Our Goal

This book has a straightforward goal: to teach you how to engage with mathematics.

Let's unpack this. By "mathematics," I mean the universe of books, papers, talks, and blog posts that contain the meat of mathematics: formal definitions, theorems, proofs, conjectures, and algorithms. By "engage" I mean that for any mathematical topic, you have the cognitive tools to make progress toward understanding that topic. I will "teach" you by introducing you to—or having you revisit—a broad foundation of topics and techniques that support the rest of mathematics. I say "with" because mathematics requires active participation.

We will define and study many basic objects of mathematics, such as polynomials, graphs, and matrices. More importantly, I'll explain *how to think* about those objects as seasoned mathematicians do. We will examine the hierarchies of mathematical abstraction, along with many of the softer skills and insights that constitute "mathematical intuition." Along the way we'll hear the voices of mathematicians—both famous historical figures and my friends and colleagues—to paint a picture of mathematics as both a messy amalgam of competing ideas and preferences, and a story with delightfully surprising twists and connections. In the end, I will show you how mathematicians think about mathematics.

So why would someone like you[1] want to engage with mathematics? Many software engineers, especially the sort who like to push the limits of what can be done with programs, eventually realize a deep truth: mathematics unlocks a *lot* of cool new programs. These are truly novel programs. They would simply be impossible to write (if not inconceivable!) without mathematics. That includes programs in this book about cryptography, data science, and art, but also to many revolutionary technologies in industry, such as signal processing, compression, ranking, optimization, and artificial intelligence. As importantly, a wealth of opportunity makes programming more fun! To quote Randall Munroe in his XKCD comic *Forgot Algebra,* "The only things you HAVE to know are how to make enough of a living to stay alive and how to get your taxes done. All the fun parts of life are optional." If you want your career to grow beyond shuffling data around to meet arbitrary business goals, you should learn the tools that enable you to write programs that captivate and delight you. Mathematics is one of those tools.

Programmers are in a privileged position to engage with mathematics. Your comfort

[1] Hopefully you're a programmer; otherwise, the title of this book must have surely caused a panic attack.

with functions, logic, and protocols gives you an intuitive familiarity with basic topics such as boolean algebra, recursion, and abstraction. You can rely on this to make mathematics less foreign, progressing all the faster to more nuanced and stimulating topics. By contrast, most educational math content is for students with no background. Such content focuses on rote exercises and passing tests. This book will omit most of that. Programming also allows me to provide immediate applications that ground the abstract ideas in code. In each chapter, we'll fashion our mathematical designs into a program you couldn't have written before. All programs are written in Python 3. The code is available on Github,[2] with a directory for each chapter.

All told, this book is *not* a textbook. I won't drill you with exercises, though drills have their place. We won't build up any particular field of mathematics from scratch. Though we'll visit calculus, linear algebra, and many other topics, this book is far too short to cover everything a mathematician ought to know about these topics. Moreover, while much of the book is appropriately rigorous, I will occasionally and judiciously loosen rigor when it facilitates a better understanding and relieves tedium. I will note when this occurs, and we'll discuss the role of rigor in mathematics more broadly.

Indeed, rather than read an encyclopedic reference, you want to become *comfortable* with the process of learning mathematics. In part that means becoming comfortable with discomfort, with the struggle of understanding a new concept, and the techniques that mathematicians use to remain productive and sane. Many people find calculus difficult, or squeaked by a linear algebra course without grokking it. After this book you should have a core nugget of understanding of these subjects, along with the cognitive tools that will enable you dive as deeply as you like.

As a necessary consequence, in this book you'll learn how to read and write proofs. The simplest and broadest truth about mathematics is that it revolves around proofs. Proofs are both the primary vehicle of insight and the fundamental measure of judgment. They are the law, the currency, and the fine art of mathematics. Most of what makes mathematics mysterious and opaque—the rigorous definitions, the notation, the overloading of terminology, the mountains of theory, and the unspoken obligations on the reader—is due to the centrality of proofs. A dominant obstacle to learning math is an unfamiliarity with this culture. In this book I'll cover the basic methods of proof, and each chapter will use proofs to build the subject matter. To be sure, you don't have to understand every proof to finish this book, and you will probably be confounded by a few. Embrace your humility. Each proof contains layers of insight that are genuinely worthwhile, but few gain a complete picture of a topic in a single sitting. As you grow into mathematics, the act of reading even previously understood proofs provides both renewed and increased wisdom. So long as you identify the value gained by your struggle, your time is well spent.

I'll also teach you how to read between the mathematical lines of a text, and understand the implicit directions and cultural cues that litter textbooks and papers. As we proceed through the chapters, we'll gradually become more terse, and you'll have many opportu-

[2] `pimbook.org`

nities to practice parsing, interpreting, and understanding math. All of the topics in this book are explained by hundreds of other sources, and each chapter's exercises include explorations of concepts beyond these pages. Finally, I'll discuss how mathematicians approach problems, and how their process influences the culture of math.

You will not learn everything you want to know in this book, nor will you learn everything this book has to offer in one sitting. Those already familiar with math may find early chapters offensively slow and detailed. Those genuinely new to math may find the later chapters offensively fast. This is by design. I want you to be exposed to as much mathematics as possible. Learn the main definitions. See new notations, conventions, and attitudes. Take the opportunity to explore topics that pique your interest.

A number of readers have reached out to me to describe their struggles with proofs. They found it helpful to read a companion text on the side with extra guidance on sets, functions, and methods of proof—particularly for the additional exercises and consistently gradual pace. In this second edition, I added two appendices that may help with readers struggling with the pace. Appendix B contains more detail about the formalities underlying proofs, along with strategies for problem solving. Appendix C contains a list of books, and specifically a section for books on "Fundamentals and Foundations" that cover the basics of set theory, proofs, and problem solving strategies.

A number of topics are conspicuously missing from this book, my negligence of which approaches criminal. Except for a few informal cameos, we ignore complex numbers, probability and statistics, differential equations, and formal logic. In my humble opinion, none of these topics is as fundamental for mathematical computer science as those I've chosen to cover. After becoming comfortable with the topics in this book, for example, probability will be very accessible. Chapter 12 on eigenvalues includes a miniature introduction to differential equations. The notes for Chapter 16 on groups briefly summarizes complex numbers. Probability underlies our discussion of random graphs in Chapter 6 and machine learning in Chapter 14. Moreover, many topics in this book are prerequisites for these other areas. And, of course, as a single human self-publishing this book on nights and weekends, I have only so much time.

The first step on our journey is to confirm that mathematics has a culture worth becoming acquainted with. We'll do this with a comparative tour of the culture of software that we understand so well.

Chapter 1

Like Programming, Mathematics has a Culture

Mathematics knows no races or geographic boundaries; for mathematics, the cultural world is one country.

–David Hilbert

Do you remember when you started to really *learn* programming? I do. I spent two years in high school programming games in Java. Those two years easily contain the worst and most embarrassing code I have ever written. My code absolutely reeked. Hundred-line functions and thousand-line classes, magic numbers, unreachable blocks of code, ridiculous comments, a complete disregard for sensible object orientation, and type-coercion that would make your skin crawl. The code worked, but it was filled with bugs and mishandled edge-cases. I broke every rule, but for all my shortcomings I considered myself a hot-shot (at least, among my classmates!). I didn't know how to design programs, or what made a program "good," other than that it ran and I could impress my friends with a zombie shooting game.

Even after I started studying software in college, it was another year before I knew what a stack frame or a register was, another year before I was halfway competent with a terminal, another year before I appreciated functional programming, and to this day I *still* have an irrational fear of systems programming and networking. I built up a base of knowledge over time, with fits and starts at every step.

In a college class on C++ I was programming a Checkers game, and my task was to generate a list of legal jump-moves from a given board state. I used a depth-first search and a few recursive function calls. Once I had something I was pleased with, I compiled it and ran it on my first non-trivial example. Despite following test-driven development, I saw those dreaded words: `Segmentation fault`. Dozens of test cases and more than twenty hours of confusion later, I found the error: my recursive call passed a reference when it should have been passing a pointer. This wasn't a bug in syntax or semantics—I understood pointers and references well enough—but a design error. As most programmers can relate, the most aggravating part was that changing four characters (swapping a few ampersands with asterisks) fixed it. Twenty hours of work for four characters! Once I begrudgingly verified it worked, I promptly took the rest of the day off to play Starcraft.

Such drama is the seasoning that makes a strong programmer. One must study the topics incrementally, learn from a menagerie of mistakes, and spend hours in a befuddled stupor before becoming "experienced." This gives rise to all sorts of programmer culture, Unix jokes, urban legends, horror stories, and reverence for the masters of C that make the programming community so lovely. It's like a secret club where you know all the handshakes, but should you forget one, a crafty use of `grep` and `sed` will suffice. The struggle makes you appreciate the power of debugging tools, slick frameworks, historically enshrined hacks, and new language features that stop you from shooting your own foot.

When programmers turn to mathematics, they seem to forget these trials. The same people who invested years grokking the tools of their trade treat new mathematical tools and paradigms with surprising impatience. I can see a few reasons why. For one, we were forced to take math classes for many year in school. That forced investment shouldn't have been pointless. But the culture of mathematics and the culture of mathematics education—elementary through lower-level college courses—are completely different.

Even college math majors have to reconcile this. I've had many conversations with such students, including friends, colleagues, and even family, who by their third year decided they didn't really enjoy math. The story often goes like this: a student who was good at math in high school reaches the point of a math major at which they must read and write proofs in earnest. It requires an ambiguous, open-ended exploration that they don't enjoy. Despite being a stark departure from the rigid structure of high school math, incoming students are not warned in advance. After coming to terms with their unfortunate situation, they decide that their best option is to persist until they graduate, at which point they return to the comfortable setting of pre-collegiate math, this time in the teacher's chair.

I don't mean to insult teaching as a profession—I love teaching and understand why one would choose to do it full time. There are many excellent teachers who excel at both the math and the trickier task of engaging aloof teenagers to think critically about it. But this pattern of disenchantment among math teachers is prevalent, and it widens the conceptual gap between secondary and "college level" mathematics. Programmers often have similar feelings. The subject they once were good at is suddenly impenetrable. It's a negative feedback loop in the education system. Math takes the blame.

Another reason programmers feel impatient is because they do so many things that relate to mathematics in deep ways. They use graph theory for data structures and search. They study enough calculus to make video games. They hear about the Curry-Howard correspondence between proofs and programs. They hear that Haskell is based on a complicated math thing called category theory. They even use mathematical results in an interesting way. I worked at a "blockchain" company that implemented a Bitcoin wallet, which is based on elliptic curve cryptography. The wallet worked, but the implementer didn't understand why. They simply adapted pseudocode found on the internet. At the risk of a dubious analogy, it's akin to a "script kiddie" who uses hacking tools as black boxes, but has little idea how they work. Mathematicians are on the other end of the spectrum. Why things work takes priority over practical implementation.

There's nothing inherently wrong with using mathematics as a black box, especially the sort of applied mathematics that comes with provable guarantees. But many programmers *want* to dive deeper. This isn't surprising, given how much time engineers spend studying source code and the internals of brittle, technical systems. Systems that programmers rely on, such as dependency management, load balancers, search engines, alerting systems, and machine learning, all have rich mathematical foundations. We're naturally curious.

