to mechanical techniques for place-value arithmetic using "Arabic" numerals. People trained in the fast and reliable execution of these procedures were called *algorists* or *computators*, or more simply, *computers*.

0.2 Multiplication

Although they have been a topic of formal academic study for only a few decades, algorithms have been with us since the dawn of civilization. Descriptions of step-by-step arithmetic computation are among the earliest examples of written human language, long predating the expositions by Fibonacci and al-Khwārizmī, or even the place-value notation they popularized.

Lattice Multiplication

The most familiar method for multiplying large numbers, at least for American students, is the ***lattice algorithm***. This algorithm was popularized by Fibonacci in *Liber Abaci*, who learned it from Arabic sources including al-Khwārizmī, who in turn learned it from Indian sources including Brahmagupta's 7th-century treatise *Brāhmasphuṭasiddhānta*, who may have learned it from Chinese sources. The oldest surviving descriptions of the algorithm appear in *The Mathematical Classic of Sunzi*, written in China between the 3rd and 5th centuries, and in Eutocius of Ascalon's commentaries on Archimedes' *Measurement of the Circle*, written around 500CE, but there is evidence that the algorithm was known much earlier. Eutocius credits the method to a lost treatise of Apollonius of Perga, who lived around 300BCE, entitled *Okytokion* (Ὠκυτόκιον).[7] The Sumerians recorded multiplication tables on clay tablets as early as 2600BCE, suggesting that they may have used the lattice algorithm.[8]

The lattice algorithm assumes that the input numbers are represented as explicit strings of digits; I'll assume here that we're working in base ten, but the algorithm generalizes immediately to any other base. To simplify notation,[9] the

[7]Literally "medicine that promotes quick and easy childbirth"! Pappus of Alexandria reproduced several excerpts of *Okytokion* about 200 years before Eutocius, but his description of the lattice multiplication algorithm (if he gave one) is *also* lost.

[8]There is ample evidence that ancient Sumerians calculated accurately with extremely large numbers using their base-60 place-value numerical system, but I am not aware of any surviving record of the actual methods they used. In addition to standard multiplication and reciprocal tables, tables listing the squares of integers from 1 to 59 have been found, leading some math historians to conjecture that Babylonians multiplied using an identity like $xy = ((x+y)^2 - x^2 - y^2)/2$. But this trick only works when $x + y < 60$; history is silent on how the Babylonians might have computed x^2 when $x \geq 60$.

[9]but at the risk of inflaming the historical enmity between Greece and Egypt, or Lilliput and Blefuscu, or Macs and PCs, or people who think zero is a natural number and people who are wrong

input consists of a pair of arrays $X[0\,..\,m-1]$ and $Y[0\,..\,n-1]$, representing the numbers

$$x = \sum_{i=0}^{m-1} X[i] \cdot 10^i \quad \text{and} \quad y = \sum_{j=0}^{n-1} Y[j] \cdot 10^j,$$

and similarly, the output consists of a single array $Z[0\,..\,m+n-1]$, representing the product

$$z = x \cdot y = \sum_{k=0}^{m+n-1} Z[k] \cdot 10^k.$$

The algorithm uses addition and *single-digit* multiplication as primitive operations. Addition can be performed using a simple for-loop. In practice, single-digit multiplication is performed using a lookup table, either carved into clay tablets, painted on strips of wood or bamboo, written on paper, stored in read-only memory, or memorized by the computator. The entire lattice algorithm can be summarized by the formula

$$x \cdot y \;=\; \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \bigl(X[i] \cdot Y[j] \cdot 10^{i+j}\bigr).$$

Different variants of the lattice algorithm evaluate the partial products $X[i] \cdot Y[j] \cdot 10^{i+j}$ in different orders and use different strategies for computing their sum. For example, in *Liber Abaco*, Fibonacci describes a variant that considers the mn partial products in increasing order of significance, as shown in modern pseudocode below.

```
FibonacciMultiply($X[0\,..\,m-1], Y[0\,..\,n-1]$):
  $hold \leftarrow 0$
  for $k \leftarrow 0$ to $n+m-1$
      for all $i$ and $j$ such that $i+j=k$
          $hold \leftarrow hold + X[i] \cdot Y[j]$
      $Z[k] \leftarrow hold \bmod 10$
      $hold \leftarrow \lfloor hold/10 \rfloor$
  return $Z[0\,..\,m+n-1]$
```

Fibonacci's algorithm is often executed by storing all the partial products in a two-dimensional table (often called a "tableau" or "grate" or "lattice") and then summing along the diagonals with appropriate carries, as shown on the right in Figure 0.1. American elementary-school students are taught to multiply one factor (the "multiplicand") by each digit in the other factor (the "multiplier"), writing down all the multiplicand-by-digit products before adding them up, as shown on the left in Figure 0.1. This was also the method described by Eutocius, although he fittingly considered the multiplier digits from left to right, as shown in Figure 0.2. Both of these variants (and several others) are described and illustrated side by side in the anonymous 1458 textbook *L'Arte dell'Abbaco*, also known as the *Treviso Arithmetic*, the first *printed* mathematics book in the West.

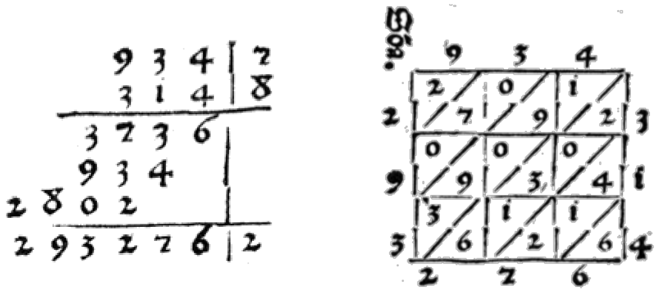

Figure 0.1. Computing 934 × 314 = 293276 using "long" multiplication (with error-checking by casting out nines) and "lattice" multiplication, from *L'Arte dell'Abbaco* (1458). (See Image Credits at the end of the book.)

