
10 Recurrences

A recurrence describes a sequence of numbers. Early terms are specified explicitly and later terms are expressed as a function of their predecessors. As a trivial example, this recurrence describes the sequence 1, 2, 3, etc.:

$$\begin{aligned}T_1 &= 1 \\T_n &= T_{n-1} + 1 \quad (\text{for } n \geq 2).\end{aligned}$$

Here, the first term is defined to be 1 and each subsequent term is one more than its predecessor.

Recurrences turn out to be a powerful tool. In this chapter, we’ll emphasize using recurrences to analyze the performance of recursive algorithms. However, recurrences have other applications in computer science as well, such as enumeration of structures and analysis of random processes. And, as we saw in Section 9.4, they also arise in the analysis of problems in the physical sciences.

A recurrence in isolation is not a very useful description of a sequence. One can not easily answer simple questions such as, “What is the hundredth term?” or “What is the asymptotic growth rate?” So one typically wants to *solve* a recurrence; that is, to find a closed-form expression for the n th term.

We’ll first introduce two general solving techniques: guess-and-verify and plug-and-chug. These methods are applicable to every recurrence, but their success requires a flash of insight—sometimes an unrealistically brilliant flash. So we’ll also introduce two big classes of recurrences, linear and divide-and-conquer, that often come up in computer science. Essentially all recurrences in these two classes are solvable using cookbook techniques; you follow the recipe and get the answer. A drawback is that calculation replaces insight. The “Aha!” moment that is essential in the guess-and-verify and plug-and-chug methods is replaced by a “Huh” at the end of a cookbook procedure.

At the end of the chapter, we’ll develop rules of thumb to help you assess many recurrences without any calculation. These rules can help you distinguish promising approaches from bad ideas early in the process of designing an algorithm.

Recurrences are one aspect of a broad theme in computer science: reducing a big problem to progressively smaller problems until easy base cases are reached. This same idea underlies both induction proofs and recursive algorithms. As we’ll see, all three ideas snap together nicely. For example, one might describe the running time of a recursive algorithm with a recurrence and use induction to verify the solution.

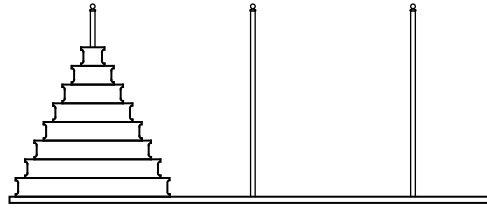


Figure 10.1 The initial configuration of the disks in the Towers of Hanoi problem.

10.1 The Towers of Hanoi

According to legend, there is a temple in Hanoi with three posts and 64 gold disks of different sizes. Each disk has a hole through the center so that it fits on a post. In the misty past, all the disks were on the first post, with the largest on the bottom and the smallest on top, as shown in Figure 10.1.

Monks in the temple have labored through the years since to move all the disks to one of the other two posts according to the following rules:

- The only permitted action is removing the top disk from one post and dropping it onto another post.
- A larger disk can never lie above a smaller disk on any post.

So, for example, picking up the whole stack of disks at once and dropping them on another post is illegal. That’s good, because the legend says that when the monks complete the puzzle, the world will end!

To clarify the problem, suppose there were only 3 gold disks instead of 64. Then the puzzle could be solved in 7 steps as shown in Figure 10.2.

The questions we must answer are, “Given sufficient time, can the monks succeed?” If so, “How long until the world ends?” And, most importantly, “Will this happen before the final exam?”

10.1.1 A Recursive Solution

The Towers of Hanoi problem can be solved recursively. As we describe the procedure, we’ll also analyze the running time. To that end, let T_n be the minimum number of steps required to solve the n -disk problem. For example, some experimentation shows that $T_1 = 1$ and $T_2 = 3$. The procedure illustrated above shows that T_3 is at most 7, though there might be a solution with fewer steps.

The recursive solution has three stages, which are described below and illustrated in Figure 10.3. For clarity, the largest disk is shaded in the figures.

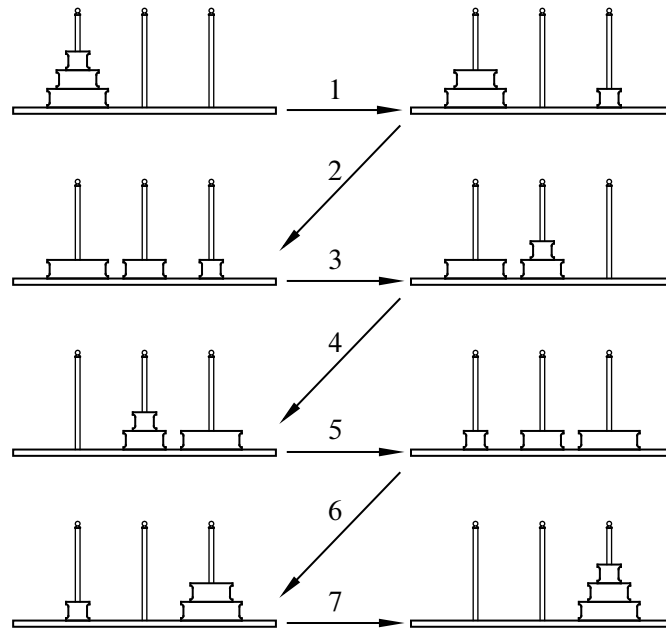


Figure 10.2 The 7-step solution to the Towers of Hanoi problem when there are $n = 3$ disks.

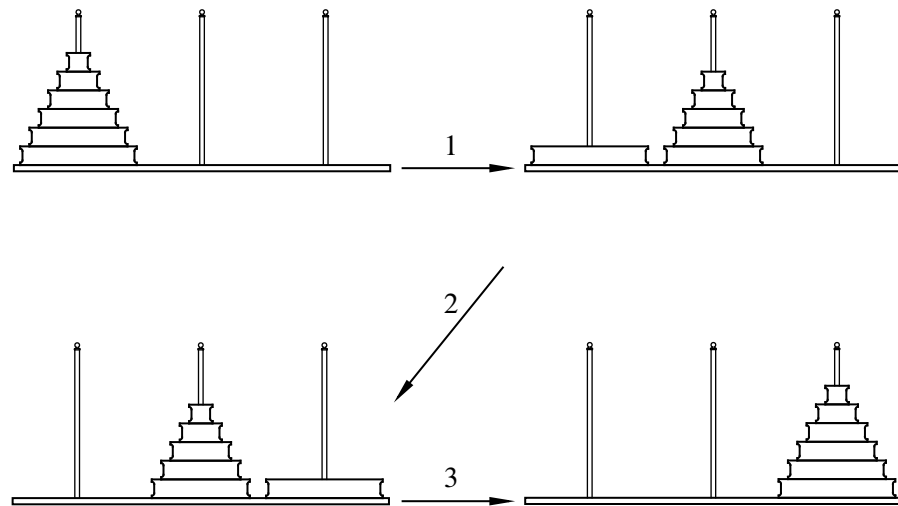


Figure 10.3 A recursive solution to the Towers of Hanoi problem.

Stage 1. Move the top $n - 1$ disks from the first post to the second using the solution for $n - 1$ disks. This can be done in T_{n-1} steps.

Stage 2. Move the largest disk from the first post to the third post. This takes just 1 step.

Stage 3. Move the $n - 1$ disks from the second post to the third post, again using the solution for $n - 1$ disks. This can also be done in T_{n-1} steps.