A second obstacle is that math writers are too terse. The purest fields of mathematics take a sort of pretentious pride in how abstract and compact their work is. I can think of a handful of famous books, for which my friends spent weeks or months on a single chapter! This acts as a barrier to entry, especially since minute details matter for applications.

Yet another hindrance is that mathematics has no centralized documentation. Instead it has a collection of books, papers, journals, and conferences, each with subtle differences, citing each other in a haphazard manner. Dealing with this is not easy. One often needs to translate between two different notations or jargons. Students of mathematics solve these problems with knowledgeable teachers. Working mathematicians "just do it." They reconcile the differences themselves with coffee and contemplation.

What programmers consider "sloppy" notation is one symptom of the problem, but there there are other expectations on the reader that, for better or worse, decelerate the pace of reading. Unfortunately I have no solution here. Part of the power and expressiveness of mathematics is the ability for its practitioners to overload, redefine, and omit in a suggestive manner. Mathematicians also have thousands of years of "legacy" math that require backward compatibility. Enforcing a single specification for all of mathematics—a suggestion I frequently hear from software engineers—would be horrendously counterproductive.

Ideas we take for granted today, such as algebraic notation, drawing functions in the Euclidean plane, and summation notation, were at one point actively developed technologies. Each of these notations had a revolutionary effect on science, and also, to quote Bret Victor, on our capacity to "think new thoughts." One can draw a line from the proliferation of algebraic notation to the invention of the computer.[1] Borrowing software terminology, algebraic notation is among the most influential and scalable technologies humanity has ever invented. And as we'll see in Chapter 10 and Chapter 16, we can find algebraic structure hiding in exciting places. Algebraic notation helps us understand this structure not only because we can compute, but also because we can visually see the symmetries in the formulas. This makes it easier for us to identify, analyze, and encapsulate structure when it occurs.

[1] Leibniz, one of the inventors of calculus, dreamed of a machine that could automatically solve mathematical problems. Ada Lovelace (up to some irrelevant debate) designed the first program for computing Bernoulli numbers, which arise in algebraic formulas for sums of powers of integers. In the early 1900's Hilbert posed his Tenth Problem on algorithms for solving Diophantine equations, and later his Entscheidungsproblem, which was solved concurrently by Church and Turing and directly led to Turing's code-breaking computer.

Finally, the best mathematicians study concepts that connect decades of material, while simultaneously inventing new concepts which have no existing words to describe them. Without flexible expression, such work would be impossible. It reduces cognitive load, a theme that will follow us throughout the book. Unfortunately, it only does so for the readers who have *already* absorbed the basic concepts of discussion. By contrast, good software practice assumes a lower bar. Code is encouraged to be simple enough for new grads to understand, and heavily commented otherwise. Surprising behavior is considered harmful. As such, the uninitiated programmer often has a much larger cognitive load when reading math than when reading a program.

There are good reasons why mathematics is the way it is, though the reasons may not always be clear. I like to summarize the contrast by claiming that mathematical notation is closer to spoken language than to code. There is a historical and cultural context missing from many criticisms of math. It's a legacy system, yes, but a well-designed one. We should understand it, learn from its advantages, and discard the obsolete parts. Those obsolete parts are present, but rarer than they seem.

To fairly evaluate mathematics, we must first learn some mathematics. Only then can we compare and contrast programming and mathematics in terms of their driving questions, their values, their methods, their measures of success, and their cultural expectations. Programming, at its core, focuses on how to instruct a computer to perform some task. But the broader driving questions include how to design a flexible system, how to efficiently store and retrieve data, how to design systems that can handle various modes of failure, how to scale, and how to tame growth and complexity.

Contrast this with mathematics which, at its core, focuses on how to describe a mathematical object and how to prove theorems about its behavior. The broader driving questions include how to design a unified framework for related patterns, how to find computationally useful representations of an object, how to find interesting patterns to study, and most importantly, how to think more clearly about mathematical subjects.

A large chunk of this book expands on this summary through interludes between each chapter and digressions after introducing technical concepts. The rest covers the fundamental objects and methods of a typical mathematical education. So let's begin our journey into the mathematical mists with an open mind.

Read on, and welcome to the club.

Chapter 2

Polynomials

We are not trying to meet some abstract production quota of definitions, theorems and proofs. The measure of our success is whether what we do enables people *to understand and think more clearly and effectively about mathematics.*

–William Thurston

We begin with polynomials. In studying polynomials, we'll discuss mathematical definitions, work carefully through two nontrivial proofs, and implement a system for "sharing secrets" using something called *polynomial interpolation.* To whet your appetite, this secret sharing scheme allows one to encode a secret message in 10 parts so that any 6 can be used to reconstruct the secret, but with fewer than 6 pieces it's impossible to determine even a single bit of the original message. The numbers 10 and 6 are just examples, and the scheme we'll present works for any pair of integers.

2.1 Polynomials, Java, and Definitions

We start with the definition of a polynomial. The problem, if you're the sort of person who struggled with math, is that reading the definition as a formula will make your eyes glaze over. In this chapter we're going to overcome this.

The reason I'm so confident is that I'm certain you've overcome the same obstacle in the context of programming. For example, my first programming language was Java. And my first program, which I didn't write but rather copied verbatim, was likely similar to this monstrosity.

```
/*****************************************
 *  Compilation: javac HelloWorld.java
 *  Execution:  java HelloWorld
 *
 *  Prints "Hello, World".
 *****************************************/
public class HelloWorld {
   public static void main(String[] args) {
      // Prints "Hello, World" to stdout on the terminal.
      System.out.println("Hello, World");
   }
}
```

It was roughly six months before I understood what all the different pieces of this program did, despite the fact that I had written 'public static void main' so many times I had committed it to memory. Computers don't generally require you to understand a code snippet to run. But at *some point,* we all stopped to ask, "what do those words actually mean?" That's the step when my eyes stop glazing over. That's the same procedure we need to invoke for a mathematical definition, preferably faster than six months.

Now I'm going to throw you in the thick of the definition of a polynomial. But stay with me! I want you to start by taking out a piece of paper and literally copying down the definition (the entire next paragraph), character for character, as one would type out a program from scratch. This is not an idle exercise. Taking notes by hand uses a part of your brain that both helps you remember what you wrote, and helps you *read* it closely. Each individual word and symbol of a mathematical definition affects the concept being defined, so it's important to parse everything slowly.

Definition 2.1. A single variable *polynomial with real coefficients* is a function f that takes a real number as input, produces a real number as output, and has the form

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n,$$

where the a_i are real numbers. The a_i are called *coefficients* of f. The *degree* of the polynomial is the integer n.

Let's analyze the content of this definition in three ways. First, *syntactically,* which also highlights some general features of written definitions. Second, *semantically,* where we'll discuss what a polynomial should represent as a concept in your mind. Third, we'll inspect this definition *culturally,* which includes the unspoken expectations of the reader upon encountering a definition in the wild. As we go, we'll clarify some nuance to the definition related to certain "edge cases."

Syntax

A *definition* is an English sentence or paragraph in which italicized words refer to the concepts being defined. In this case, Definition 2.1 defines three things: a *polynomial with real coefficients* (the function f), *coefficients* (the numbers a_i), and a polynomial's *degree* (the integer n).

A proper mathematical treatment might also define what a "real number" is, but we simply don't have the time or space.[1] For now, think of a real number as a floating point number without the emotional baggage that comes from trying to fit all decimals into a finite number of bits.

An array of numbers a, which in most programming languages would be indexed using square brackets like `a[i]`, is almost always indexed in math using subscripts a_i. For two-dimensional arrays, we comma-separate the indices in the subscript, i.e. $a_{i,j}$ is equivalent to `a[i][j]`. Hence, the coefficients are an array of real numbers. Many mathematicians index arrays from 1 instead of 0, and we will do both in this book.

We used a strange phrase in Definition 2.1, that "f has the form" of some expression. This means that we're choosing specific values for the data defining f. It's making a particular instance of the definition, as if it were a class definition in a program. In this case the choices are:

1. The names for all the variables involved. The definition has chosen f for the function, x for the input variable name, a for the array of coefficients, and n for the degree. One can choose other names as desired.
2. A value for the degree.
3. A value for the array of coefficients $a_0, a_1, a_2, \dots, a_n$, where n must match the chosen degree.

Specifying all of these results in a concrete polynomial.

Semantics

Let's start with a simple example polynomial, where I pick g for the function name, t for the input name, b for the coefficients, and define $n = 3$, and $b_0, b_1, b_2, b_3 = 2, 0, 4, -1$. By definition, g has the form

$$g(t) = 2 + 0t + 4t^2 + (-1)t^3.$$

We take some liberties and usually write g more briefly as $g(t) = 2 + 4t^2 - t^3$. As you might expect, g is a function you can evaluate, and evaluating it at an input $t = 2$ means substituting 2 for t and doing the requisite arithmetic to get

$$g(2) = 2 + 4(2^2) - 2^3 = 10.$$

According to the definition, a polynomial is a function that is written in a certain form. Really what's being said is that a polynomial is any function of a single input that *can be* written in the required form, even if you might write it a different way sometimes. This

[1] If you're truly interested in how real numbers are defined from scratch, Spivak's text *Calculus* devotes Chapter 29 to a gold-standard treatment. You might be ready for it after working through a few chapters of this book, but be warned: Spivak starts Chapter 29 with, "The mass of drudgery which this chapter necessarily contains..."

makes our internal concept of a polynomial more general than the letter of Definition 2.1. A polynomial is any function of a single numeric input that can be expressed using only addition and multiplication and constants, along with the input variable itself. So the following is a polynomial:

$$g(t) = (t - 1)(t + 6)^2$$

You recover the precise form of Definition 2.1 by algebraically simplifying and grouping terms. The form described in Definition 2.1 is not ideal for every occasion! For example, if you want to evaluate a polynomial quickly on a computer, you might represent the polynomial so that evaluating it doesn't redundantly compute the powers $t^1, t^2, t^3, \dots, t^n$. One such scheme is called Horner's method, which we'll return to in an Exercise. The form in Definition 2.1 might be called a "canonical" or "standard" form, and it's often useful for manipulation in proofs. As we'll see later in this chapter, it's easy to express a generic sum or difference of two polynomials in the standard form.

Suffice it to say, there are many representations of the same abstract polynomial. You can do arithmetic and renaming to get to a standard representation. $f(x) = x + 1$ is the same polynomial as $g(t) = 1 + t$, though they differ syntactically.

There are other ways to think about polynomials, and we'll return to polynomials in future chapters with new and deeper ideas about them. Here are some previews of that. The first is that a polynomial, as with any function, can be represented as a set of pairs called *points*. That is, if you take each input t and pair it with its output $f(t)$, you get a set of tuples $(t, f(t))$, which can be analyzed from the perspective of set theory. We will return to this perspective in Chapter 4.

Second, a polynomial's graph can be plotted as a curve in space, so that the horizontal direction represents the input and the vertical represents the output. Figure 2.1 shows a plot of one part of the curve given by the polynomial $f(x) = x^5 - x - 1$.