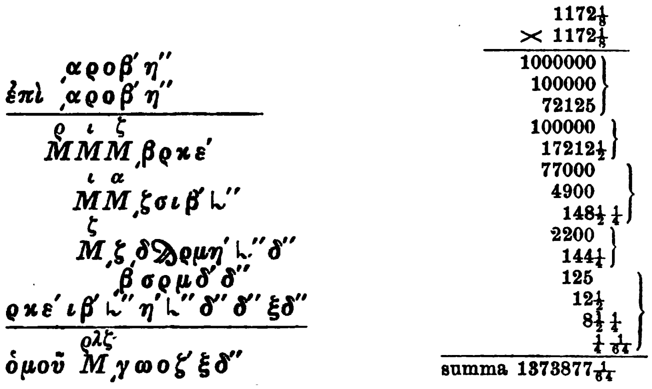

Figure 0.2. Eutocius's 6th-century calculation of $1178\frac{1}{8} \times 1178\frac{1}{8} = 1373877\frac{1}{64}$, in his commentary on Archimedes' *Measurement of the Circle*, transcribed (left) and translated into modern notation (right) by Johan Heiberg (1891). (See Image Credits at the end of the book.)

All of these variants of the lattice algorithm—and other similar variants described by Sunzi, al-Khwārizmī, Fibonacci, *L'Arte dell'Abbaco*, and many other sources—compute the product of any m-digit number and any n-digit number in ***O*(*mn*) *time***; the running time of every variant is dominated by the number of single-digit multiplications.

Duplation and Mediation

The lattice algorithm is not the oldest multiplication algorithm for which we have direct recorded evidence. An even older and arguably simpler algorithm, which does not rely on place-value notation, is sometimes called *Russian peasant multiplication*, *Ethiopian peasant multiplication*, or just ***peasant multiplication***.

A variant of this algorithm was copied into the Rhind papyrus by the Egyptian scribe Ahmes around 1650BC, from a document he claimed was (then) about 350 years old.[10] This algorithm was still taught in elementary schools in Eastern Europe in the late 20th century; it was also commonly used by early digital computers that did not implement integer multiplication directly in hardware.

The peasant multiplication algorithm reduces the difficult task of multiplying arbitrary numbers to a sequence of four simpler operations: (1) determining parity (even or odd), (2) addition, (3) ***duplation*** (doubling a number), and (4) ***mediation*** (halving a number, rounding down).

```
PeasantMultiply($x, y$):
  $prod \leftarrow 0$
  while $x > 0$
      if $x$ is odd
          $prod \leftarrow prod + y$
      $x \leftarrow \lfloor x/2 \rfloor$
      $y \leftarrow y + y$
  return $prod$
```

x	y		$prod$
			0
123	+ 456	=	456
61	+ 912	=	1368
30	~~1824~~		
15	+ 3648	=	5016
7	+ 7296	=	12312
3	+ 14592	=	26904
1	+ 29184	=	**56088**

Figure 0.3. Multiplication by duplation and mediation

The correctness of this algorithm follows by induction from the following recursive identity, which holds for all non-negative integers x and y:

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

Arguably, this recurrence *is* the peasant multiplication algorithm!

As stated, PeasantMultiply performs $O(\log x)$ parity, addition, and mediation operations, but we can improve this bound to $O(\log \min\{x, y\})$ by swapping the two arguments when $x > y$. Assuming the numbers are represented using any reasonable place-value notation (like binary, decimal, Babylonian hexagesimal, Egyptian duodecimal, Roman numeral, Chinese counting rods, bead positions on an abacus, and so on), each operation requires at most $O(\log(xy)) = O(\log \max\{x, y\})$ single-digit operations, so the overall running time of the algorithm is $O(\log \min\{x, y\} \cdot \log \max\{x, y\}) = O(\log x \cdot \log y)$.

In other words, this algorithm requires ***$O(mn)$ time*** to multiply an m-digit number by an n-digit number; up to constant factors, this is the same running

[10]The version of this algorithm actually used in ancient Egypt does not use mediation or parity, but it does use comparisons. To avoid halving, the algorithm pre-computes two tables by repeated doubling: one containing all the powers of 2 not exceeding x, the other containing the same powers of 2 multiplied by y. The powers of 2 that sum to x are then found by greedy subtraction, and the corresponding entries in the other table are added together to form the product.

time as the lattice algorithm. This algorithm requires (a constant factor!) more paperwork to execute by hand than the lattice algorithm, but the necessary primitive operations are arguably easier for humans to perform. In fact, the two algorithms are equivalent when numbers are represented in binary.

Compass and Straightedge

Classical Greek geometers identified numbers (or more accurately, *magnitudes*) with line segments of the appropriate length, which they manipulated using two simple mechanical tools—the compass and the straightedge—versions of which had already been in common use by surveyors, architects, and other artisans for centuries. Using *only* these two tools, these scholars reduced several complex geometric constructions to the following primitive operations, starting with one or more identified reference points.

- Draw the unique line passing through two distinct identified points.
- Draw the unique circle centered at one identified point and passing through another.
- Identify the intersection point (if any) of two lines.
- Identify the intersection points (if any) of a line and a circle.
- Identify the intersection points (if any) of two circles.

In practice, Greek geometry students almost certainly drew their constructions on an *abax* (ἄβαξ), a table covered in dust or sand.[11] Centuries earlier, Egyptian surveyors carried out many of the same constructions using ropes to determine straight lines and circles on the ground.[12] However, Euclid and other Greek geometers presented compass and straightedge constructions as precise mathematical *abstractions*—points are *ideal* points; lines are *ideal* lines; and circles are *ideal* circles.

Figure 0.4 shows an algorithm, described in Euclid's *Elements* about 2500 years ago, for multiplying or dividing two magnitudes. The input consists of four distinct points A, B, C, D, and the goal is to construct a point Z such that $|AZ| = |AC||AD|/|AB|$. In particular, if we define $|AB|$ to be our unit of length, then the algorithm computes the product of $|AC|$ and $|AD|$.

Notice that Euclid first defines a new primitive operation RightAngle by (as modern programmers would phrase it) writing a subroutine. The correctness of the algorithm follows from the observation that triangles ACE and AZF are

[11]The written numerals 1 through 9 were known in Europe at least two centuries before Fibonacci's *Liber Abaci* as "gobar numerals", from the Arabic word *ghubār* meaning dust, ultimately referring to the Indian practice of performing arithmetic on tables covered with sand. The Greek word ἄβαξ is the origin of the Latin *abacus*, which also originally referred to a sand table.