This algorithm shows that T_n , the minimum number of steps required to move n disks to a different post, is at most $T_{n-1} + 1 + T_{n-1} = 2T_{n-1} + 1$. We can use this fact to upper bound the number of operations required to move towers of various heights:

$$\begin{aligned} T_3 &\leq 2 \cdot T_2 + 1 = 7 \\ T_4 &\leq 2 \cdot T_3 + 1 \leq 15 \end{aligned}$$

Continuing in this way, we could eventually compute an upper bound on T_{64} , the number of steps required to move 64 disks. So this algorithm answers our first question: given sufficient time, the monks can finish their task and end the world. This is a shame. After all that effort, they’d probably want to smack a few high-fives and go out for burgers and ice cream, but nope—world’s over.

10.1.2 Finding a Recurrence

We can not yet compute the exact number of steps that the monks need to move the 64 disks, only an upper bound. Perhaps, having pondered the problem since the beginning of time, the monks have devised a better algorithm.

In fact, there is no better algorithm, and here is why. At some step, the monks must move the largest disk from the first post to a different post. For this to happen, the $n - 1$ smaller disks must all be stacked out of the way on the only remaining post. Arranging the $n - 1$ smaller disks this way requires at least T_{n-1} moves. After the largest disk is moved, at least another T_{n-1} moves are required to pile the $n - 1$ smaller disks on top.

This argument shows that the number of steps required is at least $2T_{n-1} + 1$. Since we gave an algorithm using exactly that number of steps, we can now write an expression for T_n , the number of moves required to complete the Towers of Hanoi problem with n disks:

$$\begin{aligned} T_1 &= 1 \\ T_n &= 2T_{n-1} + 1 \quad (\text{for } n \geq 2). \end{aligned}$$

This is a typical recurrence. These two lines define a sequence of values, T_1, T_2, T_3, \dots . The first line says that the first number in the sequence, T_1 , is equal to 1. The second line defines every other number in the sequence in terms of its predecessor. So we can use this recurrence to compute any number of terms in the sequence:

$$\begin{aligned} T_1 &= 1 \\ T_2 &= 2 \cdot T_1 + 1 = 3 \\ T_3 &= 2 \cdot T_2 + 1 = 7 \\ T_4 &= 2 \cdot T_3 + 1 = 15 \\ T_5 &= 2 \cdot T_4 + 1 = 31 \\ T_6 &= 2 \cdot T_5 + 1 = 63. \end{aligned}$$

10.1.3 Solving the Recurrence

We could determine the number of steps to move a 64-disk tower by computing T_7, T_8 , and so on up to T_{64} . But that would take a lot of work. It would be nice to have a closed-form expression for T_n , so that we could quickly find the number of steps required for any given number of disks. (For example, we might want to know how much sooner the world would end if the monks melted down one disk to purchase burgers and ice cream *before* the end of the world.)

There are several methods for solving recurrence equations. The simplest is to *guess* the solution and then *verify* that the guess is correct with an induction proof. As a basis for a good guess, let’s look for a pattern in the values of T_n computed above: 1, 3, 7, 15, 31, 63. A natural guess is $T_n = 2^n - 1$. But whenever you guess a solution to a recurrence, you should always verify it with a proof, typically by induction. After all, your guess might be wrong. (But why bother to verify in this case? After all, if we’re wrong, it’s not the end of the... no, let’s check.)

Claim 10.1.1. $T_n = 2^n - 1$ satisfies the recurrence:

$$\begin{aligned} T_1 &= 1 \\ T_n &= 2T_{n-1} + 1 \quad (\text{for } n \geq 2). \end{aligned}$$

Proof. The proof is by induction on n . The induction hypothesis is that $T_n = 2^n - 1$. This is true for $n = 1$ because $T_1 = 1 = 2^1 - 1$. Now assume that $T_{n-1} = 2^{n-1} - 1$ in order to prove that $T_n = 2^n - 1$, where $n \geq 2$:

$$\begin{aligned} T_n &= 2T_{n-1} + 1 \\ &= 2(2^{n-1} - 1) + 1 \\ &= 2^n - 1. \end{aligned}$$

The first equality is the recurrence equation, the second follows from the induction assumption, and the last step is simplification. ■

Such verification proofs are especially tidy because recurrence equations and induction proofs have analogous structures. In particular, the base case relies on the first line of the recurrence, which defines T_1 . And the inductive step uses the second line of the recurrence, which defines T_n as a function of preceding terms.

Our guess is verified. So we can now resolve our remaining questions about the 64-disk puzzle. Since $T_{64} = 2^{64} - 1$, the monks must complete more than 18 billion billion steps before the world ends. Better study for the final.

10.1.4 The Upper Bound Trap

When the solution to a recurrence is complicated, one might try to prove that some simpler expression is an upper bound on the solution. For example, the exact solution to the Towers of Hanoi recurrence is $T_n = 2^n - 1$. Let’s try to prove the “nicer” upper bound $T_n \leq 2^n$, proceeding exactly as before.

Proof. (Failed attempt.) The proof is by induction on n . The induction hypothesis is that $T_n \leq 2^n$. This is true for $n = 1$ because $T_1 = 1 \leq 2^1$. Now assume that $T_{n-1} \leq 2^{n-1}$ in order to prove that $T_n \leq 2^n$, where $n \geq 2$:

$$\begin{aligned} T_n &= 2T_{n-1} + 1 \\ &\leq 2(2^{n-1}) + 1 \\ &\not\leq 2^n \quad \text{Uh-oh!} \end{aligned}$$

The first equality is the recurrence relation, the second follows from the induction hypothesis, and the third step is a flaming train wreck. ■

The proof doesn’t work! As is so often the case with induction proofs, the argument only goes through with a *stronger* hypothesis. This isn’t to say that upper bounding the solution to a recurrence is hopeless, but this is a situation where induction and recurrences do not mix well.

10.1.5 Plug and Chug

Guess-and-verify is a simple and general way to solve recurrence equations. But there is one big drawback: you have to *guess right*. That was not hard for the Towers of Hanoi example. But sometimes the solution to a recurrence has a strange form that is quite difficult to guess. Practice helps, of course, but so can some other methods.

Plug-and-chug is another way to solve recurrences. This is also sometimes called “expansion” or “iteration”. As in guess-and-verify, the key step is identifying a pattern. But instead of looking at a sequence of *numbers*, you have to spot a pattern in a sequence of *expressions*, which is sometimes easier. The method consists of three steps, which are described below and illustrated with the Towers of Hanoi example.

Step 1: Plug and Chug Until a Pattern Appears

The first step is to expand the recurrence equation by alternately “plugging” (applying the recurrence) and “chugging” (simplifying the result) until a pattern appears. Be careful: too much simplification can make a pattern harder to spot. The rule to remember—indeed, a rule applicable to the whole of college life—is *chug in moderation*.

$$\begin{aligned}
 T_n &= 2T_{n-1} + 1 \\
 &= 2(2T_{n-2} + 1) + 1 && \text{plug} \\
 &= 4T_{n-2} + 2 + 1 && \text{chug} \\
 &= 4(2T_{n-3} + 1) + 2 + 1 && \text{plug} \\
 &= 8T_{n-3} + 4 + 2 + 1 && \text{chug} \\
 &= 8(2T_{n-4} + 1) + 4 + 2 + 1 && \text{plug} \\
 &= 16T_{n-4} + 8 + 4 + 2 + 1 && \text{chug}
 \end{aligned}$$

Above, we started with the recurrence equation. Then we replaced T_{n-1} with $2T_{n-2} + 1$, since the recurrence says the two are equivalent. In the third step, we simplified a little—but not too much! After several similar rounds of plugging and chugging, a pattern is apparent. The following formula seems to hold:

$$\begin{aligned}
 T_n &= 2^k T_{n-k} + 2^{k-1} + 2^{k-2} + \dots + 2^2 + 2^1 + 2^0 \\
 &= 2^k T_{n-k} + 2^k - 1
 \end{aligned}$$

Once the pattern is clear, simplifying is safe and convenient. In particular, we’ve collapsed the geometric sum to a closed form on the second line.