Using the curves they "carve out" in space, polynomials can be regarded as geometric objects with geometric properties like "curvature" and "smoothness." In Chapter 8 we'll return to this more formally, but until then one can guess how they might faithfully describe a plot like the one in Figure 2.1. The connection between polynomials as geometric objects and their algebraic properties is a deep one that has occupied mathematicians for centuries. For example, the degree gives some information about the shape of the curve. Figure 2.2 shows plots of generic polynomials of degrees 3 through 6. As the degree goes up, so does the number of times the polynomial "changes direction" between increasing and decreasing. Making this mathematically rigorous requires more nuance—after all, the degree five polynomial in Figure 2.1 only changes direction twice—but the pattern suggested by Figure 2.2 is no coincidence.

Finally, polynomials can be thought of as "building blocks" for complicated structures. That is, polynomials are a family of increasingly expressive objects, which get more complex as the degree increases. This idea is the foundation of the application for this chapter (sharing secrets), and it will guide us to use Taylor polynomials to approximate things in Chapters 8 and 14.

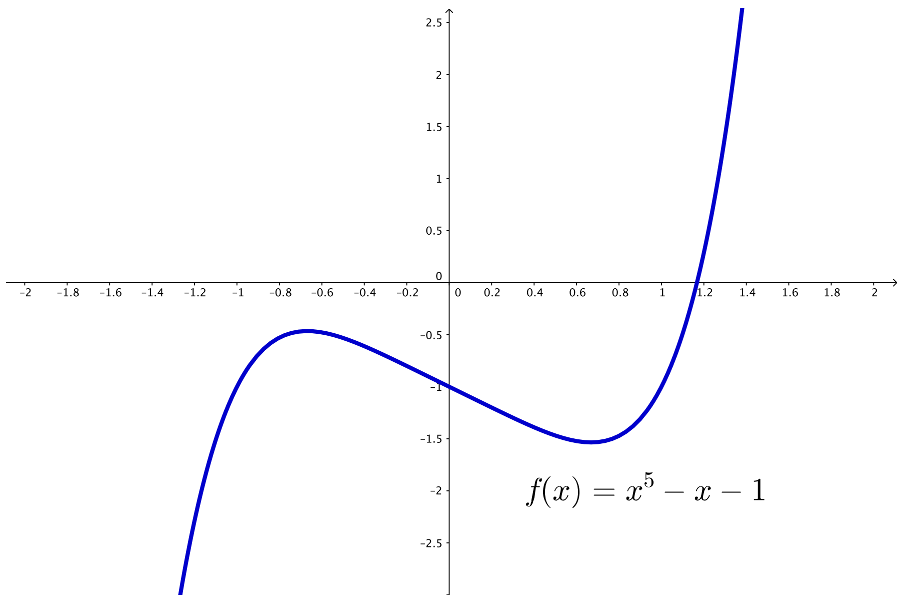

Figure 2.1: A polynomial as a curve in the plane.

Polynomials occur with stunning ubiquity across mathematics. It makes one wonder exactly why they are so central. It's because polynomials encapsulate the full expressivity of addition and multiplication. As programmers, we know that even such simple operations as binary AND, OR, and NOT, when combined arbitrarily, allow us to build circuits that make a computer. Those three operations yield the full gamut of algorithms. Polynomials fill a similar role for arithmetic. Indeed, polynomials with multiple variables can represent AND, OR, and NOT, if you restrict the values of the variables to be zero and one (interpreted as *false* and *true*, respectively).

$$\begin{aligned} \text{AND}(x, y) &= xy \\ \text{NOT}(x) &= 1 - x \\ \text{OR}(x, y) &= 1 - (1 - x)(1 - y) \end{aligned}$$

Any logical condition can be represented using a combination of these polynomials. Polynomials are expressive enough to capture all of boolean logic. This suggests that even single-variable polynomials *should* have strikingly complex behavior. The rest of the chapter will display bits of that dazzling performance.

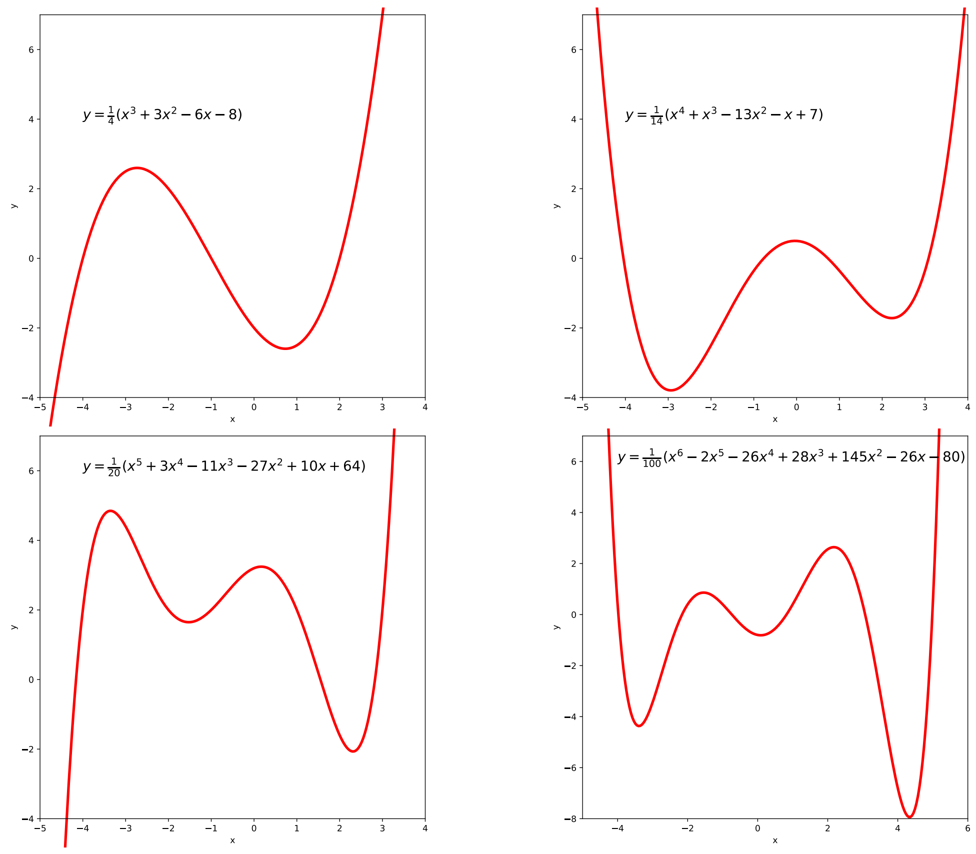

Figure 2.2: Polynomials of varying degrees.

Culture

The most important cultural expectation, one every mathematician knows, is that the second you see a definition in a text **you must immediately write down examples.** Generous authors provide examples of genuinely new concepts, but an author is never obligated to do so. The unspoken rule is that the reader should not continue unless the reader understands what the definition is saying. That is, you aren't expected to *master* the concept, most certainly not at the same speed you read it. But you should have some idea going forward of what the defined words refer to.

Software testing provides a good analogy. You start with the simplest possible tests, usually setting as many values as you can to zero or one, then work your way up to more complicated examples. Later, when you get stuck on some theorem or proof—an unavoidable occupational hazard—you return to those examples and test how the claims in the proof apply to them. This is how one builds so-called "mathematical intuition." In the long term, that intuition allows you to absorb new ideas faster.

So let's write down some examples of polynomials according to Definition 2.1, starting from the simplest. To make you pay attention, I'll slip in some examples that are not polynomials and your job is to check them against the definition. Take your time, and you can check your answers in the Chapter Notes.

$$
\begin{aligned}
f(x) &= 0 \\
g(x) &= 12 \\
h(x) &= 1 + x + x^2 + x^3 \\
i(x) &= x^{1/2} \\
j(x) &= \frac{1}{2} + x^2 - 2x^4 + 8x^8 \\
k(x) &= 4.5 + \frac{1}{x} - \frac{5}{x^2} \\
l(x) &= \pi - \frac{1}{e}x^5 + e\pi^3 x^{10} \\
m(x) &= x + x^2 - x^\pi + x^e
\end{aligned}
$$

Like software testing, examples weed out pesky edge cases and clarify what is permitted by the definition. For example, the exponents of a polynomial must be nonnegative integers, though I only stated it implicitly in the definition.

When reading a definition, one often encounters the phrase “by convention.” This can refer to a strange edge case or a matter of taste. A common example is the factorial $n! = 1 \cdot 2 \cdot \dots \cdot n$, where $0! = 1$ by convention. This makes formulas cleaner and provides a natural default value of an “empty product,” a sensible base case for a loop that computes the product of a (possibly empty) list of numbers.

For polynomials, convention strikes when we inspect the example $f(x) = 0$. What is the degree of f? On one hand, it makes sense to say that the zero polynomial has degree $n = 0$ and $a_0 = 0$. On the other hand, it also makes sense (in a strict, syntactical sense) to say that f has degree $n = 1$ with $a_0 = 0$ and $a_1 = 0$, or $n = 2$ with three zeros. But we don’t want a polynomial to have multiple distinct possibilities for degree. Indeed, this would allow $f(x) = 0$ to have *every* positive degree (by adding extra zeros), depriving the word “degree” of a consistent interpretation.

To avoid this, we amend Definition 2.1 so that the last coefficient a_n is required to be nonzero. But then the function $f(x) = 0$ is not allowed to be a polynomial! So, by convention, we define a special exception, the function $f(x) = 0$, as the *zero polynomial*. By convention, the zero polynomial is defined to have degree -1. Note that every time a definition includes the phrase “by convention,” a computer program gains an edge case.[2]

This edge case made us reconsider the right definition of a polynomial, but it was mostly a superficial change. Other times, as we will confront head on in Chapter 8 when we define limits, dealing with an edge case reveals the soul of a concept. It’s curious how mathematical books tend to start with the final product, instead of the journey to the

[2] You may wonder: is it possible to represent the same polynomial with two formulas that have different degrees? Theorem 2.3 can be used to prove this is impossible. Exercise 4 asks you to prove it using elementary means.

right definition. Perhaps teaching the latter is much harder and more time consuming, with fewer tangible benefits. But in advanced mathematics, deep understanding comes in fits and starts. Often, no such distilled explanation is known.

In any case, examples are the primary method to clarify the features of a definition. Having examples in your pocket as you continue to read is important, and *coming up* with the examples yourself is what helps you internalize a concept.

It is a bit strange that mathematicians choose to write definitions with variable names by example, rather than using the sort of notation one might use to define a programming language syntax. Using a loose version of Backus-Naur form (BNF), which is used in parsers to define syntax, I might define a polynomial as:

```
coefficient = number
variable = 'x'
term = coefficient * variable ^ int
polynomial = term
           | term + polynomial
```

The problem is that this definition doesn't tell you what polynomials are all about. While Definition 2.1 isn't perfect, it signals that a polynomial is a function of a single input. BNF only provides a sequence of named tokens. As a human, we want to understand that a polynomial is a function with particular structure, and that's not captured by a purely syntactic definition. This theme, that most mathematics is designed for human-to-human communication, will follow us throughout the book. Mathematical discourse is about getting a vivid and rigorous idea from your mind into someone else's mind.

That's why an author usually starts with a conceptual definition like Definition 2.1 many pages before discussing a computer-friendly representation. It's why mathematicians will seamlessly convert between representations—such as the functional, set-theoretic, and geometric representations I described earlier—as if mathematics were the JavaScript type system on steroids. In Java you have to separate an interface from the class which implements it, and in C++ templates are distinct from their usage. In math, we often have multiple equivalent definitions—some closer to an interface and some closer to a syntactic representation—and we have to prove that they are equivalent to justify switching between them. Once we've built up a collection of these definitions, we often settle on one as the "clearest" and most generally useful definition that is presented first. Underlying all the definitions is an abstract concept we keep in our minds. The definition is one way to make that concept concrete while also expressing one particular facet of its properties for the task at hand.