[12]Remember what "geometry" means? Democritus would later refer to these Egyptian surveyors, somewhat derisively, as *arpedonaptai* (ἀρπεδονάπται), meaning "rope-fasteners".

```
⟨⟨Construct the line perpendicular to $\ell$ passing through $P$.⟩⟩
RightAngle($\ell, P$):
    Choose a point $A \in \ell$
    $A, B \leftarrow$ Intersect(Circle($P, A$), $\ell$)
    $C, D \leftarrow$ Intersect(Circle($A, B$), Circle($B, A$))
    return Line($C, D$)

⟨⟨Construct a point $Z$ such that $|AZ| = |AC||AD|/|AB|$.⟩⟩
MultiplyOrDivide($A, B, C, D$):
    $\alpha \leftarrow$ RightAngle(Line($A, C$), $A$)
    $E \leftarrow$ Intersect(Circle($A, B$), $\alpha$)
    $F \leftarrow$ Intersect(Circle($A, D$), $\alpha$)
    $\beta \leftarrow$ RightAngle(Line($E, C$), $F$)
    $\gamma \leftarrow$ RightAngle($\beta, F$)
    return Intersect($\gamma$, Line($A, C$))
```

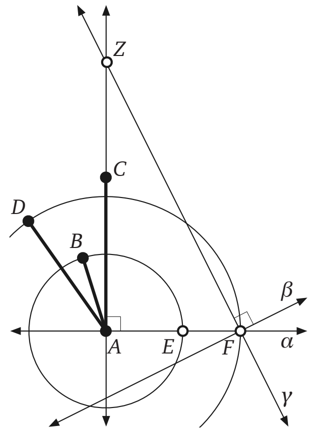

Figure 0.4. Multiplication by compass and straightedge.

similar. The second and third lines of the main algorithm are ambiguous, because α intersects any circle centered at A at *two* distinct points, but the algorithm is actually correct no matter which intersection points are chosen for E and F.

Euclid's algorithm reduces the problem of multiplying two magnitudes (lengths) to a series of primitive compass-and-straightedge operations. These operations are difficult to implement precisely on a modern digital computer, but Euclid's algorithm wasn't *designed* for a digital computer. It was designed for the Platonic Ideal Geometer, wielding the Platonic Ideal Compass and the Platonic Ideal Straightedge, who could execute each operation perfectly in constant time *by definition*. In this model of computation, MultiplyOrDivide runs in $O(1)$ time!

0.3 Congressional Apportionment

Here is another real-world example of an algorithm of significant political importance. Article I, Section 2 of the United States Constitution requires that

> Representatives and direct Taxes shall be apportioned among the several States which may be included within this Union, according to their respective Numbers.... The Number of Representatives shall not exceed one for every thirty Thousand, but each State shall have at Least one Representative....

Because there are only a finite number of seats in the House of Representatives, *exact* proportional representation requires either shared or fractional representatives, neither of which are legal. As a result, over the next several decades, many different apportionment algorithms were proposed and used to round the ideal fractional solution fairly. The algorithm actually used today, called the ***Huntington-Hill method*** or the ***method of equal proportions***, was first suggested by Census Bureau statistician Joseph Hill in 1911, refined by Harvard mathematician Edward Huntington in 1920, adopted into Federal law (2 U.S.C. §2a) in 1941, and survived a Supreme Court challenge in 1992.[13]

The Huntington-Hill method allocates representatives to states one at a time. First, in a preprocessing stage, each state is allocated one representative. Then in each iteration of the main loop, the next representative is assigned to the state with the highest *priority*. The priority of each state is defined to be $P/\sqrt{r(r+1)}$, where P is the state's population and r is the number of representatives already allocated to that state.

The algorithm is described in pseudocode in Figure 0.5. The input consists of an array $Pop[1..n]$ storing the populations of the n states and an integer R equal to the total number of representatives. (Currently, in the United States, $n = 50$ and $R = 435$.) The output array $Rep[1..n]$ records the number of representatives allocated to each state.

```
ApportionCongress($Pop[1..n], R$):
    $PQ \leftarrow$ NewPriorityQueue
    for $i \leftarrow 1$ to $n$
        $Rep[i] \leftarrow 1$
        Insert$\left(PQ, i, Pop[i]/\sqrt{2}\right)$
        $R \leftarrow R - 1$

    for $i \leftarrow n+1$ to $R$
        $s \leftarrow$ ExtractMax($PQ$)
        $Rep[s] \leftarrow Rep[s] + 1$
        $priority \leftarrow Pop[s] \big/ \sqrt{Rep[s]\,(Rep[s]+1)}$
        Insert($PQ, s, priority$)

    return $Rep[1..n]$
```

Figure 0.5. The Huntington-Hill apportionment algorithm

This implementation of Huntington-Hill uses a priority queue that supports the operations NewPriorityQueue, Insert, and ExtractMax. (The actual law doesn't say anything about priority queues, of course.) The output of the algorithm, and therefore its correctness, does not depend *at all* on how this priority queue is implemented. The Census Bureau uses a sorted array, stored in a single column of an Excel spreadsheet, which is recalculated from scratch

[13]Overruling an earlier ruling by a federal district court, the Supreme Court unanimously held that ***any*** apportionment method adopted in good faith by Congress is constitutional (*United States Department of Commerce v. Montana*). The current congressional apportionment algorithm is described in gruesome detail at the U.S. Census Department web site http://www.census.gov/topics/public-sector/congressional-apportionment.html. A good history of the apportionment problem can be found at http://www.thirty-thousand.org/pages/Apportionment.htm. A report by the Congressional Research Service describing various apportionment methods is available at http://www.fas.org/sgp/crs/misc/R41382.pdf.

at every iteration. You (should have) learned a more efficient implementation in your undergraduate data structures class.

Similar apportionment algorithms are used in multi-party parliamentary elections around the world, where the number of seats allocated to each party is supposed to be proportional to the number of votes that party receives. The two most common are the *D'Hondt method*[14] and the *Webster–Sainte-Laguë method*,[15] which respectively use priorities $P/(r+1)$ and $P/(2r+1)$ in place of the square-root expression in Huntington-Hill. The Huntington-Hill method is essentially unique to the United States House of Representatives, thanks in part to the constitutional requirement that each state must be allocated at least one representative.

