Previous Chapter Return to Table of Contents Next Chapter

CHAPTER 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

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

(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

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

(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

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

(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

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

(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 $T(n) \le cn \lg n$ 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 $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)$. Substituting into the recurrence yields

$$T(n) \le 2(c \lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)) + n$$

$$\le cn \lg(n/2) + n$$

$$= cn \lg n - cn \lg 2 + n$$

$$= cn \lg n - cn + n$$

$\leq cn \lg n,$

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 $T(n) \le cn$ for an appropriate choice of the constant c. Substituting our guess in the recurrence, we obtain

$$T(n) \le c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1$$

$$= cn + 1 ,$$

which does not imply $T(n) \le cn$ 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 $T(n) \le cn - b$, where $b \ge 0$ is constant. We now have

$$T(n) \le (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1$$

$$= cn - 2b + 1$$

$$\le cn - b ,$$

as long as $b \ge 1$. 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 $T(n) \le cn$ and then arguing

$$T(n) \le 2(c\lfloor n/2 \rfloor) + n$$

$\leq cn + n$

$= O(n)$, $\Leftarrow$ *wrong!!*

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

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:

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

$$= n + 3\left(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)\right)$$

$$= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor)))$$

$$= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor),$$

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.

When a recurrence contains floor and ceiling functions, the math can become especially complicated. Often, it helps to assume that the recurrence is defined only on exact powers of a number. In our example, if we had assumed that $n = 4^k$ for some integer k, the floor functions could have been conveniently omitted. Unfortunately, proving the bound $T(n) = O(n)$ solely for exact powers of 4 is technically an abuse of the O-notation. The definitions of asymptotic notation require that bounds be proved for *all* sufficiently large integers, not just those that are powers of 4. We shall see in Section 4.3 that for a large class of recurrences, this technicality can be overcome. Problem 4-5 also gives conditions under which an analysis for exact powers of an integer can be extended to all integers.

Recursion trees

Exercises

4.3 The master method

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

$$T(n) = aT(n/b) + \hat{a}(n),$$

(4.5)

where $a \geq 1$ and $b > 1$ are constants and $\hat{a}(n)$ 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 $\hat{a}(n)$. (That is, using the notation from Section 1.3.2, $\hat{a}(n) = D(n) + C(n)$.) For example, the recurrence arising from the MERGE-SORT procedure has $a = 2$, $b = 2$, and $\hat{a}(n) = \Theta(n)$.

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

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

$$T(n) = aT(n/b) + \hat{a}(n),$$

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 $\hat{a}(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $\hat{a}(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.

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

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 â(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 $T(n) = \Theta(\hat{a}(n^{\log_b a})$. If, as in case 3, the function â(n) is the larger, then the solution is $T(n) = \Theta(\hat{a}(n))$. If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\hat{a}(n) \lg n)$.

Beyond this intuition, there are some technicalities that must be understood. In the first case, not only must â(n) be smaller than $n^{\log_b a}$, it must be *polynomially* smaller. That is, â(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 â(n) be larger than $n^{\log_b a}$, it must be polynomially larger and in addition satisfy the "regularity" condition that $a\hat{a}(n/b) \leq c\hat{a}(n)$. 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 â(n). There is a gap between cases 1 and 2 when â(n) is smaller than $n^{\log_b a}$ but not polynomially smaller. Similarly, there is a gap between cases 2 and 3 when â(n) is larger than $n^{\log_b a}$ but not polynomially larger. If the function â(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 $a = 9$, $b = 3$, â(n) = n, and thus $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since â($n$) = $O(n^{\log_3 9 - \epsilon})$, 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 $a = 1$, $b = 3/2$, â(n) = 1, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since â(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 $a = 3$, $b = 4$, â(n) = $n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since â($n$) = $\Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we show that the regularity condition holds for â(n). For sufficiently large n, $a\hat{a}(n/b) = 3(n/4) \lg(n/4) \le (3/4)n \lg n = c\hat{a}(n)$ 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: $a = 2$, $b = 2$, â(n) = n1g n, and $n^{\log_b a} = n$. It seems that case 3 should apply, since â(n) = n1g n is asymptotically larger than $n^{\log_b a} = n$ but not polynomially larger. The ratio â(n)/$n^{\log_b a}$ = (n 1g n)/n = 1g 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 $a\hat{a}(n/b) \le c\hat{a}(n)$ for some constant $c < 1$, which is part of case 3 of the master theorem. Give an example of a simple function $\hat{a}(n)$ that satisfies all the conditions in case 3 of the master theorem except the regularity condition.