I want to make this extremely clear because in mathematics it's implicit. My math teachers in college and grad school *never explicitly* discussed why one would use one definition over another, because somehow along the arduous journey through a math education, the folks who remained understood it. It also explains why understanding a definition is such an important prerequisite to reading the mathematics that follows.

Polynomials may seem frivolous to illustrate the difference between an object-as-abstract-concept and the representational choices that go into understanding a definition, but the same pattern lurks behind more complicated definitions. First the author will start with the best conceptual definition—the one that seems to them, with the hindsight of years of study, to be the most useful way to communicate the idea behind the concept. For us that's Definition 2.1. Often these definitions seem totally useless from a programming perspective.

Then ten pages later (or a hundred!) the author introduces another definition, often a data definition, which turns out to be *equivalent* to the first. Any properties defined in the first definition automatically hold in the second and vice versa. But the *data definition* is the one that allows for nice programs. You might think the author was crazy not to start with the data definition, but it's the conceptual definition that sticks in your mind, generalizes, and guides you through proofs. This interplay between intuitive and data definitions will take center stage in Chapter 10, our first exposure to linear algebra.

It's also worth noting that the multiplicity of definitions arose throughout history. Polynomials have been studied for many centuries, but parser-friendly forms of polynomials weren't needed until the computer age. Likewise, algebra was studied before the graphical representations of Descartes allowed us to draw polynomials as curves. Each new perspective and definition was driven by an additional need. As a consequence, the "best" definition of a concept can change. Throughout history math has been shaped and reshaped to refine, rigorize, and distill the core insights, often to ease the fashionable calculations of the time.

In any case, the point is that we will fluidly convert between the many ways of thinking about polynomials: as expressions defined abstractly by picking a list of numbers, or as functions with a special structure. Effective mathematics is flexible in this way.

2.2 A Little More Notation

When defining a function, one often uses the compact arrow notation $f : A \to B$ to describe the allowed inputs and outputs. All possible inputs are collectively called the *domain*, and all possible outputs are called the *range*. There is one caveat I'll explain via programming. Say you have a function that doubles the input, such as

```
int f(int x) {
  return 2*x;
}
```

The inputs are integers, and the *type* of the output is also integer, but 3 is not a possible output of this particular function.

In math we disambiguate this with two words. *Range* is the set of actual outputs of a function, and the "type" of outputs is called the *codomain*. The notation $f : A \to B$ specifies the domain A and codomain B, while the range depends on the semantics of f. When one introduces a function, as programmers do with type signatures and function headers, we state the notation $f : A \to B$ before the function definition.

Because mathematicians were not originally constrained by ASCII, they developed other symbols for types. The symbol for the set of real numbers is $\mathbb{R}$. The font is called "blackboard-bold," and it's the standard font for denoting number systems. Applying the arrow notation, a polynomial is $f : \mathbb{R} \to \mathbb{R}$. A common phrase is to say a polynomial is "over the reals" to mean it has real coefficients. As opposed to, say, a polynomial over the integers that has integer coefficients.

Most famous number types have special symbols. The symbol for integers is $\mathbb{Z}$, and the positive integers are denoted by $\mathbb{N}$, often called the *natural numbers*.[3] There is an amusing dispute of no real consequence between logicians and other mathematicians on whether zero is a natural number, with logicians demanding it is.

Finally, I'll use the $\in$ symbol, read "in," to assert or assume membership in some set. For example $q \in \mathbb{N}$ is the claim that q is a natural number. It is literally short hand for the phrase, "q is in the natural numbers," or "q is a natural number." It can be used in a condition (preceded by "if"), an assertion (preceded by "suppose"), or a question.

2.3 Existence & Uniqueness

Having seen some definitions, we're ready to develop the main tool we need for secret sharing: the existence and uniqueness theorem for polynomials passing through a given set of points.

First, a word about existence and uniqueness. Existence proofs are classic in mathematics. They come in all shapes and sizes. Mathematicians like to take interesting properties they see on small objects, write down the property in general, and then ask things like, "Are there arbitrarily large objects with this property?" or, "Are there infinitely many objects with this property?" I imagine a similar pattern in physics. Given equations that govern the internal workings of a star you might ask, would these equations support arbitrarily massive stars?

One simple uniqueness question is quite famous: are there are infinitely many pairs of prime numbers of the form $p, p + 2$? For example, 11 and 13 work, but 23 is not part of such a pair.[4] It's an open question whether there are infinitely many such pairs. The assertion that there are is called the Twin Prime Conjecture.

In some cases you get lucky, and the property you defined is specific enough to single out a *unique* mathematical object. This is what will happen to us with polynomials. Other times, the property (or list of properties) you defined are too restrictive, and there are no mathematical objects that can satisfy it. For example, Kleinberg's Impossibility Theorem for Clustering lays out three natural properties for a clustering algorithm (an algorithm that finds dense groups of points in a geometric dataset) and proves that no algorithm can satisfy all three simultaneously. See the Chapter Notes for more on this. Though such theorems are often heralded as genius, more often than not mathematicians avoid impossibility by turning small examples into broad conjectures.

That's how we'll approach existence and uniqueness for polynomials. Here is the theo-

[3] The Z stands for Zahlen, the German word for "numbers."
[4] See how I immediately wrote down examples?

rem we'll prove, stated in its most precise form. Don't worry, we'll go carefully through every bit of it, but try to read it now.

Theorem 2.2. *For any integer $n \geq 0$ and any list of $n+1$ points $(x_1, y_1), (x_2, y_2), \dots, (x_{n+1}, y_{n+1})$ in $\mathbb{R}^2$ with $x_1 < x_2 < \cdots < x_{n+1}$, there exists a unique polynomial $p(x)$ of degree at most n such that $p(x_i) = y_i$ for all i.*

The one piece of new notation is the exponent on $\mathbb{R}^2$. This means "pairs" of real numbers. Likewise, $\mathbb{Z}^3$ would be triples of integers, and $\mathbb{N}^{10}$ length-10 tuples of natural numbers.

A briefer, more informal way to state the theorem: there is a unique degree n polynomial passing through a choice of $n+1$ points.[5] Now just like with definitions, the first thing we need to do when we see a new theorem is write down the simplest possible examples. In addition to simplifying the theorem, it will give us examples to work with while going through the proof. Write down some examples now. As mathematician Alfred Whitehead said, "We think in generalities, but we live in details."

Back already? I'll show you examples I'd write down, and you can compare your process to mine. The simplest example is $n = 0$, so that $n + 1 = 1$ and we're working with a single point. Let's pick one at random, say $(7, 4)$. The theorem asserts that there is a unique degree zero polynomial passing through this point. What's a degree zero polynomial? Looking back at Definition 2.1, it's a function like $a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d$ (I'm using d for the degree here because n is already taken), where we've chosen to set $d = 0$. Setting $d = 0$ means that f has the form $f(x) = a_0$. So what's such a function with $f(7) = 4$? There is no choice but $f(x) = 4$. It should be clear that it's the only degree zero polynomial that does this. Indeed, the datum that defines a degree-zero polynomial is a single number, and the constraint of passing through the point $(7, 4)$ forces that one piece of data to a specific value.

Let's move on to a slightly larger example which I'll allow you to work out for yourself before going through the details. When $n = 1$ and we have $n + 1 = 2$ points, say $(2, 3), (7, 4)$, the theorem claims a unique degree 1 polynomial f with $f(2) = 3$ and $f(7) = 4$. Find it by writing down the definition for a polynomial in this special case and solving the two resulting equations.[6]

Alright. A degree 1 polynomial has the form

$$f(x) = a_0 + a_1 x.$$

Writing down the two equations $f(2) = 3, f(7) = 4$, we must simultaneously solve:

$$a_0 + a_1 \cdot 2 = 3$$
$$a_0 + a_1 \cdot 7 = 4$$

[5] To say a function $f(x)$ "passes" through a point (a, b) means that $f(a) = b$. When we say this we're thinking of f as a geometric curve. It's 'passing' through the point because we imagine a dot on the curve moving along it. That perspective allows for colorful language in place of notation.

[6] If you're comfortable solving basic systems of equations, you may want to skip ahead to the next section.

If we solve for a_0 in the first equation, we get $a_0 = 3 - 2a_1$. Substituting that into the second equation we get $(3 - 2a_1) + a_1 \cdot 7 = 4$, which solves for $a_1 = 1/5$. Plugging this back into the first equation gives $a_0 = 3 - 2/5$. This has forced the polynomial to be exactly

$$f(x) = \left(3 - \frac{2}{5}\right) + \frac{1}{5}x = \frac{13}{5} + \frac{1}{5}x.$$

Geometrically, a degree 1 polynomial is a line. Our example above reinforces a fact we already know, that there is a unique line between any two points. Well, it's not *quite* the same fact. What is different about this scenario? The statement of the theorem said, "$x_1 < x_2 < \cdots < x_{n+1}$". In our example, this means we require $x_1 < x_2$. So this is where we run a sanity check. What happens if $x_1 = x_2$? Think about it, and if you can't tell then you should try to prove it wrong: try to find a degree 1 polynomial passing through the points $(2, 3), (2, 5)$.

The problem could be that there is *no* degree 1 polynomial passing through those points, violating existence. Or, the problem might be that there are *many* degree 1 polynomials passing through these two points, violating uniqueness. It's your job to determine what the problem is. And despite it being pedantic, you should work straight from the definition of a polynomial! Don't use any mnemonics or heuristics you may remember; we're practicing reading from precise definitions.

In case you're stuck, let's follow our pattern from before. If we call $a_0 + a_1x$ our polynomial, saying it passes through these two points is equivalent to saying that there is a simultaneous solution to the following two equations $f(2) = 3$ and $f(2) = 5$.

$$\begin{aligned} a_0 + a_1 \cdot 2 &= 3 \\ a_0 + a_1 \cdot 2 &= 5 \end{aligned}$$

What happens when you try to solve these equations like we did before? Try it.

What about for three points or more? Well, that's the point at which it might start to get difficult to compute. You can try by setting up equations like those I wrote above, and with some elbow grease you'll solve it. Such things are best done in private so you can make plentiful mistakes without being judged for it.

Now that we've worked out two examples of the theorem in action, let's move on to the proof. The proof will have two parts, existence and uniqueness. That is, first we'll show that a polynomial satisfying the requirements exists, and then we'll show that if two polynomials both satisfied the requirements, they'd have to be the same. In other words, there can only be one polynomial with that property.

Existence of Polynomials Through Points

We will show existence by direct construction. That is, we'll "be clever" and find a general way to write down a polynomial that works. Being clever sounds scary, but the process is actually quite natural, and it follows the same pattern as we did for reading and understanding definitions: you start with the simplest possible example (but this time the example will be generic) and then you work up to more complicated examples. By the time we get to $n = 2$ we will notice a pattern, that pattern will suggest a formula for the general solution, and we will prove it's correct. In fact, once we understand how to build the general formula, the proof that it works will be trivial.