Step 2: Verify the Pattern

The next step is to verify the general formula with one more round of plug-and-chug.

$$\begin{aligned}
 T_n &= 2^k T_{n-k} + 2^k - 1 \\
 &= 2^k (2T_{n-(k+1)} + 1) + 2^k - 1 && \text{plug} \\
 &= 2^{k+1} T_{n-(k+1)} + 2^{k+1} - 1 && \text{chug}
 \end{aligned}$$

The final expression on the right is the same as the expression on the first line, except that k is replaced by $k + 1$. Surprisingly, this effectively *proves* that the formula is correct for all k . Here is why: we know the formula holds for $k = 1$, because that’s the original recurrence equation. And we’ve just shown that if the formula holds for some $k \geq 1$, then it also holds for $k + 1$. So the formula holds for all $k \geq 1$ by induction.

Step 3: Write T_n Using Early Terms with Known Values

The last step is to express T_n as a function of early terms whose values are known. Here, choosing $k = n - 1$ expresses T_n in terms of T_1 , which is equal to 1. Simplifying gives a closed-form expression for T_n :

$$\begin{aligned}
 T_n &= 2^{n-1} T_1 + 2^{n-1} - 1 \\
 &= 2^{n-1} \cdot 1 + 2^{n-1} - 1 \\
 &= 2^n - 1.
 \end{aligned}$$

We’re done! This is the same answer we got from guess-and-verify.

Let’s compare guess-and-verify with plug-and-chug. In the guess-and-verify method, we computed several terms at the beginning of the sequence, T_1, T_2, T_3 , etc., until a pattern appeared. We generalized to a formula for the n th term, T_n . In contrast, plug-and-chug works backward from the n th term. Specifically, we started with an expression for T_n involving the preceding term, T_{n-1} , and rewrote this using progressively earlier terms, T_{n-2}, T_{n-3} , etc. Eventually, we noticed a pattern, which allowed us to express T_n using the very first term, T_1 , whose value we knew. Substituting this value gave a closed-form expression for T_n . So guess-and-verify and plug-and-chug tackle the problem from opposite directions.

10.2 Merge Sort

Algorithms textbooks traditionally claim that sorting is an important, fundamental problem in computer science. Then they smack you with sorting algorithms until life as a disk-stacking monk in Hanoi sounds delightful. Here, we’ll cover just *one* well-known sorting algorithm, Merge Sort. The analysis introduces another kind of recurrence.

Here is how Merge Sort works. The input is a list of n numbers, and the output is those same numbers in nondecreasing order. There are two cases:

- If the input is a single number, then the algorithm does nothing, because the list is already sorted.
- Otherwise, the list contains two or more numbers. The first half and the second half of the list are each sorted recursively. Then the two halves are merged to form a sorted list with all n numbers.

Let’s work through an example. Suppose we want to sort this list:

10, 7, 23, 5, 2, 8, 6, 9.

Since there is more than one number, the first half (10, 7, 23, 5) and the second half (2, 8, 6, 9) are sorted recursively. The results are 5, 7, 10, 23 and 2, 6, 8, 9. All that remains is to merge these two lists. This is done by repeatedly emitting the smaller of the two leading terms. When one list is empty, the whole other list is emitted. The example is worked out below. In this table, underlined numbers are about to be emitted.

First Half	Second Half	Output
5, 7, 10, 23	<u>2</u> , 6, 8, 9	
<u>5</u> , 7, 10, 23	6, 8, 9	2
7, 10, 23	<u>6</u> , 8, 9	2, 5
<u>7</u> , 10, 23	8, 9	2, 5, 6
10, 23	<u>8</u> , 9	2, 5, 6, 7
10, 23	<u>9</u>	2, 5, 6, 7, 8
<u>10</u> , <u>23</u>		2, 5, 6, 7, 8, 9
		2, 5, 6, 7, 8, 9, 10, 23

The leading terms are initially 5 and 2. So we output 2. Then the leading terms are 5 and 6, so we output 5. Eventually, the second list becomes empty. At that point, we output the whole first list, which consists of 10 and 23. The complete output consists of all the numbers in sorted order.

10.2.1 Finding a Recurrence

A traditional question about sorting algorithms is, “What is the maximum number of comparisons used in sorting n items?” This is taken as an estimate of the running time. In the case of Merge Sort, we can express this quantity with a recurrence. Let T_n be the maximum number of comparisons used while Merge Sorting a list of n numbers. For now, assume that n is a power of 2. This ensures that the input can be divided in half at every stage of the recursion.

- If there is only one number in the list, then no comparisons are required, so $T_1 = 0$.
- Otherwise, T_n includes comparisons used in sorting the first half (at most $T_{n/2}$), in sorting the second half (also at most $T_{n/2}$), and in merging the two halves. The number of comparisons in the merging step is at most $n - 1$. This is because at least one number is emitted after each comparison and one more number is emitted at the end when one list becomes empty. Since n items are emitted in all, there can be at most $n - 1$ comparisons.

Therefore, the maximum number of comparisons needed to Merge Sort n items is given by this recurrence:

$$T_1 = 0$$

$$T_n = 2T_{n/2} + n - 1 \quad (\text{for } n \geq 2 \text{ and a power of } 2).$$

This fully describes the number of comparisons, but not in a very useful way; a closed-form expression would be much more helpful. To get that, we have to solve the recurrence.

10.2.2 Solving the Recurrence

Let’s first try to solve the Merge Sort recurrence with the guess-and-verify technique. Here are the first few values:

$$T_1 = 0$$

$$T_2 = 2T_1 + 2 - 1 = 1$$

$$T_4 = 2T_2 + 4 - 1 = 5$$

$$T_8 = 2T_4 + 8 - 1 = 17$$

$$T_{16} = 2T_8 + 16 - 1 = 49.$$

We’re in trouble! Guessing the solution to this recurrence is hard because there is no obvious pattern. So let’s try the plug-and-chug method instead.

Step 1: Plug and Chug Until a Pattern Appears

First, we expand the recurrence equation by alternately plugging and chugging until a pattern appears.

$$\begin{aligned}
 T_n &= 2T_{n/2} + n - 1 \\
 &= 2(2T_{n/4} + n/2 - 1) + (n - 1) && \text{plug} \\
 &= 4T_{n/4} + (n - 2) + (n - 1) && \text{chug} \\
 &= 4(2T_{n/8} + n/4 - 1) + (n - 2) + (n - 1) && \text{plug} \\
 &= 8T_{n/8} + (n - 4) + (n - 2) + (n - 1) && \text{chug} \\
 &= 8(2T_{n/16} + n/8 - 1) + (n - 4) + (n - 2) + (n - 1) && \text{plug} \\
 &= 16T_{n/16} + (n - 8) + (n - 4) + (n - 2) + (n - 1) && \text{chug}
 \end{aligned}$$

A pattern is emerging. In particular, this formula seems holds:

$$\begin{aligned}
 T_n &= 2^k T_{n/2^k} + (n - 2^{k-1}) + (n - 2^{k-2}) + \dots + (n - 2^0) \\
 &= 2^k T_{n/2^k} + kn - 2^{k-1} - 2^{k-2} \dots - 2^0 \\
 &= 2^k T_{n/2^k} + kn - 2^k + 1.
 \end{aligned}$$

On the second line, we grouped the n terms and powers of 2. On the third, we collapsed the geometric sum.

