) While mathematical texts index sequences from 1, computer scientists start from 0, and we will use the latter convention for our ranges. Interestingly, although 0-based indexing in computer science was initially used as a way to indicate the offset in memory, this convention turns out to be more natural for mathematical reasons:

- For a sequence with n elements, the indices are in the range $[0, n)$ and any iteration is bounded by the length.
- Rotating n elements to the right by k transforms an index i to the index $i + k \bmod n$.

When we swap two ranges, often only one of them needs to be specified explicitly:

```
template <ForwardIterator I0, ForwardIterator I1>
I1 swap_ranges(I0 f0, I0 l0, I1 f1) {
  while (f0 != l0) swap(*f0++, *f1++);
  return f1;
}
```

There's no point in specifying the end of the second range, since in order for the swap to work, it must contain (at least) the same number of elements as the first range. Why do we return `f1`? Because it might be useful to the caller — it's information that the caller doesn't have — and it costs us nothing.

This is a good time to review *The Law of Useful Return*, which we introduced in Section 6.6:

The Law of Useful Return

When writing code, it's often the case that you end up computing a value that the calling function doesn't currently need. However, it often turns out that later, this value will be important when the code is called in a different situation. In this situation, you should obey the *law of useful return*:

A procedure should return all the potentially useful information it computed.

The quotient-remainder function that we saw in Chapter 6 is a good example: when we first wrote it, we only needed the remainder, but we had already done all the work to find the quotient. Later, we saw other applications that use both values.

Note that the law does *not* imply doing unneeded extra computations, nor does it imply that useless information should be returned. For example, in the code above, it's not useful to return `f0`, because the algorithm guarantees that at the end it's equal to `l0`, which the caller already has. It's useless to give me information I already have.

The Law of Separating Types

The `swap_ranges` code above illustrates another important programming principle, the *law of separating types*:

Do not assume that two types are the same when they may be different.

Note that our function was declared with two iterator types, like this:

```
template <ForwardIterator I0, ForwardIterator I1>
   I1 swap_ranges(I0 f0, I0 l0, I1 f1);
```

rather than assuming that they're the same type, like this:

```
template <ForwardIterator I>
   I swap_ranges(I f0, I l0, I f1);
```

The first way gives us more generality and allows us to use the function in situations we wouldn't otherwise be able to, without incurring any additional computation cost. For example, we could swap a range of elements in a linked list with a range of elements in an array and it would just work.

In situations where we're not sure if the second range is long enough to perform all the needed swaps, we can make both ranges explicit and have our `while` test check to make sure neither iterator has run off the end of the range:

```
template <ForwardIterator I0, ForwardIterator I1>
pair<I0, I1> swap_ranges(I0 f0, I0 l0, I1 f1, I1 l1) {
  while (f0 != l0 && f1 != l1) {
    swap(*f0++, *f1++);
  }
  return pair<I0, I1>(f0, f1);
}
```

This time we do return both `f0` and `f1`, because one range may be exhausted first, so we won't necessarily have reached `l0` and `l1`.

The Law of Completeness

When designing an interface, consider all the related procedures.

If there are different ways of invoking an algorithm, provide interfaces for those related functions. In our swap example, we've already provided two related interfaces for bounded ranges, and we'll also provide interfaces for bounded ranges.

Note that this rule doesn't mean that you should have a single interface that provides disparate uses, such as a data structure with an"insert_or_erase" function.

Swapping counted ranges is almost the same, except that instead of checking to see if we reached the end of the range, we count down from n to 0:

```
template <ForwardIterator I0, ForwardIterator I1, Integer N>
pair<I0, I1> swap_ranges_n(I0 f0, I1 f1, N n) {
  while (n != N(0)) {
    swap(*f0++, *f1++);
    --n;
  }
  return pair<I0, I1>(f0, f1);
}
```

Counted ranges are easier for the compiler, because it knows the number of iterations, called the *trip count*, in advance. This allows them to make certain optimizations, such as loop unrolling, or software pipelining.

Exercise 10.1

Why don't we provide the following interface?

```
pair<I0, I1> swap_ranges_n(I0 f0, I1 f1, N0 n0, N1 n1)
```

10.2 Rotation

One of the most important algorithms you've probably never heard of is `rotate`. It's a fundamental tool used behind the scenes in many common computing applications, such as buffer manipulation in text editors. It implements the mathematical operation *rotation*.

A permutation of n elements by k where $k \geq 0$:

$$(k \bmod n, k+1 \bmod n, \ldots, k+n-2 \bmod n, k+n-1 \bmod n)$$

is called an n by k rotation.

If you imagine all n elements laid out in a circle, we're shifting each one "clockwise" by k.

What should the interface to the `rotate` algorithm be? Designing interfaces, like designing programs, is a multi-pass activity. We can't really design an ideal interface until we have seen how the algorithm will be used, and not all the uses are immediately apparent.

At first glance, it might seem that rotation could be implemented with a modular shift, taking the beginning and end of the range, together with the amount to shift, as arguments. However, doing modular arithmetic on every operation is quite expensive. Also, it turns out that rotation is equivalent to exchanging different length blocks, a task which is extremely useful for many applications. Viewed this way, it is convenient to present rotation with three iterators: f, m and l where $[f, m)$ and $[m, l)$ are valid ranges.[1] Rotation then interchanges ranges $[f, m)$ and $[m, l)$. If the client wants to rotate k positions in a range $[f, l)$, then it should pass a value of m equal to $l - k$. As an example, this rotation:

```
0 1 2 3 4 5 6
f   m         l
```

produces this result:

```
  2 3 4 5 6 0 1
```

Note that we are doing a rotation with $k = 5$; essentially, we're moving each item 5 spaces to the right (and wrapping around when we run off the end).

Exercise 10.2

Prove that if we do `rotate(f, m, l)` then it performs distance(f, l) by distance(m, l) rotation.

An important `rotate` algorithm was developed by David Gries, a professor at Cornell University, in collaboration with IBM researcher Harlan Mills [cite]:

```
template <ForwardIterator I>
void gries_mills_rotate(I f, I m, I l) {
  // u = distance(f, m) && v = distance(m, l)
  if (f == m || m == l) return; // u == 0 || v == 0
  pair<I, I> p = swap_ranges(f, m, m, l);
  while(p.first != m || p.second != l) {
    if (p.first == m) {         // u < v
      f = m; m = p.second;      // v = v - u
    } else {                    // v < u
```

[1] The names f, m, and l are meant to be mnemonic for *first*, *middle*, and *last*.

```
      f = p.first;              // u = u - v
    }
    p = swap_ranges(f, m, m, l);
  }
  return;                       // u == v
}
```

The algorithm first uses the regular swap_ranges function to swap as many elements as we can — as many elements as there are in the shorter range. The `if` test checks whether we've exhausted the first range or the second range. Depending on the result, we reset the start positions of f and m. Then we do another swap, and repeat the process until neither range has remaining elements.

This is easier to follow with an example. Let's look at how a range is transformed, and how the iterators move, as the algorithm runs:

```
0 1 2 3 4 5 6
f   m       l

2 3 0 1 4 5 6
    f   m   l

2 3 4 5 0 1 6
        f   m l

2 3 4 5 6 1 0
          f m l

2 3 4 5 6 0 1
            f m
              l
```

Now look at the comments in the `gries_mills_rotate` code above (shown in red). We call u the length of the 1st range $[f, m)$, and v the length of the 2nd range $[m, l)$. We can observe something remarkable: the annotations are performing our familiar subtractive GCD! At the end of the algorithm, $u = v =$ gcd.

Exercise 10.3

If you inline `swap_ranges` you see that the algorithm does unnecessary iterator comparisons. Re-write the algorithm so that no unnecessary iterator comparisons are done.

It turns out that many applications benefit if rotate returns a new middle: a position where the first element moved. If rotate returns this new middle, then `rotate(f, rotate(f, m, l), l)` is an identity permutation. The task is to find a way to return the desired value without doing any extra work. It took years to figure out how to do this. First, we need the following "auxiliary rotate" algorithm:

```
template <ForwardIterator I>
I rotate_unguarded(I f, I m, I l, I m1) {
  // assert(f != m && m != l)
  pair<I, I> p = swap_ranges(f, m, m, l);
  while (p.first != m || p.second != l) {
    f = p.first;
    if (m == f) m = p.second;
    p = swap_ranges(f, m, m, l);
  }
  return m1;
}
```

The central loop is the same as in the Gries-Mills algorithm, just written differently. (We could have written it this way before, but wanted to make the u and v computations clearer.) But this function does one more thing: it takes an extra argument, and returns that argument. The reason for this will be clear below.

We need to find m' — the element whose distance to the last is the same as m's distance from the first. It's the value returned by the first call to swap_ranges. To get it back, we can embed a call to `rotate_unguarded` in our final version of `rotate`, which works as long as we have forward iterators:

```
template <ForwardIterator I>
I rotate(I f, I m, I l, forward_iterator_tag) {
  if (f == m) return l;
  if (m == l) return f;
  pair<I, I> p = swap_ranges(f, m, m, l);
  while (p.first != m || p.second != l) {
    if (p.second == l)
      return rotate_unguarded(p.first, m, l, p.first);
    f = m;
    m = p.second;
    p = swap_ranges(f, m, m, l);
  }
  return m;
}
```

How much work does this algorithm do? Until the last iteration of the main loop, every swap puts one element in the right place, and moves another element out of the way. But in the final call to `swap_ranges`, the two ranges are the same length, so every swap puts both elements it is swapping into their final positions, i.e. we get an extra move for free on every swap. So the total number of swaps is the total number of elements n, minus the number of free swaps we saved in the last step. How many swaps did we save in the last step? The length of the ranges at the end, which as we saw above, is $\gcd(n, k)$, where n = distance(f, l) and k = distance(m, l). So the total number of swaps is $n - \gcd(n, k)$. And since each swap takes three assignments (`tmp = a; a = b; b = tmp`), the total number of assignments is $3(n - \gcd(n, k))$.

10.3 Using Cycles

Can we find a faster algorithm? We can if we exploit the fact that rotations, like any permutation, have cycles. Consider a rotation of $k = 2$ for $n = 6$ elements:

```
0 1 2 3 4 5
4 5 0 1 2 3
```

The item in position 0 ends up in position 2, 2 ends up in 4, and 4 ends up back in position 0. These three elements form a cycle. Similarly, item 1 ends up in 3, 3 in 5, and 5 back in 1, forming another cycle. So this rotation has two cycles. Recall from Section 9.4 that we can perform any permutation in $n - u + v$ assignments, where n is the number of elements, u is the number of trivial cycles, and v is the number of non-trivial cycles. Since we normally don't have any non-trivial cycles, we need $n + v$ assignments.

Exercise 10.x

Prove that if a rotation of n elements has a trivial cycle, then it has n trivial cycles. (In other words, a rotation either moves all elements or no elements.)

It turns out that the number of cycles is $\gcd(k, n)$, so we should be able to do the rotation in $n + \gcd(k, n)$ assignments,[2] instead of the $3(n - \gcd(n, k))$ we needed for the Gries-Mills algorithm. Furthermore, in practice gcd is very small; in fact is 1 (that is, there is only one cycle) about 60% of the time. [cite] So a rotate algorithm that exploits cycles always does fewer assignments.

There is one catch: Gries-Mills only required moving one step forward; it works even for singly-linked lists. But if we want to take advantage of cycles, we need to be able to do

[2] See EoP pp. 178-179 for the proof.

long jumps. Such an algorithm requires stronger requirements on the iterators, namely the ability to do random access.

To create our new rotate function, we'll first write a helper function that moves every element in a cycle to its next position. But instead of saying "what position does the item in position x move to," we'll say "what position do we find the item that's going to move to position x." (Even though these two operations are symmetric mathematically, it turns out that the latter is more efficient, since it only needs one saved temporary per cycle, instead of every step.)

Here's our helper function:

```
template <ForwardIterator I, Transformation F>
void cycle_from(I i, F from) {
  typedef typename iterator_traits<I>::value_type V;
  V tmp = *i;
  I start = i;
  for (I j = from(i); j != start; j = from(j)) {
    *i = *j;
    i = j;
  }
  *i = tmp;
}
```

(The line beginning with `typedef` is a bit of C++ syntax that simply lets us define `V` to be the *value type* of the iterator `I`, that is, the type of the items the iterator is iterating over.) How does `cycle_from` know what position an item comes from? That information will be encapsulated in a function `from` that we pass in as an argument. You can think of `from(i)` as "compute where the element moving into position i comes from."

A *function object*, also known as a *functor*, is an object that can be called as a function, and which can maintain additional state. Function objects are typically used when we want to pass a function, rather than a conventional piece of data, to a function. They are also much faster than the traditional C-style approach of passing a pointer to a function, since the operator code is inlined — that is, there is no function call overhead.

The function object we're going to pass to `cycle_from` will be an instance of `rotate_transform`:

```
template <RandomAccessIterator I>
struct rotate_transform {
  typedef typename iterator_traits<I>::difference_type N;
  N plus;
  N minus;
  I m1;

  rotate_transform(I f, I m, I l) :
```

```
    plus(m - f), minus(m - l), m1(f + (l - m)){}

  I operator()(I i) const {
    return i + ((i < m1) ?  plus : minus);
  }
};
```

The idea is that even though we are conceptually "rotating" elements, in practice some items move forward and some move backward (because the rotation caused them to wrap around the end of our range). When `rotate_transform` is instantiated for a given set of ranges, it precomputes (1) how much to move forward for items that should move forward, (2) how much to move backward for things that move backward, and (3) what the crossover point is for deciding when to move forward and when to move backward.

Now we can write the modified algorithm for rotation, discovered by Fletcher and Silver in 1965 [cite]:

```
template <RandomAccessIterator I>
I rotate(I f, I m, I l, random_access_iterator_tag) {
  if (f == m) return l;
  if (m == l) return f;
  typedef iterator_traits<I>::difference_type N;
  N d = gcd(m - f, l - m);
  rotate_transform<I> rotator(f, m, l);
  while (d-- > 0) cycle_from(f + d, rotator);
  return rotator.m1;
}
```

After handling some trivial boundary cases, the algorithm first computes the number of cycles (the gcd) and constructs a `rotate_transform` object. Then it calls `cycle_from` to shift all the elements along each cycle, and repeats this for every cycle.

Notice that the signatures of this `rotate` function and the previous one differ by the type of the last argument. In the next section, we'll see how the fastest implementation for a given situation can be automatically invoked.

When Is an Algorithm Faster In Practice?

We have seen above an example where one algorithm does fewer assignments than another algorithm. Does that mean it will run faster? Not necessarily. In practice, the ability to fit relevant data in cache can make a dramatic difference in its speed. An algorithm that involves large jumps in memory — that is, one that has poor locality of reference — may

end up being slower than one which requires more assignments but does not have this property.

10.4 Reverse

Another fundamental algorithm is *reverse*, which (obviously) reverses the order of the elements of a sequence. More formally, reverse permutes a k-element list such that item 0 and item $k-1$ are swapped, item 1 and item $k-2$ are swapped, and so on.

If we have `reverse`, we can implement `rotate` in just three lines of code:

```
template <BidirectionalIterator I>
void three_reverse_rotate(I f, I m, I l) {
     reverse(f, m);
     reverse(m, l);
     reverse(f, l);
}
```

Exercise 10.4

How many assignments does 3-reverse rotate perform?

This elegant algorithm, whose inventor is unknown, works for bidirectional iterators. However, it has one problem: It doesn't return the new middle position. To solve this, we're going to break the final `reverse` call into two parts. We'll need a new function that reverses elements until one of the two iterators reaches the end:

```
template <BidirectionalIterator I>
pair<I, I> reverse_until(I f, I m, I l) {
  while (f != m && m != l) swap(*f++, *--l);
  return pair<I, I>(f, l);
}
```

At the end of this function, the iterator that didn't hit the end will be pointing to the new middle.

Now we can write a general `rotate` function for bidirectional iterators. When it gets to what would have been the third `reverse` call, it does `reverse_until` instead, saves the new middle position, and then finishes reversing the rest of the range:

```
template <BidirectionalIterator I>
I rotate(I f, I m, I l, bidirectional_iterator_tag) {
```

```
    reverse(f, m);
    reverse(m, l);
    pair<I, I> p = reverse_until(f, m, l);
    reverse(p.first, p.second);
    if (m == p.first) return p.second;
    return p.first;
}
```

We have seen three different implementations of `rotate`, each optimized for different types of iterators. However, we'd like to hide this complexity from the programmer who's going to be using these functions. We can do this with a bit of *template metaprogramming*, a programming technique that uses essentially uses the compiler to do some of the work of executing the code:

```
template <ForwardIterator I>
inline
I rotate(I f, I m, I l) {
  typename iterator_traits<I>::iterator_category c;
  return rotate(f, m, l, c);
}
```

The programmer only needs to call a single `rotate` function; the compiler will extract the type of the iterator being used and invoke the appropriate implementation. Note that no additional code is generated, so there is no performance cost to doing this dispatch.