Let's start with a single point (x_1, y_1) and $n = 0$. I'm not specifying the values of x_1 or y_1 because I don't want the construction to depend on my arbitrary specific choices. I must ensure that $f(x_1) = y_1$, and that f has degree zero. Simply enough, we set the first coefficient of f to y_1, the rest zero.

$$f(x) = y_1$$

On to two points. Call them $(x_1, y_1), (x_2, y_2)$ (note the variable is just plain x, and my example inputs are $x_1, x_2, \dots$). Now here's an interesting idea: I can write the polynomial in this strange way:

$$f(x) = y_1 \frac{x - x_2}{x_1 - x_2} + y_2 \frac{x - x_1}{x_2 - x_1}$$

Let's verify that this works. If I evaluate f at x_1, the second term gets $x_1 - x_1 = 0$ in the numerator and so the second term is zero. The first term, however, becomes $y_1 \frac{x_1 - x_2}{x_1 - x_2} = y_1 \cdot 1$, which is what we wanted: we gave x_1 as input and the output was y_1. Also note that we have explicitly disallowed $x_1 = x_2$ by the conditions in the theorem, so the fractions will never be $0/0$.

Likewise, if you evaluate $f(x_2)$ the first term is zero and the second term evaluates to y_2. So we have both $f(x_1) = y_1$ and $f(x_2) = y_2$, and the expression is a degree 1 polynomial. How do I know it's degree one when I wrote f in that strange way? For one, I could rewrite f like this:

$$f(x) = \frac{y_1}{x_1 - x_2}(x - x_2) + \frac{y_2}{x_2 - x_1}(x - x_1),$$

and simplify with typical algebra to get the form required by the definition:

$$f(x) = \frac{x_1 y_2 - x_2 y_1}{x_1 - x_2} + \left(\frac{y_1 - y_2}{x_1 - x_2} \right) x$$

What a headache! Instead of doing all that algebra, I could observe that no powers of x appear in the formula for f that are larger than 1, and we never multiply two x's together. Since these are the only ways to get degree bigger than 1, we can skip the algebra and be confident that the degree is 1.

The key to the above idea, and the reason we wrote it down in that strange way, is so that each constraint (e.g., $f(x_1) = y_1$) could be isolated in its own term, while all the other terms evaluate to zero. For three points $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ we just have to beef up the terms to maintain the same property: when you plug in x_1, all terms except the first evaluate to zero and the fraction in the first term evaluates to 1. When you plug in x_2, the second term is the only one that stays nonzero, and likewise for the third. Here is the generalization that does the trick.

$$f(x) = y_1 \tfrac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} + y_2 \tfrac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} + y_3 \tfrac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}$$

Now you come in. Evaluate f at x_1 and verify that the second and third terms are zero, and that the first term simplifies to y_1. The symmetry in the formula should convince you that the same holds true for x_2, x_3 without having to go through all the steps two more times. Then argue why f is degree 2.

The general formula for n points $(x_1, y_1), \dots, (x_n, y_n)$, should follow the same pattern. Add up a bunch of terms, and for the i-th term you multiply y_i by a fraction you construct according to the rule: the numerator is the product of $x - x_j$ for every j *except* i, and the denominator is a product of all the $(x_i - x_j)$ for the same j's as the numerator. It works for the same reason that our formula works for three terms above. By now, the process is clear enough that you could write a program to build these polynomials quite easily, and we'll walk through such a program together at the end of the chapter.

Here is the notation version of the process we just described in words. It's a mess, but we'll break it down.

$$f(x) = \sum_{i=1}^{n} y_i \cdot \left(\prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \right)$$

What a mouthful! I'll assume the $\sum, \prod$ symbols are new to you. They are read semantically as "sum" and "product," or typographically as "sigma" and "pi". They essentially represent loops of arithmetic. That is, the statement $\sum_{i=1}^{n}(\text{expr})$ is equivalent to the following code snippet.

```
int i;
sometype theSum = defaultValue;

for (i = 1; i <= n; i++) {
  theSum += expr(i);
}

return theSum;
```

Note by indexing from 1 and including the upper limit of the for loop condition, we are deviating from the standard programming style. Indexing from zero, like $\sum_{i=0}^{n}$, produces $n + 1$ terms in the resulting sum.

I used the undefined tokens `defaultValue` and `sometype` to highlight that the meaning of the sum depends on what the conventional 'zero object' is in that setting. For adding numbers the zero object is zero, and for adding polynomials it's the zero polynomial. It gets exotic with more advanced mathematics, which we'll see in Chapter 16 when we study groups. The point is that $\sum$ does not imply a type. It's merely a shorthand for the symbol $+$.

Moreover, explaining $\sum$ using code allows me to define $\prod$ by analogy: you just replace `+=` with `*=` and reinterpret the "default value" as what makes sense for multiplication. Functional programmers will know this pattern well, because both are a "fold" (or "reduce") function with a particular choice of binary operation and initial value.

The notation $\prod_{j \neq i}$ adds three caveats. First, recall that in this context i is fixed by the outer loop, so j is the looping variable (unfortunately, the reader is required to keep track of scope when it comes to nested sums and products). Second, the bounds on j are not stated; we have to infer them from the context. There are two hints: we're comparing j to i, so it should probably have the same range as i unless otherwise stated, and we can see where in the expression we're using j. We're using it as an index on the x's. Since the x indices go from 1 to n, we'd expect j to have that range. Being so loose might seem hazardous, but if mathematicians consider it "easy" to infer the intent of a notation, then it is considered rigorous enough.[7]

Though it sometimes makes me cringe to say it, give the author the benefit of the doubt. When things are ambiguous, pick the option that doesn't break the math. In this respect, you have to act as both the tester, the compiler, and the bug fixer when you're reading math. The best default assumption is that the author is far smarter than we are, and if you don't understand something, it's likely a user error and not a bug. In the occasional event that the author is wrong, it's often a simple mistake or typo, to which an experienced reader would say, "The author obviously meant 'foo' because otherwise none of this makes sense," and continue unscathed.

Finally, the $j \neq i$ part is an implied filter on the range of j. Inside the `for` loop you add an extra `if` statement to skip that iteration if $j = i$. Read out loud, $\prod_{j \neq i}$ would be "the product over j not equal to i." If we wanted to write out the product-nested-in-a-sum as a nested loop, it would look like this:

```
int i, j;
sometype theSum = defaultSumValue;

for (i = 1; i <= n; i++) {
  othertype product = defaultProductValue;

  for (j = 1; j <= n; j++) {
    if (j != i) {
      product *= foo(i, j);
    }
  }

  theSum += bar(i) * product;
}

return theSum;
```

$$f(x) = \sum_{i=1}^{n} \mathrm{bar}(i) \left(\prod_{j \neq i} \mathrm{foo}(i, j) \right)$$

Compare the math and code, and make sure you can connect the structural pieces. Often the inner parentheses are omitted, with the default assumption that everything to the right of a $\sum$ or $\prod$ is in the body of that loop.

[7] Another reason is that mathematicians get tired of writing these "obvious" details over and over again. Mathematicians don't have text editor tools like programmers do.

If the formula on the right still seems impenetrable, take solace in your own experience: the reason you find the left side so easy to read is that you've spent years building up the cognitive pathways in your brain for reading code. You can identify what's filler and what's important; you automatically filter out the noise in the syntax. Over time, you'll achieve this for mathematical formulas, too. You'll know how to zoom in to one expression, understand what it's saying, and zoom out to relate it to the formula as a whole. Everyone struggles with this, myself included.

One additional difficulty of reading mathematics is that the author will almost never go through these details for the reader. It's a rather subtle point to be making so early in our journey, but it's probably the first thing you notice when you read math books. Instead of doing the details, a typical proof of the existence of these polynomials looks like this.

Proof. Let $(x_1, y_1), \dots, (x_{n+1}, y_{n+1})$ be a list of $n+1$ points with no two x_i the same. To show existence, construct $f(x)$ as

$$f(x) = \sum_{i=1}^{n+1} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

Clearly the constructed polynomial $f(x)$ has degree at most n because each term has degree at most n. For each i, plugging in x_i causes all but the i-th term in the sum to vanish,[8] and the i-th term evaluates to y_i, as desired.

... Uniqueness part (we'll complete this proof in the next section) ...

$\square$

The square $\square$ is called a *tombstone* and marks the end of a proof. It's a modern replacement for QED borrowed from magazines.

The proof writer gives a relatively brief overview and you are expected to fill in the details to your satisfaction. It sucks, but if you do what's expected of you—that is, write down examples of the construction before reading on—then you build up those neural pathways, and eventually you realize that the explanation is as simple and clear as it can be. Meanwhile, your job is to evaluate the statements made in the proof on your examples. Practice allows you to judge how much work you need to put into understanding a construction or definition before continuing. And, more importantly, you'll understand it more thoroughly for all your testing.

Uniqueness of Polynomials Through Points

Now for the uniqueness part. This is a straightforward proof, but it relies on a special fact about polynomials. We'll state the fact as a theorem that we won't prove. Some terminology: a *root* of a polynomial $f : \mathbb{R} \to \mathbb{R}$ is a value z for which $f(z) = 0$.

Theorem 2.3. *The zero polynomial is the only polynomial over $\mathbb{R}$ of degree at most n which has more than n distinct roots.*

[8] To "vanish" means to evaluate to zero.

On to the uniqueness proof. It works by supposing we actually have *two* polynomials f and g, both of degree n, passing through the desired set of points $(x_1, y_1), \dots, (x_{n+1}, y_{n+1})$. We don't assume we know anything else about the polynomials ahead of time. They could be different, or they could be the same. If you wrote down two different looking polynomials with the two properties, they might just *look* different (maybe one is in factored form). So the proof operates by making no other assumptions, and showing that actually f and g have to be the same.

So suppose f, g are two such polynomials. Consider the polynomial $(f - g)(x)$, which we define as $(f - g)(x) = f(x) - g(x)$. Note that $f - g$ is a polynomial because, if the coefficients of f are a_i and the coefficients of g are b_i, the coefficients of $f - g$ are $c_i = a_i - b_i$. If f and g have different degrees, then c_i is simply a_i or $-b_i$, depending on which of f, g has a larger degree. It is crucial to this proof that $f - g$ is a polynomial.

What do we know about $f - g$? It's degree is certainly *at most* n, because you can't magically produce a coefficient of x^7 if you subtract two polynomials whose highest-degree terms are x^5. Moreover, we know that $(f - g)(x_i) = 0$ for all i. Recall that x is the generic input variable, while x_i are the input values of the specific list of points $(x_1, y_1), \dots, (x_{n+1}, y_{n+1})$ that f and g are assumed to agree on. Indeed, for every i, $f(x_i) = g(x_i) = y_i$, so subtracting them gives zero.

Now we apply Theorem 2.3. If we call d the degree of $f - g$, we know that $d \leq n$, and hence that $f - g$ can have no more than n roots unless it's the zero polynomial. But there are $n + 1$ points x_i where $f - g$ is zero! Theorem 2.3 implies that $f - g$ must be the zero polynomial, meaning f and g have the same coefficients.

Just for completeness, I'll write the above argument more briefly and put the whole proof of the theorem together as it would show up in a standard textbook. That is, extremely tersely.