Step 2: Verify the Pattern

Next, we verify the pattern with one additional round of plug-and-chug. If we guessed the wrong pattern, then this is where we’ll discover the mistake.

$$\begin{aligned}
 T_n &= 2^k T_{n/2^k} + kn - 2^k + 1 \\
 &= 2^k (2T_{n/2^{k+1}} + n/2^k - 1) + kn - 2^k + 1 && \text{plug} \\
 &= 2^{k+1} T_{n/2^{k+1}} + (k + 1)n - 2^{k+1} + 1 && \text{chug}
 \end{aligned}$$

The formula is unchanged except that k is replaced by $k + 1$. This amounts to the induction step in a proof that the formula holds for all $k \geq 1$.

Step 3: Write T_n Using Early Terms with Known Values

Finally, we express T_n using early terms whose values are known. Specifically, if we let $k = \log n$, then $T_{n/2^k} = T_1$, which we know is 0:

$$\begin{aligned} T_n &= 2^k T_{n/2^k} + kn - 2^k + 1 \\ &= 2^{\log n} T_{n/2^{\log n}} + n \log n - 2^{\log n} + 1 \\ &= nT_1 + n \log n - n + 1 \\ &= n \log n - n + 1. \end{aligned}$$

We’re done! We have a closed-form expression for the maximum number of comparisons used in Merge Sorting a list of n numbers. In retrospect, it is easy to see why guess-and-verify failed: this formula is fairly complicated.

As a check, we can confirm that this formula gives the same values that we computed earlier:

n	T_n	$n \log n - n + 1$
1	0	$1 \log 1 - 1 + 1 = 0$
2	1	$2 \log 2 - 2 + 1 = 1$
4	5	$4 \log 4 - 4 + 1 = 5$
8	17	$8 \log 8 - 8 + 1 = 17$
16	49	$16 \log 16 - 16 + 1 = 49$

As a double-check, we could write out an explicit induction proof. This would be straightforward, because we already worked out the guts of the proof in step 2 of the plug-and-chug procedure.

10.3 Linear Recurrences

So far we’ve solved recurrences with two techniques: guess-and-verify and plug-and-chug. These methods require spotting a pattern in a sequence of numbers or expressions. In this section and the next, we’ll give cookbook solutions for two large classes of recurrences. These methods require no flash of insight; you just follow the recipe and get the answer.

10.3.1 Climbing Stairs

How many different ways are there to climb n stairs, if you can either step up one stair or hop up two? For example, there are five different ways to climb four stairs:

1. step, step, step, step

2. hop, hop
3. hop, step, step
4. step, hop step
5. step, step, hop

Working through this problem will demonstrate the major features of our first cookbook method for solving recurrences. We’ll fill in the details of the general solution afterward.

Finding a Recurrence

As special cases, there is 1 way to climb 0 stairs (do nothing) and 1 way to climb 1 stair (step up). In general, an ascent of n stairs consists of either a step followed by an ascent of the remaining $n - 1$ stairs or a hop followed by an ascent of $n - 2$ stairs. So the total number of ways to climb n stairs is equal to the number of ways to climb $n - 1$ plus the number of ways to climb $n - 2$. These observations define a recurrence:

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(n) &= f(n - 1) + f(n - 2) \quad \text{for } n \geq 2. \end{aligned}$$

Here, $f(n)$ denotes the number of ways to climb n stairs. Also, we’ve switched from subscript notation to functional notation, from T_n to f_n . Here the change is cosmetic, but the expressiveness of functions will be useful later.

This is the Fibonacci recurrence, the most famous of all recurrence equations. Fibonacci numbers arise in all sorts of applications and in nature. Fibonacci introduced the numbers in 1202 to study rabbit reproduction. Fibonacci numbers also appear, oddly enough, in the spiral patterns on the faces of sunflowers. And the input numbers that make Euclid’s GCD algorithm require the greatest number of steps are consecutive Fibonacci numbers.

Solving the Recurrence

The Fibonacci recurrence belongs to the class of linear recurrences, which are essentially all solvable with a technique that you can learn in an hour. This is somewhat amazing, since the Fibonacci recurrence remained unsolved for almost six centuries!

In general, a *homogeneous linear recurrence* has the form

$$f(n) = a_1 f(n - 1) + a_2 f(n - 2) + \dots + a_d f(n - d)$$

where a_1, a_2, \dots, a_d and d are constants. The *order* of the recurrence is d . Commonly, the value of the function f is also specified at a few points; these are called *boundary conditions*. For example, the Fibonacci recurrence has order $d = 2$ with coefficients $a_1 = a_2 = 1$ and $g(n) = 0$. The boundary conditions are $f(0) = 1$ and $f(1) = 1$. The word “homogeneous” sounds scary, but effectively means “the simpler kind”. We’ll consider linear recurrences with a more complicated form later.

Let’s try to solve the Fibonacci recurrence with the benefit centuries of hindsight. In general, linear recurrences tend to have exponential solutions. So let’s guess that

$$f(n) = x^n$$

where x is a parameter introduced to improve our odds of making a correct guess. We’ll figure out the best value for x later. To further improve our odds, let’s neglect the boundary conditions, $f(0) = 0$ and $f(1) = 1$, for now. Plugging this guess into the recurrence $f(n) = f(n - 1) + f(n - 2)$ gives

$$x^n = x^{n-1} + x^{n-2}.$$

Dividing both sides by x^{n-2} leaves a quadratic equation:

$$x^2 = x + 1.$$

Solving this equation gives *two* plausible values for the parameter x :

$$x = \frac{1 \pm \sqrt{5}}{2}.$$

This suggests that there are at least two different solutions to the recurrence, neglecting the boundary conditions.

$$f(n) = \left(\frac{1 + \sqrt{5}}{2}\right)^n \quad \text{or} \quad f(n) = \left(\frac{1 - \sqrt{5}}{2}\right)^n$$

A charming feature of homogeneous linear recurrences is that any linear combination of solutions is another solution.

Theorem 10.3.1. *If $f(n)$ and $g(n)$ are both solutions to a homogeneous linear recurrence, then $h(n) = sf(n) + tg(n)$ is also a solution for all $s, t \in \mathbb{R}$.*

Proof.

$$\begin{aligned} h(n) &= sf(n) + tg(n) \\ &= s(a_1 f(n-1) + \dots + a_d f(n-d)) + t(a_1 g(n-1) + \dots + a_d g(n-d)) \\ &= a_1(sf(n-1) + tg(n-1)) + \dots + a_d(sf(n-d) + tg(n-d)) \\ &= a_1 h(n-1) + \dots + a_d h(n-d) \end{aligned}$$

The first step uses the definition of the function h , and the second uses the fact that f and g are solutions to the recurrence. In the last two steps, we rearrange terms and use the definition of h again. Since the first expression is equal to the last, h is also a solution to the recurrence. ■

The phenomenon described in this theorem—a linear combination of solutions is another solution—also holds for many differential equations and physical systems. In fact, linear recurrences are so similar to linear differential equations that you can safely snooze through that topic in some future math class.