0.4 A Bad Example

As a prototypical example of a sequence of instructions that is *not* actually an algorithm, consider "Martin's algorithm":[16]

```
BeAMillionaireAndNeverPayTaxes( ):
  Get a million dollars.
  If the tax man comes to your door and says, "You have never paid taxes!"
      Say "I forgot."
```

Pretty simple, except for that first step; it's a doozy. A group of billionaire CEOs, Silicon Valley venture capitalists, or New York City real-estate hustlers might consider this an algorithm, since for them the first step is both unambiguous and trivial,[17] but for the rest of us poor slobs, Martin's procedure is too vague to be considered an actual algorithm. On the other hand, this is a perfect example of a ***reduction***—it *reduces* the problem of being a millionaire and never paying taxes to the "easier" problem of acquiring a million dollars. We'll see reductions over and over again in this class. As hundreds of businessmen and politicians have demonstrated, if you know how to solve the easier problem, a reduction tells you how to solve the harder one.

Martin's algorithm, like some of our previous examples, is not the kind of algorithm that computer scientists are used to thinking about, because it

[14]developed by Thomas Jefferson in 1792, used for U.S. Congressional apportionment from 1792 to 1832, rediscovered by Belgian mathematician Victor D'Hondt in 1878, and refined by Swiss physicist Eduard Hagenbach-Bischoff in 1888.

[15]developed by Daniel Webster in 1832, used for U.S. Congressional apportionment from 1842 to 1911, rediscovered by French mathematician André Sainte-Laguë in 1910, and rediscovered again by German physicist Hans Schepers in 1980.

[16]Steve Martin, "You Can Be A Millionaire", *Saturday Night Live*, January 21, 1978. Also appears on *Comedy Is Not Pretty*, Warner Bros. Records, 1979.

[17]Something something secure quantum blockchain deep-learning something.

is phrased in terms of operations that are difficult for computers to perform. This book focuses (almost!) exclusively on algorithms that can be reasonably implemented on a standard digital computer. Each step in these algorithms is either directly supported by common programming languages (such as arithmetic, assignments, loops, or recursion) or something that you've already learned how to do (like sorting, binary search, tree traversal, or singing "*n* Bottles of Beer on the Wall").

0.5 Describing Algorithms

The skills required to effectively *design and analyze* algorithms are entangled with the skills required to effectively *describe* algorithms. At least in my classes, a complete description of any algorithm has four components:

- **What:** A precise specification of the problem that the algorithm solves.
- **How:** A precise description of the algorithm itself.
- **Why:** A proof that the algorithm solves the problem it is supposed to solve.
- **How fast:** An analysis of the running time of the algorithm.

It is not necessary (or even advisable) to *develop* these four components in this particular order. Problem specifications, algorithm descriptions, correctness proofs, and time analyses usually evolve simultaneously, with the development of each component informing the development of the others. For example, we may need to tweak the problem description to support a faster algorithm, or modify the algorithm to handle a tricky case in the proof of correctness. Nevertheless, *presenting* these components separately is usually clearest for the reader.

As with any writing, it's important to aim your descriptions at the right audience; I recommend writing for a competent but skeptical programmer *who is not as clever as you are*. Think of yourself six months ago. As you develop any new algorithm, you will naturally build up lots of intuition about the problem and about how your algorithm solves it, and your informal reasoning will be guided by that intuition. But anyone *reading* your algorithm later, or the code you derive from it, won't share your intuition or experience. Neither will your compiler. Neither will you six months from now. All they will have is your written description.

Even if you never have to explain your algorithms to anyone else, it's still important to develop them with an audience in mind. Trying to communicate clearly forces you to *think* more clearly. In particular, writing for a *novice* audience, who will interpret your words *exactly* as written, forces you to work through fine details, no matter how "obvious" or "intuitive" your high-level ideas may seem at the moment. Similarly, writing for a *skeptical* audience forces you to develop robust arguments for correctness and efficiency, instead of trusting your intuition or your intelligence.[18]

I cannot emphasize this point enough: **Your primary job as an algorithm designer is *teaching other people* how and why your algorithms work.** If you can't communicate your ideas to other human beings, they may as well not exist. Producing correct and efficient executable code is an important but secondary goal. Convincing yourself, your professors, your (prospective) employers, your colleagues, or your students that you are smart is at best a distant third.

Specifying the Problem

Before we can even start developing a new algorithm, we have to agree on what problem our algorithm is supposed to solve. Similarly, before we can even start *describing* an algorithm, we have to *describe* the problem that the algorithm is supposed to solve.

Algorithmic problems are often presented using standard English, in terms of real-world objects, not in terms of formal mathematical objects. It's up to us, the algorithm designers, to restate these problems in terms of abstract mathematical objects—numbers, arrays, lists, graphs, trees, and so on—that we can reason about formally. We must also determine if the problem statement carries any hidden assumptions, and state those assumptions explicitly. (For example, in the song "n Bottles of Beer on the Wall", n is always a non-negative integer.[19])

We may need to refine our specification as we develop the algorithm. For example, our algorithm may require a particular input representation, or produce a particular output representation, that was left unspecified in the original informal problem description. Or our algorithm might actually solve a *more general* problem than we were originally asked to solve. (This is a common feature of recursive algorithms.)

The specification should include just enough detail that someone else could *use* our algorithm as a black box, without knowing how or why the algorithm actually works. In particular, we must describe the type *and meaning* of each input parameter, and exactly how the eventual output depends on the input parameters. On the other hand, our specification should *deliberately hide* any details that are *not* necessary to use the algorithm as a black box. Let that which does not matter truly slide.

For example, the lattice and duplation-and-mediation algorithms both solve the same problem: Given two non-negative integers x and y, each represented

[18]In particular, I assume that *you* are a skeptical novice!

[19]I've never heard anyone sing "$\sqrt{2}$ Bottles of Beer on the Wall." Occasionally I *have* heard set theorists singing "$\aleph_0$ bottles of beer on the wall", but for some reason they always gave up before the song was over.

as an array of digits, compute the product $x \cdot y$, also represented as an array of digits. To someone *using* these algorithms, the choice of algorithm is completely irrelevant. On the other hand, the Greek straightedge-and-compass algorithm solves a *different problem*, because the input and output values are represented by line segments instead of arrays of digits.