Theorem 2.4. *For any integer $n \geq 0$ and any list of $n + 1$ points $(x_1, y_1), (x_2, y_2), \dots, (x_{n+1}, y_{n+1})$ in $\mathbb{R}^2$ with $x_1 < x_2 < \cdots < x_{n+1}$, there exists a unique polynomial $p(x)$ of degree at most n such that $p(x_i) = y_i$ for all i.*

Proof. Let $(x_1, y_1), \dots, (x_{n+1}, y_{n+1})$ be a list of points with no two x_i the same. To show existence, construct $f(x)$ as

$$f(x) = \sum_{i=1}^{n+1} y_i \left(\prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \right)$$

Clearly the constructed polynomial $f(x)$ is degree $\leq n$ because each term has degree at most n. For each i, plugging in x_i causes all but the i-th term in the sum to vanish, and the i-th term clearly evaluates to y_i, as desired.

To show uniqueness, let $g(x)$ be another polynomial that passes through the same set of points given in the theorem. We will show that $f = g$. Examine $f - g$. It is a polynomial with degree at most n which has all of the $n + 1$ values x_i as roots. By Theorem 2.3, we conclude that $f - g$ is the zero polynomial, or equivalently that $f = g$.

$\square$

We spent quite a few pages expanding the details of a ten-line proof. This is par for the course. When you encounter a mysterious or overly brief theorem or proof it becomes your job to expand and clarify it as needed. Much like with reading programs written by others, as your mathematical background and experience grows you'll need less work to fill in the details.

Now that we've shown the existence and uniqueness of a degree at most n polynomial passing through a given list of $n + 1$ points, we're allowed to give "it" a name. It's called the *interpolating polynomial* of the given points. The verb *interpolate* means to take a list of points and find the unique minimum-degree polynomial passing through them.

2.4 Realizing it in Code

Let's write a Python program that computes the interpolating polynomial. I'm going to assume the existence of a polynomial class that accepts as input a list of coefficients (in the same order as Definition 2.1, starting from the degree zero term) and has methods for adding, multiplying, and evaluating at a given value. All of this code, including the polynomial class, is available at this book's Github repository.[9] Note the polynomial class is not intended to be perfect. The goal is not to be industry-strength, but to help you understand the constructions we've seen in the chapter.

Here are some examples of constructing polynomials.

```
# special syntax for the zero polynomial
ZERO = Polynomial([])

f = Polynomial([1, 2, 3]) # 1 + 2 x + 3 x^2
g = Polynomial([-8, 17, 0, 5]) # -8 + 17 x + 5 x^3

f + g == Polynomial([-7, 19, 3, 5])
f(1) == 6
```

Now we write the main interpolate function. It uses the yet-to-be-defined function `single_term` that computes a single term of the interpolating polynomial for a given degree. Note we use Python list comprehensions, for which `[EXPRESSION for x in my_list]` is a shorthand expression for the following.

```
output_list = []

for x in my_list:
  output_list.append(EXPRESSION)

# the list comprehension expression evaluates to this list
output_list
```

Now the interpolate function:

[9] See `pimbook.org`.

```python
def interpolate(points):
    """ Return the unique polynomial of degree at most n passing
        through the given n+1 points.
    """
    if len(points) == 0:
        raise ValueError('Must provide at least one point.')

    x_values = [p[0] for p in points]
    if len(set(x_values)) < len(x_values):
        raise ValueError('Not all x values are distinct.')

    terms = [single_term(points, i) for i in range(0, len(points))]
    return sum(terms, ZERO)
```

The first two blocks check for the edge cases: an empty input or repeating x-values. The last block creates a list of terms of the sum from the proof of Theorem 2.2. The return statement sums all the terms, using the zero polynomial as the starting value. Now for the `single_term` function.

```python
def single_term(points, i):
    """ Return one term of an interpolated polynomial.

    Arguments:
      - points: a list of (float, float)
      - i: an integer indexing a specific point
    """
    the_term = Polynomial([1.])
    xi, yi = points[i]

    for j, p in enumerate(points):
        if j == i:
            continue
        xj = p[0]
        the_term = the_term * Polynomial(
            [-xj / (xi - xj), 1.0 / (xi - xj)]
        )

    return the_term * Polynomial([yi])
```

We had to break up the degree-1 polynomial $(x - x_j)/(x_i - x_j)$ into its coefficients, which are $a_0 = -x_j/(x_i - x_j)$ and $a_1 = 1/(x_i - x_j)$. The rest computes the product over the relevant terms. Some examples:

```
>>> points1 = [(1, 1)]
>>> points2 = [(1, 1), (2, 0)]
>>> points3 = [(1, 1), (2, 4), (7, 9)]
>>> interpolate(points1)
1.0
>>> interpolate(points2)
2.0 + -1.0 x^1
>>> f = interpolate(points3)
>>> f
-2.666666666666666 + 3.9999999999999996 x^1 + -0.3333333333333334 x^2
>>> [f(xi) for (xi, yi) in points3]
[1.0, 3.999999999999999, 8.999999999999993]
```

Ignoring the rounding errors, we can see the interpolation is correct.

2.5 Application: Sharing Secrets

Next we'll use polynomial interpolation to "share secrets" in a secure way. Here's the scenario. Say I have five daughters, and I want to share a secret with them, represented as a binary string and interpreted as an integer. Perhaps the secret is the key code for a safe which contains my will. The problem is that my daughters are greedy. If I just give them the secret one might do something nefarious, like forge a modified will that leaves her all my riches at the expense of the others.

Moreover, I'm afraid to even give them *part* of the key code. They might be able to brute force the rest and gain access. Any daughter of mine will be handy with a computer. Even worse, three of the daughters might get together with their pieces of the key code, guess the rest, and exclude the other two daughters. So what I really want is a scheme that has the following properties.

1. Each daughter gets a "share," i.e., some string unique to them.
2. If four of the daughters collude without the fifth, they cannot use their shares to reconstruct the secret.
3. If all five of the daughters combine their shares, they can reconstruct the secret.

In fact, I'd be happier if I could prove, not only that any four out of the five daughters couldn't pool their shares to determine the secret, but that they'd provably have *no information at all* about the secret. They can't even determine a single bit of information about the secret, and they'd have an easier time breaking open the safe with a jackhammer.

The magical fact is that there is such a scheme. Not only is it possible, but it's possible no matter how many daughters I have (say, n), and no matter what minimum size group I want to allow to reconstruct the secret (say, $k \leq n$). So I might have 20 daughters, and I may want any 14 of them to be able to reconstruct the secret, but prevent any group of 13 or fewer from doing so.

Polynomial interpolation gives us all of these guarantees. Here is the scheme. First represent the secret s as an integer. Construct a random polynomial $f(x)$ so that $f(0) = s$. We'll say in a moment what degree d to use for $f(x)$. If we know d, generating f is easy. Call $a_0, \dots, a_d$ the coefficients of f. Set $a_0 = s$ and randomly pick the other coefficients, while ensuring $a_d \neq 0$. If you have n people, the shares you distribute are values of $f(x)$ at $f(1), f(2), \dots, f(n)$. In particular, to person i you give the point $(i, f(i))$.

What do we know about subsets of points? If any k people get together, they can construct the unique degree $k - 1$ polynomial $g(x)$ passing through all those points. The question is, will the resulting $g(x)$ be the same as $f(x)$? If so, they can compute $g(0) = f(0)$ to get the secret! This is where we pick d, to control how many shares are needed. If we want k to be the minimum number of shares needed to reconstruct the secret, we make our polynomial degree $d = k - 1$. Then if k people get together and interpolate $g(x)$, they can appeal to Theorem 2.2 to be sure that $g(x) = f(x)$.

Let's be more explicit and write down an example. Say we have $n = 5$ daughters, and we want any $k = 3$ of them to be able to reconstruct the secret. Pick a polynomial $f(x)$ of degree $d = k - 1 = 2$. If the secret is 109, we generate f as

$$f(x) = 109 + \text{random} \cdot x + \text{random} \cdot x^2$$

Note that if you're going to actually use this to distribute secrets that matter, you need to be a bit more careful about the range of these random numbers. For the sake of this example let's say they're random 10-bit integers, but in reality you'd want to do everything with modular arithmetic. See the Chapter Notes for further discussion.

Next, we distribute one point to each daughter as their share.

$$(1, f(1)), (2, f(2)), (3, f(3)), (4, f(4)), (5, f(5))$$

To give concrete numbers to the examples, if

$$f(x) = 109 - 55x + 271x^2,$$

then the secret is $f(0) = 109$ and the shares are

$$(1, 325), (2, 1083), (3, 2383), (4, 4225), (5, 6609).$$

The polynomial interpolation theorem tells us that with any three points we can completely reconstruct $f(x)$, and then plug in zero to get the secret.

For example, using our polynomial interpolation algorithm, if we feed in the first, third, and fifth shares we reconstruct the polynomial exactly:

```
>>> points = [(1, 325), (3, 2383), (5, 6609)]
>>> interpolate(points)
109.0 + -55.0 x^1 + 271.0 x^2
>>> f = interpolate(points); int(f(0))
109
```

At this point you should be asking yourself: how do I know there's not some other way to get $f(x)$ (or even just $f(0)$) if you have fewer than k points? You should clearly understand the claim being made. It's not just that one can reconstruct $f(0)$ when given enough points on f, but also that *no algorithm* can reconstruct $f(0)$ with fewer than k points.

Indeed it's true, and two little claims show why. Say f is degree d and you have d points (just one fewer than the theorem requires to reconstruct). The first claim is that there are infinitely many different degree d polynomials passing through those same d points. Indeed, if you pick any new x value, say $x = 0$, and any y value, and you add (x, y) to your list of points, then you get an interpolated polynomial for that list whose "decoded secret" is different. Due to Theorem 2.2, each choice of y gives a *different* interpolating polynomial.

The second claim is a consequence of the first. If you only have d points, then not only can $f(0)$ be different, but it can be *anything you want it to be!* For *any* value y that you think might be the secret, there is a choice of a new point that you could add to the list to make y the "correct" decoded value $f(0)$.

Let's think about this last claim. Say your secret is an English sentence s = "Hello, world!" and you encode it with a degree 10 polynomial $f(x)$ so that $f(0)$ is a binary representation of s, and you have the shares $f(1), \dots, f(10)$. Let y be the binary representation of the string "Die, rebel scum!" Then I can take those same 10 points, $f(1), f(2), \dots, f(10)$, and I can make a polynomial passing through them *and* for which $y = f(0)$. In other words, your knowledge of the 10 points gives you no information to distinguish between whether the secret is "Hello world!" or "Die, rebel scum!" Same goes for the difference between "John is the sole heir" and "Joan is the sole heir," a case in which a single-character difference could change the entire meaning of the message.

To drive this point home, let's go back to our small example secret 109 and encoded polynomial

$$f(x) = 109 - 55x + 271x^2$$

I give you just two points, $(2, 1083), (5, 6609)$, and a desired "fake" decrypted message, 533. The claim is that I can come up with a polynomial that has $f(2) = 1083$ and $f(5) = 6609$, and also $f(0) = 533$. Indeed, we already wrote the code to do this! Figure 2.3 demonstrates this with four different "decoded secrets."

```
>>> points = [(2, 1083), (5, 6609)]
>>> interpolate(points + [(0, 533)])
533.0 + -351.7999999999999 x^1 + 313.4 x^2
>>> f = interpolate(points + [(0, 533)]); int(f(0))
533.0
```

Note that the coefficients of the fake secret polynomial are no longer integers, but this problem is fixed when you do everything with modular arithmetic instead of floating point numbers (again, see the Chapter Notes).

