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

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 recursive 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 recurrence can be expressed algorithmically as follows:

```
PeasantMultiply($x, y$):
  if $x = 0$
      return 0
  else
      $x' \gets \lfloor x/2 \rfloor$
      $y' \gets y + y$
      $prod \gets$ PeasantMultiply($x', y'$)     ⟨⟨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 Édouard 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.

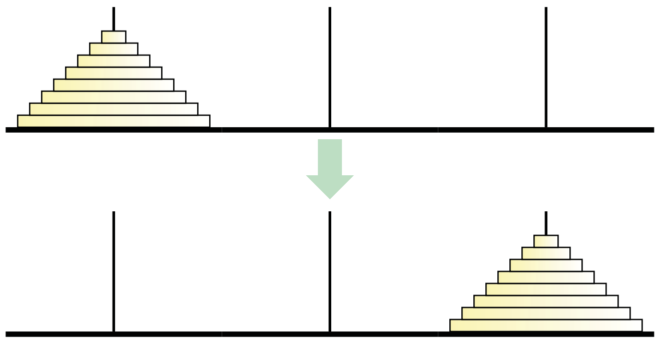

Figure 1.1. The (8-disk) Tower of Hanoi puzzle

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 spare 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 $n-1$ smaller disks to the spare peg. Once that's done, we can move the largest disk directly to its destination. Finally, to finish the puzzle, we have to move the $n-1$ smaller disks from the spare peg to their destination.

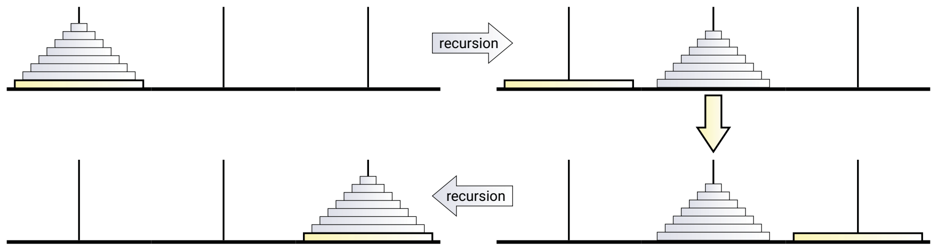

Figure 1.2. The Tower of Hanoi algorithm; ignore everything but the bottom disk.

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.

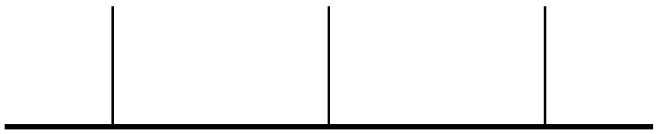

Figure 1.3. The vacuous base case for the Tower of Hanoi algorithm. There is no spoon.

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 necessary nor helpful. Our ***only*** task is to reduce the problem instance we're given to one or more simpler instances, or to solve the problem directly if such a reduction is impossible. Our recursive Tower of Hanoi algorithm is trivially correct when $n = 0$. For any $n \geq 1$, the Recursion Fairy correctly moves the top $n-1$ disks (more formally, the Inductive Hypothesis implies that our recursive algorithm correctly moves the top $n-1$ disks) so our algorithm is correct.

The recursive Hanoi algorithm is expressed in pseudocode in Figure 1.4. The algorithm moves a stack of n disks from a source peg (*src*) to a destination peg (*dst*) using a third temporary peg (*tmp*) as a placeholder. Notice that the algorithm correctly does nothing at all when $n = 0$.

```
HANOI($n, src, dst, tmp$):
  if $n > 0$
      HANOI($n-1, src, tmp, dst$)     ⟨⟨Recurse!⟩⟩
      move disk $n$ from $src$ to $dst$
      HANOI($n-1, tmp, dst, src$)     ⟨⟨Recurse!⟩⟩
```

Figure 1.4. A recursive algorithm to solve the Tower of Hanoi

Let $T(n)$ denote the number of moves required to transfer n disks—the running time of our algorithm. Our vacuous base case implies that $T(0) = 0$, and the more general recursive algorithm implies that $T(n) = 2T(n-1)+1$ for any $n \geq 1$. By writing out the first several values of $T(n)$, we can easily guess that $\boldsymbol{T(n) = 2^n - 1}$; a straightforward induction proof implies that this guess is correct. In particular, moving a tower of 64 disks requires $2^{64} - 1 =$ 18,446,744,073,709,551,615 individual moves. Thus, even at the impressive rate of one move per second, the monks at Benares will be at work for approximately 585 billion years ("plus de *cinq milliards de siècles*") before tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.