We've been using the `reverse` function, but how might we implement it? For bidirectional iterators, the code is fairly straightforward; we have a pointer to the beginning that moves forward, and a pointer to the end that moves backward, and we keep swapping the elements they point to until they run into each other:

```
template <BidirectionalIterator I>
void reverse(I f, I l, bidirectional_iterator_tag) {
  while (f != l && f != --l) swap(*f++, *l);
}
```

It might appear that according to the law of useful return we should return `pair<I, I>(f, l)`. However, there is no evidence that it is useful; therefore the law does not apply.

In many circumstances, the above function is suboptimal, because every time through the loop, we have to do two comparisons. Often we know, or can easily compute in advance, how many times we'll need to iterate (the *trip count*). Then we could use a function that does $n/2$ swaps:

```
template <BidirectionalIterator I, Integer N>
void reverse_n(I f, I l, N n) {
  n >>= 1;
  while (n-- > N(0)) {
    swap(*f++, *--l);
  }
}
```

In particular, if we have a random access iterator, we can compute the trip count in constant time, and implement `reverse` using `reverse_n`, like this:

```
template <RandomAccessIterator I>
void reverse(I f, I l, random_access_iterator_tag) {
  reverse_n(f, l, l - f);
}
```

Exercise 10.5

Unroll the loop of `reverse_n` 4 times.

What if we only have forward iterators and we still want to reverse? Earlier, we implemented `rotate` with the help of `reverse`. Now we're going to do the opposite. We'll use a recursive algorithm that keeps partitioning the range in half (the `h` in the code). The argument `n` keeps track of the length of the sequence being reversed:

```
template <ForwardIterator I, BinaryInteger N>
I reverse_recursive(I f, N n) {
  if (n == 0) return f;
  if (n == 1) return ++f;
  N h = n >> 1;
  I m = reverse_recursive(f, h);
  std::advance(m, n & 1);
  I l = reverse_recursive(m, h);
  swap_ranges_n(f, m, h);
  return l;
}
```

The function returns the end of the range, so the first recursive call returns the midpoint. Then we advance the midpoint by 1 or 0 depending on whether the length is even or odd. The interesting thing about this function is that it's not clear where the actual work happens — essentially, the stack does all the work.

Now we can write our `reverse` function for forward iterators:

```
template <ForwardIterator I>
void reverse(I f, I l, forward_iterator_tag) {
  reverse_recursive(f, distance(f, l));
}
```

Finally, we can write the generic version of `reverse` that works for any iterator type, just as we did for `rotate` earlier:

```
template <ForwardIterator I>
inline
void reverse(I f, I l) {
  typename iterator_traits<I>::iterator_category c;
  reverse(f, l, c);
}
```

10.5 Space Complexity

The idea of *computational complexity* was invented at General Electric Research in 1964 by Juris Hartmanis and Richard Stearns. Hartmanis and Stearns originally worked on *time* complexity, but later, with their colleague Phil Lewis, extended the idea to *space* complexity.

When talking about concrete algorithms, programmers need to think about where they fall in terms of time and space complexity. There are many levels of time complexity (e.g. constant, logarithmic, quadratic, etc.). However, there are traditionally considered to be only two types of algorithms with respect to space complexity: those which perform their computations *in place*, and those which do not.

An algorithm is *in-place* if for an input of length n it uses $O(p(\log n))$ additional space where p is a polynomial.[3] (Initially, in-place algorithms were often defined as using constant space, but this was too narrow a restriction. For example, a divide-and-conquer sort algorithm like quicksort, which uses log space on the stack, was intended to be included.)

Algorithms that are not in-place use more space — usually, enough to create a copy of their data.

Let's see how a non-in-place algorithm can be faster than an in-place algorithm. First, we need this helper function, which copies elements in reverse order, starting at the end of a range:

```
template <BidirectionalIterator I, OutputIterator O>
O reverse_copy(I f, I l, O r) {
```

[3] Such algorithms are also called *polylog space* algorithms.

```
  while (f ! = l) *r++ = *--l;
  return r;
}
```

Now we can write a non-in-place `reverse` algorithm; copies all the data to a buffer, then copies it back in reverse order:

```
template <ForwardIterator I, Integer n, BidirectionalIterator B>
I reverse_n_with_buffer(I f, N n, B buffer) {
  return reverse_copy(buffer, copy_n(f, n, buffer), f);
}
```

This function only takes $2n$ assignments, instead of the $3n$ we needed for the swap-based implementations. (In practice the actual running time depends largely on whether the data fits in cache.)

10.6 Memory-Adaptive Algorithms

In practice, the dichotomy of in-place and not in-place algorithms is not very useful. While the assumption of unlimited memory is not realistic, neither is the assumption of only polylog extra memory. Usually 25%, 10%, 5% or at least 1% of extra memory is available, and can be exploited to get significant performance gains. Algorithms need to adapt to however much is available; they need to be *memory-adaptive*.

Let's create a memory-adaptive algorithm for `reverse`. It takes a buffer `b` that we can use as temporary space, and an argument `b_n` indicating how big the buffer is. The overall algorithm is recursive, but the recursion only happens on large chunks, so the overhead is acceptable. The idea is that for a given invocation of the function, if the length of the sequence being reversed fits in the buffer, we do the fast reverse with buffer. If not, we recurse, splitting the sequence in half:

```
template <ForwardIterator I, Integer n, BidirectionalIterator B>
I reverse_n_adaptive(I f, N n, B b, N b_n) {
  if (n == N(0)) return f;
  if (n == N(1)) return ++f;
  if (n <= b_n) return reverse_n_with_buffer(f, n, b);
  N h = n >> 1;
  I m = reverse_n_adaptive(f, h, b, b_n);
  advance(m, n & 1);
  I l = reverse_n_adaptive(m, h, b, b_n);
  swap_ranges_n(f, m, h);
  return l;
}
```

The caller of this function should ask the system how much memory is available, and pass that value as `b_n`. Unfortunately, such a call is not provided in most operating systems.

The Sad Story of `get_temporary_buffer`

When the C++ STL library was created, the designer realized that it would be helpful to have a function `get_temporary_buffer` that takes a size and returns the largest available temporary buffer no greater than that size which fits into physical memory. As a placeholder, he wrote a crude and impractical implementation, which repeatedly calls `malloc` asking for initially huge and gradually smaller chunks of memory until it returns a valid pointer. He put in a prominent comment in the code saying something like "this is a bogus implementation, replace it!" To his surprise, he discovered years later that all the major vendors who provide STL implementations are still using this terrible implementation — but they removed his comment.

10.7 Lessons of the Journey

In this journey, we followed a single algorithm, gcd, from its origins among the Pythagoreans in ancient times to its surprising appearance in modern algorithms like `rotate`. Along the way, we've seen how mathematicians from Stevin to Gauss to Dirichlet were able to generalize the requirements further and further, culminating in Noether's work on abstract algebra.

All programmers, even the most junior, are heirs of Pythagoras. Even if we're working on the most mundane task, we should write code in the spirit of this tradition. We should take inspiration from the works of all the mathematicians who came before us. Computer scientists should venerate not only today's Turing Award winners like Hartmanis and Knuth, but also people like Poincaré and Noether who laid the foundation for their work.

Journey Three: Successors of Peano

Chapter 11

Numbers from the Ground Up

This journey is about the simplest algorithm we've encountered yet: adding 1. But it will take us to ideas that are used in modern programming, like iterators and searching data structures. Along the way, we'll encounter some of the strangest entities in mathematics — multiple levels of infinities, self-referential propositions, and more.

We'll start by looking at some seemingly basic ideas about how we can use axioms to define everyday arithmetic. Many people believe that mathematicians first sit down and write axioms, then derive theorems from them. This is rarely the case. In practice, mathematicians explore a new area and begin discovering theorems. Only later do they step back and ask what minimal set of assumptions are needed. As we shall see, axioms play a role in programming as well; C++ concepts (which we'll cover in Chapter 14) are essentially a way of grouping axioms.

11.1 Euclid and the Axiomatic Method

Euclid was the first mathematician who developed the *axiomatic method*, in which an entire mathematical system was built on the basis of a few formal principles. In fact, for centuries Euclid's were the only known examples of axioms, and they applied only to geometry.

Euclid divided his principles into three groups: Definitions, Postulates, and Common Notions.

Euclid gave 23 definitions, including these:

> 1. A point is that which has not parts.
> ...
> 23. Parallel straight lines are straight lines which, being in the same plane and being produced indefinitely in both directions, do not meet one another in either direction.

Notice that definition 23 explicitly mentions the fact that the lines must be in the same plane, something that some casual definitions forget.

He gave the following five "common notions":

1. Things which are equal to the same thing are also equal to one another.
2. If equals be added to equals, the whole are equal.
3. If equals be subtracted from equals, the remainders are equal.
4. Things which coincide with one another are equal to one another.
5. The whole is greater than the part.

Today we would express these notions as follows:

1. $a = c \wedge b = c \implies a = b$
2. $a = b \wedge c = d \implies a + c = b + d$
3. $a = b \wedge c = d \implies a - c = b - d$
4. $a \equiv b \implies a = b$
5. $a < a + b$

What's interesting about these is that they are not limited to geometry; they apply to integers, complex numbers, etc. In fact, these common notions, such as transitivity of equality, are essential to programming.[1]

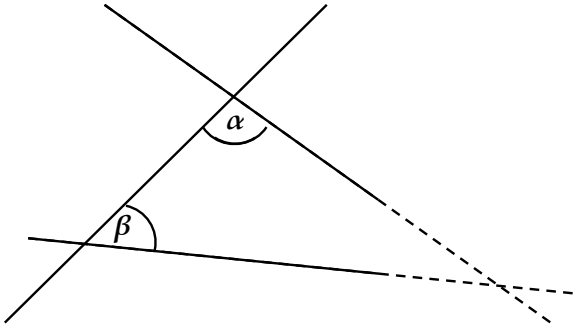
Finally, he stated his famous five postulates. These are stated in terms of allowable operations in the "computational machinery" of his geometric system; you can read the first three as being prefixed with a statement like "It is possible...":

1. To draw a straight line from any point to any point.
2. To produce a finite straight line continuously in a straight line.
3. To describe a circle with any centre and distance.
4. That all right angles are equal to one another.
5. That, if a straight line falling on two straight lines make the interior angles on the same side less than two right angles, the two straight lines, if produced indefinitely, meet on that side on which are the angles less than the two right angles.

[1] The definition of regular types in the first journey is derived from these Euclidean notions.

If we were writing Euclid's system today, we would consider both "common notions" and "postulates" to be *axioms* — unprovable assumptions on which the rest of the system is built.

Euclid's fifth postulate is the most important axiom in the history of mathematics. Also known as the *parallel postulate*, it expresses the relation shown in the following diagram:



$\alpha + \beta < 180°$

However, there are many equivalent ways to express the same notion:

- Given a line and a point not on it, at most one parallel to the given line can be drawn through the point.[2]
- There exists a triangle whose angles add up to 180°.
- There exist two similar triangles that are not congruent.

11.2 The Overthrow of Euclidean Geometry

> It is no exaggeration to call Lobatchewsky the Copernicus of Geometry, for geometry is only a part of the vaster domain which he renovated; it might even be just to designate him as a Copernicus of all thought.
>
> E.T. Bell, *Men of Mathematics* [cite]

Almost from the time Euclid stated the five postulates, mathematicians felt that there was something different about the fifth one. Intuitively, they felt that the first four postulates were somehow more fundamental; perhaps the fifth postulate could be derived

[2]This formulation, which is often taught as "the parallel postulate" in secondary school geometry, was actually published by Scottish mathematician John Playfair in 1795, and is properly known as *Playfair's Axiom*.

from the others, and therefore was not a true axiom. Thus began a 2000-year search for a proof of the fifth postulate, one pursued by such luminaries as the astronomer (and mathematician) Ptolemy (90-168), the poet (and mathematician) Omar Khayyam (1050-1153), and the Italian priest (and mathematician) Girolomo Saccheri (1667-1733). Saccheri wrote a book called *Euclidus Vindicatus* ("Euclid Vindicated") in which he constructed a whole geometrical system based on the the assumption that the fifth postulate is false, then claimed that the results would be so bizarre that the postulate must be true.

While most 18th century mathematicians didn't care about axioms, the mood shifted in the 19th century. Mathematicians started to focus on the foundations of their work. They revisited geometry, no longer taking Euclid for granted, but questioning his work.

Around 1824, Russian mathematician Nikolai Lobachevsky was working on the problem. At some point, he realized that the parallel postulate was just one possible assumption, and that other assumptions were equally valid. Instead of saying "there is at most one line through a point parallel to a given line," Lobachevsky essentially said, "there are many lines..." Unlike, Saccheri, Lobachevsky realized that the resulting system of geometry was entirely consistent. In other words, he invented an entirely new non-Euclidean geometry, sometimes called *hyperbolic* geometry.

In Lobachevsky's geometry, there are no similar triangles except for congruent ones. By way of analogy, think of triangles on the surface of a sphere. For small triangles, the surface is almost planar, so the angles add up to almost 180°. But as the triangles get bigger, the angles need to get bigger because of the curvature of the surface. Lobachevsky's model was similar, but with space curved in the opposite way, so that bigger triangles corresponded to smaller angles.

Lobachevsky's results, first published in 1826, were met with dismissal and scorn from the Russian mathematical community, and Lobachevsky himself was marginalized. One person who recognized the validity of Lobachevsky's work was Gauss, who learned Russian to read Lobachevsky's book, and who nominated him for membership in the Göttingen Academy of Sciences near the end of his career. But in general, it would take many years before his work became an accepted part of mathematics. Today, as the quotation at the start of the section shows, Lobachevsky's discovery is considered to be a monumental turning point in the history of mathematics.

Nikolai Ivanovich Lobachevsky (1792–1856)

[Biography of Lobachevsky goes here. Some points to include:

- He's a provincial guy from a small insignificant place called Kazan on the Volga river. At first there's no university there, but one is founded in 1808, and he joins their first class. (Tolstoy and Lenin would later go there, but that's decades in the future.)

- Taught by Johann Bartels, Professor of Gauss, at Kazan University.
- Becomes Professor, then rector (president) of his university.
- His work on Non-Euclidean Geometry first reported in 1826, then "rejected by the philistines" over the next several years.
- Elected to Göttingen Academy of Sciences (1842).
- Last major book (Pangeometry) in 1855

]

Often when a new idea emerges in math or science, it is discovered independently by multiple people at roughly the same time. This was the case with non-Euclidean geometry. At about the same time Lobachevsky was working in Kazan, a young Hungarian mathematician named János Bolyai made a similar discovery. A few years later, Bolyai's father Farkas Bolyai, a well-known math professor and friend of Gauss, included the son's results as an appendix to one of his own books. Farkas sent Gauss the book. Although Gauss privately remarked that young Bolyai was a genius, the letter he sent Farkas had a discouraging message:

> If I commenced by saying that I am unable to praise this work, you would certainly be surprised for a moment. But I cannot say otherwise. To praise it would be to praise myself. Indeed the whole contents of the work, the path taken by your son, the results to which he is led, coincide almost entirely with my meditations, which have occupied my mind partly for the last thirty or thirty-five years.

The letter is typical of Gauss, both in his refusal to give credit to others and in his insistence that his own unpublished thoughts gave him priority. (We now know that Gauss had indeed discovered many of the same ideas, but had decided not to publish because he was afraid of the reaction.) Why he acknowledged Lobachevsky's work but condemned Bolyai's we will never know. But whatever the reason, the results were tragic. Bolyai was devastated by Gauss's rebuke, and never worked in mathematics again. Even sadder, he became mentally unstable. When he came across Lobachevsky's book some time later, he was convinced that "Lobachevsky" was actually a pseudonym for Gauss, who had stolen Bolyai's ideas.