Returning to the Fibonacci recurrence, this theorem implies that

$$f(n) = s \left(\frac{1 + \sqrt{5}}{2} \right)^n + t \left(\frac{1 - \sqrt{5}}{2} \right)^n$$

is a solution for all real numbers s and t . The theorem expanded two solutions to a whole spectrum of possibilities! Now, given all these options to choose from, we can find one solution that satisfies the boundary conditions, $f(0) = 0$ and $f(1) = 1$. Each boundary condition puts some constraints on the parameters s and t . In particular, the first boundary condition implies that

$$f(0) = s \left(\frac{1 + \sqrt{5}}{2} \right)^0 + t \left(\frac{1 - \sqrt{5}}{2} \right)^0 = s + t = 1.$$

Similarly, the second boundary condition implies that

$$f(1) = s \left(\frac{1 + \sqrt{5}}{2} \right)^1 + t \left(\frac{1 - \sqrt{5}}{2} \right)^1 = 1.$$

Now we have two linear equations in two unknowns. The system is not degenerate, so there is a unique solution:

$$s = \frac{1}{\sqrt{5}} \cdot \frac{1 + \sqrt{5}}{2} \quad t = -\frac{1}{\sqrt{5}} \cdot \frac{1 - \sqrt{5}}{2}.$$

These values of s and t identify a solution to the Fibonacci recurrence that also satisfies the boundary conditions:

$$\begin{aligned} f(n) &= \frac{1}{\sqrt{5}} \cdot \frac{1 + \sqrt{5}}{2} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \cdot \frac{1 - \sqrt{5}}{2} \left(\frac{1 - \sqrt{5}}{2} \right)^n \\ &= \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^{n+1} - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^{n+1}. \end{aligned}$$

It is easy to see why no one stumbled across this solution for almost six centuries! All Fibonacci numbers are integers, but this expression is full of square roots of five! Amazingly, the square roots always cancel out. This expression really does give the Fibonacci numbers if we plug in $n = 0, 1, 2$, etc.

This closed-form for Fibonacci numbers has some interesting corollaries. The first term tends to infinity because the base of the exponential, $(1 + \sqrt{5})/2 \approx 1.618\dots$ is greater than one. This value is often denoted ϕ and called the “golden ratio”. The second term tends to zero, because $(1 - \sqrt{5})/2 \approx -0.618033988\dots$ has absolute value less than 1. This implies that the n th Fibonacci number is:

$$f(n) = \frac{\phi^{n+1}}{\sqrt{5}} + o(1).$$

Remarkably, this expression involving irrational numbers is actually very close to an integer for all large n —namely, a Fibonacci number! For example:

$$\frac{\phi^{20}}{\sqrt{5}} = 6765.000029\dots \approx f(19).$$

This also implies that the ratio of consecutive Fibonacci numbers rapidly approaches the golden ratio. For example:

$$\frac{f(20)}{f(19)} = \frac{10946}{6765} = 1.618033998\dots$$

10.3.2 Solving Homogeneous Linear Recurrences

The method we used to solve the Fibonacci recurrence can be extended to solve any homogeneous linear recurrence; that is, a recurrence of the form

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + \dots + a_d f(n-d)$$

where a_1, a_2, \dots, a_d and d are constants. Substituting the guess $f(n) = x^n$, as with the Fibonacci recurrence, gives

$$x^n = a_1 x^{n-1} + a_2 x^{n-2} + \dots + a_d x^{n-d}.$$

Dividing by x^{n-d} gives

$$x^d = a_1 x^{d-1} + a_2 x^{d-2} + \dots + a_{d-1} x + a_d.$$

This is called the *characteristic equation* of the recurrence. The characteristic equation can be read off quickly since the coefficients of the equation are the same as the coefficients of the recurrence.

The solutions to a linear recurrence are defined by the roots of the characteristic equation. Neglecting boundary conditions for the moment:

- If r is a nonrepeated root of the characteristic equation, then r^n is a solution to the recurrence.
- If r is a repeated root with multiplicity k then $r^n, nr^n, n^2r^n, \dots, n^{k-1}r^n$ are all solutions to the recurrence.

Theorem 10.3.1 implies that every linear combination of these solutions is also a solution.

For example, suppose that the characteristic equation of a recurrence has roots s , t , and u twice. These four roots imply four distinct solutions:

$$f(n) = s^n \quad f(n) = t^n \quad f(n) = u^n \quad f(n) = nu^n.$$

Furthermore, every linear combination

$$f(n) = a \cdot s^n + b \cdot t^n + c \cdot u^n + d \cdot nu^n \tag{10.1}$$

is also a solution.

All that remains is to select a solution consistent with the boundary conditions by choosing the constants appropriately. Each boundary condition implies a linear equation involving these constants. So we can determine the constants by solving a system of linear equations. For example, suppose our boundary conditions were $f(0) = 0$, $f(1) = 1$, $f(2) = 4$, and $f(3) = 9$. Then we would obtain four equations in four unknowns:

$$\begin{aligned} f(0) = 0 &\Rightarrow a \cdot s^0 + b \cdot t^0 + c \cdot u^0 + d \cdot 0u^0 = 0 \\ f(1) = 1 &\Rightarrow a \cdot s^1 + b \cdot t^1 + c \cdot u^1 + d \cdot 1u^1 = 1 \\ f(2) = 4 &\Rightarrow a \cdot s^2 + b \cdot t^2 + c \cdot u^2 + d \cdot 2u^2 = 4 \\ f(3) = 9 &\Rightarrow a \cdot s^3 + b \cdot t^3 + c \cdot u^3 + d \cdot 3u^3 = 9 \end{aligned}$$

This looks nasty, but remember that s , t , and u are just constants. Solving this system gives values for a , b , c , and d that define a solution to the recurrence consistent with the boundary conditions.

10.3.3 Solving General Linear Recurrences

We can now solve all linear homogeneous recurrences, which have the form

$$f(n) = a_1 f(n - 1) + a_2 f(n - 2) + \dots + a_d f(n - d).$$

Many recurrences that arise in practice do not quite fit this mold. For example, the Towers of Hanoi problem led to this recurrence:

$$\begin{aligned} f(1) &= 1 \\ f(n) &= 2f(n - 1) + 1 \quad (\text{for } n \geq 2). \end{aligned}$$

The problem is the extra $+1$; that is not allowed in a homogeneous linear recurrence. In general, adding an extra function $g(n)$ to the right side of a linear recurrence gives an *inhomogeneous linear recurrence*:

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + \dots + a_d f(n-d) + g(n).$$

Solving inhomogeneous linear recurrences is neither very different nor very difficult. We can divide the whole job into five steps:

1. Replace $g(n)$ by 0, leaving a homogeneous recurrence. As before, find roots of the characteristic equation.
2. Write down the solution to the homogeneous recurrence, but do not yet use the boundary conditions to determine coefficients. This is called the *homogeneous solution*.
3. Now restore $g(n)$ and find a single solution to the recurrence, ignoring boundary conditions. This is called a *particular solution*. We’ll explain how to find a particular solution shortly.
4. Add the homogeneous and particular solutions together to obtain the *general solution*.
5. Now use the boundary conditions to determine constants by the usual method of generating and solving a system of linear equations.