Describing the Algorithm

Computer programs are concrete representations of algorithms, but algorithms are *not* programs. Rather, algorithms are abstract mechanical procedures that can be implemented in *any* programming language that supports the underlying primitive operations. The idiosyncratic syntactic details of your favorite programming language are utterly irrelevant; focusing on these will only distract you (and your readers) from what's *really* going on.[20] A good algorithm description is closer to what we should write in the *comments* of a real program than the code itself. Code is a poor medium for storytelling.

On the other hand, a plain English prose description is usually not a good idea either. Algorithms have lots of idiomatic structure—especially conditionals, loops, function calls, and recursion—that are far too easily hidden by unstructured prose. Colloquial English is full of ambiguities and shades of meaning, but algorithms must be described as unambiguously as possible. Prose is a poor medium for precision.

In my opinion, the clearest way to present an algorithm is using a combination of *pseudo*code and structured English. Pseudocode uses the *structure* of formal programming languages and mathematics to break algorithms into primitive steps; the primitive steps themselves can be written using mathematical notation, pure English, or an appropriate mixture of the two, *whatever is clearest*. Well-written pseudocode reveals the internal structure of the algorithm but hides irrelevant implementation details, making the algorithm easier to understand, analyze, debug, and implement.

Whenever we describe an algorithm, our description should include every detail necessary to fully specify the algorithm, prove its correctness, and analyze

[20]This is, of course, a matter of religious conviction. Linguists argue incessantly over the *Sapir-Whorf hypothesis*, which states (more or less) that people think only in the categories imposed by their languages. According to an extreme formulation of this principle, some concepts in one language simply cannot be understood by speakers of other languages, not just because of technological advancement—How would you translate "jump the shark" or "Fortnite streamer" into Aramaic?—but because of inherent structural differences between languages and cultures. For a more skeptical view, see Steven Pinker's *The Language Instinct*. There is admittedly some strength to this idea when applied to different programming paradigms. (What's the Y combinator, again? How do templates work? What's an Abstract Factory?) Fortunately, those differences are too subtle to have any impact on the material in *this* book. For a compelling counterexample, see Chris Okasaki's thesis/monograph *Functional Data Structures* and its more recent descendants.

its running time. At the same time, it should *exclude* any details that are *not* necessary to fully specify the algorithm, prove its correctness, and analyze its running time. (Slide.) At a more practical level, our description should allow a competent but skeptical programmer *who has not read this book* to quickly and correctly implement the algorithm in *their* favorite programming language, *without understanding why it works*.

I don't want to bore you with the rules I follow for writing pseudocode, but I must caution against one especially pernicious habit. ***Never*** describe repeated operations informally, as in "Do [this] first, then do [that] second, ***and so on***." or "Repeat ***this process*** until [something]". As anyone who has taken one of those frustrating "What comes next in this sequence?" tests already knows, describing the first few steps of an algorithm says little or nothing about what happens in later steps. If your algorithm has a loop, write it as a loop, and explicitly describe what happens in an *arbitrary* iteration. Similarly, if your algorithm is recursive, write it recursively, and explicitly describe the case boundaries and what happens in each case.

0.6 Analyzing Algorithms

It's not enough just to write down an algorithm and say "Behold!" We must also convince our audience (and ourselves!) that the algorithm actually does what it's supposed to do, and that it does so efficiently.

Correctness

In some application settings, it is acceptable for programs to behave correctly most of the time, on all "reasonable" inputs. Not in this book; we require algorithms that are *always* correct, for *all possible* inputs. Moreover, we must *prove* that our algorithms are correct; trusting our instincts, or trying a few test cases, isn't good enough. Sometimes correctness is truly obvious, especially for algorithms you've seen in earlier courses. On the other hand, "obvious" is all too often a synonym for "wrong". Most of the algorithms we discuss in this course require real work to prove correct. In particular, correctness proofs usually involve induction. We *like* induction. Induction is our *friend*.[21]

Of course, before we can formally prove that our algorithm does what it's supposed to do, we have to formally describe what it's supposed to do!

Running Time

The most common way of ranking different algorithms for the same problem is by how quickly they run. Ideally, we want the fastest possible algorithm for any

[21]If induction is *not* your friend, you will have a hard time with this book.

particular problem. In many application settings, it is acceptable for programs to run efficiently most of the time, on all "reasonable" inputs. Not in this class; we require algorithms that *always* run efficiently, even in the worst case.

But how do we measure running time? As a specific example, how long does it take to sing the song BottlesOfBeer(n)? This is obviously a function of the input value n, but it also depends on how quickly you can sing. Some singers might take ten seconds to sing a verse; others might take twenty. Technology widens the possibilities even further. Dictating the song over a telegraph using Morse code might take a full minute per verse. Downloading an mp3 over the Web might take a tenth of a second per verse. Duplicating the mp3 in a computer's main memory might take only a few microseconds per verse.

What's important here is how the singing time changes as n grows. Singing BottlesOfBeer($2n$) requires about twice much time as singing BottlesOfBeer(n), no matter what technology is being used. This is reflected in the asymptotic singing time $\Theta(n)$.

We can measure time by counting how many times the algorithm executes a certain instruction or reaches a certain milestone in the "code". For example, we might notice that the word "beer" is sung three times in every verse of BottlesOfBeer, so the number of times you sing "beer" is a good indication of the total singing time. For this question, we can give an exact answer: BottlesOfBeer(n) mentions beer exactly $3n + 3$ times.

Incidentally, there are *lots* of other songs with quadratic singing time. This one is probably familiar to most English-speakers:

```
NDaysOfChristmas(gifts[2..n]):
  for $i \leftarrow 1$ to $n$
      Sing "On the $i$th day of Christmas, my true love gave to me"
      for $j \leftarrow i$ down to 2
          Sing "$j$ gifts[$j$]"
      if $i > 1$
          Sing "and"
      Sing "a partridge in a pear tree."
```

The input to NDaysOfChristmas is a list of $n-1$ gifts, represented here as an array. It's quite easy to show that the singing time is $\Theta(n^2)$; in particular, the singer mentions the name of a gift $\sum_{i=1}^{n} i = n(n+1)/2$ times (counting the partridge in the pear tree). It's also easy to see that during the first n days of Christmas, my true love gave to me exactly $\sum_{i=1}^{n}\sum_{j=1}^{i} j = n(n+1)(n+2)/6 = \Theta(n^3)$ gifts.