* 4.4 Proof of the master theorem

This section contains a proof of the master theorem (Theorem 4.1) for more advanced readers. The proof need not be understood in order to apply the theorem.

The proof is in two parts. The first part analyzes the "master" recurrence (4.5), under the simplifying assumption that $T(n)$ is defined only on exact powers of $b > 1$, that is, for $n = 1, b, b^2, \ldots$. This part gives all the intuition needed to understand why the master theorem is true. The second part shows how the analysis can be extended to all positive integers n and is merely mathematical technique applied to the problem of handling floors and ceilings.

In this section, we shall sometimes abuse our asymptotic notation slightly by using it to describe the behavior of functions that are only defined over exact powers of b. Recall that the definitions of asymptotic notations require that bounds be proved for all sufficiently large numbers, not just those that are powers of b. Since we could make new asymptotic notations that apply to the set $\{b^i : i = 0,1, \ldots\}$, instead of the nonnegative integers, this abuse is minor.

Nevertheless, we must always be on guard when we are using asymptotic notation over a limited domain so that we do not draw improper conclusions. For example, proving that $T(n) = O(n)$ when n is an exact power of 2 does not guarantee that $T(n) = O(n)$. The function $T(n)$ could be defined as

$$T(n) = \begin{cases} n & \text{if } n = 1, 2, 4, 8, \ldots, \\ n^2 & \text{otherwise}, \end{cases}$$

in which case the best upper bound that can be proved is $T(n) = O(n^2)$. Because of this sort of drastic consequence, we shall never use asymptotic notation over a limited domain without making it absolutely clear from the context that we are doing so.

4.4.1 The proof for exact powers

The first part of the proof of the master theorem analyzes the master recurrence (4.5),

$$T(n) = aT(n/b) + \hat{a}(n),$$

under the assumption that n is an exact power of $b > 1$, where b need not be an integer.

The analysis is broken into three lemmas. The first reduces the problem of solving the master recurrence to the problem of evaluating an expression that contains a summation. The second determines bounds on this summation. The third lemma puts the first two together to prove a version of the master theorem for the case in which *n* is an exact power of *b*.

Lemma 4.2

Let $a \geq 1$ and $b > 1$ be constants, and let â(*n*) be a nonnegative function defined on exact powers of *b*. Define *T(n)* on exact powers of *b* by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ aT(n/b) + f(n) & \text{if } n = b^i , \end{cases}$$

where *i* is a positive integer. Then

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) . \qquad (4.6)$$

(4.6)

Proof Iterating the recurrence yields

$T(n) = \hat{a}(n) + aT(n/b)$

$= \hat{a}(n) + a\hat{a}(n/b) + a^2T(n/b^2)$

$= \hat{a}(n) + a\hat{a}(n/b) + a^2\hat{a}(n/b^2) + . . .$

$+ a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n}T(1) .$

Since $a^{\log_b n} = n^{\log_b a}$, the last term of this expression becomes

$a^{\log_b n} T(1) = \Theta(n^{\log_b a})$;

using the boundary condition $T(1) = \Theta(1)$. The remaining terms can be expressed as the sum

$$\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) ;$$

thus,

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) ,$$

which completes the proof.

The recursion tree

Before proceeding, let's try to develop some intuition by using a recursion tree. Figure 4.3 shows the tree corresponding to the iteration of the recurrence in Lemma 4.2. The root of the tree has cost â(n), and it has a children, each with cost â(n/b). (It is convenient to think of a as being an integer, especially when visualizing the recursion tree, but the mathematics does not require it.) Each of these children has a children with cost â(n/b^2), and thus there are a^2 nodes that are distance 2 from the root. In general, there are a^2 nodes that are distance j from the root, and each has cost â(n/b^j). The cost of each leaf is $T(1) = \Theta(1)$, and each leaf is distance $\log_b n$ from the root, since $n/b^{\log_b n} = 1$. There are $a^{\log_b n} = n^{\log_b a}$ leaves in the tree.

We can obtain equation (4.6) by summing the costs of each level of the tree, as shown in the figure. The cost for a level j of internal nodes is a^jâ(n/b^j), and so the total of all internal node levels is

$$\sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) .$$

In the underlying divide-and-conquer algorithm, this sum represents the costs of dividing problems into subproblems and then recombining the subproblems. The cost of all the leaves, which is the cost of doing all $n^{\log_b a}$ subproblems of size 1, is $\Theta(n^{\log_b a})$.