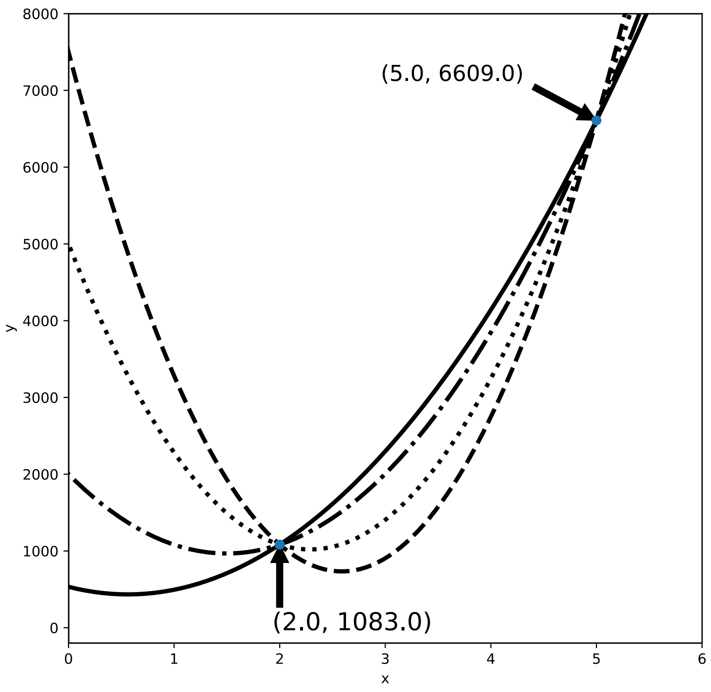

Figure 2.3: A plot of four different curves that agree on the two points $(2, 1083)$, $(5, 6609)$, but have a variety of different "decoded secret" values.

The property of being able to "decode" to any possible plaintext given an encrypted text is called *perfect secrecy*, and it's an early topic on a long journey through mathematical cryptography.

2.6 Cultural Review

1. Whenever you see a definition, you must immediately write down examples. They are your test cases and form a foundation for intuition.

2. In mathematics, we place a special emphasis on the communication of ideas from human to human.

3. A mathematical concept usually has multiple definitions. We prefer to work with the conceptual definition that is easiest to maintain in our minds, and we often don't say when we switch between two representations.

2.7 Exercises

2.1. Prove the following:

1. If f is a degree-2 polynomial and g is a degree-1 polynomial, then their product $f \cdot g$ is a degree-3 polynomial.
2. Generalize the above: if f is a degree-n polynomial and g is a degree-m polynomial, then their product $f \cdot g$ has degree $n + m$.
3. Does the above fact work when f or g are the zero polynomial, using our convention that the zero polynomial has degree -1? If not, can you think of a better convention?
4. Prove that two polynomial formulas with different degrees cannot be equal as functions. That is, there must be some input on which they disagree.

2.2. Write down examples for the following definitions:

1. Two integers a, b are said to be *relatively prime* if their only common divisor is 1. Let n be a positive integer, and define by $\varphi(n)$ (for $n > 1$) the number of positive integers less than n that are relatively prime to n. Describe why one might reasonably add the restriction $n > 1$.
2. A polynomial is called *monic* if its leading coefficient a_n is 1.
3. A *factor* of a polynomial f is a polynomial g of smaller degree so that $f(x) = g(x)h(x)$, for some polynomial h. It is said that f can be "factored" into g and h. Note that g and h must both have real coefficients and be of smaller degree than f.
4. Two polynomials are called *relatively prime* if they have no non-constant smaller-degree polynomial factors in common. A polynomial is called *irreducible* if it cannot be factored into smaller polynomials. The *greatest common divisor* of two polynomials f, g is the monic polynomial of largest degree that is a factor of both f and g.

2.3. Verify the following theorem using the examples from the previous exercise. That is, write down examples and check that the theorem works as stated. If a, n are relatively prime integers, then $a^{\varphi(n)}$ has remainder 1 when dividing by n. This result is known as Euler's theorem (pronounced "OY-lurr"), and it is the keystone of the RSA cryptosystem.

2.4. Look up Horner's method for evaluating a polynomial as a function. Implement a polynomial data structure that uses Horner's method for evaluation, and compare its runtime against naive evaluation methods.

2.5. A number x is called *algebraic* if it is the root of a polynomial whose coefficients are rational numbers (fractions of integers). Otherwise it is called *transcendental.* Numbers like $\sqrt{2}$ are algebraic, while numbers like π and e are famously not algebraic. The golden ratio is the number $\phi = \frac{1+\sqrt{5}}{2}$. Is it algebraic? What about $\sqrt{2} + \sqrt{3}$?

2.6. Prove the product of two algebraic numbers is algebraic. Similarly (but much harder), prove the sum of two algebraic numbers is algebraic. Despite the fact that π

and e are *not* algebraic, it is not known whether $\pi + e$ or πe are algebraic. Look up a proof that they cannot *both* be algebraic. Note that many such proofs appeal to vector spaces, the topic of Chapter 10.

2.7. Let $f(x) = a_0 + a_1 x + \cdots + a_n x^n$ be a degree n polynomial, and suppose it has n real roots $r_1, \dots, r_n$.[10] Prove Vieta's formulas, which are

$$\sum_{i=1}^{n} r_i = -\frac{a_{n-1}}{a_n}$$
$$\prod_{i=1}^{n} r_i = (-1)^n \frac{a_0}{a_n}.$$

Hint: if r is a root, then $f(x)$ can be written as $f(x) = (x-r)g(x)$ for some smaller degree $g(x)$. This formula shows one way the coefficients of a polynomial encode information about the roots.

2.8. Look up a proof of Theorem 2.3. There are many different proofs. Either read one and understand it using the techniques we described in this chapter (writing down examples and tests), or, if you cannot, then write down the words in the proofs that you don't understand and look for them later in this book.

2.9. There are many ways to skin a cat. The polynomial interpolation construction from this chapter is just one, often called *Lagrange interpolation*. Another is called *Newton interpolation*. Find a source that explains what it is, try to understand how these two interpolation methods differ, and implement Newton interpolation. Compare the two interpolation methods in terms of efficiency.

2.10. Bézier curves are single-variable polynomials that draw a curve controlled by a given set of "control points." The polynomial separately controls the x and y coordinates of the Bézier curve, allowing for complex shapes. Look up the definition of quadratic and cubic Bézier curves, and understand how it works. Write a program that computes a generic Bézier curve, and animates how the curve is traced out by the input. Bézier curves are most commonly seen in vector graphics and design applications as the "pen tool."

2.11. It is a natural question to ask whether the roots of a polynomial f are sensitive to changes in the coefficients of f. Wilkinson's polynomial, defined below, shows that they are

$$w(x) = \prod_{i=1}^{20} (x - i)$$

[10] This also works for possibly complex roots.

The coefficient of x^{19} in $w(x)$ is -210, and if it's decreased by 2^{-23} the position of many of the roots change by more than 0.5. Read more details online, and find an explanation of why this polynomial is so sensitive to changes in its coefficients.[11]

2.12. Write a web app that implements the distribution and reconstruction of the secret sharing protocol using the polynomial interpolation algorithm presented in this chapter, using modular arithmetic with a 32-bit modulus p.

2.13. The extended Euclidean algorithm computes the greatest common divisor of two numbers, but it also works for polynomials. Write a program that implements the Euclidean algorithm to compute the greatest common divisor of two monic polynomials. Note that this requires an algorithm to compute polynomial long division as a subroutine.

2.14. The Chinese Remainder Theorem is stated as follows. Suppose $M > 1$ is an integer and $M = m_1 \cdot m_2 \cdots m_k$ where each $m_i > 1$ is an integer. Suppose further that for each i, j, the greatest common divisor of m_i and m_j is 1. Let $r_1, \dots, r_k$ be integers such that $0 \leq r_i < m_i$ (r_i is considered a desired remainder when dividing by m_i). Then there is a unique x with $0 \leq x < M$ such that $x = r_i \mod m_i$ for all i. One can construct the desired number directly, provided one knows how to find multiplicative inverses, and the proof is identical to the polynomial interpolation theorem. Find a source that expands on the details and try to understand them.

2.15. Perhaps the biggest disservice in this chapter is ignoring the so-called Fundamental Theorem of Algebra, that every single-variable monic polynomial of degree k can be factored into linear terms $p(x) = (x - a_1)(x - a_2) \cdots (x - a_k)$. The reason is that the values a_i are not necessarily real numbers. They might be complex. Moreover, all of the proofs of the Fundamental Theorem are quite hard. In fact, one litmus test for the "intellectual potency" of a new mathematical theory is whether it provides a new proof of the Fundamental Theorem of Algebra! There is an entire book dedicated to these often-repeated proofs.[12] Sadly, we avoid complex numbers in this book. Luckily, there is a "baby" fundamental theorem, which says that every single-variable polynomial with real coefficients can be factored into a product of linear and degree-2 terms

$$p(x) = (x - a_1)(x - a_2) \cdots (x - a_m)(x^2 + b_{m+1}x + a_{m+1}) \cdots (x^2 + b_k x + a_k),$$

where none of the quadratic terms can be factored into smaller degree-1 terms. One of history's most famous mathematicians, Carl Friedrich Gauss, provided the first proof as his doctoral thesis in 1799. As part of this exercise, look up some different proofs of the Fundamental Theorem, but instead of trying to understand them, take note of the different areas of math that are used in the proofs.

[11] In "The Perfidious Polynomial," Wilkinson wrote, "I regard [the discovery of this polynomial] as the most traumatic experience in my career as a numerical analyst."
[12] Fine & Rosenberger's "The Fundamental Theorem of Algebra."

2.8 Chapter Notes

Which are Polynomials?

The polynomials were $f(x)$, $g(x)$, $h(x)$, $j(x)$, and $l(x)$. The reason i is not a polynomial is because $\sqrt{x} = x^{1/2}$ does not have an integer power. Similarly, $k(x)$ is not a polynomial because its terms have negative integer powers. Finally, $m(x)$ is not because its powers, π, e, are not integers. Of course, if you were to define π and e to be particular constants that happened to be integers, then the result would be a polynomial. But without any indication, we assume they're the famous constants.

Twin Primes

The Twin Prime Conjecture, the assertion that there are infinitely many pairs of prime numbers of the form $p, p+2$, is one of the most famous open problems in mathematics. Its origin is unknown, though the earliest record of it in print is in the mid 1800's in a text of de Polignac. In an exciting turn of events, in 2013 an unknown mathematician named Yitang Zhang[13] published a breakthrough paper making progress on Twin Primes.

His theorem is not about Twin Primes, but a relaxation of the problem. This is a typical strategy in mathematics: if you can't solve a problem, make the problem easier until you can solve it. Insights and techniques that successfully apply to the easier problem often work, or can be made to work, on the harder problem. Zhang successfully solved the following relaxation of Twin Primes, which had been attempted many times before.

Theorem. *There is a constant M, such that infinitely many primes p exist such that the next prime q after p satisfies $q - p \leq M$.*

If M is replaced with 2, then you get Twin Primes. The thinking is that perhaps it's easier to prove that there are infinitely many primes pairs with distance 6 of each other, or 100. In fact, Zhang's paper established it for M approximately 70 million. But it was the first bound of its kind, and it won Zhang a MacArthur "genius award" in addition to his choice of professorships.