Other quadratic-time songs include "Old MacDonald Had a Farm", "There Was an Old Lady Who Swallowed a Fly", "Hole in the Bottom of the Sea", "Green Grow the Rushes O", "The Rattlin' Bog", "The Court Of King Caractacus","The Barley-Mow", "If I Were Not Upon the Stage", "Star Trekkin'", "Ist das nicht ein Schnitzelbank?",[22] "Il Pulcino Pio", "Minkurinn í hænsnakofanum", "Echad Mi Yodea", and "To κοκοράκι". For more examples, consult your favorite preschooler.

```
ALOUETTE(lapart[1..n]):
  Chantez « Alouette, gentille alouette, alouette, je te plumerais. »
  pour tout $i$ de 1 á $n$
      Chantez « Je te plumerais lapart[$i$]. Je te plumerais lapart[$i$]. »
      pour tout $j$ de $i-1$ á bas á 1
          Chantez « Et lapart[$j$]! Et lapart[$j$]! »
      Chantez « Alouette, alouette, ooooooo! »
      Chantez « Alouette, gentille allouette, alouette, je te plumerais. »
```

A few songs have even more bizarre singing times. A fairly modern example is "The TELNET Song" by Guy Steele, which actually takes $\Theta(2^n)$ time to sing the first n verses; Steele recommended $n = 4$. Finally, there are some songs that *never* end.[23]

Except for "The TELNET Song", all of these songs are iterative algorithms, expressed as a small set of nested loops, so their ~~running~~ singing times can be computed using nested summations. The running time of a *recursive* algorithm is more easily expressed as a recurrence. For example, the peasant multiplication algorithm can be expressed recursively as follows:

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

Let $T(x, y)$ denote the number of parity, addition, and mediation operations required to compute $x \cdot y$. This function satisfies the recursive inequality $T(x, y) \le T(\lfloor x/2 \rfloor, 2y) + 2$ with base case $T(0, y) = 0$. Techniques described in the next chapter imply the upper bound $T(x, y) = O(\log x)$.

Sometimes the running time of an algorithm depends on a particular implementation of some underlying data structure of subroutine. For example, the Huntington-Hill apportionment algorithm APPORTIONCONGRESS runs in $O(N + RI + (R-n)E)$ time, where N denotes the running time of NEWPRIORITYQUEUE, I denotes the running time of INSERT, and E denotes the running time of EXTRACTMAX. Under the reasonable assumption that $R \ge 2n$ (on average, each state gets at least two representatives), we can simplify this bound to $O(N + R(I + E))$. The precise running time depends on the implementation of the underlying priority queue. The Census Bureau implements the priority

[22]Ja, das ist Otto von Schnitzelpusskrankengescheitmeyer!

[23]They just go on and on, my friend.

queue as an unsorted array, which gives us $N = I = \Theta(1)$ and $E = \Theta(n)$, so the Census Bureau's implementation of ApportionCongress runs in ***$O(Rn)$ time***. However, if we implement the priority queue as a binary heap or a heap-ordered array, we have $N = \Theta(1)$ and $I = R = O(\log n)$, so the overall algorithm runs in **$O(R \log n)$** time.

Finally, sometimes we are interested in computational resources other than time, such as space, number of coin flips, number of cache or page faults, number of inter-process messages, or the number of gifts my true love gave to me. These resources can be analyzed using the same techniques used to analyze running time. For example, lattice multiplication of two n-digit numbers requires $O(n^2)$ space if we write down all the partial products before adding them, but only $O(n)$ space if we add them on the fly.

Exercises

0. Describe and analyze an efficient algorithm that determines, given a legal arrangement of standard pieces on a standard chess board, which player will win at chess from the given starting position if both players play perfectly. *[Hint: There is a trivial one-line solution!]*

♥1. (a) Identify (or write) a song that requires $\Theta(n^3)$ time to sing the first n verses.
   (b) Identify (or write) a song that requires $\Theta(n \log n)$ time to sing the first n verses.
   (c) Identify (or write) a song that requires some other weird amount of time to sing the first n verses.

2. Careful readers might complain that our analysis of songs like "n Bottles of Beer on the Wall" or "The n Days of Christmas" is overly simplistic, because larger numbers take longer to sing than shorter numbers. More generally, because there are only so many words of a given length, larger sets of words necessarily contain longer words.[24] We can more accurately estimate singing time by counting the number of *syllables* sung, rather than the number of *words*.
   (a) How long does it take to sing the integer n?
   (b) How long does it take to sing "n Bottles of Beer on the Wall"?
   (c) How long does it take to sing "The n Days of Christmas"?

   As usual, express your answers in the form $O(f(n))$ for some function f.

[24]Ja, das ist das Subatomarteilchenbeschleunigungsnaturmäßigkeitsuntersuchungsmaschine!

3. The cumulative drinking song "The Barley Mow" has been sung throughout the British Isles for centuries. The song has many variants; Figure 0.6 contains pseudolyrics for one version traditionally sung in Devon and Cornwall, where *vessel*$[i]$ is the name of a vessel that holds 2^i ounces of beer.[25]

```
BarleyMow(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  for $i \leftarrow 1$ to $n$
      "We'll drink it out of the vessel[$i$], boys,"
      "Here's a health to the barley-mow!"
      for $j \leftarrow i$ downto 1
          "The vessel[$j$],"
      "And the jolly brown bowl!"
      "Here's a health to the barley-mow!"
      "Here's a health to the barley-mow, my brave boys,"
      "Here's a health to the barley-mow!"
```

Figure 0.6. "The Barley Mow".

(a) Suppose each name *vessel*$[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

(b) If you want to sing this song for arbitrarily large values of n, you'll have to make up your own vessel names. To avoid repetition, these names must become progressively longer as n increases. Suppose *vessel*$[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

[25] In practice, the song uses some subset of the following vessels; nipperkin, quarter-gill, half-a-gill, gill, quarter-pint, half-a-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel/kilderkin, barrel, hogshead, pipe/butt, tun, well, river, and ocean. With a few exceptions (especially at the end), every vessel in this list has twice the volume of its predecessor. Irish and Scottish versions of the song have slightly different lyrics, and they usually switch to people (barmaid, landlord, drayer, and so on) after "gallon".