In terms of the recursion tree, the three cases of the master theorem correspond to cases in which the total cost of the tree is (1) dominated by the costs in the leaves, (2) evenly distributed across the levels of the tree, or (3) dominated by the cost of the root.

The summation in equation (4.6) describes the cost of the dividing and combining steps in the underlying divide-and-conquer algorithm. The next lemma provides asymptotic bounds on the summation's growth.

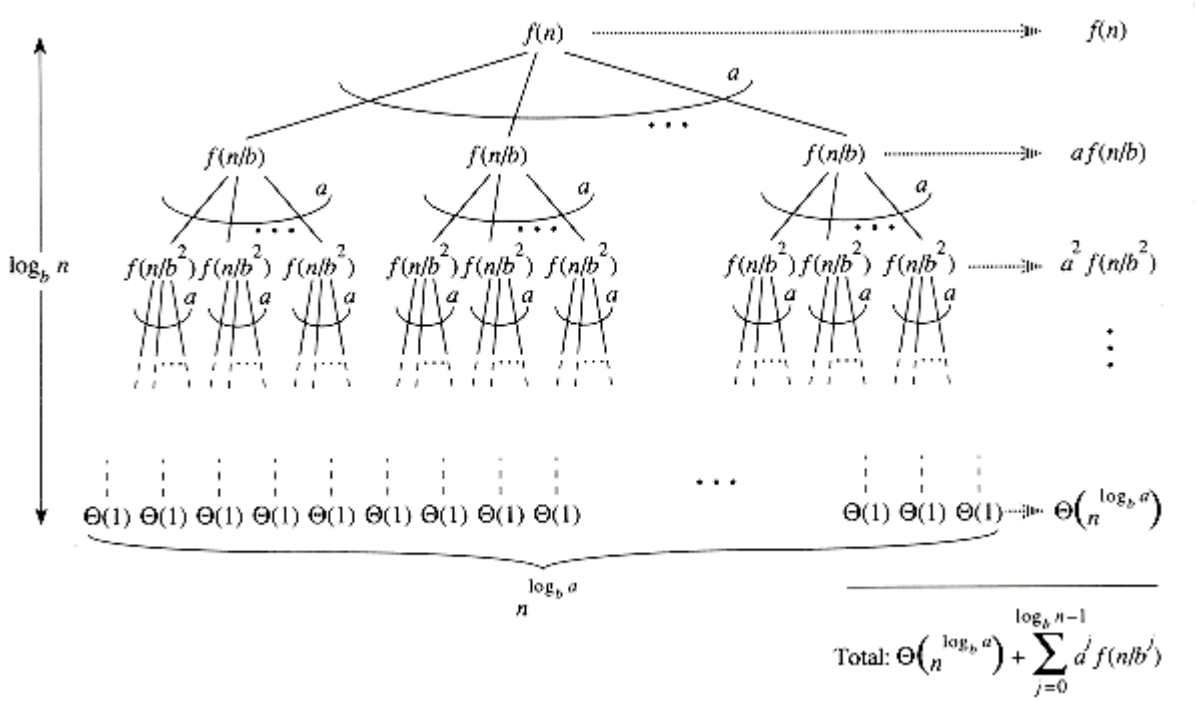

Figure 4.3 The recursion tree generated by T(n) = aT(n/b)+ f(n). The tree is a complete a-ary tree with $n^{\log_b a}$ leaves and height $\log_b a$. The cost of each level is shown at the right, and their sum is given in equation (4.6).

Lemma 4.3

Let $a \geq 1$ and $b > 1$ be constants, and let â(n) be a nonnegative function defined on exact powers of b. A function $g(n)$ defined over exact powers of b by

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \qquad (4.7)$$

(4.7)

can then be bounded asymptotically for exact powers of b as follows.

1. If â(n) = $O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $g(n) = O(n^{\log_b a})$.

2. If â(n) = $\Theta(n^{\log_b a})$, then $g(n) = \Theta(n^{\log_b a} \lg n)$.

3. If aâ(n/b) $\leq$ câ(n) for some constant $c < 1$ and all $n \geq b$, then $g(n) = \Theta$(â(n)).

Proof For case 1, we have â(n) = $O(n^{\log_b a - \epsilon})$, implying that â($n/b^j$) = $O((n/b^j)^{\log_b a - \epsilon})$. Substituting into equation (4.7) yields

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right) . \qquad (4.8)$$

(4.8)