Once non-Euclidean geometry was discovered, many mathematicians wrestled with what they considered to be an important question: Which geometry is actually correct, Euclid's or Lobachevsky's? Gauss took the question quite seriously, and proposed an ingenious experiment to test the theory:

First, find three mountains forming a triangle that are some distance apart, but close enough so that a person standing on top of each with a telescope can see the others. Then set up surveying equipment on each peak to accurately measure the angles of the triangle. If the angles add up to 180°, then Euclid is right; if less than 180, then Lobachevsky is.

As far as we know, the actual experiment was never conducted. But over time, the question became moot. Beltrami, Poincaré, and Klein would ultimately prove the independence of the fifth postulate, showing that if Euclidean geometry is valid, then so is Lobachevskian. More importantly, mathematicians began to treat questions of reality as irrelevant. While math was originally invented to understand aspects of the world we live in, by the end of the 19th century, it began to be seen as a purely abstract exercise.

11.3 Hilbert's Approach

> It does not matter if we call the things *chairs*, *tables* and *beer mugs* or *points*, *lines* and *planes*.
>
> – David Hilbert [cite]

David Hilbert, perhaps the greatest mathematician of the early 20th century, was a strong advocate of this abstract approach In a view that eventually became standard throughout mathematics, he said that if a theory was consistent, it was true.

Hilbert spent ten years rethinking Euclid and constructing his own axiomatic system for geometry. While all of Euclid's theorems and proofs are correct, by modern standards the axioms are somewhat shaky. It took 2400 years before anyone tried to come up with a better foundation for geometry. Hilbert's system contained many more axioms than Euclid, making explicit many things that Euclid took for granted. Hilbert had:

- 7 axioms of connection (e.g. If two points lie on a plane, all points on the line going through these points are on this plane)
- 4 axioms of order (e.g. There is a point between any two points on a line)
- 1 axiom of parallels
- 6 axioms of congruence (e.g. Two triangles are congruent if side-angle-side...)
- 1 Archimedes' axiom
- 1 Completeness axiom

Hilbert's geometric system is quite complex, and could itself be the basis of an entire course. Unfortunately, by the time he was done constructing the axioms, he had no energy left to prove any geometric theorems. Hilbert's work on the axioms of Euclidean geometry was the last major work done on that subject.

David Hilbert 1862–1943

[Biography of Hilbert goes here. His work includes:

- Invariant theory,
- Theory of algebraic integers,
- Foundations of geometry – spends 10 years rethinking Euclid.
- Hilbert Spaces,
- Mathematical Physics – Co-invents General Relativity Theory (curved space and all), independently and at roughly the same time as Einstein.
- Foundations of Mathematics. Proposed to develop a (possibly abstract) machine to prove all theorems. There is a direct path from this work to the development of computers.

]

In 1900, Hilbert gave a lecture at the Sorbonne in Paris where he listed 10 important unsolved problems in mathematics, and challenged the community to work on them. The list was later expanded to 23 problems in a published paper. Work on these problems, which became known as *Hilbert's problems*, defined much of mathematics in the 20th century. We'll talk about the first problem, the *continuum hypothesis*, in the next chapter; it concerns the sizes of certain infinite sets. We shall also see how attempts to solve the second problem, the consistency of arithmetic, let to the theoretic origins of computing.

11.4 Peano and His Axioms

> Certainly it is permitted to anyone to put forward whatever hypotheses he wishes, and to develop the logical consequences contained in those hypotheses. But in order that this work merit the name of Geometry, it is necessary that these hypotheses or postulates express the result of the more simple and elementary observations of physical figures.
>
> – Giuseppe Peano [cite]

Before Hilbert announced his program, others had been working on similar ideas about formalizing mathematical systems. One of these was Italian mathematician Giuseppe

Peano. As the above quotation shows, Peano was still interested in the connections between mathematics and reality. In 1891, he began writing *Formulario Mathematico* ("Mathematical Formulas"), what would become a multi-volume work containing all the important theorems expressed in a symbolic notation he invented. Some of his notation, such as the symbols for quantifiers, are still used today.

In 1889, Peano published a set of axioms that provided a formal basis for arithmetic. There were five, just like Euclid's:

There is a set $\mathbb{N}$ called the *natural numbers*:

1. $\exists 0 \in \mathbb{N}$
2. $\forall n \in \mathbb{N} \ : \ \exists n' \in \mathbb{N}$ – called its *successor*
3. $\forall \mathbb{S} \subset \mathbb{N} \ : \ (0 \in \mathbb{S} \wedge \forall n \ : \ n \in \mathbb{S} \implies n' \in \mathbb{S}) \implies \mathbb{S} = \mathbb{N}$
4. $\forall n, m \in \mathbb{N} \ : \ n' = m' \implies n = m$
5. $\forall n \in \mathbb{N} \ : \ n' \neq 0$

The third axiom, known as the *axiom of induction*, is the most important. It says that if we take any subset S of $\mathbb{N}$ that contains zero and obeys the rule that the successor of every element is also in S, then S is the same as $\mathbb{N}$. Another way to put this is "there are no unreachable natural numbers"; if you start with zero and keep taking successor, you'll eventually get to every natural number. Many modern texts put this axiom last, but we'll keep Peano's order.[3]

Peano's axioms transformed arithmetic. In fact, he was building on earlier work by Dedekind (whom we met in Journey 2) and Hermann Grassman, both of whom showed how to derive some basic principles of arithmetic. But Peano went farther, and in fact, mathematicians since then talk about *Peano arithmetic*, not just arithmetic.

Giuseppe Peano (1858–1932)

[Biography of Peano goes here. Some points to include:

- He was a peasant boy from Italy, born right at the time the country is being united.
- Worked on space-filling curve in 1890. Hilbert published the same thing in the same journal 2 years later, without acknowledging Peano.

[3] Modern texts often also start natural numbers with 1 rather than 0, and replace the *second order* induction axiom (i.e. it uses operations on sets, not just on elements of a set) with a *first order* induction axiom schema.

- Although Peano's work was ahead of Hilbert's in many ways, Peano had the misfortune of being at a provincial Italian university (Turin) when Göttingen was at its peak.
- Peano was going to write in French but decided it was too ambiguous, so invented his own unambiguous language for scientists, *Latine sine Flexione* ("Latin without Inflection"), later renamed "Interlingua." Therefore his great work, *Formulario Mathematico* is largely unknown. It has never been translated and is not in print.
- Most people don't know anything about the book except the 1st page, where he lays out the five axioms listed above. Here's what they looked like in his language:
  0. N_0 es classe, vel "numero" es nomen commune.
  1. Zero es numero.
  2. Si *a* es numero, tunc suo successivo es numero.
  3. N_0 es classe minimo, que satisfac ad conditione 0, 1, 2; [...]
  4. Duo numero, que habe successivo aequale, es aequale inter se.
  5. 0 non seque ullo numero.

]

To prove that every axiom is needed, we need to remove each one from the set and demonstrate that the remaining set has consequences that do not meet our intent — in this case, that they do not correspond to what we mean by natural numbers.

Removing existence of 0 axiom. If we remove this axiom, we are forced to drop all axioms that refer to zero. Since we have no elements to start with, the other axioms never apply, and can be satisfied by the empty set.

Removing totality of successor axiom. If we remove the requirement that every value have a successor, then we end up allowing finite sets like {0} or {0, 1, 2}. Clearly, no finite set satisfies our notion of natural numbers. (However, on computers, we give up this axiom, since all of our data types are finite; for example, a `uint64` can only express the first 2^{64} integers.)

Removing induction axiom. If we remove the induction axiom, than we end up with the situation where we have more integer-like things than there are integers. These "unreachable" numbers are called *transfinite ordinals*, and are designated by ω. So we could end up with sets like $\{0, 1, 2, 3, \ldots, \omega, \omega + 1, \omega + 2, \ldots\}$, $\{0, 1, 2, 3, \ldots, \omega_1, \omega_1 + 1, \omega_1 + 2, \ldots, \omega_2, \omega_2 + 1, \omega_2 + 2, \ldots\}$ and so on.

Removing invertibility of successor axiom. If we remove the requirement that equal successors have equal predecessors, then we're allowing "ρ-shaped" structures where an item can have multiple predecessors, some earlier in the sequence and some later, such as $\{0, 1, 1, 1, \ldots\}$, $\{0, 1, 2, 1, 2, \ldots\}$, or $\{0, 1, 2, 3, 4, 5, 3, 4, 5, \ldots\}$.

Removing "Nothing has 0 as its successor" axiom. If we remove this axiom, then we'd allow sets that loop back to zero, like $\{0, 0, \ldots\}$ and $\{0, 1, 0, 1, \ldots\}$.

11.5 Building Arithmetic

Now that we have established that all of Peano's axioms are independent, and therefore necessary for our notion of natural numbers, we can define arithmetic operations based on these axioms.

Definition of Addition:

$$a + 0 = a \tag{11.1}$$
$$a + b' = (a + b)' \tag{11.2}$$

Note that we are not *proving* these statements; we are defining addition to *be* these statements. All properties of adding natural numbers follow from this definition. For example, here's how we would prove that 0 is the left additive identity, i.e. $0 + a = a$.

$$\begin{aligned} \text{base step}: &\ 0 + 0 = 0 \\ \text{induction step}: &\ 0 + a = a \implies 0 + a' = (0 + a)' = a' \end{aligned}$$

In the base step, we assert that it's true when a is zero. We know this because of equation 11.1 in the definition of addition. In the induction step, we assume it's true for any a. By equation 11.2, we know that $0 + a' = (0 + a)'$. But by the assumption of the induction step, we can substitute a for $0 + a$, so our result is a', and therefore $0 + a' = a'$.

Definition of Multiplication:

$$a \cdot 0 = 0 \tag{11.3}$$
$$a \cdot b' = (a \cdot b) + a \tag{11.4}$$

We can now prove that $0 \cdot a = 0$, much as we did above:

$$\begin{aligned} \text{base step}: &\ 0 \cdot 0 = 0 \\ \text{induction step}: &\ 0 \cdot \text{a} = 0 \implies 0 \cdot a' = 0 \cdot a + 0 = 0 \end{aligned}$$

Definition of 1. We also define 1 as the successor of 0:

$$1 = 0'$$

Now we know how to add 1:

$$a + 1 = a + 0' = (a + 0)' = a'$$

and how to multiply by 1:

$$a \cdot 1 = a \cdot 0' = a \cdot 0 + a = 0 + a = a$$

We can also derive fundamental properties of addition; they also follow from the axioms.

Associativity of Addition: $(a + b) + c = a + (b + c)$

$$\begin{aligned}
&\text{base step :}\\
&(a+b)+0 = a+b = a+(b+0)\\
&\text{induction step :}\\
&(a+b)+c = a+(b+c) \implies\\
&(a+b)+c' = ((a+b)+c)' =\\
&(a+(b+c))' = a+(b+c)' = a+(b+c')
\end{aligned}$$

(This relies on the fact that $b = b + 0$)

To get commutativity, we'll start by proving it for the special case $a + 1 = 1 + a$:

$$\begin{aligned}
&\text{base step :}\\
&1+0 = 1+0 = 1\\
&\text{induction step :}\\
&a+1 = 1+a \implies\\
&a'+1 = a'+0' =\\
&(a'+0)' = ((a+1)+0)' =\\
&(a+1)' = (1+a)' = 1+a'
\end{aligned}$$

Commutativity of Addition: $a + b = b + a$

$$\begin{aligned}
&\text{base step :}\\
&a+0 = a = 0+a\\
&\text{induction step :}\\
&a+b = b+a \implies\\
&a+b' = a+(b+1) =\\
&(a+b)+1 = (b+a)+1 =\\
&b+(a+1) = b+(1+a) =\\
&(b+1)+a = b'+a
\end{aligned}$$

Exercise 11.1

Using induction, prove:

- associativity and commutativity of multiplication;
- distributivity of multiplication over addition.

Exercise 11.2

Using induction, define total ordering between natural numbers.

Exercise 11.3

Using induction, define the following partial functions on natural numbers:

- predecessor;
- successor.

Do Peano axioms define natural numbers? No; as Peano put it, "... number (positive integer) cannot be defined (seeing that the ideas of order, succession, aggregate, etc., are as complex as that of number)." If you don't already know what they are, Peano's definitions won't tell you. Instead, they *describe* our existing idea of numbers, formalizing our notions of arithmetic, which helps provide a way to structure proofs.

In general, we can say that axioms explain, not define – and the explanation could be very slow, or not constructive at all. But they still serve a useful purpose; they get us to think about which properties are essential and which are not. This is a good attitude to take when studying the documentation for a programming interface: Why is that requirement imposed? What kind of consequences would be allowed if it were not there?

Chapter 12

Sets and Infinities

> No one shall expel us from the Paradise that Cantor has created.
>
> — David Hilbert, *Über das Unendliche Mathematische Annalen* 95 (1925)

As we saw in the last chapter, Peano used a tricky word "set" (*classe* in his language) when he wrote the axioms of arithmetic. Other mathematicians used the word before, but tended to avoid talking about infinite sets. Could you take the entire group of natural numbers and treat them as a single entity? This seemed like a dangerous idea. The Greeks had largely ignored infinities — everything had to be done in a finite number of steps. But infinite sets, and other types of infinities, kept cropping up in mathematics. In this chapter, we shall see how mathematicians eventually came to deal with them.

12.1 Tackling the Infinite

In the late 1340s and early 1350s, Europe was devastated by an outbreak of bubonic plague, a pandemic that killed 30-60% of the population. French and English kings battled over control of Normandy, Brittany, and other parts of France in the Hundred Years War. Popes of the Catholic Church ruled from Avignon rather than Rome, eventually leading to a schism where both popes and "anti-popes" competed for power. Despite this tragedy and turmoil, the 14th century was actually a time when the first stirrings of the Renaissance could be felt. In Italy, Dante, Bocaccio, and Petrarch were writing their great works. Great logicians such as William of Ockham and Jean Buridan were teaching. And in France, a bishop named Nicole Oresme (pronounced "oh-RHEM") began working on how to solve infinite series.[1]

[1] Archimedes had dealt with infinite series, but only in the context of geometry. Oresme was the first to deal with them algebraically.

Consider the series

$$\sum_{i=1}^{\infty} \frac{i}{2^{i-1}} = 1 + 2 \cdot \frac{1}{2} + 3 \cdot \frac{1}{4} + \cdots + n \cdot \frac{1}{2^{n-1}} + \ldots$$

How could we compute the sum? Oresme devised the following strategy: First, he rearranged and grouped the terms like this:

$$\begin{aligned}(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{n-1}} + \ldots) + \\ (\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{n-1}} + \ldots) + \\ (\frac{1}{4} + \cdots + \frac{1}{2^{n-1}} + \ldots) + \cdots =\end{aligned}$$

Notice that by adding columns vertically, there is 1 copy of 1, then 2 copies of $\frac{1}{2}$, then 3 copies of $\frac{1}{4}$, and so on, just as the original expression specifies.

Then he notes that each row is $\frac{1}{2}$ of the row above, so we can rewrite it like this:

$$\begin{aligned}1 \cdot (1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{n-1}} + \ldots) + \\ \frac{1}{2} \cdot (1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{n-1}} + \ldots) + \\ \frac{1}{4} \cdot (1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{n-1}} + \ldots) + \cdots =\end{aligned}$$

Since every coefficient is applied to the same quantity, we can use the distributive law to pull out all the coefficients and put them together into a sum, like this:

$$(1 + \frac{1}{2} + \frac{1}{4} + \ldots) \cdot (1 + \frac{1}{2} + \frac{1}{4} + \ldots) =$$

Notice that the first term, made up of all the coefficients, is the same as the second term; they're both $1 + \frac{1}{2} + \frac{1}{4} + \ldots$. But this sum was already known; it's equal to 2. So our original sum is $2 \cdot 2$, or 4.

Nicole Oresme (1320-1382)

[Biography of Oresme goes here. Points to include:

- Probably studied with Buridan, whose was rector of the Sorbonne.
- Becomes friends with King Charles V, who asks him to translate works of Aristotle into French (from Latin).
- Came up with idea of Cartesian coordinates long before Descartes.
- Proves Merton theorem about bodies in motion.
- Came up with an early version of what we now call *Gresham's Law* that bad money drives out good.
- Invented what we now call monetarism: Wrote that kings (governments) shouldn't use monetary policy (minting money) to get revenue; if they want revenue they should collect taxes, not debase the currency.