As an example, let’s consider a variation of the Towers of Hanoi problem. Suppose that moving a disk takes time proportional to its size. Specifically, moving the smallest disk takes 1 second, the next-smallest takes 2 seconds, and moving the n th disk then requires n seconds instead of 1. So, in this variation, the time to complete the job is given by a recurrence with a $+n$ term instead of a $+1$:

$$\begin{aligned} f(1) &= 1 \\ f(n) &= 2f(n-1) + n \quad \text{for } n \geq 2. \end{aligned}$$

Clearly, this will take longer, but how much longer? Let’s solve the recurrence with the method described above.

In Steps 1 and 2, dropping the $+n$ leaves the homogeneous recurrence $f(n) = 2f(n-1)$. The characteristic equation is $x = 2$. So the homogeneous solution is $f(n) = c2^n$.

In Step 3, we must find a solution to the full recurrence $f(n) = 2f(n-1) + n$, without regard to the boundary condition. Let’s guess that there is a solution of the

form $f(n) = an + b$ for some constants a and b . Substituting this guess into the recurrence gives

$$\begin{aligned} an + b &= 2(a(n-1) + b) + n \\ 0 &= (a+1)n + (b-2a). \end{aligned}$$

The second equation is a simplification of the first. The second equation holds for all n if both $a+1=0$ (which implies $a=-1$) and $b-2a=0$ (which implies that $b=-2$). So $f(n) = an + b = -n - 2$ is a particular solution.

In the Step 4, we add the homogeneous and particular solutions to obtain the general solution

$$f(n) = c2^n - n - 2.$$

Finally, in step 5, we use the boundary condition, $f(1) = 1$, determine the value of the constant c :

$$\begin{aligned} f(1) = 1 &\Rightarrow c2^1 - 1 - 2 = 1 \\ &\Rightarrow c = 2. \end{aligned}$$

Therefore, the function $f(n) = 2 \cdot 2^n - n - 2$ solves this variant of the Towers of Hanoi recurrence. For comparison, the solution to the original Towers of Hanoi problem was $2^n - 1$. So if moving disks takes time proportional to their size, then the monks will need about twice as much time to solve the whole puzzle.

10.3.4 How to Guess a Particular Solution

Finding a particular solution can be the hardest part of solving inhomogeneous recurrences. This involves guessing, and you might guess wrong.¹ However, some rules of thumb make this job fairly easy most of the time.

- Generally, look for a particular solution with the same form as the inhomogeneous term $g(n)$.
- If $g(n)$ is a constant, then guess a particular solution $f(n) = c$. If this doesn't work, try polynomials of progressively higher degree: $f(n) = bn + c$, then $f(n) = an^2 + bn + c$, etc.
- More generally, if $g(n)$ is a polynomial, try a polynomial of the same degree, then a polynomial of degree one higher, then two higher, etc. For example, if $g(n) = 6n + 5$, then try $f(n) = bn + c$ and then $f(n) = an^2 + bn + c$.

¹In Chapter 12, we will show how to solve linear recurrences with generating functions—it's a little more complicated, but it does not require guessing.

- If $g(n)$ is an exponential, such as 3^n , then first guess that $f(n) = c3^n$. Failing that, try $f(n) = bn3^n + c3^n$ and then $an^23^n + bn3^n + c3^n$, etc.

The entire process is summarized on the following page.

10.4 Divide-and-Conquer Recurrences

We now have a recipe for solving general linear recurrences. But the Merge Sort recurrence, which we encountered earlier, is not linear:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(n/2) + n - 1 \quad (\text{for } n \geq 2). \end{aligned}$$

In particular, $T(n)$ is not a linear combination of a fixed number of immediately preceding terms; rather, $T(n)$ is a function of $T(n/2)$, a term halfway back in the sequence.

Merge Sort is an example of a divide-and-conquer algorithm: it divides the input, “conquers” the pieces, and combines the results. Analysis of such algorithms commonly leads to *divide-and-conquer* recurrences, which have this form:

$$T(n) = \sum_{i=1}^k a_i T(b_i n) + g(n)$$

Here a_1, \dots, a_k are positive constants, b_1, \dots, b_k are constants between 0 and 1, and $g(n)$ is a nonnegative function. For example, setting $a_1 = 2$, $b_1 = 1/2$, and $g(n) = n - 1$ gives the Merge Sort recurrence.

10.4.1 The Akra-Bazzi Formula

The solution to virtually all divide and conquer solutions is given by the amazing *Akra-Bazzi formula*. Quite simply, the asymptotic solution to the general divide-and-conquer recurrence

$$T(n) = \sum_{i=1}^k a_i T(b_i n) + g(n)$$

is

$$T(n) = \Theta \left(n^p \left(1 + \int_1^n \frac{g(u)}{u^{p+1}} du \right) \right) \quad (10.2)$$

Short Guide to Solving Linear Recurrences

A linear recurrence is an equation

$$f(n) = \underbrace{a_1 f(n-1) + a_2 f(n-2) + \dots + a_d f(n-d)}_{\text{homogeneous part}} + \underbrace{g(n)}_{\text{inhomogeneous part}}$$

together with boundary conditions such as $f(0) = b_0, f(1) = b_1$, etc. Linear recurrences are solved as follows:

1. Find the roots of the characteristic equation

$$x^n = a_1 x^{n-1} + a_2 x^{n-2} + \dots + a_{k-1} x + a_k.$$

2. Write down the homogeneous solution. Each root generates one term and the homogeneous solution is their sum. A nonrepeated root r generates the term cr^n , where c is a constant to be determined later. A root r with multiplicity k generates the terms

$$d_1 r^n \quad d_2 n r^n \quad d_3 n^2 r^n \quad \dots \quad d_k n^{k-1} r^n$$

where d_1, \dots, d_k are constants to be determined later.

3. Find a particular solution. This is a solution to the full recurrence that need not be consistent with the boundary conditions. Use guess-and-verify. If $g(n)$ is a constant or a polynomial, try a polynomial of the same degree, then of one higher degree, then two higher. For example, if $g(n) = n$, then try $f(n) = bn + c$ and then $an^2 + bn + c$. If $g(n)$ is an exponential, such as 3^n , then first guess $f(n) = c3^n$. Failing that, try $f(n) = (bn + c)3^n$ and then $(an^2 + bn + c)3^n$, etc.
4. Form the general solution, which is the sum of the homogeneous solution and the particular solution. Here is a typical general solution:

$$f(n) = \underbrace{c2^n + d(-1)^n}_{\text{homogeneous solution}} + \underbrace{3n + 1}_{\text{inhomogeneous solution}}$$

5. Substitute the boundary conditions into the general solution. Each boundary condition gives a linear equation in the unknown constants. For example, substituting $f(1) = 2$ into the general solution above gives

$$\begin{aligned} 2 &= c \cdot 2^1 + d \cdot (-1)^1 + 3 \cdot 1 + 1 \\ \Rightarrow -2 &= 2c - d. \end{aligned}$$

Determine the values of these constants by solving the resulting system of linear equations.

where p satisfies

$$\sum_{i=1}^k a_i b_i^p = 1. \tag{10.3}$$

A rarely-troublesome requirement is that the function $g(n)$ must not grow or oscillate too quickly. Specifically, $|g'(n)|$ must be bounded by some polynomial. So, for example, the Akra-Bazzi formula is valid when $g(n) = x^2 \log n$, but not when $g(n) = 2^n$.