We bound the summation within the O-notation by factoring out terms and simplifying, which leaves an increasing geometric series:

$$\begin{aligned}
\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon} &= n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} \left(\frac{a b^\epsilon}{b^{\log_b a}}\right) \\
&= n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j \\
&= n^{\log_b a - \epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right) \\
&= n^{\log_b a - \epsilon} \left(\frac{n^\epsilon - 1}{b^\epsilon - 1}\right) .
\end{aligned}$$

Since b and ϵ are constants, the last expression reduces to $n^{\log_b a - \epsilon} O(n^\epsilon) = O(n^{\log_b a})$. Substituting this expression for the summation in equation (4.8) yields

$g(n) = O(n^{\log_b a})$,

and case 1 is proved.

Under the assumption that $\hat{a}(n) = \Theta(n^{\log_b a})$ for case 2, we have that $\hat{a}(n/b^j) = \Theta((n/b^j)^{\log_b a})$. Substituting into equation (4.7) yields

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) . \qquad (4.9)$$

(4.9)

We bound the summation within the Θ as in case 1, but this time we do not obtain a geometric series. Instead, we discover that every term of the summation is the same:

$$\begin{aligned}\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^{\log_b a}}\right) \\ &= n^{\log_b a} \sum_{j=0}^{\log_b n - 1} 1 \\ &= n^{\log_b a} \log_b n .\end{aligned}$$

Substituting this expression for the summation in equation (4.9) yields

$g(n) = \Theta(n^{\log_b a} \log_b n)$

$= \Theta(n^{\log_b a} \lg n)$,

and case 2 is proved.

Case 3 is proved similarly. Since $f(n)$ appears in the definition (4.7) of $g(n)$ and all terms of $g(n)$ are nonnegative, we can conclude that $g(n) = \Omega(f(n))$ for exact powers of b. Under the assumption that $af(n/b) \le cf(n)$ for some constant $c < 1$ and all $n \ge b$, we have $a^j f(n/b^j) \le c^j f(n)$. Substituting into equation (4.7) and simplifying yields a geometric series, but unlike the series in case 1, this one has decreasing terms:

$$\begin{aligned}g(n) &\le \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \\ &\le \sum_{j=0}^{\log_b n - 1} c^j f(n) \\ &\le f(n) \sum_{j=0}^{\infty} c^j \\ &= f(n)\left(\frac{1}{1-c}\right) \\ &= O(f(n)) ,\end{aligned}$$

since c is constant. Thus, we can conclude that $g(n) = \Theta(f(n))$ for exact powers of b. Case

3 is proved, which completes the proof of the lemma.

We can now prove a version of the master theorem for the case in which n is an exact power of b.

Lemma 4.4

Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a nonnegative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ aT(n/b) + f(n) & \text{if } n = b^i, \end{cases}$$

where i is a positive integer. Then $T(n)$ can be bounded asymptotically for exact powers of b 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 $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Proof We use the bounds in Lemma 4.3 to evaluate the summation (4.6) from Lemma 4.2. For case 1, we have

$$T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a})$$

$$= \Theta(n^{\log_b a}),$$

and for case 2,

$$T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n)$$

$$= \Theta(n^{\log_b a} \lg n).$$

For case 3, the condition $a f(n/b) \leq c f(n)$ implies $f(n) = \Omega(n^{\log_b a + \epsilon})$ (see Exercise 4.4-3). Consequently,

$$T(n) = \Theta(n^{\log_b a}) + \Theta(f(n))$$

$$= \Theta(f(n)).$$

4.4.2 Floors and ceilings

To complete the proof of the master theorem, we must now extend our analysis to the situation in which floors and ceilings are used in the master recurrence, so that the recurrence is defined for all integers, not just exact powers of b. Obtaining a lower bound on

$$T(n) = aT(\lceil n/b \rceil) + f(n)$$

(4.10)

and an upper bound on

$T(n) = aT(\lfloor n/b \rfloor) + f(n)$

(4.11)

is routine, since the bound $\lceil n/b \rceil \geq n/b$ can be pushed through in the first case to yield the desired result, and the bound $\lfloor n/b \rfloor \leq n/b$ can be pushed through in the second case. Lower bounding the recurrence (4.11) requires much the same technique as upper bounding the recurrence (4.10), so we shall only present this latter bound.

