From
Cormen, Leiserson, Rivest,
Introduction to Algorithms (1990)

4 Recurrences

As noted in Chapter 1, when an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs. For example, we saw in Chapter 1 that the worst-case running time $T(n)$ of the MERGE-SORT procedure could be described by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 , \end{cases} \tag{4.1}$$

whose solution was claimed to be $T(n) = \Theta(n \lg n)$.

This chapter offers three methods for solving recurrences—that is, for obtaining asymptotic "Θ" or "O" bounds on the solution. In the ***substitution method***, we guess a bound and then use mathematical induction to prove our guess correct. The ***iteration method*** converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence. The ***master method*** provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n),$$

where $a \geq 1$, $b > 1$, and $f(n)$ is a given function; it requires memorization of three cases, but once you do that, determining asymptotic bounds for many simple recurrences is easy.

Technicalities

In practice, we neglect certain technical details when we state and solve recurrences. A good example of a detail that is often glossed over is the assumption of integer arguments to functions. Normally, the running time $T(n)$ of an algorithm is only defined when n is an integer, since for most algorithms, the size of the input is always an integer. For example, the recurrence describing the worst-case running time of MERGE-SORT is really

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 . \end{cases} \tag{4.2}$$

Boundary conditions represent another class of details that we typically ignore. Since the running time of an algorithm on a constant-sized input is

a constant, the recurrences that arise from the running times of algorithms generally have $T(n) = \Theta(1)$ for sufficiently small n. Consequently, for convenience, we shall generally omit statements of the boundary conditions of recurrences and assume that $T(n)$ is constant for small n. For example, we normally state recurrence (4.1) as

$$T(n) = 2T(n/2) + \Theta(n) , \tag{4.3}$$

without explicitly giving values for small n. The reason is that although changing the value of $T(1)$ changes the solution to the recurrence, the solution typically doesn't change by more than a constant factor, so the order of growth is unchanged.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions. We forge ahead without these details and later determine whether or not they matter. They usually don't, but it is important to know when they do. Experience helps, and so do some theorems stating that these details don't affect the asymptotic bounds of many recurrences encountered in the analysis of algorithms (see Theorem 4.1 and Problem 4-5). In this chapter, however, we shall address some of these details to show the fine points of recurrence solution methods.

4.1 The substitution method

The substitution method for solving recurrences involves guessing the form of the solution and then using mathematical induction to find the constants and show that the solution works. The name comes from the substitution of the guessed answer for the function when the inductive hypothesis is applied to smaller values. This method is powerful, but it obviously can be applied only in cases when it is easy to guess the form of the answer.

The substitution method can be used to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n , \tag{4.4}$$

which is similar to recurrences (4.2) and (4.3). We guess that the solution is $T(n) = O(n \lg n)$. Our method is to prove that $T(n) \leq cn \lg n$ for an appropriate choice of the constant $c > 0$. We start by assuming that this bound holds for $\lfloor n/2 \rfloor$, that is, that $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n , \end{aligned}$$

where the last step holds as long as $c \geq 1$.

Mathematical induction now requires us to show that our solution holds for the boundary conditions. That is, we must show that we can choose the constant c large enough so that the bound $T(n) \leq cn \lg n$ works for the boundary conditions as well. This requirement can sometimes lead to problems. Let us assume, for the sake of argument, that $T(1) = 1$ is the sole boundary condition of the recurrence. Then, unfortunately, we can't choose c large enough, since $T(1) \leq c1 \lg 1 = 0$.

This difficulty in proving an inductive hypothesis for a specific boundary condition can be easily overcome. We take advantage of the fact that asymptotic notation only requires us to prove $T(n) \leq cn \lg n$ for $n \geq n_0$, where n_0 is a constant. The idea is to remove the difficult boundary condition $T(1) = 1$ from consideration in the inductive proof and to include $n = 2$ and $n = 3$ as part of the boundary conditions for the proof. We can impose $T(2)$ and $T(3)$ as boundary conditions for the inductive proof because for $n > 3$, the recurrence does not depend directly on $T(1)$. From the recurrence, we derive $T(2) = 4$ and $T(3) = 5$. The inductive proof that $T(n) \leq cn \lg n$ for some constant $c \geq 2$ can now be completed by choosing c large enough so that $T(2) \leq c2 \lg 2$ and $T(3) \leq c3 \lg 3$. As it turns out, any choice of $c \geq 2$ suffices. For most of the recurrences we shall examine, it is straightforward to extend boundary conditions to make the inductive assumption work for small n.

Making a good guess

Unfortunately, there is no general way to guess the correct solutions to recurrences. Guessing a solution takes experience and, occasionally, creativity. Fortunately, though, there are some heuristics that can help you become a good guesser.