Let’s solve the Merge Sort recurrence again, using the Akra-Bazzi formula instead of plug-and-chug. First, we find the value p that satisfies

$$2 \cdot (1/2)^p = 1.$$

Looks like $p = 1$ does the job. Then we compute the integral:

$$\begin{aligned} T(n) &= \Theta \left(n \left(1 + \int_1^n \frac{u-1}{u^2} du \right) \right) \\ &= \Theta \left(n \left(1 + \left(\log u + \frac{1}{u} \right) \Big|_1^n \right) \right) \\ &= \Theta \left(n \left(\log n + \frac{1}{n} \right) \right) \\ &= \Theta(n \log n). \end{aligned}$$

The first step is integration and the second is simplification. We can drop the $1/n$ term in the last step, because the $\log n$ term dominates. We’re done!

Let’s try a scary-looking recurrence:

$$T(n) = 2T(n/2) + 8/9T(3n/4) + n^2.$$

Here, $a_1 = 2$, $b_1 = 1/2$, $a_2 = 8/9$, and $b_2 = 3/4$. So we find the value p that satisfies

$$2 \cdot (1/2)^p + (8/9)(3/4)^p = 1.$$

Equations of this form don’t always have closed-form solutions, so you may need to approximate p numerically sometimes. But in this case the solution is simple: $p = 2$. Then we integrate:

$$\begin{aligned} T(n) &= \Theta \left(n^2 \left(1 + \int_1^n \frac{u^2}{u^3} du \right) \right) \\ &= \Theta(n^2(1 + \log n)) \\ &= \Theta(n^2 \log n) \end{aligned}$$

That was easy!

10.4.2 Two Technical Issues

Until now, we’ve swept a couple issues related to divide-and-conquer recurrences under the rug. Let’s address those issues now.

First, the Akra-Bazzi formula makes no use of boundary conditions. To see why, let’s go back to Merge Sort. During the plug-and-chug analysis, we found that

$$T_n = nT_1 + n \log n - n + 1.$$

This expresses the n th term as a function of the first term, whose value is specified in a boundary condition. But notice that $T_n = \Theta(n \log n)$ for *every* value of T_1 . The boundary condition doesn’t matter!

This is the typical situation: *the asymptotic solution to a divide-and-conquer recurrence is independent of the boundary conditions*. Intuitively, if the bottom-level operation in a recursive algorithm takes, say, twice as long, then the overall running time will at most double. This matters in practice, but the factor of 2 is concealed by asymptotic notation. There are corner-case exceptions. For example, the solution to $T(n) = 2T(n/2)$ is either $\Theta(n)$ or zero, depending on whether $T(1)$ is zero. These cases are of little practical interest, so we won’t consider them further.

There is a second nagging issue with divide-and-conquer recurrences that does not arise with linear recurrences. Specifically, dividing a problem of size n may create subproblems of non-integer size. For example, the Merge Sort recurrence contains the term $T(n/2)$. So what if n is 15? How long does it take to sort seven-and-a-half items? Previously, we dodged this issue by analyzing Merge Sort only when the size of the input was a power of 2. But then we don’t know what happens for an input of size, say, 100.

Of course, a practical implementation of Merge Sort would split the input *approximately* in half, sort the halves recursively, and merge the results. For example, a list of 15 numbers would be split into lists of 7 and 8. More generally, a list of n numbers would be split into approximate halves of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. So the maximum number of comparisons is actually given by this recurrence:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1 \quad (\text{for } n \geq 2). \end{aligned}$$

This may be rigorously correct, but the ceiling and floor operations make the recurrence hard to solve exactly.

Fortunately, *the asymptotic solution to a divide and conquer recurrence is unaffected by floors and ceilings*. More precisely, the solution is not changed by replacing a term $T(b_i n)$ with either $T(\lceil b_i n \rceil)$ or $T(\lfloor b_i n \rfloor)$. So leaving floors and

ceilings out of divide-and-conquer recurrences makes sense in many contexts; those are complications that make no difference.

10.4.3 The Akra-Bazzi Theorem

The Akra-Bazzi formula together with our assertions about boundary conditions and integrality all follow from the *Akra-Bazzi Theorem*, which is stated below.

Theorem 10.4.1 (Akra-Bazzi). *Suppose that the function $T : \mathbb{R} \rightarrow \mathbb{R}$ satisfies the recurrence*

$$T(x) = \begin{cases} \text{is nonnegative and bounded} & \text{for } 0 \leq x \leq x_0 \\ \sum_{i=1}^k a_i T(b_i x + h_i(x)) + g(x) & \text{for } x > x_0 \end{cases}$$

where:

1. a_1, \dots, a_k are positive constants.
2. b_1, \dots, b_k are constants between 0 and 1.
3. x_0 is large enough so that T is well-defined.
4. $g(x)$ is a nonnegative function such that $|g'(x)|$ is bounded by a polynomial.
5. $|h_i(x)| = O(x/\log^2 x)$.

Then

$$T(x) = \Theta \left(x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du \right) \right)$$

where p satisfies

$$\sum_{i=1}^k a_i b_i^p = 1.$$

The Akra-Bazzi theorem can be proved using a complicated induction argument, though we won't do that here. But let's at least go over the statement of the theorem.

All the recurrences we've considered were defined over the integers, and that is the common case. But the Akra-Bazzi theorem applies more generally to functions defined over the real numbers.

The Akra-Bazzi formula is lifted directly from the theorem statement, except that the recurrence in the theorem includes extra functions, h_i . These functions

extend the theorem to address floors, ceilings, and other small adjustments to the sizes of subproblems. The trick is illustrated by this combination of parameters

$$\begin{aligned}
 a_1 &= 1 & b_1 &= 1/2 & h_1(x) &= \left\lceil \frac{x}{2} \right\rceil \left(\frac{x}{2} \right) \\
 a_2 &= 1 & b_2 &= 1/2 & h_2(x) &= \left\lfloor \frac{x}{2} \right\rfloor \left(\frac{x}{2} \right) \\
 & & & & g(x) &= x - 1
 \end{aligned}$$

which corresponds the recurrence

$$\begin{aligned}
 T(x) &= 1 \cdot T\left(\left\lceil \frac{x}{2} \right\rceil\right) + \left(\left\lceil \frac{x}{2} \right\rceil \left(\frac{x}{2} \right) \right) + T\left(\left\lfloor \frac{x}{2} \right\rfloor\right) + \left(\left\lfloor \frac{x}{2} \right\rfloor \left(\frac{x}{2} \right) \right) + x - 1 \\
 &= T\left(\left\lceil \frac{x}{2} \right\rceil\right) + T\left(\left\lfloor \frac{x}{2} \right\rfloor\right) + x - 1.
 \end{aligned}$$

This is the rigorously correct Merge Sort recurrence valid for all input sizes, complete with floor and ceiling operators. In this case, the functions $h_1(x)$ and $h_2(x)$ are both at most 1, which is easily $O(x/\log^2 x)$ as required by the theorem statement. These functions h_i do not affect—or even appear in—the asymptotic solution to the recurrence. This justifies our earlier claim that applying floor and ceiling operators to the size of a subproblem does not alter the asymptotic solution to a divide-and-conquer recurrence.

10.4.4 The Master Theorem

There is a special case of the Akra-Bazzi formula known as the Master Theorem that handles some of the recurrences that commonly arise in computer science. It is called the *Master* Theorem because it was proved long before Akra and Bazzi arrived on the scene and, for many years, it was the final word on solving divide-and-conquer recurrences. We include the Master Theorem here because it is still widely referenced in algorithms courses and you can use it without having to know anything about integration.