]

Oresme was also the first to demonstrate the divergence of the Harmonic series $\sum_{i=1}^{2^n} \frac{1}{i}$. Again, his approach involved re-grouping the terms:

$$\begin{aligned}\sum_{i=1}^{2^n} \frac{1}{i} = 1 &+ \frac{1}{2} \\ &+ (\frac{1}{3} + \frac{1}{4}) \\ &+ (\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}) \\ &+ \dots \\ &+ (\frac{1}{2^{n-1}+1} + \dots + \frac{1}{2^n}) \geq \frac{n}{2}\end{aligned}$$

Oresme observed that since each group of terms is greater than $\frac{1}{2}$, the sum is greater than $\frac{n}{2}$, and we know that as n grows, the number of groups grows. Therefore the series diverges.

Oresme's work on infinite series led to mathematical ruminations about infinity. He was the first to introduce the idea on which all modern set theory is based: the idea of a *one-to-one correspondence* to measure the cardinality of infinite sets. Oresme observed that you can count odd integers with integers, so there are in some sense the same number of odd integers as integers. This was a controversial idea, since Euclid taught that the whole is greater than any of the parts, and odd integers are in some sense only a part of integers.

12.2 Paradoxes of the Infinite

> This is one of the difficulties which arise when we attempt, with our finite minds, to discuss the infinite, assigning to it those properties which we give to the finite and limited; but this I think is wrong, for we cannot speak of infinite quantities as being the one greater or less than or equal to another.
>
> – Galileo [cite]

About three hundred years after Oresme, the great Italian scientist Galileo wrote two revolutionary books. In one, *Dialogue Concerning the Two Chief World Systems*, he promoted the Copernican view that Earth and other heavenly bodies revolve around the Sun.[2] In the other, *Discourses and Mathematical Demonstrations Relating to Two New Sciences*, he discussed the strength of materials and the motion of objects, marking the beginning of modern Physics.

Galileo Galilei (1564–1642)

[Biography of Galileo goes here. Some points to include:

- His father wanted him to be a doctor; he became a mathematician and scientist instead, and eventually published 20 volumes of his work.
- He was a very good writer and attracted the attention of the Medici Duke of Florence, who gave him a court appointment.
- He later became friends with Pope Urban VIII, who asked him to write a dialogue about the Copernican system. Galileo introduced a character "Simplicio," representing the Pope's point of view, and made fun of him. Forced to recant.
- He wrote other book where he explains why animals can't scale up (square-cube law). He was forbidden to publish in Italy, so he sent the book to Mersenne who published it in the Netherlands.

]

In the *Two New Sciences* book, his final masterpiece, Galileo observed that there is a

[2] Interestingly, he insists that the orbits are circular, while Johannes Kepler, writing about the same time, realizes they are elliptical.

one-to-one correspondence between natural numbers and squares:

$$n \longleftrightarrow n^2$$
$$\{1, 2, 3, \dots\} \longleftrightarrow \{1, 4, 9, \dots\}$$

Yet the density of squares goes down as we go up. As Galileo put it:

> The proportionate number of squares diminishes as we pass to larger numbers, Thus up to 100 we have 10 squares, that is, the squares constitute 1/10 part of all the numbers; up to 10000, we find only 1/100 part to be squares; and up to a million only 1/1000 part; on the other hand in an infinite number, if one could conceive of such a thing, he would be forced to admit that there are as many squares as there are numbers taken all together. [cite]

If there are fewer and fewer squares as we count up, and the number approaches zero as we go to infinity, how can there be as many squares as numbers? This is one of many paradoxes of infinity that we'll encounter, and it illustrates that our intuitions about infinities are somehow broken.

A related paradox is described by Littlewood hundreds of years later in his book *A Mathematician's Miscellany* [cite]:

> (5) *An infinity paradox.* Balls numbered 1, 2, ... (or for a mathematician the numbers themselves) are put into a box as follows. At 1 minute to noon the numbers 1 to 10 are put in, and the number 1 is taken out. At $\frac{1}{2}$ minute to noon numbers 11 to 20 are put in and the number 2 is taken out. At $\frac{1}{3}$ minute 21 to 30 in and 3 out; and so on. How many are in the box at noon?

Intuitively, it seems that since at every step we put in more balls than we take out, there should be an infinite number of balls at the end. But at the same time, at the kth step we take out the ball labeled k, so for every ball there is a time it gets taken out. Therefore every ball is eventually taken out, and at the end we will have zero balls in the box.

12.3 Levels of Infinity

While other mathematicians deliberately avoided thinking about infinity, in a book called *Paradoxes of the Infinite*, Czech mathematician Bernard Bolzano exploring the idea of infinite sets in the early 19th century. He was also the first to use the term "one-to-one correspondence." Here are a few of his propositions:

§19 Not all infinite sets are equal with respect to their multiplicity.

> One could say that all infinite sets are infinite and thus one cannot compare them, but most people will agree that an interval in the real line is certainly a part and thus agree to a comparison of infinite sets.

§20 There are distinct infinite sets between which there is 1-1 correspondence. It is possible to have a 1-1 correspondence between an infinite set and a proper subset of it.

$y = 12/5x$ and $y = 5/12x$ gives a 1-1 correspondence between [0,5] and [0,12].

§21 If two sets A and B are infinite, one can not conclude anything about the equality of the sets even if there is a 1-1 correspondence.

If A and B are finite and A is a subset of B such that there is a 1-1 correspondence, then indeed $A = B$.

The above property is thus characteristic of infinite sets.

The last point, §21, is often known as "Dedekind's Axiom," but it came from Balzano. Basically, he realized that the property of being able to have a 1-1 correspondence with its proper subset is what *defines* an infinite set.

Bernard Bolzano (1781–1848)

[Biography of Bolzano goes here. Some points to include:

- Born in Prague in what was then the Austro-Hungarian empire, speaks Czech (considered a 2nd-class language) and German.
- When he's 22, Napoleon conquers Europe and there are constant wars.
- Bolzano becomes a priest, and professor of religious studies at Prague. He starts to preach that nations should work toward peace, that there shouldn't be differences between rich and poor, that Jews should be treated as equals, and that Czech people should be able to study at university in their own language. These were incredibly radical and subversive ideas at the time.
- His sermons get really popular, with thousands of people attending, until the emperor hears about it and he loses his job.
- He starts doing math, logic, and philosophy. He puts calculus on sounder footing. He invents the ϵ-δ definition of continuous functions, and proves many fundamental results such as the intermediate value theorem for an arbitrary continuous function, commonly known as Bolzano-Cauchy.
- He wrote 60 or 70 volumes, but only about 15% has been translated.

]

In his book *Theory of Science*, Bolzano moves from sets to ideas about what we can and cannot prove, a transition we will also make in this journey. He writes:

> That no proposition has truth disproves itself because it is itself a proposition and we should have to call it false in order to call it true. For, if all propositions were false, then this proposition itself, namely that all propositions are false, would be false. Thus, not all propositions are false, but there also true propositions.

As we shall see, this idea of self-referential propositions will play an important role in the development of set theory and computability.

Exercise 12.1

Using Bolzano's method, prove that there are infinitely many true propositions.

12.4 Set Theory

In 1873, Georg Cantor, a professor at the University of Halle in Germany, had an important correspondence with Richard Dedekind, about the size of various sets. Cantor asked whether it was possible to enumerate positive real numbers. Dedekind replied that this was "not an interesting question" [cite], but provided a proof of how to enumerate algebraic numbers (we'll explain what these are in a minute).[3] Cantor then said that if the answer to his original question is no, it would provide a new and more powerful proof of the existence of transcendental numbers. Then, in his next letter, Cantor announced that he had proved that the answer was indeed no.

To understand Cantor's proof, it's important to understand the notion of *countable* and *uncountable* sets. Countable sets are those that can be put into a one-to-one correspondence with natural numbers; as we have seen, Oresme and Bolzano were already using this idea to show that various sets were "the same size" as integers. Hilbert later came up with a nice way to think about these sets, which we now call Hilbert's Hotel [cite]: Imagine that we have a hotel with an infinite number of rooms, each holding one guest. Is the hotel full, or can it accommodate more guests?

- Suppose one guest arrives at the hotel. Is there room for her? Yes; the hotel simply asks the guest in room 1 to move to room 2, the guest in room 2 to move to room 3, and so on.
- Suppose infinitely many new guests arrive. Is there room for all of them? Yes; the hotel asks the guest in room 1 to move to room 2, the guest in room 2 to move to

[3] While "everyday" numbers are algebraic, it turns out that they account for only a tiny amount of all numbers; the rest are transcendental numbers like π.

room 4, and so on. Now all the odd-numbered rooms are vacant. Since there are infinitely many odd numbers, the new guests will all have rooms.

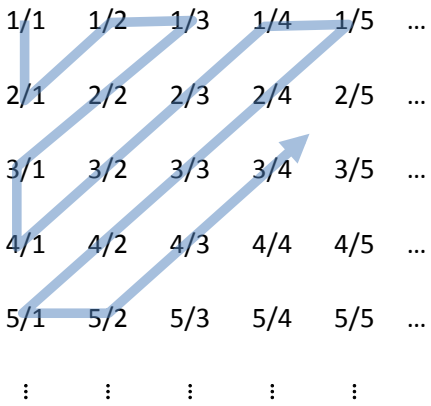
- Suppose infinitely many buses, each holding infinitely many guests, arrive. Is there room for them? Yes! The odd-numbered rooms are emptied as above. Then the people in the first bus move into rooms numbered 3^i, the people in the second bus move into rooms numbered 5^i, and so on for each prime number.[4]

What about rational numbers? At first glance it might seem like there are more rational numbers than integers, since there are infinitely many of them between each integer. Yet as with the infinite number of infinite buses, it turns out that there is a way to count them all with integers. Imagine that we have an infinite grid of rational numbers, starting with $\frac{1}{1}$. As we move from one column to the next, we'll increase the denominator by 1; as we move from one row to the next, we'll increase the numerator by 1, so the upper left corner of the grid will look like this:

1/1	1/2	1/3	1/4	1/5	...
2/1	2/2	2/3	2/4	2/5	...
3/1	3/2	3/3	3/4	3/5	...
4/1	4/2	4/3	4/4	4/5	...
5/1	5/2	5/3	5/4	5/5	...
⋮	⋮	⋮	⋮	⋮	

To count all the rational numbers, start at the upper left corner and follow this zigzag pattern:

[4] There are actually many ways to solve this case.



Since we can put rational numbers in a one-to-one correspondence with natural numbers, rational numbers are also a countably infinite set.

A real number is called *algebraic* if it is a root of a polynomial with integer coefficients. A polynomial is said to have *weight* k if k is the maximum of the absolute values of its coefficients and exponents. Since there is a finite number of polynomials of a given weight, and every one of them has finitely many roots, algebraic numbers are countable. Although people think of this as one of Cantor's proofs, it actually came from Dedekind — it's the one he mentioned in the letters described above. (Interestingly, it was Emmy Noether who published their correspondence.)

Cantor's 1874 Theorem: Uncountability of Continuum. Now we come to Cantor's theorem that the set of all real numbers (also called the *continuum*) is uncountable:

> Given a closed interval of real numbers and a sequence of real numbers, the interval contains a number that is not in the sequence.

Formally, we can state it like this:

$$\begin{aligned} &\text{If } a_0, b_0 \in \mathbb{R} \ \wedge \ a_0 < b_0 \text{ and } \{r_i\}_{i=0}^{\infty} \subseteq \mathbb{R} \\ &\quad \text{then } \exists c \in [a_0, b_0] \text{ such that } c \notin \{r_i\}. \end{aligned}$$

Informally, what it says is that you cannot enumerate every point in a line segment. The idea of the proof is to imagine that there is a list of all the real numbers, and then show that there is a number that is not on the list.

Proof:

1. Find the first pair of distinct elements in the sequence $\{r_i\}$ that is in the interval $[a_0, b_0]$. (If there is no such pair, then, obviously there is an element in $[a_0, b_0]$ such that it is not in the sequence; therefore; we are done with the proof.)

2. Designate the smaller element of the pair a_1 and the larger one b_1, and repeat steps 1 and 2 with the interval $[a_0, b_0]$ being replaced with the $[a_1, b_1]$ and the sequence $\{r_i\}$ with the "unused" elements in the sequence $\{r_i\}$.

3. Observe that r_i does not belong to the inside of the interval $[a_{2i}, b_{2i}]$.

4. So if the sequence of intervals $[a_i, b_i]$ is finite, the middle of the last interval is not in the sequence $\{r_i\}$.

5. If the sequence of intervals $[a_i, b_i]$ is infinite, no element r_i could belong to its intersection $\bigcap[a_i, b_i]$. And since we know that the intersection is not empty, an element from it will not be in $\{r_i\}$.

Exercise 12.2

Prove that the intersection of nested, closed intervals is not empty.

Georg Cantor (1845–1918)

[Bio of Cantor goes here. Born in St. Petersburg to German parents. Studied at the University of Zurich and then Berlin. Became professor at University of Halle. Met Dedekind. Leopold Kronecker, the head of the Berlin math department, said he had to "protect the youth from the deviations of Cantor."

Cantor's ideas about levels of infinity were disregarded and dismissed by other mathematicians. Poincaré, for example, said that all of set theory should be thrown out of mathematics. Eventually, Cantor became so disillusioned that his work turn an odd turn. At one point he wrote letters to the Pope and to a famous theologian, Cardinal Franzelin, telling them that his work on infinities has finally proven the existence of God. He also spent a good deal of time looking for evidence that Francis Bacon wrote Shakespeare's plays. Despite these digressions, he ended his career with some of his most important results on set theory.]

Cantor realized that there were two kinds of infinity. One was the cardinality ("size") of the natural numbers $\mathbb{N}$ and other sets equipotent to (i.e. that could be placed in a one-to-one correspondence with) $\mathbb{N}$. He labeled this number $\aleph_0$, using the first letter of the

Hebrew alphabet, *aleph*. The second was the cardinality of sets equipotent to the real numbers $\mathbb{R}$, which he called $\mathfrak{C}$ (the letter C, for "continuum," in a Gothic typeface).

Another of Cantor's important results concerned the size of the *power set* of the set of natural numbers. The power set of a set S, written 2^S or $\mathcal{P}(S)$, is the set of all subsets of S:

$$x \in 2^S \iff x \subseteq S$$

The notation derives from the fact that if S is finite, it has $2^{|S|}$ possible subsets.

Exercise 12.3

Prove that a power set of a finite set with n elements contains 2^n elements.

Cantor saw that the power set of $\aleph_0$ had the same cardinality as the real numbers:

$$|2^{\aleph_0}| = |\mathfrak{C}|$$

This followed from the fact that one set could be mapped to the other, following several steps:

1. $|\mathbb{R}| = |\mathbb{R}^+|$ through $f(x) = e^x$. That is, we can map all real numbers to just positive reals by raising e to the power of the number.

2. $|\mathbb{R}^+| = |(0,1)|$ through $f(x) = \frac{1}{x+1}$. We can map positive reals to the open interval between 0 and 1 by applying the given function.

3. $|(0,1)| = |\text{Seq}\{0,1\}|$ through binary. We can map any point in this interval to a infinite sequence of binary digits, representing a binary fraction (imagine a "binary point" before the first digit).[5] We denote the set of all such sequences Seq{0, 1}.

4. $|\text{Seq}\{0,1\}| = |2^{\aleph_0}|$ through the *characteristic function*. This is similar to the way we use bit vectors in programming. That is, we can use the first binary digit to indicate whether the first possible element is present in a given subset, the second digit to indicate whether the second possible element is present, and so on. [6]

[5] In fact, the situation is slightly more complicated, because there is more than one way to represent the same real. For example, $\frac{1}{2}$ could be represented in binary as 0.1, or as the infinitely repeating 0.01111.... However, there are ways to address this ambiguity.

[6] For example, the set of odd numbers would be represented by the bit string 010101... because the first natural number (0) is not in the set, the next one is, the next one is not, and so on.

12.5 Diagonalization

In 1903, Bertrand Russell stated what is now called *Russell's Paradox*: Consider the set $S = \{x \mid x \notin x\}$ of all sets that do not contain themselves. Is $S \in S$? If it is, it is not; if it is not, it is. This idea was similar in spirit to a proof technique Cantor developed several years earlier, when proving perhaps his most important result. Published late in his career in 1891, it is now known as *Cantor's Theorem*:

> *There is no onto function from a set to its power set.*

An onto function is one that covers all the values in its codomain. Formally, a function $f : X \to Y$ is called *onto* or *surjective* if

$$\forall y \in Y \; \exists x \in X \; : \; f(x) = y$$

Note that a function can be onto without being one-to-one (also known as *bijective*). For example, the absolute value function from integers to natural numbers is onto (every natural number is the absolute value of some integer) but not one-to-one (since two different things, an integer and its negative, both have the same absolute value).