As of this writing, subsequent progress, carried out by some of the world's most famous mathematicians in an online collaboration called the Polymath Project, brought M down to 246. Assuming a conjecture in number theory called the Elliott-Halberstam conjecture, they reduced this constant to 6.

Impossibility of Clustering

A *clustering algorithm* is a program f that takes as input:

- A list of points S,
- A distance function d that describes the distance between two points $d(x, y)$ where x, y are in S,

[13] Though he had a Ph.D, early in his career Zhang had been unable to find academic work, and had stints in a motel, as a delivery driver, and at a Subway sandwich shop before he found a position as a lecturer at the University of New Hampshire.

and produces as output a *clustering* of S, i.e., a choice of how to split S into non-overlapping subsets. The individual subsets are called "clusters."

The function d is also required to have some properties that make it reasonably interpretable as a "distance" function. In particular, all distances are nonnegative, $d(x, y) = d(y, x)$, and the distance between a point and itself is zero.

The Kleinberg Impossibility Theorem for Clustering says that no clustering algorithm f can satisfy all of the following three properties, which he calls *scale-invariance*, *richness*, and *consistency*.[14]

- **Scale-invariance**: The output of f is unchanged if you stretch or shrink all distances in d by the same multiplicative factor.
- **Richness**: Every partition of S is a possible output of f, (for some choice of d).
- **Consistency**: The output of f on input (S, d) is unchanged if you modify d by shrinking the distances between points in the same cluster and enlarging the distances between points in different clusters.

One can interpret this theorem as an explanation (in part) for why clustering is a hard problem. While there are hundreds of clustering algorithms to choose from, none "just works" the way we humans intuitively want one to. This may be, as Kleinberg suggests, because our naive brains expect these three properties to hold, despite the fact that they are mathematically incompatible.

It also suggests that the "right" clustering function depends more on the application you use it for, which raises the question: how can one pick a clustering function with principle?

It turns out, if you allow the required *number* of output clusters to be an input to the clustering algorithm, you can avoid impossibility and instead achieve uniqueness. For more, see the 2009 paper "A Uniqueness Theorem for Clustering" of Zadeh and Ben-David. The authors proceeded to study how to choose a clustering algorithm "in principle" by studying what properties uniquely determine various clustering algorithms; meaning if you want to do clustering in practice, you have to think hard about exactly what properties your application needs from a clustering. Suffice it to say, this process is a superb example of navigating the border separating impossibility, existence, and uniqueness in mathematics.

More on Secret Sharing

The secret sharing scheme presented in this chapter was originally devised by Adi Shamir (the same Shamir of RSA) in a two-page 1979 paper called "How to share a secret." In this paper, Shamir follows the terse style and does not remind the reader how the interpolating polynomial is constructed.

[14] Of incidental interest to readers of this book, Jon Kleinberg also developed an eigenvector-based search ranking algorithm that was a precursor to Google's PageRank algorithm.

He does, however, mention that in order to make this scheme secure, the coefficients of the polynomial must be computed using modular arithmetic. Here's what is meant by that, and note that we'll return to understand this in Chapter 16 from a much more general perspective.

Given an integer n and a *modulus* p (in our case a prime integer), we represent n "modulo" p by replacing it with its remainder when dividing by p. Most programming languages use the % operator for this, so that $a = n\%p$ means a is the remainder of n/p. Note that if $n < p$, then $n\%p = n$ is its own remainder. The standard notation in mathematics is to use the word "mod" and the $\equiv$ symbol (read "is equivalent to" or "is congruent to"), as in

$$a \equiv n \mod p.$$

The syntactical operator precedence is a bit weird here: "mod" is not a binary operation, but rather describes the entire equation, as if to say, "everything here is considered modulo p."

We chose a prime p for the modulus because doing so allows you to "divide." Indeed, for a given n and prime p, there is a unique k such that $(n \cdot k) \equiv 1 \mod p$. Again, an interesting example of existence and uniqueness. Note that it takes some work to find k, and the extended Euclidean algorithm is the standard method. When evaluating a polynomial function like $f(x)$ at a given x, the output is taken modulo p and is guaranteed to be between 0 and p.

Modular arithmetic is important because (1) it's faster than arithmetic on arbitrarily large integers, and (2) when evaluating $f(x)$ at an unknown integer x not modulo p, the size of the output and knowledge of the degree of f can give you some information about the input x. In the case of secret sharing, seeing the sizes of the shares reveals information about the coefficients of the underlying polynomial, and hence information about $f(0)$, the secret. This is unpalatable if we want perfect secrecy.

Moreover, when you use modular arithmetic you can prove that picking a uniformly random $(d + 1)$-th point in the secret sharing scheme will produce a uniformly random decoded "secret" $f(0)$. That is, uniformly random between 0 and p. Without bounding the allowed size of the integers, it doesn't make sense to have a "uniform" distribution. As a consequence, it is harder to define and interpret the security of such a scheme.

Finally, from discussions I've had with people using this scheme in industry, polynomial interpolation is not fast enough for modern applications. For example, one might want to do secret sharing between three parties at streaming-video rates. Rather, one should use so-called "linear" secret sharing schemes, which are based on systems of linear equations. Such schemes are best analyzed from the perspective of linear algebra, the topic of Chapter 10.

Chapter 3

On Pace and Patience

You enter the first room of the mansion and it's completely dark. You stumble around bumping into the furniture but gradually you learn where each piece of furniture is. Finally, after six months or so, you find the light switch, you turn it on, and suddenly it's all illuminated. You can see exactly where you were. Then you move into the next room and spend another six months in the dark. So each of these breakthroughs, while sometimes they're momentary, sometimes over a period of a day or two, they are the culmination of, and couldn't exist without, the many months of stumbling around in the dark that precede them.

–Andrew Wiles on what it's like to do mathematics research.

We learned a lot in the last chapter. One aspect that stands out is just how *slow* the process of learning unfamiliar math can be. I told you that every time you see a definition or theorem, you had to stop and write stuff down to understand it better. But this isn't all that different from programming. Experienced coders know when to fire up a REPL or debugger, or write test programs to isolate how a new feature works.

The main difference for us is that mathematics has no debugger or REPL. There is no reference implementation. Mathematicians often get around this hurdle by conversation, and I encourage you to find a friend to work through this book with. As William Thurston writes in his influential essay, "On Proof and Progress in Mathematics," mathematical knowledge is embedded in the minds and the social fabric of the community of people thinking about a topic. Books and papers support this, but the higher up you go, the farther the primary sources stray from textbooks.

If you are reading this book alone, you have to play the roles of the program writer, the tester, and the compiler. The writer for when you're conjuring new ideas and asking questions; the tester for when you're reading theorems and definitions; and the compiler to check your intuition and hunches for bugs. This often slows reading mathematics down to a crawl, for novices and experts alike. Mathematicians always read with a pencil and notepad handy.

When you first read a theorem, you expect to be confused. Let me say it again: the **rule** is that you are confused, the **exception** is that everything is clear. Mathematical culture requires being comfortable being almost continuously in a state of little to no understanding. It's a humble life, but once you nail down what exactly is unclear, you can make progress toward understanding. The easiest way to do this is by writing down lots of examples, but it's not always possible to do that. We've already seen an example, a theorem about the *impossibility* of having a nonzero polynomial with more roots than its degree.

In the quote at the beginning of this chapter, Andrew Wiles discusses what it's like to do mathematical research, but the same analogy holds for learning mathematics. Speaking with experienced mathematicians and reading their books makes you feel like an idiot. Whatever they're saying is the most basic idea in the world, and you barely stumble along. My favorite dramatic embodiment of this feeling is an episode of a YouTube series called Kid Snippets in which children are asked to pretend to be in a math class, while adult actors act it out using dubbed voices.[1] The older child tries to explain to the younger child how to subtract, and the little kid just doesn't get it. Aside from being absolutely hilarious, the video has a deep and probably unintentional truth, that the more mathematics you try to learn the more you feel like the poor student. The video especially resonates when, toward the end, the teacher asks, "Do you get it now?" and the student pauses and slowly says, "Yes." That yes is the fledgling mathematician saying, "I obviously don't understand, but I've accepted it and will try to understand it later."

I've been in the student's shoes a thousand times. Indeed, if I'm *not* in those shoes at least once a day then it wasn't a productive day! I say at least a dozen stupid things daily and think countlessly many more stupid thoughts in search of insight. It's a rare moment when I think, "I'm going to solve this problem I don't already know how to solve," and there is no subsequent crisis. Even in reading what should be basic mathematical material (there's a huge list of things that I am embarrassed to be ignorant about) I find myself mentally crying out, "How the hell does that statement follow!?"

I had a conversation with an immensely talented colleague, a *far* more talented mathematician than I, in which she said (I paraphrase), "If I spend an entire day and all I do is understand this one feature of this one object that I didn't understand before, then that's a *great* day." We all have to build up insight over time, and it's a slow and arduous process. In Andrew Wiles's analogy, my friend is still in the dark room, but she's feeling some object precisely enough to understand that it's a vase. She still has no idea where the light switch is, and the vase might give her no indication as to where to look next. But if piece by piece she can construct a clear enough picture of the room in her mind, then she will find the switch. What keeps her going is that she knows enough little insights will lead her to a breakthrough worth having.

Though she is working on far more complicated and abstract mathematics than you are likely to, we must all adopt her attitude if we want to learn mathematics. If it sounds like all of this will take *way* too much of your time (all day to learn a single little thing!), remember two things. First, my colleague works on much more abstract and difficult mathematics than the average programmer interested in mathematics would encounter. She's looking for the meta-insights that are many levels above the insights found in this

[1] You can watch it at `http://youtu.be/KdxEAt91D7k`

book. As we'll see in Chapter 11, insights are like a ladder, and every rung is useful. Second, the more you practice reading and absorbing mathematics, the better you get at it. When my colleague says she spent an entire day understanding something, she efficiently applied tools she had built up over time. She has a bank of examples to bolster her. She knows how to cycle through applicable proof techniques, and how to switch between different representations to see if a different perspective helps. Some of these techniques are described in Appendix B.

But most importantly, she's being inquisitive! Her journey is led as much by her task as by her curiosity. As mathematician Paul Halmos said in his book, "I Want to be a Mathematician,"

> *Don't just read it; fight it! Ask your own questions, look for your own examples, discover your own proofs.*

Mathematician Terence Tao expands on this in his essay, "Ask yourself dumb questions—and answer them!"

> *When you learn mathematics, whether in books or in lectures, you generally only see the end product—very polished, clever and elegant presentations of a mathematical topic. However, the process of discovering new mathematics is much messier, full of the pursuit of directions which were naive, fruitless or uninteresting.*
>
> *While it is tempting to just ignore all these "failed" lines of inquiry, actually they turn out to be essential to one's deeper understanding of a topic, and (via the process of elimination) finally zeroing in on the correct way to proceed.*
>
> *So one should be unafraid to ask "stupid" questions, challenging conventional wisdom on a subject; the answers to these questions will occasionally lead to a surprising conclusion, but more often will simply tell you why the conventional wisdom is there in the first place, which is well worth knowing.*

So you'll get confused. We all do. A good remedy is finding the right pace to make steady progress. And when in doubt, start slow.