If a recurrence is similar to one you have seen before, then guessing a similar solution is reasonable. As an example, consider the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \ ,$$

which looks difficult because of the added "17" in the argument to T on the right-hand side. Intuitively, however, this additional term cannot substantially affect the solution to the recurrence. When n is large, the difference between $T(\lfloor n/2 \rfloor)$ and $T(\lfloor n/2 \rfloor + 17)$ is not that large: both cut n nearly evenly in half. Consequently, we make the guess that $T(n) = O(n \lg n)$, which you can verify as correct by using the substitution method (see Exercise 4.1-5).

Another way to make a good guess is to prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainty. For example, we might start with a lower bound of $T(n) = \Omega(n)$ for the recurrence (4.4), since we have the term n in the recurrence, and we can prove an initial upper bound of $T(n) = O(n^2)$. Then, we can gradually lower the

upper bound and raise the lower bound until we converge on the correct, asymptotically tight solution of $T(n) = \Theta(n \lg n)$.

Subtleties

There are times when you can correctly guess at an asymptotic bound on the solution of a recurrence, but somehow the math doesn't seem to work out in the induction. Usually, the problem is that the inductive assumption isn't strong enough to prove the detailed bound. When you hit such a snag, revising the guess by subtracting a lower-order term often permits the math to go through.

Consider the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \ .$$

We guess that the solution is $O(n)$, and we try to show that $T(n) \leq cn$ for an appropriate choice of the constant c. Substituting our guess in the recurrence, we obtain

$$\begin{aligned} T(n) &\leq c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 \\ &= cn + 1 \ , \end{aligned}$$

which does not imply $T(n) \leq cn$ for any choice of c. It's tempting to try a larger guess, say $T(n) = O(n^2)$, which can be made to work, but in fact, our guess that the solution is $T(n) = O(n)$ is correct. In order to show this, however, we must make a stronger inductive hypothesis.

Intuitively, our guess is nearly right: we're only off by the constant 1, a lower-order term. Nevertheless, mathematical induction doesn't work unless we prove the exact form of the inductive hypothesis. We overcome our difficulty by *subtracting* a lower-order term from our previous guess. Our new guess is $T(n) \leq cn - b$, where $b \geq 0$ is constant. We now have

$$\begin{aligned} T(n) &\leq (c \lfloor n/2 \rfloor - b) + (c \lceil n/2 \rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b \ , \end{aligned}$$

as long as $b \geq 1$. As before, the constant c must be chosen large enough to handle the boundary conditions.

Most people find the idea of subtracting a lower-order term counterintuitive. After all, if the math doesn't work out, shouldn't we be increasing our guess? The key to understanding this step is to remember that we are using mathematical induction: we can prove something stronger for a given value by assuming something stronger for smaller values.

Avoiding pitfalls

It is easy to err in the use of asymptotic notation. For example, in the recurrence (4.4) we can falsely prove $T(n) = O(n)$ by guessing $T(n) \leq cn$

and then arguing

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2\rfloor) + n \\ &\le cn + n \\ &= O(n)\,, \qquad \Longleftarrow \textit{wrong!!} \end{aligned}$$

since c is a constant. The error is that we haven't proved the exact form of the inductive hypothesis, that is, that $T(n) \le cn$.

Changing variables

Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one you have seen before. As an example, consider the recurrence

$$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n\,,$$

which looks difficult. We can simplify this recurrence, though, with a change of variables. For convenience, we shall not worry about rounding off values, such as $\sqrt{n}$, to be integers. Renaming $m = \lg n$ yields

$$T(2^m) = 2T(2^{m/2}) + m\,.$$

We can now rename $S(m) = T(2^m)$ to produce the new recurrence

$$S(m) = 2S(m/2) + m\,,$$

which is very much like recurrence (4.4) and has the same solution: $S(m) = O(m \lg m)$. Changing back from $S(m)$ to $T(n)$, we obtain $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg\lg n)$.

Exercises

4.1-1
Show that the solution of $T(n) = T(\lceil n/2\rceil) + 1$ is $O(\lg n)$.

4.1-2
Show that the solution of $T(n) = 2T(\lfloor n/2\rfloor) + n$ is $\Omega(n \lg n)$. Conclude that the solution is $\Theta(n \lg n)$.

4.1-3
Show that by making a different inductive hypothesis, we can overcome the difficulty with the boundary condition $T(1) = 1$ for the recurrence (4.4) without adjusting the boundary conditions for the inductive proof.

4.1-4
Show that $\Theta(n \lg n)$ is the solution to the "exact" recurrence (4.2) for merge sort.

4.1-5
Show that the solution to $T(n) = 2T(\lfloor n/2\rfloor + 17) + n$ is $O(n \lg n)$.

4.1-6

Solve the recurrence $T(n) = 2T(\sqrt{n}) + 1$ by making a change of variables. Do not worry about whether values are integral.

4.2 The iteration method

The method of iterating a recurrence doesn't require us to guess the answer, but it may require more algebra than the substitution method. The idea is to expand (iterate) the recurrence and express it as a summation of terms dependent only on n and the initial conditions. Techniques for evaluating summations can then be used to provide bounds on the solution.