In proving his theorem, Cantor introduced a method known as a *diagonal argument* or *diagonalization*, which has since become one of the standard proof techniques used by mathematicians.

Here's Cantor's proof:

> Assume the contrary, i.e. that there exists a set $\mathbb{S}$ and a surjection $f : \mathbb{S} \to 2^{\mathbb{S}}$.
>
> Define a set $\mathbb{D}$ consisting only of elements of S that are not in the image of the mapping:
>
> $$\mathbb{D} = \{x \in \mathbb{S} | x \notin f(x)\}$$
>
> Since $\mathbb{D} \subset \mathbb{S}$, then by definition of power set, $\mathbb{D} \in 2^{\mathbb{S}}$.
> Since f is surjective, there must be an element d in S such that $f(d) = \mathbb{D}$.
> Either $d \in \mathbb{D}$ or $d \notin \mathbb{D}$.
> If $d \in \mathbb{D}$, then by the definition of D, $d \notin \mathbb{D}$.
> If $d \notin \mathbb{D}$, then by the definition of D, $d \in \mathbb{D}$.
> Contradiction.

To see why this approach is called diagonalization, imagine that there's a way to write down all the subsets of a set S. Of course, S might be (countably) infinite, so our list might be infinite, but that's okay. The way we'll write each subset is with its characteristic function: a binary string with a 1 in position i if the ith element is in this subset, and a 0 otherwise. So our list might look like this:

SUBSET # \ ELEMENT #	1	2	3	4	5	...
1	1	1	1	1	1	...
2	1	0	1	0	1	...
3	0	1	1	0	1	
4	0	0	1	1	0	...
5	1	1	0	0	1	...
	⋮	⋮	⋮	⋮	⋮	

To construct our subset D, we want to identify a new subset d that's not already on the list. How do we do this? We take the first element of the first subset and negate it — if it's a 1, then d will have a 0 in that position; if it's 0, then d will have a 1. Then we do the same for the second element of the second subset, and so on, moving along the diagonal:

SUBSET #	1	2	3	4	5	...
1	**1**	1	1	1	1	...
2	1	**0**	1	0	1	...
3	0	1	**1**	0	1	
4	0	0	1	**1**	0	...
5	1	1	0	0	**1**	...
	⋮	⋮	⋮	⋮	⋮	
Elements in set d:	**0**	**1**	**0**	**0**	**0**	...

How do we know that d isn't already on the list? Well, we know it's not the same as the first entry on the list, because that entry contains the first element of S (i.e., has a 1 in the

first position), while d does not. We know it's not the same as the second entry on the list, because that entry has a 0 in the second position, while d has a a 1. Continuing down, we see that d can't be on the list because it *always* differs from every list entry in at least one position — the one on the diagonal.

In fact, we can use this diagonal argument to prove the uncountability of the continuum much more simply than Cantor did in his earlier 1874 proof. We just treat the bits as the binary representation of a number in the interval (0,1). However many entries we add to the table, we can always use the above diagonal construction to find a number d that is not on the list.[7]

12.6 The Continuum Hypothesis

We know that $\aleph_0$ is the smallest kind of infinity, and that $\mathfrak{C}$ is much bigger. But are there other levels of infinity in between? Is there an $\aleph_1$ that is bigger than $\aleph_0$ and smaller than $\mathfrak{C}$? Cantor believed that there was not; in other words:

$$\mathfrak{C} = 2^{\aleph_0} = \aleph_1$$

This claim, which Cantor stated in 1878, is called the *continuum hypothesis* (*CH* for short), and he spent a great deal of effort trying and failing to prove it.

Hilbert made the problem famous in 1900 when he listed it as the first of his 23 problems. Mathematicians worked on it for decades, ultimately showing that it cannot be disproved (1940) or disproved (1963) within standard set theory.[8]

Whether or not there are infinities in between $\aleph_0$ and $2^{\aleph_0}$, Cantor's Theorem tells us that there are bigger ones, each formed by taking the power set of elements of the previous one: $\aleph_0, 2^{\aleph_0}, 2^{2^{\aleph_0}}, \ldots$. We call the ith element of this sequence $\beth_i$ (after the Hebrew letter *beth*). Of course, this sequence is itself infinite, and we could take the union of these sets as i grows without bound:

$$\bigcup_{i=0}^{\infty} \beth_i$$

These are very strange (and very large) "unreachable" infinities. In a 1952 book called *Les Nombres Inaccessibles* ("Inaccessible Numbers"), Émile Borel pointed out that since sentences of a natural language are countably infinite, then only a tiny subset of all numbers can be referred to with natural language sentences. So most numbers are "inaccessible"

[7] In fact, we don't even need to represent the interval in binary. We could represent the numbers on the list as decimal numbers with an implicit decimal point before them; instead of negating the ith digit, we simply add 1 mod 10, or use some other function for getting a different value. All that matters is that we make sure the ith digit of the number we're constructing is different from the one used in the ith number in the table.

[8] "Standard set theory" here refers to what are known as ZF and ZFC, which will be explained in the next section.

in the sense that they can never be referred to — in fact, if you take a random real number in the interval [0,1], it will be inaccessible with probability 1! This raises an interesting philosophical question: In what sense do these numbers exist?

12.7 Axiomatizing Set Theory

In 1907, mathematician Ernst Zermelo proposed seven axioms to create a solid foundation for set theory, with no paradoxes. We'll look at each axiom and comment on it:

> I. *Axiom of extensionality.* If every element of set M is also an element of N and the other way around, then $M = N$.

It's important to define what it means for sets to be equal. It's hard, because sets are unordered collections. In programming, it's very hard to implement sets unless you assume some total ordering.

> II. *Axiom of elementary sets.* There is a set, the *empty set* $\varnothing$, that contains no element. If a is an object of the domain, there exists a set $\{a\}$, that contains a and only a as an element. If a and b are two objects of the domain, there always exists a set $\{a, b\}$ containing as elements a and b but no object x distinct from them both.

This is analogous to the way the Lisp programming language builds up everything from `nil` and the `cons` operation.

> III. *Axiom of separation.* Whenever the propositional function $E(x)$ is defined for all elements of a set M, M possesses a subset containing all the elements x of M for which $E(x)$ is true and no other elements.

This allows you to get a subset out of a set.

> IV. *Axiom of the power set.* To every set T there corresponds a set $\mathcal{P}(T)$, the power set of T, that contains all the subsets of T and no other elements.

For Zermelo, it's only a set if it can be built up from $\varnothing$ by repeated applications of power set: $2^{\varnothing} = \{\varnothing\}, 2^{2^{\varnothing}} = \{\varnothing, \{\varnothing\}\}$, and so on.

> V. *Axiom of the union.* To every set T there corresponds a set $\cup_T$, the union of T, that contains all the elements of the elements of T and no other elements.

This "flattens" one level of a nested set.

> VI. *Axiom of Choice (AC).* If T is a set whose elements all are sets that are different from $\varnothing$, and mutually disjoint, its union $\cup_T$ includes at least one subset C_T having one and only one element in common with each element of T.

This is a very peculiar axiom; it says you can pick an element. If you think of all the subsets as bags, C_T is like getting one element from every bag. It turns out that this has some very strange consequences, such as the *Banach-Tarski* paradox, which says that by using the axiom of choice, one can cut a sphere into a finite number of pieces that can be rearranged so that we end up with two spheres of the same size as the original sphere. Because of these problems, many mathematicians have wondered if it's possible to get rid of this axiom.

> VII. *Axiom of infinity.* There exists in the domain at least one set Z that contains the empty set as an element and is so constituted that to each of its elements a there corresponds a further element of the form $\{a\}$.

When talking about infinite sets, most people today use Dedekind's axiom that it is a set having a one-to-one correspondence with a subset. But Zermelo used a constructive method, defining infinite sets recursively. The use of recursion in the definition of data structures (again, common in Lisp) dates back to his idea.

Exercise 12.4

Prove the following theorem due to Zermelo: Every set M possesses at least one subset M_0 that is not an element of M.

In 1925, Israeli mathematician Abraham Fraenkel proposed two additional axioms:

> VIII. *Axiom of regularity.* Every non-empty set contains an element disjoint from it..

> IX. *Axiom of replacement.* An image of every set is a set.

These axioms help mathematicians deal with infinities of infinities, and we won't be discussing them further. What's important is that the resulting combined theory, known as *Zermelo-Fraenkel set theory with the axiom of choice (ZFC)* became the generally accepted approach to providing a foundation for mathematics.

[Further reading: A great book on set theory: *Foundations of Set Theory* by Fraenkel, Bar-Hillel, and Levy.]

Like non-Euclidean geometry, set theory progressed from being considered a crazy idea to being widely accepted. When Cantor invented set theory in the 1870s, he was

ridiculed and considered to be a threat to youth. Twenty years later, the top researchers in the world were working on set theory, and by the 1920s, it was generally accepted as the foundation of mathematics. What's interesting is that nothing changed about the theory. It's not that the problems were addressed or that the paradoxes became any less paradoxical. What changed was the tenor of the times; ideas that seemed radical in one century were perfectly acceptable in another. We see this pattern throughout the history of mathematics — and through history in general.

Chapter 13

From Axioms to Computability

In this chapter, we'll see how attempts to formalize mathematics led to some striking results about the abilities and limits of any computational device.

[More intro here.]

13.1 Hilbert's Program

In 1921, David Hilbert proposed that all mathematics be formalized, which meant:

- Every proposition is written in a formal language.
- The system must be *complete*, that is, every true proposition can be proved within the system. If a proposition is not true, then its negation is provable.
- The system must be *consistent*, meaning that it is not possible to prove both a proposition and its negation.
- Completeness and consistency should be proven with "finitary" methods, that is, without resorting to infinities.

This became known as *Hilbert's Program*, and it became a focus of work for many important mathematicians in the 1920s.

What Hilbert was essentially proposing was a system for automatically proving or disproving theorems. This system could be thought of as a (possibly abstract) computational machine, in which one could simply "turn the crank" and the system would spit out true propositions.

As time went on, there was substantial progress. For example, in 1929, Kurt Gödel proved that First-Order Predicate Calculus (FOPC),[1] a formal system that could be used

[1] FOPC contains predicates that take individual elements as arguments; for example, we might have a predicate $P(x)$ that is true when x is a person but false otherwise. "First-Order" means that we can have predicates about elements, but we can't have predicates about predicates.

to prove some mathematical propositions, was complete. By 1930, almost everyone in the mathematical world believed that Hilbert's Program would be completed at any moment.

13.2 The Program Collapses

> "It is all over."
>
> — John von Neumann, on hearing about Gödel's Theorem [cite]

Gödel began to focus his work on the system proposed by Alfred North Whitehead and Bertrand Russell in their three-volume work, *Principia Mathematica*[2] (PM). The book, whose final volume came out in 1913, was an earlier attempt to formalize mathematics. The system described in PM was more powerful than FOPC — powerful enough to satisfy Hilbert's first requirement. But was it complete?

Gödel realized that PM's use of induction, which FOPC lacks, makes all the difference. It makes the system more powerful, but it also introduces a weakness. In a 1931 paper "On Formally Undecidable Propositions in *Principia Mathematica* and Related Systems I," Gödel published his famous *incompleteness theorems*, which stated that:

1. Any consistent theory containing Peano arithmetic contains a proposition that is true, but not provable.

2. Any consistent theory containing Peano arithmetic cannot prove its own consistency.

Essentially, what these two results said is that Hilbert's program was fundamentally impossible. Any system strong enough to express the kind of mathematical propositions needed was necessarily *incomplete*.

Gödel's proof is quite complex, but the basic idea is this: The axioms and inference rules in PM are designed to prove propositions about integers. For example, it can express, in formal symbols, an idea like "5 is prime" or "there are infinitely many even numbers." Since the system is purely mechanical, we should be able to turn the crank and have propositions like this come out. So Gödel came up with a clever way of assigning a unique integer to every possible proposition, a technique now known as *Gödel-numbering* or *Gödelization*. Since propositions in the system are about numbers, and propositions could now themselves be expressed as numbers, it was possible to have propositions that "talk about" the truth and falsity of other propositions. Gödel then showed how to express a special proposition in the language of the system, a proposition which essentially says:

[2] Although Whitehead and Russell were English, they gave their book a Latin title, meaning "Mathematical Principles," probably named after Isaac Newton's similarly named book from 200 years earlier. Confusingly, Russell also wrote an entirely different book called *Principles of Mathematics*.

This proposition is not provable.

If this proposition could be generated by turning the crank — that is, proved within the system — then the statement is false, so the system has proved a false proposition, and is therefore inconsistent. If it can't be generated by the system, then it is a true statement that can't be proven, and the system is incomplete.

It is difficult to convey today how shocking Gödel's result was. Prior to 1931, there was a general sense that all mathematical truths, and perhaps eventually all scientific truths, would ultimately be derivable by a mechanistic process, and that researchers were on the verge of doing so. After Gödel's Theorem, these hopes were dashed. As the quote from von Neumann at the beginning of this section suggests, there was a feeling that the foundations of their intellectual enterprise were crumbling, and nothing was certain anymore.

Yet decades later, we are more comfortable with the idea of these limits. Mathematics existed before anyone tried to formalize its foundations, and continued to exist afterwards. No bridges collapsed because of Gödel's result.

Interestingly, the idea of Gödel-numbering, which was so novel in 1931, is quite familiar to computer scientists today: We know that a string of bits can represent a number or an instruction in machine language, and the instruction can act on other numbers or even on itself.

Kurt Gödel (1906–1978)

[Biography of Gödel goes here. Points to include:

- Kurt Gödel (pronounced almost like the English word "girdle," but without finishing the "r" sound) was born in Brno, in what was then the Austro-Hungarian empire; always considered himself Austrian and later moved to Vienna.
- In addition to incompleteness results, worked on recursive function theory (now part of the theory of computation).
- Came up with interesting proof about the length of proofs – just because something is simple to state doesn't mean it's simple to prove.
- He escaped from the Nazis via the Trans-Siberian railway and through Alaska.
- He worked at the Institute for Advanced Study and became good friends with Einstein.
- He became increasingly eccentric and spent the last 10 years of his life working on a formal proof of God's existence.

- He became paranoid, and refused to eat any food not cooked by his wife. When she had a long hospitalization, he died of starvation.

]

13.3 Church, Turing, and the Origins of Computer Science

Is there a mechanical procedure for deciding whether a mathematical statement is true or false? This question, called the "decision problem" (*entscheidungsproblem* in German), is closely related to Hilbert's program, and it was the focus of many researchers in the 1930s.

One of these was Alonzo Church. Church came up with a system for defining computational procedures, which he called *λ-calculus*. And he formulated an important hypothesis, known as *Church's Thesis*, about the nature of computation:

> *Whatever can be computed by a modern computer (or by a Turing machine) includes everything that ever can be computed.*

Church didn't actually state it in this way; early versions of the idea referred to the kinds of computational mechanisms that had been proposed at the time, like λ-calculus. But the basic idea is the same: once you have a basic mechanism for computation, you can (theoretically) compute everything computable. Every computational device is in some sense the same; there isn't some special kind of über-computer that can compute "harder" functions.[3]

Of course, Church's Thesis isn't provable, because we'd have to quantify over all possible future computational systems that don't exist yet. Nevertheless, it is generally accepted, and is in a sense the basis of Computer Science: We know that if we design an algorithm, we can implement it on any general-purpose computer.

Alonzo Church (1903–1995)

[Biography of Church goes here. Some highlights:

Alonzo Church grew up in Virginia. He realized that the kind of problems he and others were working on were really a new area of mathematics, which is called *mathematical logic* or *symbolic logic*. While teaching at Princeton, Church did more than anyone else to establish and promote this new field. First, he published *A Bibliography of Symbolic Logic*, identifying all of the most important writings. Next, he founded the *Journal of Symbolic*

[3] We're assuming that we have as much memory as we need and we don't care how long it takes.

Logic, which still exists today. Later, he wrote *Introduction to Mathematical Logic*, which was for many years the definitive textbook in the field. Church's own work was highly influential, but even more than this, his legacy is in his students, which include many of the pioneers of the new field of computer science. These include multiple Turing Award winners, as well as Alan Turing himself. Church continued teaching until age 87, and was continuing to publish papers up to the year of his death at 92.]