An early version of the song entitled "Give us once a drink" appears in the play *Jack Drum's Entertainment (or the Comedie of Pasquill and Katherine)* written by John Marston around 1600. ("Giue vs once a drinke for and the black bole. Sing gentle Butler *bally moy*!") There is some disagreement whether Marston wrote the "high Dutch Song" specifically for the play, whether "bally moy" is a mondegreen for "barley mow" or vice versa, or whether it's actually the same song at all. These discussions are best had over n bottles of beer.

(c) Suppose each time you mention the name of a vessel, you actually drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each *vessel*$[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BarleyMow(n)? (Give an *exact* answer, not just an asymptotic bound.)

4. Recall that the input to the Huntington-Hill algorithm ApportionCongress is an array $P[1..n]$, where $P[i]$ is the population of the ith state, and an integer R, the total number of representatives to be allotted. The output is an array $r[1..n]$, where $r[i]$ is the number of representatives allotted to the ith state by the algorithm.

   Let $P = \sum_{i=1}^{n} P[i]$ denote the total population of the country, and let $r_i^* = R \cdot P[i]/P$ denote the ideal number of representatives for the ith state.

   (a) Prove that $r[i] \geq \lfloor r_i^* \rfloor$ for all i.

   (b) Describe and analyze an algorithm that computes exactly the same congressional apportionment as ApportionCongress in $O(n \log n)$ time. (Recall that the running time of ApportionCongress depends on R, which could be arbitrarily larger than n.)

   ♥(c) If a state's population is small relative to the other states, its ideal number r_i^* of representatives could be close to zero; thus, tiny states are over-represented by the Huntington-Hill apportionment process. Surprisingly, this can also be true of very large states. Let $\alpha = (1+\sqrt{2})/2 \approx 1.20711$. Prove that for any $\varepsilon > 0$, there is an input to ApportionCongress with $\max_i P[i] = P[1]$, such that $r[1] > (\alpha - \varepsilon)\, r_1^*$.

   ♥(d) Can you improve the constant α in the previous question?

The control of a large force is the same principle as the control of a few men:
it is merely a question of dividing up their numbers.

— Sun Zi, *The Art of War* (c. 400CE), translated by Lionel Giles (1910)

Our life is frittered away by detail. . . . Simplify, simplify.

— Henry David Thoreau, *Walden* (1854)

Now, don't ask me what Voom is. I never will know.
But, boy! Let me tell you, it DOES clean up snow!

— Dr. Seuss [Theodor Seuss Geisel], *The Cat in the Hat Comes Back* (1958)

Do the hard jobs first. The easy jobs will take care of themselves.

— attributed to Dale Carnegie

1 Recursion

1.1 Reductions

Reduction is the single most common technique used in designing algorithms. Reducing one problem X to another problem Y means to write an algorithm for X that uses an algorithm for Y as a black box or subroutine. Crucially, the correctness of the resulting algorithm for X cannot depend in any way on *how* the algorithm for Y works. The only thing we can assume is that the black box solves Y correctly. The inner workings of the black box are simply *none of our business*; they're somebody else's problem. It's often best to literally think of the black box as functioning purely by magic.

For example, the peasant multiplication algorithm described in the previous chapter reduces the problem of multiplying two arbitrary positive integers to three simpler problems: addition, mediation (halving), and parity-checking. The algorithm relies on an abstract "positive integer" data type that supports those three operations, but the correctness of the multiplication algorithm does not depend on the precise data representation (tally marks, clay tokens, Babylonian hexagesimal, quipu, counting rods, Roman numerals, finger positions, augrym stones, gobar numerals, binary, negabinary, Gray code, balanced ternary, phinary, quater-imaginary, . . .), or on the precise implementations of those operations. Of course, the *running time* of the multiplication algorithm depends on the *running time* of the addition, mediation, and parity operations, but that's a separate issue from *correctness*. Most importantly, we can create a more efficient multiplication algorithm just by switching to a more efficient number representation (from tally marks to place-value notation, for example).

Similarly, the Huntington-Hill algorithm reduces the problem of apportioning Congress to the problem of maintaining a priority queue that supports the operations INSERT and EXTRACTMAX. The abstract data type "priority queue" is a black box; the correctness of the apportionment algorithm does not depend on any specific priority queue data structure. Of course, the *running time* of the apportionment algorithm depends on the *running time* of the INSERT and EXTRACTMAX algorithms, but that's a separate issue from the *correctness* of the algorithm. The beauty of the reduction is that we can create a more efficient apportionment algorithm by simply swapping in a new priority queue data structure. Moreover, the designer of that data structure does not need to know or care that it will be used to apportion Congress.

When we design algorithms, we may not know exactly how the basic building blocks we use are implemented, or how our algorithms might be used as building blocks to solve even bigger problems. That ignorance is uncomfortable for many beginners, but it is both unavoidable and extremely useful. Even when you do know precisely how your components work, it is often *extremely* helpful to pretend that you don't.

1.2 Simplify and Delegate

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

- If the given instance of the problem can be solved directly, solve it directly.
- Otherwise, reduce it to one or more ***simpler instances of the same problem***.

If the self-reference is confusing, it may be helpful to imagine that someone else is going to solve the simpler problems, just as you would assume for other types of reductions. I like to call that someone else the ***Recursion Fairy***. Your *only* task is to *simplify* the original problem, or to solve it directly when simplification is either unnecessary or impossible; the Recursion Fairy will solve all the simpler subproblems for you, using Methods That Are None Of Your Business So *Butt*

Out.[1] Mathematically sophisticated readers might recognize the Recursion Fairy by its more formal name: the ***Induction Hypothesis***.

There is one mild technical condition that must be satisfied in order for any recursive method to work correctly: There must be no infinite sequence of reductions to simpler and simpler instances. Eventually, the recursive reductions must lead to an elementary ***base case*** that can be solved by some other method; otherwise, the recursive algorithm will loop forever. The most common way to satisfy this condition is to reduce to one or more ***smaller*** instances of the same problem. For example, if the original input is a skreeble with n glurps, the input to each recursive call should be a skreeble with strictly less than n glurps. Of course this is impossible if the skreeble has no glurps at all—You can't have negative glurps; that would be silly!—so in that case we must grindlebloff the skreeble using some other method.