Theorem 10.4.2 (Master Theorem). *Let T be a recurrence of the form*

$$T(n) = aT\left(\frac{n}{b}\right) + g(n).$$

Case 1: *If $g(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ for some constant $\epsilon > 0$, then*

$$T(n) = \Theta\left(n^{\log_b(a)}\right)$$

Case 2: If $g(n) = \Theta(n^{\log_b(a)} \log^k(n))$ for some constant $k \geq 0$, then

$$T(n) = \Theta(n^{\log_b(a)} \log^{k+1}(n))$$

Case 3: If $g(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some constant $\epsilon > 0$ and $ag(n/b) < cg(n)$ for some constant $c < 1$ and sufficiently large n , then

$$T(n) = \Theta(g(n)).$$

The Master Theorem can be proved by induction on n or, more easily, as a corollary of Theorem 10.4.1. We will not include the details here.

10.4.5 Pitfalls with Asymptotic Notation and Induction

We’ve seen that asymptotic notation is quite useful, particularly in connection with recurrences. And induction is our favorite proof technique. But mixing the two is risky business; there is great potential for subtle errors and false conclusions!

False Claim. If

$$\begin{aligned} T(1) &= 1 \quad \text{and} \\ T(n) &= 2T(n/2) + n, \end{aligned}$$

then $T(n) = O(n)$.

The Akra-Bazzi theorem implies that the correct solution is $T(n) = \Theta(n \log n)$ and so this claim is false. But where does the following “proof” go astray?

Bogus proof. The proof is by strong induction. Let $P(n)$ be the proposition that $T(n) = O(n)$.

Base case: $P(1)$ is true because $T(1) = 1 = O(1)$.

Inductive step: For $n \geq 2$, assume $P(1), P(2), \dots, P(n-1)$ to prove $P(n)$. We have

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + n \\ &= 2 \cdot O(n/2) + n \\ &= O(n). \end{aligned}$$

The first equation is the recurrence, the second uses the assumption $P(n/2)$, and the third is a simplification. ■

Where’s the bug? The proof is already far off the mark in the second sentence, which defines the induction hypothesis. The statement “ $T(n) = O(n)$ ” is either true or false; it’s validity does not depend on a particular value of n . Thus the very idea of trying to prove that the statement holds for $n = 1, 2, \dots$, is wrong-headed.

The safe way to reason inductively about asymptotic phenomena is to *work directly with the definition of the asymptotic notation*. Let’s try to prove the claim above in this way. Remember that $f(n) = O(n)$ means that there exist constants n_0 and $c > 0$ such that $|f(n)| \leq cn$ for all $n \geq n_0$. (Let’s not worry about the absolute value for now.) If all goes well, the proof attempt should fail in some blatantly obvious way, instead of in a subtle, hard-to-detect way like the earlier argument. Since our perverse goal is to demonstrate that the proof won’t work for *any* constants n_0 and c , we’ll leave these as variables and assume only that they’re chosen so that the base case holds; that is, $T(n_0) \leq cn$.

Proof Attempt. We use strong induction. Let $P(n)$ be the proposition that $T(n) \leq cn$.

Base case: $P(n_0)$ is true, because $T(n_0) \leq cn$.

Inductive step: For $n > n_0$, assume that $P(n_0), \dots, P(n - 1)$ are true in order to prove $P(n)$. We reason as follows:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2c(n/2) + n \\ &= cn + n \\ &= (c + 1)n \\ &\not\leq cn. \quad \blacksquare \end{aligned}$$

The first equation is the recurrence. Then we use induction and simplify until the argument collapses!

In general, it is a good idea to stay away from asymptotic notation altogether while you are doing the induction. Once the induction is over and done with, then you can safely use big-Oh to simplify your result.

10.5 A Feel for Recurrences

We’ve guessed and verified, plugged and chugged, found roots, computed integrals, and solved linear systems and exponential equations. Now let’s step back and look for some rules of thumb. What kinds of recurrences have what sorts of solutions?

Here are some recurrences we solved earlier:

	Recurrence	Solution
Towers of Hanoi	$T_n = 2T_{n-1} + 1$	$T_n \sim 2^n$
Merge Sort	$T_n = 2T_{n/2} + n - 1$	$T_n \sim n \log n$
Hanoi variation	$T_n = 2T_{n-1} + n$	$T_n \sim 2 \cdot 2^n$
Fibonacci	$T_n = T_{n-1} + T_{n-2}$	$T_n \sim (1.618 \dots)^{n+1} / \sqrt{5}$

Notice that the recurrence equations for Towers of Hanoi and Merge Sort are somewhat similar, but the solutions are radically different. Merge Sorting $n = 64$ items takes a few hundred comparisons, while moving $n = 64$ disks takes more than 10^{19} steps!

Each recurrence has one strength and one weakness. In the Towers of Hanoi, we broke a problem of size n into two subproblem of size $n - 1$ (which is large), but needed only 1 additional step (which is small). In Merge Sort, we divided the problem of size n into two subproblems of size $n/2$ (which is small), but needed $(n - 1)$ additional steps (which is large). Yet, Merge Sort is faster by a mile!

This suggests that *generating smaller subproblems is far more important to algorithmic speed than reducing the additional steps per recursive call*. For example, shifting to the variation of Towers of Hanoi increased the last term from $+1$ to $+n$, but the solution only doubled. And one of the two subproblems in the Fibonacci recurrence is just *slightly* smaller than in Towers of Hanoi (size $n - 2$ instead of $n - 1$). Yet the solution is exponentially smaller! More generally, linear recurrences (which have big subproblems) typically have exponential solutions, while divide-and-conquer recurrences (which have small subproblems) usually have solutions bounded above by a polynomial.

All the examples listed above break a problem of size n into two smaller problems. How does the number of subproblems affect the solution? For example, suppose we increased the number of subproblems in Towers of Hanoi from 2 to 3, giving this recurrence:

$$T_n = 3T_{n-1} + 1$$

This increases the root of the characteristic equation from 2 to 3, which raises the solution exponentially, from $\Theta(2^n)$ to $\Theta(3^n)$.

Divide-and-conquer recurrences are also sensitive to the number of subproblems. For example, for this generalization of the Merge Sort recurrence:

$$T_1 = 0$$

$$T_n = aT_{n/2} + n - 1.$$

the Akra-Bazzi formula gives:

$$T_n = \begin{cases} \Theta(n) & \text{for } a < 2 \\ \Theta(n \log n) & \text{for } a = 2 \\ \Theta(n^{\log a}) & \text{for } a > 2. \end{cases}$$

So the solution takes on three completely different forms as a goes from 1.99 to 2.01!

How do boundary conditions affect the solution to a recurrence? We’ve seen that they are almost irrelevant for divide-and-conquer recurrences. For linear recurrences, the solution is usually dominated by an exponential whose base is determined by the number and size of subproblems. Boundary conditions matter greatly only when they give the dominant term a zero coefficient, which changes the asymptotic solution.

So now we have a rule of thumb! The performance of a recursive procedure is usually dictated by the size and number of subproblems, rather than the amount of work per recursive call or time spent at the base of the recursion. In particular, if subproblems are smaller than the original by an additive factor, the solution is most often exponential. But if the subproblems are only a fraction the size of the original, then the solution is typically bounded by a polynomial.

MIT OpenCourseWare
<http://ocw.mit.edu>

6.042J / 18.062J Mathematics for Computer Science
Fall 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.