Independently of Church, British mathematician Alan Turing developed his own approach to the *entscheidungsproblem*. Turing proposed an abstract universal machine — what we now call a *Turing machine* and proved that given an arbitrary program, there was no way to decide whether the machine would halt when running the program. Since this *halting problem* was undecidable, the could be no solution to Hilbert's more general decision problem.

Turing later studied with Church, and their early work overlapped a great deal. Today we refer to the devices in Church's Thesis as "Turing-complete" — that is, computational systems equivalent in power to a Turing machine. (In fact, Church's Thesis is often called the *Church-Turing Thesis* due to Turing's contributions.) But it was Turing's idea of the universal machine, more than anything else, that greatly influenced the design of the modern computer.

Alan Turing (1912–1954)

[Biography of Turing goes here. Some highlights:

Attended King's College at Cambridge. After work on Turing machine, etc., worked on cryptography in WWII and built machines that broke the Enigma code used by the Nazis. After the war, wrote a paper proposing the architecture of a practical computer, the Pilot ACE. Prototype gets built at Manchester. Later writes one of the seminal papers in AI, proposing the Turing Test. Also worked in biology (morphogenesis). Prosecuted for homosexual acts and died of cyanide poisoning at age 43. At the time it was widely believed that he committed suicide, but people are no longer sure.]

13.4 Diagonalization Revisited

As discussed in Section 12.5, Cantor's diagonal argument became the basis of a standard proof technique in mathematical logic and computer science. The argument relies on self-reference (specifically, in the case of Cantor's Theorem, in the definition of the set D as $\{x \in S | x \notin f(x)\}$).

Self-referential propositions have a long history, dating back at least 600 BC when the Cretan philosopher Epimenides said "All Cretans are liars."[4] Medieval philosophers wrestled with these so-called *insolubilia* ("unsolvables").

In Journey 2, we saw how we could start with a specific algorithm — greatest common measure of two line segments — and, through repeated abstraction, apply it to more and more general domains. Could we do the same thing for diagonalization? What would a generic form of Cantor's proof look like?

To do this, we'll need to start by defining a particular set of functions:

Definition 13.1. *A set* $\mathbb{F}$ *of (possibly partial) functions of one (*$\mathbb{F}_1$*) and two (*$\mathbb{F}_2$*) variables from domain* T *to codomain* T' *is called a* unary-binary family.

(Here, "unary" and "binary" refer to the *arity* of the function, i.e. how many arguments it has, not to unary or binary numbers.) Some examples of unary-binary families are:

- total functions from $\mathbb{N}$ to $\mathbb{N}$ and from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$
- continuous functions from $\mathbb{R}$ to $\mathbb{R}$ and from $\mathbb{R} \times \mathbb{R}$ to $\mathbb{R}$
- computable functions from $\mathbb{N}$ to $\mathbb{N}$ and from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$

(Functions with more than two arguments may also be included in the family, but not needed for any of our results.)

Definition 13.2. *In a unary-binary family* $\mathbb{F}$ *with the domain* T *a binary function* $\mathfrak{I} \in \mathbb{F}_2$ *is called an* interpreter *if*

$$\forall f \in \mathbb{F}_1 \ \exists c \in T \ : \ \forall x \in T \ \mathfrak{I}(c, x) = f(x)$$

We can think of c as representing code, x representing the input data we're going to run the code on, and $\mathfrak{I}$ (the letter I in gothic typeface) as the interpreter. We normally think of an interpreter as something that takes some code and some data and executes the code on the data. However, here we are generalizing the notion of code (and the notion of interpreter); the code here is simply a way to indicate to the interpreter which function to compute. The interpreter can use any method it chooses to compute the function — executing instructions, using a lookup table, or even asking a genie. The code and data have to be the same type, but that's not a very stringent restriction — they could be bits, s-expressions in a Lisp-like language, and so on.

The programming language Lisp includes an `eval` function that evaluates on expression, applying the first element of the list to the arguments constituting the rest of the

[4] Actually, Epimenides wasn't trying to state a paradox; in the original context, he was criticizing public opinion, and may not have realized the irony of his statement. In any case, his statement eventually came to be a canonical example of a liar paradox.

list. We can define our own two-argument version[5] of this that meets our criteria for an interpreter:

```
(defun eval1 (x y)
       (eval (cons x y)))
```

But our definition of interpreters doesn't require anything as powerful as this. Here's an example of a very simple interpreter: Consider a unary-binary family L containing unary functions $f_k(x) = x + k$ and binary functions $w_k(x, y) = x + y + k$. Then, $w_0(x, y) = x + y$ is an interpreter in L, since $w(k, x) = f_k(x)$.

Now we introduce a strange idea: an *anti*-interpreter. If an interpreter computes whatever its given code is supposed to do on its given data, an anti-interpreter computes *anything but* that. Let's define it formally:

Definition 13.3. *In a unary-binary family $\mathbb{F}$ with the domain T, a function $\mathfrak{A} \in \mathbb{F}_2$ is called an* anti-interpreter *if*

$$\forall f \in \mathbb{F}_1 \ \exists c \in T \ : \ \forall x \in T \ \mathfrak{A}(c, x) \neq f(x)$$

($\mathfrak{A}$ is the gothic letter A, for anti-interpreter.)

Let's construct an anti-interpreter: As before, L is a unary-binary family containing unary functions $f_k(x) = x + k$ and binary functions $w_k(x, y) = x + y + k$. Then, $w_1(x, y) = x + y + 1$ is an anti-interpreter in L, since $w_1(k, x) = x + k + 1 = f_k(x) + 1 \neq f_k(x)$.

What we're saying here is that if our code was supposed to add k to its argument, we'll add $k + 1$, and therefore by definition we will be computing something other than what the code was supposed to compute.

Could we make an analogous anti-interpreter for Lisp based on `eval`? To answer the question, we need one more definition:

Definition 13.4. *A unary-binary family $\mathbb{D}$ from T to T' is called a* diagonalizable family *if*

$$\forall g \in \mathbb{D}_2 \ \exists f \in \mathbb{D}_1 \ : \ \forall x \in T \ g(x, x) = f(x)$$

In other words, a family of functions is diagonalizable if you can give the same value to both arguments and still end up with a function in the family. For example, Lisp functions are diagonalizable, because for any binary function `foo` we can define a function `bar`:

```
(defun bar (x) (foo x x))
```

This brings us to the *Anti-Interpreter Theorem*:

A diagonalizable family does not have an anti-interpeter.

[5] Lisp has a similar two-argument function called `apply`, but its first argument is a function, not an s-expression.

Proof: Let us assume that we have a diagonalizable family that *does* have an anti-interpreter $\mathfrak{A}$. Since our family is diagonalizable, we know that there's a function

$$\mathfrak{a}(x) = \mathfrak{A}(x, x)$$

in the family. Since $\mathfrak{A}$ is an anti-interpreter, by definition it gives "the wrong answer" for every function, and in particular, for our particular function $\mathfrak{a}(x)$:

$$\exists c \in T \ : \ \forall x \ \ \mathfrak{a}(x) \neq \mathfrak{A}(c, x)$$

But if we use the code c as our argument x, we have:

$$\mathfrak{a}(c) = \mathfrak{A}(c, c) \ \wedge \ \mathfrak{a}(c) \neq \mathfrak{A}(c, c)$$

Contradiction.

What about our simple family L? We've already shown that L has both an interpreter and an anti-interpreter. It is not diagonalizable because a function $w_k(x, x) = x + x + k = 2x + k$ is not in the family.

To come up with a general rule about when we have an interpreter, we need one more definition, a *composable-diagonalizable (C-D)* family:

Definition 13.5. *A diagonalizable family C with domain T and co-domain T' is* composable *if:*

$$\exists(\mathfrak{f} : T' \to T') \, \forall(w \in C) \ : \ \ \mathfrak{f}(x) \neq x \ \ \wedge \ \ \mathfrak{f}(w(x, y)) \in C$$

$\mathfrak{f}$ is called the *non-fixed-point function* of the family since it always returns something different from its argument. The definition says that the family is C-D if there is at least one non-fixed point function in the family that, when composed with any binary function in the family, is still in the family. It's a generalization of the "adding 1" idea we used above to make an anti-interpreter in our example family L.

Examples of C-D families include:

- Total computable functions over integers
- Polynomial time functions over integers
- Continuous functions over real numbers
- Differentiable functions over real numbers

In other words, they are exactly the kinds of functions we deal with every day.

Non-Existence of Interpreter Theorem: A C-D family does not contain an interpreter.
Proof: Let us assume that an interpreter $\mathfrak{I}$ exists. Then we can compose it with our function $\mathfrak{f}$. But the result will always give a different answer than $\mathfrak{I}$, so

$$\mathfrak{A}(x, y) = \mathfrak{f}(\mathfrak{I}(x, y))$$

is an anti-interpreter. But we have already proven that a diagonalizable family can't have an anti-interpreter, because this will lead to contradiction.

Recall that the universal Turing machine is a Turing machine that is an interpreter for Turing machines. If we accept Church's thesis, such an interpreter exists in any equivalent formulation:

Interpreter Thesis: Any non-trivial set of computable functions that contains an interpreter is Turing-complete.

Using the interpreter idea, we can easily prove Turing's halting problem:

> *There is no computable function* halt(c, x) *that returns true if code c terminates on input x and false otherwise.*

Proof: If such a function existed, we would be able use it to construct an anti-interpreter:

```
integer anti_interpreter(integer c,  integer x) {
  if (halt(c, x)) {
    return interpreter(c, x) + 1;
  } else {
    return 0;
  }
}
```

But we have proven that an anti-interpreter cannot exist. So the function `halt` cannot exist.

Chapter 14

Theories and Concepts

In this chapter, we'll explore how some of the mathematical ideas discussed in this journey appear in modern computer programming. In particular, we'll see how the notion of *successor*, which was a core part of Peano's basis for arithmetic, plays a role in the concept of *iterators* and searching in data structures.

14.1 Aristotle's Organization of Knowledge

> All humans naturally desire to know.
>
> – Aristotle, *Metaphysics I,1*

The School of Athens, a famous painting by Italian Renaissance painter Raphael, depicts many ancient Greek philosophers. At the center are Plato and Aristotle, the two most important philosophers of the ancient world. Plato, whom we met in Journey 2, is pointing upward, while his student Aristotle holds his hand out over the ground. According to popular interpretation, Plato is pointing to the heavens, indicating that we should contemplate the eternal, while Aristotle is indicating that we need to study the world. In fact, it could be said that Plato invented mathematics and Aristotle invented the study of everything else, especially science. Aristotle's works cover everything from Ethics to Zoology.

Aristotle (384 BC – 322 BC)

[Biography of Aristotle goes here. Some points to include:

- Came from Stageira, which was the distant frontier of Greece at the time. Philip of Macedon burned the town and sold the residents into slavery.

- Aristotle went to Athens in search of wisdom and studied at the Academy for 20 years until Plato's death. He then left Athens, perhaps disappointed that he was not chosen as Plato's successor.
- He was appointed to train the son of Philip of Macedon, who becomes Alexander the Great.
- Then he moved back to Athens and founded a competing school, the Lyceum, which teaches all subjects, from gymnastics to poetry.
- Aristotle bases his work on actual observation, not just thinking. For example, when wrote about political science, he didn't (like Plato) make up his idea of an ideal state. Instead, he had his students go to different city-states and report back on how their government was constituted.
- Although Aristotle wrote many dialogues, none survive; the writings we have were probably intended as lecture notes, possibly written by his students.

]

Artistotle's writings were preserved toward the end of the first millennium A.D. by Arab philosophers. In the 12th century, when Christian kingdoms recaptured Spain from the Islamic state known as Al-Andalus, they found a library in the Spanish city of Toledo containing a great number of books in Arabic, including translations of Aristotle's works. As the books were translated into Latin, a kind of Aristotelian renaissance spread through Europe. Aristotle became known as simply "the Philosopher," and his works became a part of generally accepted knowledge. The great Andalusian philosopher Ibn Rushd (known as Averroes to Europeans) wrote widely-read commentaries on Aristotle, reconciling his philosophy with the teachings of Islam; he became known simply as "the Commentator." In the 13th century, Christian scholars such as Thomas Aquinas similarly showed that Aristotelianism was consistent with Christianity; this was often described as "baptizing" Aristotle. As a result, Aristotle's works were part of the required study at European universities for literally hundreds of years.

Among Aristotle's most important works is the *Organon*, a collection of six treatises on various aspects of logic which would define the field for the next 2600 years.[1] In *Categories*, the first part of the *Organon*, Aristotle introduced the notion of *abstraction*. He wrote about the distinction between an *individual*, a *species*, and a *genus*. While today most people think of these as biological distinctions, for Aristotle they applied to everything. A species incorporates all the "essential" properties of a type of thing. A genus may contain

[1] Unlike his other works, a Latin version of the *Organon* was available to Europeans much earlier; Boethius provided the translation in the early 6th century.

many species, each identified by its *differentia* — the things that differentiate it from other species in the genus.

It is Aristotle's idea of genus that inspired the term *generic programming* — a way to think about programming that focuses on the level of *genera* (the plural of genus) rather than species.

14.2 Theories and Models

People use the word "theory" in many ways, often meaning something like "conjecture" — that is, an unproven explanation. But to mathematicians, "theory" has a very specific meaning, which does not include this sense of being unproven.

Definition 14.1. *A* theory *is a set of true propositions.*

From now on, when we use the word "theory," we'll mean this specific mathematical sense. Here are some important facts about theories:

- A theory can be generated by a set of axioms plus a set of inference rules.
- A theory is *finitely-axiomatizable* if it can be generated from a finite set of axioms.
- A set of axioms is *independent* if removing one will decrease the set of true propositions.
- A theory is *complete* is for any proposition, either it or its negation is in the theory.
- A theory is *consistent* if for no proposition it contains it and its negation.

Let's look at specific example. Recall the notion of *groups* introduced in Journey 2; this is a theory:

$$
\begin{aligned}
\text{operations}: &\quad x \circ y,\ \ x^{-1} \\
\text{constant}: &\quad e\ (\text{identity}) \\
\text{axioms}: & \\
& x \circ (y \circ z) = (x \circ y) \circ z \\
& x \circ e = e \circ x = x \\
& x \circ x^{-1} = x^{-1} \circ x = e \\
\text{theorems}: & \\
& \forall y\, \forall x\ :\ x \circ y = x \implies y = e \\
& (x \circ y)^{-1} = y^{-1} \circ x^{-1} \\
& \ldots
\end{aligned}
$$

Note that we aren't enumerating all the true propositions (theorems) that constitute the theory of groups. Rather, we're generating the propositions by deriving them from the axioms and from previously proven propositions. For example, we can prove the first theorem by multiplying both sides of the equation by x^{-1}. Like basis vectors in linear algebra, axioms form a basis for the theory. Also like the linear algebra case, we can have more than one basis for the same theory.

Closely associated with the notion of a theory is that of a *model*. Again, the mathematical meaning is quite different from the everyday meaning:

Definition 14.2. *A set of elements that satisfies all the propositions in a theory is called its* model.

In a sense, a model is a particular *implementation* of a theory. Just as there can be multiple implementations of, say, an algorithm, there can be multiple models of a theory.

The more[2] propositions there are in a theory, the fewer different models there are; more propositions also implies more axioms. This makes sense intuitively: axioms and propositions are *constraints* on a theory; the more of them you have, the harder it is to satisfy all of them, so the fewer elements there will be that do so.

Conversely, the more models there are for a theory, the fewer propositions there are. If there are more ways to do something, there must be fewer constraints on how you can do it.

A (consistent) theory is called *categorical* or *univalent* if all of its models are isomorphic.[3] We could almost define categorical theories as those that have only one model, but it turns out that there are often multiple models that are effectively the same.

An inconsistent theory has no model — there's no way to satisfy all the propositions without a contradiction.

Categorical Theories vs. STL

For a long time, people believed that only categorical theories were good for programming. When the C++ Standard Template Library (STL) was first proposed, many computer scientists opposed it on the grounds that many of its fundamental concepts, such as iterator, were underspecified. In fact, it is this underspecification that gives the library its generality. If you can get by with fewer axioms, you allow for a wider range of implementations.

[2] The notion of "more" that we're interested in here is that of *additional*. If the set of propositions for theory A contains all the propositions of theory B, plus some additional ones, then we say A has more, even if both have a countably infinite number of propositions.

[3] This is the original definition of Oswald Veblen. Modern logicians talk about κ-categorical theories: all models of the cardinality κ are isomorphic.