As an example, consider the recurrence

$$T(n) = 3T(\lfloor n/4 \rfloor) + n \ .$$

We iterate it as follows:

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) \\ &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) \ , \end{aligned}$$

where $\lfloor \lfloor n/4 \rfloor /4 \rfloor = \lfloor n/16 \rfloor$ and $\lfloor \lfloor n/16 \rfloor /4 \rfloor = \lfloor n/64 \rfloor$ follow from the identity (2.4).

How far must we iterate the recurrence before we reach a boundary condition? The ith term in the series is $3^i \lfloor n/4^i \rfloor$. The iteration hits $n = 1$ when $\lfloor n/4^i \rfloor = 1$ or, equivalently, when i exceeds $\log_4 n$. By continuing the iteration until this point and using the bound $\lfloor n/4^i \rfloor \le n/4^i$, we discover that the summation contains a decreasing geometric series:

$$\begin{aligned} T(n) &\le n + 3n/4 + 9n/16 + 27n/64 + \cdots + 3^{\log_4 n}\Theta(1) \\ &\le n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) \\ &= 4n + o(n) \\ &= O(n) \ . \end{aligned}$$

Here, we have used the identity (2.9) to conclude that $3^{\log_4 n} = n^{\log_4 3}$, and we have used the fact that $\log_4 3 < 1$ to conclude that $\Theta(n^{\log_4 3}) = o(n)$.

The iteration method usually leads to lots of algebra, and keeping everything straight can be a challenge. The key is to focus on two parameters: the number of times the recurrence needs to be iterated to reach the boundary condition, and the sum of the terms arising from each level of the iteration process. Sometimes, in the process of iterating a recurrence, you can guess the solution without working out all the math. Then, the iteration can be abandoned in favor of the substitution method, which usually requires less algebra.

4.3 The master method

The master method provides a "cookbook" method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n) , \tag{4.5}$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. The master method requires memorization of three cases, but then the solution of many recurrences can be determined quite easily, often without pencil and paper.

The recurrence (4.5) describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time $T(n/b)$. The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$. (That is, using the notation from Section 1.3.2, $f(n) = D(n) + C(n)$.) For example, the recurrence arising from the MERGE-SORT procedure has $a = 2$, $b = 2$, and $f(n) = \Theta(n)$.

As a matter of technical correctness, the recurrence isn't actually well defined because n/b might not be an integer. Replacing each of the a terms $T(n/b)$ with either $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ doesn't affect the asymptotic behavior of the recurrence, however. (We'll prove this in the next section.) We normally find it convenient, therefore, to omit the floor and ceiling functions when writing divide-and-conquer recurrences of this form.

The master theorem

The master method depends on the following theorem.

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Before applying the master theorem to some examples, let's spend a moment trying to understand what it says. In each of the three cases, we are comparing the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the solution to the recurrence is determined by the larger of the two functions. If, as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \Theta(n^{\log_b a})$. If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \Theta(f(n))$. If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Beyond this intuition, there are some technicalities that must be understood. In the first case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be *polynomially* smaller. That is, $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^ϵ for some constant $\epsilon > 0$. In the third case, not only must $f(n)$ be larger than $n^{\log_b a}$, it must be polynomially larger and in addition satisfy the "regularity" condition that $af(n/b) \leq cf(n)$. This condition is satisfied by most of the polynomially bounded functions that we shall encounter.

It is important to realize that the three cases do not cover all the possibilities for $f(n)$. There is a gap between cases 1 and 2 when $f(n)$ is smaller than $n^{\log_b a}$ but not polynomially smaller. Similarly, there is a gap between cases 2 and 3 when $f(n)$ is larger than $n^{\log_b a}$ but not polynomially larger. If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, the master method cannot be used to solve the recurrence.

Using the master method

To use the master method, we simply determine which case (if any) of the master theorem applies and write down the answer.

As a first example, consider

$$T(n) = 9T(n/3) + n .$$

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider

$$T(n) = T(2n/3) + 1,$$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

For the recurrence

$$T(n) = 3T(n/4) + n \lg n ,$$

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

The master method does not apply to the recurrence

$$T(n) = 2T(n/2) + n \lg n,$$

even though it has the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. It seems that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$ but not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^ϵ for any positive constant ϵ. Consequently, the recurrence falls into the gap between case 2 and case 3. (See Exercise 4.4-2 for a solution.)

Exercises

4.3-1

Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 4T(n/2) + n$.

b. $T(n) = 4T(n/2) + n^2$.

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

4.3-2

The running time of an algorithm A is described by the recurrence $T(n) = 7T(n/2) + n^2$. A competing algorithm A' has a running time of $T'(n) = aT'(n/4) + n^2$. What is the largest integer value for a such that A' is asymptotically faster than A?

4.3-3

Use the master method to show that the solution to the recurrence $T(n) = T(n/2) + \Theta(1)$ of binary search (see Exercise 1.3-5) is $T(n) = \Theta(\lg n)$.

4.3-4 ⋆

Consider the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$, which is part of case 3 of the master theorem. Give an example of a simple function $f(n)$ that satisfies all the conditions in case 3 of the master theorem except the regularity condition.