We've already seen one instance of this pattern in the peasant multiplication algorithm, which is based directly on the following identity.

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

The same identity can be expressed algorithmically as follows:

```
MULTIPLY($x, y$):
  if $x = 0$
      return 0
  else
      $x' \gets \lfloor x/2 \rfloor$
      $y' \gets y + y$
      $prod \gets$ MULTIPLY($x', y'$)    ⟨⟨Recurse!⟩⟩
      if $x$ is odd
          $prod \gets prod + y$
      return $prod$
```

A lazy Egyptian scribe could execute this algorithm by computing x' and y', *asking a more junior scribe to multiply x' and y'*, and then possibly adding y to the junior scribe's response. The junior scribe's problem is simpler because $x' < x$, and repeatedly decreasing a positive integer eventually leads to 0. How the junior scribe actually computes $x' \cdot y'$ is none of the senior scribe's business (and it's none of your business, either).

[1]When I was an undergraduate, I attributed recursion to "elves" instead of the Recursion Fairy, referring to the Brothers Grimm story about an old shoemaker who leaves his work unfinished when he goes to bed, only to discover upon waking that elves ("Wichtelmänner") have finished everything overnight. Someone more entheogenically experienced than I might recognize these Rekursionswichtelmänner as Terence McKenna's "self-transforming machine elves".

1.3 Tower of Hanoi

The Tower of Hanoi puzzle was first published—as an actual physical puzzle!—by the French teacher and recreational mathematician Éduoard Lucas in 1883,[2] under the pseudonym "N. Claus (de Siam)" (an anagram of "Lucas d'Amiens"). The following year, Henri de Parville described the puzzle with the following remarkable story:[3]

> In the great temple at Benares[4] . . . beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

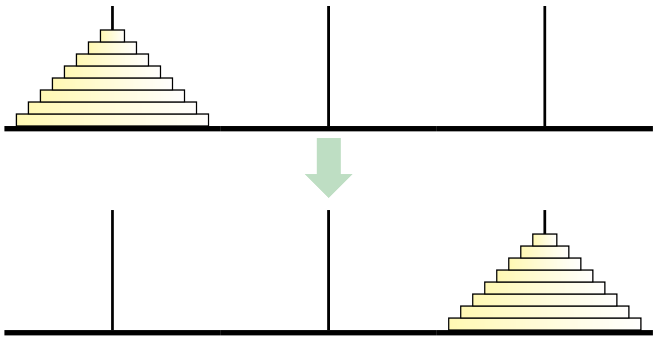

Figure 1.1. The Tower of Hanoi puzzle

Of course, as good computer scientists, our first instinct on reading this story is to substitute the variable n for the hardwired constant 64. And because most physical instances of the puzzle are made of wood instead of diamonds and gold, I will call the three possible locations for the disks "pegs" instead of

[2]Lucas later claimed to have invented the puzzle in 1876.

[3]This English translation is taken from W. W. Rouse Ball's 1892 book *Mathematical Recreations and Essays*.

[4]The "great temple at Benares" is almost certainly the Kashi Vishvanath Temple in Varanasi, Uttar Pradesh, India, located approximately 2400km west-north-west of Hà Nội, Việt Nam, where the fictional N. Claus supposedly resided. Coincidentally, the French Army invaded Hanoi in 1883, the same year Lucas released his puzzle, ultimately leading to its establishment as the capital of French Indochina.

"needles". How can we move a tower of n disks from one peg to another, using a third peg as an occasional placeholder, without ever placing a disk on top of a smaller disk?

As N. Claus (de Siam) pointed out in the pamphlet included with his puzzle, the secret to solving this puzzle is to think recursively. Instead of trying to solve the entire puzzle at once, let's concentrate on moving just the largest disk. We can't move it at the beginning, because all the other disks are in the way. So first we have to move those $n-1$ disks to the third peg. And then after we move the largest disk, we have to move those $n-1$ disks back on top of it.

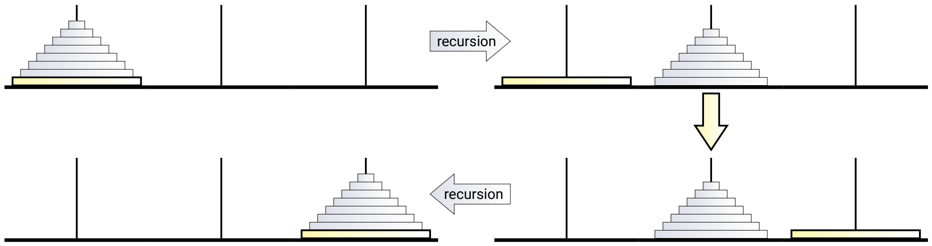

Figure 1.2. The Tower of Hanoi algorithm; ignore everything but the bottom disk.

So now all we have to figure out is how to—

NO!! STOP!!

That's it! We're done! We've successfully reduced the n-disk Tower of Hanoi problem to two instances of the $(n-1)$-disk Tower of Hanoi problem, which we can gleefully hand off to the Recursion Fairy—or to carry Lucas's metaphor further, to the junior monks at the temple. *Our* job is finished. If we didn't trust the junior monks, we wouldn't have hired them; let them do their job in peace.

Our reduction does make one subtle but extremely important assumption: *There is a largest disk*. Our recursive algorithm works for any *positive* number of disks, but it breaks down when $n = 0$. We must handle that case using a different method. Fortunately, the monks at Benares, being good Buddhists, are quite adept at moving zero disks from one peg to another in no time at all, by doing nothing.

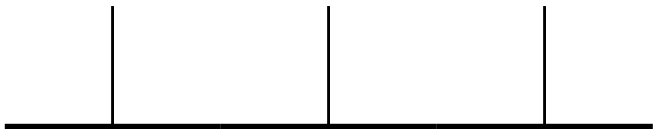

Figure 1.3. The vacuous base case for the Tower of Hanoi algorithm. There is no spoon.

It may be tempting to think about how all those smaller disks move around—or more generally, what happens when the recursion is unrolled—but really, don't do it. For most recursive algorithms, unrolling the recursion is neither