Let's look at an example of a non-categorical theory. While there is only one non-isomorphic group of order 1, 2, or 3, there are two non-isometric groups of order 4, the cyclic group $\mathbb{Z}_4$ (the additive group of remainders mod 4) and something called the Klein group. These have the following multiplication tables:

	e	**a**	**b**	**c**
e	e	a	b	c
a	a	b	c	e
b	b	c	e	a
c	c	e	a	b

Cyclic group Z_4

	e	**a**	**b**	**c**
e	e	a	b	c
a	a	e	c	b
b	b	c	e	a
c	c	b	a	e

Klein group

In the table on the left, addition is our operation and we can think of "e" as being 0 (the additive identity) and a, b, and c representing the integers 1, 2, and 3. So, for example, $a \circ b = 1 + 2 = 3 = c$, therefore the value at row a and column b is c.

Are these two groups are really different (i.e. not isomorphic), or is there a way to rearrange the rows and corresponding columns to get the same table? In other words, is there a distinguishing proposition for groups of order 4? Yes, the proposition:

$$\forall x \in G \ : \ x^2 = e$$

is true for the Klein group but false for $\mathbb{Z}_4$. We can see this by looking at the diagonal of the multiplication table.

There are actually different models of $\mathbb{Z}_4$: the additive group of remainders modulo 4 (consisting of {0, **1**, 2, **3**}, which we used above) and the multiplicative group of non-zero remainders modulo 5 (consisting of {1, **2**, **3**, 4}). But these two models are isomorphic; we could map elements of one to elements of the other. In the first model, the values 1 and 3 are *generators* of the group — we could start with either of them, raise it to a power using the group operation, and get all the other elements. In the second model, 2 and 3 are generators. This gives us a hint about how we could do the mapping: a generator from one group has to map to a generator from the other group, which in this case gives us two different mappings.

Similarly, there are multiple models of the Klein group: the multiplicative group of units modulo 8 (consisting of {1, 3, 5, 7}) and the group of isometries transforming a rectangle into itself (the identity transform, vertical symmetry, horizontal symmetry, and 180° rotation).

14.3 Values and Types

Now we will see how some of the mathematical ideas we've been discussing fit in to computer programming. First, we need a few definitions:

Definition 14.3. *A* datum *is a sequence of bits.*

101 is an example of a datum.

Definition 14.4. *A* value *is a datum together with its interpretation.*

A datum without an interpretation has no meaning. The datum 101 might have the interpretation of the integer 5, or the integer -3 (if we were using a 3-bit 2's complement representation). Every value must be associated with a datum in memory; there is no way to refer to disembodied values in a language like C++ or Java.

Definition 14.5. *A* value-type *is a set of values sharing a common interpretation.*

Definition 14.6. *An* object *is a collection of bits in memory that contain a value of a given value-type.*

Note that there is nothing in the definition that says that all the bits of an object must be contiguous. In fact, it's quite common for parts of an object to be located at different places in memory.

An object is *immutable* if the value never changes, and *mutable* otherwise. An object is *unrestricted* if it can contain any value of its value type.

Definition 14.7. *An* object type *is a uniform method of storing and retrieving values of a given value type from a particular object when given its* address.

What we call "types" in programming languages are object types. C++, Java, and other programming languages do not provide mechanisms for defining value-type.[4] Every type resides in memory and is an object type.

14.4 Concepts

The essence of generic programming lies in the idea of *concepts*. A concept is a way of describing a family of related object types. The relationship between concept and type is exactly the relationship between theory and model, and between genus and species. Here are some examples of concepts and some of their types:

- Integral: `int8_t`, `uint8_t`, `int16_t`, ...

[4] The iterator trait `value_type` in C++ actually returns the object type of the value returned by dereferencing the iterator.

- UnsignedIntegral: `uint8_t`, `uint16_t`, ...
- SignedIntegral: `int8_t`, `int16_t`, ...

The idea of concepts exists in many languages, but very few languages provide an explicit way to talk about concepts. [5]

Many programming languages provide a mechanism to specify the interface of a type to be implemented later: abstract classes in C++, interfaces in Java, and so on. However, these mechanism completely specify the interface, including strict requirements on the types of arguments and return values. In contrast, concepts allow interfaces to be specified in terms of families of related types. For example, in both Java and C++, you can specify an interface containing a function `size()` returning a value of type `int32`. In the world of concepts, you can have an interface with a function `size()` that returns a value of an integral type — `int16`, `int32`, or `int64`.

A concept can viewed as a predicate on types, assuring that they meet a set of requirements concerning the *operations* they provide, their *semantics*, and their *time/space complexity*. A type is said to *satisfy* a concept if it meets these requirements.

When first encountering concepts, programmers often wonder why the third requirement, for space/time complexity, is included. Isn't complexity just an implementation detail? To answer this question, consider a real-world example. Suppose you defined the abstract data type *stack*, but you implemented it as an array, where every time you pushed something onto the array, you had to move every existing element to the next position in order to make room. Instead of pushing on the stack being a fast operation (O(1)), it's now a slow operation (O(n)). This violates a programmer's assumption of how stacks should behave. In a sense, a stack that doesn't have fast push and pop is *not really a stack*. Therefore these very basic complexity constraints are part of what it means to satisfy a concept.

Two very useful ideas are *type functions* and *type attributes*. A type function is a function that, given a type, returns an affiliated type. For example, it would be nice to have type functions like this:

- `value_type(Sequence)`
- `coefficient_type(Polynomial)`
- `ith_element_type(Tuple, size_t)`

Unfortunately, mainstream programming languages do not contain type functions, even though they would be easy to implement. (After all, the compiler already knows things like the type of elements in a sequence.)

A type attribute is a function that, given a type, returns one of its numerical attributes, for example:

[5] There have been proposals to include concepts in C++; the work is still in progress. There are also concept-like features in some functional programming languages such as Haskell.

- `sizeof`
- `alignment_of`
- number of members in a struct

Some languages provide some type attributes, like `sizeof()` in C and C++.

Let's look at some very general concepts. The first one is Regular.[6] A type is regular if it supports these operations:

- copy construction
- assignment
- equality
- destruction

Having a copy constructor implies having a default constructor, since `T a(b)` is equivalent to `T a; a = b;`. To describe the semantics of Regular, we'll express the requirements as axioms:

$$\begin{aligned} \forall a\, \forall b\, \forall c &: T\ a(b) \implies (b = c \implies a = c) \\ \forall a\, \forall b\, \forall c &: a \leftarrow b \implies (b = c \implies a = c) \\ \forall f \in \text{RegularFunction} &: a = b \implies f(a) = f(b) \end{aligned}$$

For example, the second axiom says that if you assign b to a, then anything that was equal to b will now also be equal to a. Equality is a tricky concept, and we need to invoke the notion of a *regular function* (not to be confused with a regular type), which is one that produces equal results given equal inputs.

The complexity requirements on Regular are that each operation is linear in the area of the object, where area includes all space occupied by the object, include its remote parts. [7]

Note that the concept Regular is universal — it's not specific to any programming language. A type in any language is regular if it satisfies the requirements.

A related concept is Semiregular, which is just like Regular except that equality is not explicitly defined. This is needed in a few situations where it is very difficult to implement an equality predicate. Even in these situations, equality is assumed to be implicitly defined, so that axioms that control copying and assignment are still valid. After all, as we saw above, the meaning of assigning a to b is that b's value will be *equal* to $a's$ value afterward.

[6] By convention, we write concept names with initial capitals, and display them with a Sans Serif typeface.
[7] To see a more formal treatment of the concept Regular, see *Elements of Programming* Sec. 1.5.

14.5 Three Fundamental Programming Tasks

Programmers often spend time learning arcane data structures and algorithms that handle specialized situations they might encounter. While these can be useful in certain contexts, many programmers would actually benefit from a deeper understanding of some of the very basic tasks that occur over and over in a variety of situations. Perhaps the three most fundamental problems in programming are *swap*, *minimum* (and its analogue, *maximum*), and *linear search*. They are much harder than is generally appreciated. If you can understand these, then you can reason about programming in general. We'll discuss swap and minimum here, and save linear search for the next chapter.

The *swap* operation requires only that the types of its arguments satisfy the concept Semiregular:

```
template <Semiregular T>
void swap(T& x, T& y) {
  T tmp(x);
  x = y;
  y = tmp;
}
```

We can see that `swap` requires the ability to copy-construct, assign, and destruct. It does not need to explicitly test for equality, so we do not need the types to be Regular. When we design an algorithm, we'll want to know what concepts the types need to satisfy, but we'll also want to make sure not to place extra requirements we don't need.

The *minimum* and *maximum* functions rely on the notion of *ordering*.

[A bit more about min and max.]

Chapter 15

Iterators and Search

In this chapter we will consider the third fundamental programming task, linear search, and its cousin, binary search. Linear search is based on the notion of successor, which is at the heart of the concept of *iterators*.

15.1 Iterators

An an *iterator* is a concept used to express where we are in a sequence. In fact, iterators were originally going to be called "coordinates" or "positions"; they may be viewed as a generalization of pointers.

Iterators require three operations:

- regular type operations (including equality)
- successor
- dereferencing

The essence of an iterator is the notion of *successor*.

Indeed, iterators come to us directly from Peano's axioms; essentially, the concept Iterator is "a theory with successor." However, our iterator concepts will be weaker, because we don't require all of Peano's axioms. For example, in Peano arithmetic, every number has a successor, while with iterators, this is not always the case — sometimes we get to the end of our data. Peano also tells us that if successors are equal, the predecessors must be equal, and that we can't have loops. These requirements are also not the case for programmers; we're allowed to have data structures that link back to earlier elements, and form loops; sometimes this is exactly what we need.

In short, an iterator is "something that lets you do linear search in linear time."

The second iterator operation, *dereferencing*, is a way to get from a pointer to (in a sense, the name of) a value to the value itself. Dereferencing has a time complexity requirement; it's assumed to be "fast," which means that there is not a faster way of getting to data than through the iterator. Iterators are sometimes bigger than traditional pointers, in situations where they store some additional state to enable fast navigation. Iterators may also support special values indicating that we are past the end of the object, as well as singular values like the null pointer that cannot be dereferenced. It's okay that dereferencing is a partial function (i.e. that it isn't defined for all values) — after all, mathematicians have no trouble saying what division is, even though it's undefined for zero.

Dereferencing and successor are closely connected,[1] and this relationship imposes the following restrictions:

- Dereferencing is defined on an iterator if and only if successor is defined.
- There are no non-empty ranges containing no values.
- If you are not at the end of the range, you can dereference.

Why do we need equality as a requirement for iterators? Because we need to be able to see when one iterator reaches another — we don't always want to iterate to the end of our data.

15.2 Iterator Categories, Operations, and Traits

There are several kinds of iterators, which we call *iterator categories*. Here are the most important:

- *Input Iterators* support one-directional traversal, but only once, as is found in single-pass algorithms. The canonical model of an input iterator is an *input stream*. Bytes are coming over the wire and we can process them one at a time, but once they are processed, they are gone. However, all one-pass algorithms that use input iterators are not limited to input streams, and work perfectly well on multi-pass data structures such as linked lists; merge of two ordered sequences is one such algorithm.
- *Forward Iterators* also support only one-directional traversal, but this traversal can be repeated as needed, as in multi-pass algorithms. The canonical model of a forward iterator is the *singly-linked list*.[2]
- *Bidirectional Iterators* support bidirectional traversal, repeated as needed (i.e. they also can be used in multi-pass algorithms). The canonical model of a bidirectional

[1] It is possible to separate the operations explicitly, as is done in EoP.

[2] Note that we assume that link structure of the list is not modified as it is traversed.

iterator is a *doubly-linked list*. Bidirectional iterators have an invertible successor function: if an element x has a successor y, then y has a predecessor.

- *Random-Access Iterators* support random-access algorithms, that is, they allow access to any element in constant time (both *far* and *fast*). The canonical model is an *array*.

In addition, there is another common iterator category which behaves slightly differently from the others:

- *Output Iterators* support alternating successor (`++`) and dereference ($*$) operations, but dereferencing can only be used as an l-value, and they provide no equality function. The canonical model of an output iterator is an *output stream*. We can't define equality because we can't even get to the elements.

While the iterators described above are the only ones included in C++, there are many other useful iterator concepts as well, for example:

- *Linked Iterators* work in situations where the successor function is mutable (for example, a linked list where the link structure is modified[3]).

- *Segmented Iterators* are for cases where the data is stored in noncontiguous *segments*, each containing contiguous sequences. The data structure `std::deque`, which is implemented as a segmented array, would immediately benefit; instead of needing each successor operation to check whether the end of the segment has been reached, a "top level" iterator could find the next segment and know its bounds, while the "bottom level" iterator could iterate through that segment.

A simple but important thing we may want to do is find the distance between two iterators. For an input iterator, we might write our `distance()` function like this:

```
template <InputIterator I>
DifferenceType(I) distance(I f, I l, std::input_iterator_tag) {
  // precondition: valid_range(f, l)
  DifferenceType(I) n(0);
  while (f != l) {
    ++f;
    ++n;
  }
return n;
}
```

[3]For example, by a function like `rplacd` in Lisp or `set-cdr` in Scheme.

There are three notable things about this code: the use of the type function `DifferenceType()`, the use of the iterator tag argument, and the precondition. We'll discuss all of these below, but before we do, let's compare this to a different implementation, one that's optimized for random access iterators:

```
template <RandomAccessIterator I>
DifferenceType(I) distance(I f, I l, std::random_access_iterator_tag) {
    // precondition: valid_range(f, l)
    return l - f;
}
```

Since we have random access, we don't have to repeatedly increment (and count) from one iterator to the other; we can just use a constant time operation — subtraction — to find the distance.

The *difference type* of an iterator is an integral type that is large enough to encode the largest possible range. For example, if our iterators were pointers, the difference type could be `ptrdiff_t`. But in general we don't know in advance which type the iterator will be, so we need a type function to get the difference type. Although C++ does not have a general mechanism for type functions, STL iterators have a special set of attributes known as *iterator traits*:

- `value_type`
- `reference`
- `pointer`
- `difference_type`
- `iterator_category`

The `reference` and `pointer` traits are rarely used in current architectures[4], but the others are very important.

Since the syntax for accessing iterator traits is rather verbose, we'll implement our own macro:

```
#define DifferenceType(X) typename std::iterator_traits<X>::difference_type
```

This gives us the `DifferenceType()` function used in the code above.

The iterator trait `iterator_category` tells us what sort of iterator we're dealing with — forward iterator, bidirectional iterator, and so on. It's an object with no data, only a type.

[4]Earlier versions of the Intel processor architecture included different types for shorter and longer pointers, so it was important to know which to use for a given iterator. Today, if the value type of an iterator is `T`, the `pointer` iterator trait would simply be `T*`.

Now we can return to the use of the iterator tag argument in the `distance` functions above. The iterator tags shown in the examples (`std::input_iterator_tag` and `std::random_access_iterator_tag`) are possible values of the iterator category trait, so by including them as arguments, we are distinguishing the type signature of the two function implementations. This allows us to perform *category dispatch* on the `distance` function: We can write a general form of the function for any iterator category, and the fastest one will be invoked:

```
template <InputIterator I>
inline
DifferenceType(I) distance(I f, I l) {
  return distance(f, l, (std::iterator_traits<I>::iterator_category)());
}
```

Note that the third argument is actually a constructor call creating an instance of the appropriate type, because we cannot pass types to functions. When the client calls `distance()`, it uses the two-argument version shown above. That function then invokes the implementation that matches the iterator category. This dispatch happens at compile time and the general function is inline, so there is literally no performance penalty for choosing the right version of the function. We could achieve the category dispatch with virtual functions and inheritance, but that would incur an unnecessary runtime performance cost.

15.3 Ranges Revisited

Finally, let's return to the third notable feature of our `distance` functions: the valid range precondition. It would be nice if we could have a function `valid_range` that returned true if the range specified by the two iterators was valid and false otherwise, but unfortunately, it's not possible to implement such a function. For example, if two iterators each represent cells in a linked list (`cons` cells, in Lisp terminology), we have no way of knowing if there's a path from one to the other. But even if we're dealing with simple pointers, we still cannot compute `valid_range`: C and C++ don't guarantee that two pointers can be compared unless they point to the same array — something we don't know in advance when we're trying to see if the range is valid. Even if compiler implementations actually allowed comparison of two arbitrary pointers, there is no guarantee that their positions relative to each other won't change — heap-allocated arrays can be moved during program execution.

So we can't write a *valid_range* function, but we can still use it as a precondition. Instead of guaranteeing the correct behavior in code, we'll use axioms that, if satisfied, insure that our `distance` function will behave as intended. Specifically, we postulate the

following two axioms:

$$\text{container}(c) \implies \text{valid}(\text{begin}(c), \text{end}(c))$$
$$\text{valid}(x, y) \wedge x \neq y \implies \text{valid}(\text{successor}(x), y)$$

The first axiom says that if it's a container, the range from `begin()` to `end()` is valid. The second axiom says that if (x,y) is a nonempty valid range, then the range (successor(x), y) is also valid. All STL-style containers must obey these axioms. This allows us to prove the algorithms correct. For example, if you go back to our original `distance` function on page 227, you'll see that the second axiom insures that if we start with a valid range, we'll still have one each time through the loop.

In addition to the successor (++) and `distance` operations, it's useful to have a way to move an iterator by several positions at once. We call this function `advance`. As before, we'll implement two versions, one for input iterators:

```
template <InputIterator I>
inline
void advance(I& x, DifferenceType(I) n, std::input_iterator_tag) {
  while (n) {
    --n;
    ++x;
  }
}
```

And another for random access iterators:

```
template <RandomAccessIterator I>
inline
void advance(I& x, DifferenceType(I) n, std::random_access_iterator_tag) {
  x += n;
}
```

Together with a top-level function for doing the dispatch:

```
template <InputIterator I>
inline
void advance(I& x, DifferenceType(I) n) {
  advance(x, n, (std::iterator_traits<I>::iterator_category)());
}
```

15.4 Linear Search

At the end of the last chapter, we discussed the three fundamental algorithms all programmers should understand: swap, min/max, and linear search. Now we are ready to

discuss the last of the three. The simplest idea of linear search is to scan a linear list until we find a specific element. But we will generalize that to a function that scans the list until it finds an element that satisfies a given predicate. So in addition to being able to search for a specific value, we could find, for example, the first odd element, or the first element that has no vowels, or whatever else we like. Of course, there might not be such an element, so we need some way to indicate that no item is found. We'll call our function `find_if` — i.e. "find it if it's there"[5]:

```
template <InputIterator I, Predicate P>
I find_if(I f, I l, P p) {
  while (f != l && !p(*f)) ++f;
  return f;
}
```

As shown in red, the function relies on equality, dereference, and successor. Note that if no item that satisfies the predicate, the returned value of `f` will be the same as the `l`, the iterator that points past the end of the range. The calling function uses this comparison to know whether an matching item has been found. C and C++ guarantee that a pointer is valid one position past the end of an array.

This function has an implicit semantic precondition: the value type of the iterator and the argument type of the predicate must be the same, otherwise there is no way to apply the predicate to the items in the range.

Sometimes we can only traverse the data once — for example, if we are searching an input stream. As discussed earlier, in this case we'll need to use input iterators and restrict ourselves to single-pass algorithms. With input iterators, `i == j` does not imply `++i == ++j`; for example, if you've already consumed a character from an input stream, you can't consume the same character again with a different pointer.

Here's a variation of our linear search function for the input iterator case; although we could have overloaded the name, we've deliberately added `_n` to emphasize that this version uses a counted range:

```
template <InputIterator I, Predicate P>
std::pair<I, difference_type(I)>
find_if_n(I f, difference_type(I) n, P p) {
  while (n && !p(*f)) { ++f; --n; }
  return std::make_pair(f, n);
}
```

Why do we return a pair? Wouldn't it be sufficient to return the iterator that points to the found element, as we did in the previous version? No. In the previous version, the caller had the "last" iterator to compare to; here, it does not. So the second returned value tells

[5]This name originated in Common Lisp.

the caller whether we're at the end, in which case the item was not found and the returned iterator cannot be dereferenced. But just as importantly, if we do find a matching item, this allows us to *re-start the search where we left off*. Without this, there would be no way to search a range for anything but the first occurence of a desired item.

This illustrates an important point: Just as there can be bugs in code, there can also be bugs in the interface. We'll see an example of one in the next section.

15.5 Binary Search

We'll finish our journey with a look at another core problem, *binary search*. The idea of binary search originated in the *Intermediate Value Theorem* (IVT), also known as the *Bolzano-Cauchy Theorem*:

> If f is a continuous function in an interval $[a, b]$ such that $f(a) < f(b)$,
> then $\forall u \in [f(a), f(b)]$ there is $c \in [a, b]$ such that $u = f(c)$.

The proof of the theorem consists of doing binary search. Suppose, as an example, we have a continuous function such that $f(a)$ is -3 and $f(b)$ is 5. The IVT tells us that for a particular point in the image of the function — let's say, 0 — there is a point c in the domain such that $f(c) = 0$. How can we find that point? We can start by finding the point x_1 that is half the distance between a and b, and computing $f(x_1)$. If it equals 0, we're done; we've found c. If it's less than 0, we repeat with a point x_2 that's half the distance from a to x_1. If it's greater, we take half the distance from x_1 to b. If we keep repeating the process, we'll eventually find c.

Simon Stevin already had a similar idea in 1594, when he devised a version of the IVT for polynomials. But since Steven was interested in doing everything with decimals, he actually divided the interval into tenths rather than halves, and examined all of them until he found the tenth that contained the desired value. It was Lagrange who first described the binary approach for polynomials in 1795. Bolzano and Cauchy generalized the IVT in the early 19th century, and it is their version used by mathematicians today.

Augustin-Louis Cauchy (1789–1857)

[Biography of Cauchy goes here. Some points to include:

- Cauchy invented modern analysis and many other things.
- He wrote 800 papers.

- He refused to take an oath to Louis Philippe after Charles X was deposed. As a result, he was exiled. Later he was allowed to return, but could not get a salary (because he still refused to take the oath).
- After the next revolution, he was allowed to get his salary, but he gave it all to the poor.

]

Binary search was first discussed as a programming technique in 1946 by physicist and computing pioneer John Mauchly, co-creator of the ENIAC (the first electronic general-purpose computer). However, many details were left unspecified. The first "correct" algorithm for performing binary search was published in 1960 by D.H. Lehmer, a mathematician who had worked on the ENIAC years earlier. However, Lehmer's version did not use the correct interface, and this error was repeated for several decades afterwards.

An example of the erroneous interface appears in the UNIX `bsearch()` function. According to the POSIX standard:

> The bsearch() function shall return a pointer to a matching member of the array, or a null pointer if no match is found. If two or more members compare equal, which member is returned is unspecified.

[Cite http://www.unix.com/man-page/POSIX/3posix/bsearch/]

There are two fundamental flaws with this interface. The first concerns the return of a null pointer to indicate that the item is not found. Often you are doing the search because you want to insert an item if it isn't there, *at the place it would have been if it were there*. With this interface, you have to start from scratch to find your insert position, this time using linear search! Furthermore, there are many applications where you actually want to find the closest or next value to where the missing item would be; the item you're searching for may simply be a prefix of the kinds of items you hope to find.

The second flaw concerns the situation where there are multiple matches. The matches may be keys of items that you need to retrieve. So how do you obtain the entire list of equal items, if you don't know which one you're on? You have to do linear search both backwards and forwards to find the ends of the matching list.

The right way to implement binary search begins with the idea of a *partition point*. Assume that we have a list of items that has been sorted such that a certain predicate is true of the first m items, but not the rest.[6] Then the partition point is the position m.

[6] In retrospect, it would have been better to have the false items first, since the boolean value false sorts before true; unfortunately the "wrong" order is now part of the C++ standard.

Formally, if we have a function that computes the partition point, then its precondition is:

$$\exists m \in [f, n) \ : \ (\forall i \in [f, m) \ : \ p(i)) \ \wedge \ (\forall i \in [m, f + n) \ : \ \neg p(i))$$

(i.e. the elements are already partitioned as described above) and its postcondition is that it returns the value m from the precondition.

For a counted range, the partition point algorithm looks like this:

```
template <ForwardIterator I, Predicate P>
I partition_point_n(I f, DifferenceType(I) n, P p) {
  while (n) {
    I middle(f);
    DifferenceType(I) half(n >> 1);
    advance(middle, half);
    if (!p(*middle)) {
      n = half;
    } else {
      f = ++middle;
      n = n - (half + 1);
    }
  }
  return f;
}
```

This is an extremely important algorithm, and it's worth spending some time to make sure you understand it. It uses a binary-search-style strategy just like the IVT. Recall that the goal is to return the first "bad" element, that is, the first element for which the predicate is false. The outer loop continues until `n` is zero. Inside the loop, we position the iterator `middle` to a point halfway between `f` and `f + n`. If our predicate is false for the element at that position, we set `n` to half of its previous value and repeat. Otherwise we know the predicate is true, so we set our starting point `f` to the next value after the middle, adjust `n` to reflect the number of items left, and repeat. When we are done, the return value will be the partition point between the true and false values.

Notice that the above function uses `advance` to move the iterator. Since we don't know the iterator type, we can't assume that addition is allowed. However, if we have a random access iterator, the `advance` function we wrote earlier will call the fast implementation. (If we don't have a random access iterator, we might have to make as many as n moves, but in either case we'll never have to do more than $\log n$ comparisons.)

If we are given a bounded range instead, we simply compute the distance and invoke the counted range version:

```
template <ForwardIterator I, Predicate P>
I partition_point(I f, I l, P p) {
```

```
  return partition_point_n(f, std::distance(f, l), p);
}
```

[Further reading: For more information about sorted ranges, see section 6.5 of EoP.]

Now let's return to the general binary search problem. To solve it, we're going to make use of the following:

Binary Search Lemma. For any sorted range $[v_i, v_j)$ and a value a, there are two iterators, lower bound b_l and upper bound b_u such that:

$$\begin{aligned} &1.\ \forall k \in [i, b_l) &&: v_k < a \\ &2.\ \forall k \in [b_l, b_u) &&: v_k = a \\ &3.\ \forall k \in [b_u, j) &&: v_k > a \end{aligned}$$

These bounds always exist, though in the special case where there is no matching item, $b_l = b_u$.

Exercise 15.1

Prove the Binary Search Lemma.

Now we can use our partition point function to perform binary search. The C++ standard template library actually provides a few functions that perform binary search, depending on our task. If we want to find the *first* position where the item is found, we use the `lower_bound` function. Using the features in C++11, we can write `lower_bound` like this:

```
template <ForwardIterator I>
I lower_bound(I f, I l, ValueType(I) a) {
  return std::partition_point(f, l,
                              [=](ValueType(I) x) { return x < a; });
}
```

The last line defines a new anonymous function (also known as a *lambda expression*) that returns true if its argument is less than `x`, then passes that function as the predicate used by `partition_point`.

In the case where the returned position is not `l`, we still need to know whether we found the item or not. To do this, the caller needs to check whether the dereferenced return value is equal to a.

If we instead want to find the *last* position where the item is found, we use `upper_bound`. The code is almost the same, except that we define the predicate to check if the value is less than or equal to `a`, rather than strictly less than:

```
template <ForwardIterator I>
I lower_bound(I f, I l, ValueType(I) a) {
  return std::partition_point(f, l,
                              [=](ValueType(I) x) { return x <= a; });
}
```

If we don't have C++11, our implementations are a bit more complex:

```
template <ForwardIterator I>
I lower_bound(I f, I l, ValueType(I) a) {
  std::less<ValueType(I)> cmp;
  return std::partition_point(f, l, std::bind2nd(cmp, a));
}

template <ForwardIterator I>
I upper_bound(I f, I l, ValueType(I) a) {
  std::less_equal<ValueType(I)> comp;
  return std::partition_point(f, l, std::bind2nd(cmp, a));
}
```

First we instantiate the function object `std::less` (or `std::less_equal`) to get a comparison predicate. Then we use the adapter `std::bind2nd` to convert the binary comparator to a unary predicate that always compares with the value `a`.

Some readers may be wondering which function is the "real" binary search. The answer is that it depends on the task. If you want to find the position of the first matching element, then `lower_bound` is the "real" binary search. If you want to find the position of the last matching element, then it's `upper_bound`. If you want to know the entire range of matching elements, C++ provides a third function, `equal_range`. And if all you care about is whether or not there was a match, you can use the function `binary_search` — but keep in mind that all it's doing is calling `lower_bound` and testing whether the dereferenced return value is equal to the item.

15.6 Lessons of the Journey

In this journey, we've seen how the mathematical foundations of arithmetic led to the computational concept — or theory — of iterators. Many people think of iterators as a generalization of pointers, and in a sense that is true. But the idea for iterators didn't come from pointers; it came from Peano.

Theories like iterators are just as important to computer science, and are just as sound, as theories like groups in abstract algebra. While most computer scientists don't concentrate on proving theorems, neither did the ancient Greeks who developed early mathematics; they developed algorithms for straightedge and compass.

Epilogue

The three journeys in this book followed the history of three different algorithms, tracing them from their origins in ancient Egypt and Greece to their uses in programming today. While people often think of programming as primarily a 20th-century invention, many of the ideas have much older roots. As we saw in the first journey, the first known algorithm is four thousand years old, and the first book of Euclid's *Elements* may be viewed as the first attempt to build what we might now call a set of components, essentially containing 48 functions which build on each other.

The three journeys also explored bits of the history of three important areas of mathematics. Hopefully you have seen how some of the key ideas from these fields are linked to programming.

Some of the major themes that repeatedly arose are:

Abstraction. One of the things mathematics teaches is abstraction. If you have a procedure for achieving some task, what sorts of things does it work for? Is there a more general version that works for more things? Are there other applications where you could apply it? One example of this was the Egyptian multiplication algorithm, which we generalized to computing power for an arbitrary binary operation and applied to tasks from graph theory to cryptography. The lesson for programming is clear: start with a specific efficient solution and, whenever possible, try to relax the type requirements. This will insure that it has the widest possible applicability, and reduces the brittleness of the interface when requirements change.

Organizing Knowledge. This is a task that has occupied both philosophers and mathematicians for millennia. What sorts of things are there in the world? How do they relate to each other? If we want to represent something, what do we need to know about it? What does it mean to be a certain kind of thing? For mathematicians, it means satisfying a set of axioms. That idea carries over into programming through the notion of concepts, which can be viewed as a set of requirements that types must satisfy.

Constraints, Contradictions, and Creativity. Often the greatest breakthroughs result from situations where someone pushed up against the boundary of an existing approach or found a contradiction. Examples are Pythagoreans and the discovery of irrational numbers, Lobachevsky and the discovery of non-Euclidean geometry, and Stein working on a very slow computer. Don't think that just because there is a standard way of doing some-

thing, there is not an entirely different approach waiting to be discovered — even if the standard one has been around for hundreds of years.

Correctness and Completeness of Programming Interfaces. Just as we can write correct and incorrect code, so we can write correct and incorrect programming interfaces. An incorrect interface cripples the application by limiting what it can do, and/or by making it much less efficient. For example, a binary search function that returns only a boolean value instead of the position of a found item makes it impossible to see if a second matching item exists. Also, algorithms usually need to be implemented in families, where different variations have slightly different interfaces (e.g. counted range vs. bounded range) to handle different use cases. Just as you need to rewrite a program a few times to get it right, you need to redesign the interface; the correct interface won't be clear until you've already implemented an algorithm and explored its use cases.

Theory and Practice. There is a perception among programmers that mathematics, particularly in its more abstract areas, has little practical value, and that mathematicians are people who don't know or care about practical concerns. Looking at the history, we can see that both of these perceptions are false. The greatest mathematicians eagerly worked on extremely practical problems — for example Gauss worked on one of the first electromechanical telegraphs, and Poincaré spent years developing time zones. Perhaps more importantly, it is impossible to know which theoretical ideas are going to have practical applications. Proving theorems about the properties of prime numbers is as theoretical as it gets, yet these theorems led to the creation of cryptographic protocols used every day in online commerce. And arguing about whether an axiomatic system is complete might seem like an Ivory Tower exercise, yet it led to the development of modern computers. Theory and practice are not in opposition; they are two sides of the same coin.

So next time you set out to write a program, remember that you are the inheritor of a long mathematical tradition of algorithmic thought. In using modern tools of programming, you are already benefiting from the work of those who came before, from Euclid to Stevin to Noether to Turing. By designing beautiful, general algorithms, you are adding your own small contribution to their work.

Further Reading

[Further reading goes here]

Appendix: C++ Language Features

[Appendix on C++ language features goes here]

Bibliography

[Bibliography goes here]

Index

[Index goes here]