We wish to iterate the recurrence (4.10), as was done in Lemma 4.2. As we iterate the recurrence, we obtain a sequence of recursive invocations on the arguments

n,

$\lceil n/b \rceil$,

$\lceil \lceil n/b \rceil / b \rceil$,

$\lceil \lceil \lceil n/b \rceil / b \rceil / b \rceil$,

$\vdots$

Let us denote the ith element in the sequence by n_i, where

$$n_i = \begin{cases} n & \text{if } i = 0 , \\ \lceil n_{i-1}/b \rceil & \text{if } i > 0 . \end{cases} \qquad (4.12)$$

(4.12)

Our first goal is to determine the number of iterations k such that n_k is a constant. Using the inequality $\lceil x \rceil \leq x + 1$, we obtain

$$\begin{aligned} n_0 &\leq n , \\ n_1 &\leq \frac{n}{b} + 1 , \\ n_2 &\leq \frac{n}{b^2} + \frac{1}{b} + 1 , \\ n_3 &\leq \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1 , \\ &\vdots \end{aligned}$$

In general,

$$\begin{aligned} n_i &\le \frac{n}{b^i} + \sum_{j=0}^{i-1} \frac{1}{b^j} \\ &\le \frac{n}{b^i} + \frac{b}{b-1} , \end{aligned}$$

and thus, when $i = \lfloor \log_b n \rfloor$, *we obtain* $n_i \le b + b/(b - 1) = O(1)$.

We can now iterate recurrence (4.10), obtaining

$$\begin{aligned} T(n) &= f(n_0) + aT(n_1) \\ &= f(n_0) + af(n_1) + a^2 T(n_2) \\ &\le f(n_0) + af(n_1) + a^2 f(n_2) + \cdots \\ &\quad + a^{\lfloor \log_b n \rfloor - 1} f(n_{\lfloor \log_b n \rfloor - 1}) + a^{\lfloor \log_b n \rfloor} T(n_{\lfloor \log_b n \rfloor}) \\ &= \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) , \end{aligned} \qquad (4.13)$$

(4.13)

which is much the same as equation (4.6), except that n is an arbitrary integer and not restricted to be an exact power of b.

We can now evaluate the summation

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \qquad (4.14)$$

(4.14)

from (4.13) in a manner analogous to the proof of Lemma 4.3. Beginning with case 3, if $af(\lceil n/b \rceil) \le cf(n)$ for $n > b + b/(b - 1)$, where $c < 1$ is a constant, then it follows that $a^j f(n_j) \le c^j f(n)$. Therefore, the sum in equation (4.14) can be evaluated just as in Lemma 4.3. For case 2, we have $f(n) = \Theta(n^{\log_b a})$. If we can show that $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$, then the proof for case 2 of Lemma 4.3 will go through. Observe that $j \le \lfloor \log_b n \rfloor$ *implies* $b^j/n \le 1$. The bound $f(n) = O(n^{\log_b a})$ implies that there exists a constant $c > 0$ such that for sufficiently large n_j,

$$\begin{aligned} f(n_j) &\le c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} \\ &= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\ &\le c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} \\ &\le O\left(\frac{n^{\log_b a}}{a^j}\right) , \end{aligned}$$

since $c(1 + b/(b - 1))^{\log_b a}$ is a constant. Thus, case 2 is proved. The proof of case 1 is almost identical. The key is to prove the bound $f(n_j) = O(n^{\log_b a - \epsilon})$, which is similar to the corresponding

proof of case 2, though the algebra is more intricate.

We have now proved the upper bounds in the master theorem for all integers n. The proof of the lower bounds is similar.

Exercises

4.4-1

Give a simple and exact expression for n_i in equation (4.12) for the case in which b is a positive integer instead of an arbitrary real number.

4.4-2

Show that if $f(n) = \Theta(n^{\log_b a} \lg^k n)$, *where* $k \geq 0$, then the master recurrence has solution $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. For simplicity, confine your analysis to exact powers of b.

4.4-3

Show that case 3 of the master theorem is overstated, in the sense that the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ implies that there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$.

Problems

4-1 Recurrence examples

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

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

b. $T(n) = T(9n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n - 1) + n$.

h. $T(n) = T(\sqrt{n}) + 1$.

4-2 Finding the missing integer

An array $A[1 . . n]$ contains all the integers from 0 to n except one. It would be easy to determine the missing integer in $O(n)$ time by using an auxiliary array $B[0 . . n]$ to record which numbers appear in A. In this problem, however, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of $A[i]$," which takes constant time.

Show that if we use only this operation, we can still determine the missing integer in $O(n)$ time.

4-3 Parameter-passing costs

Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time = $\Theta(1)$.

2. An array is passed by copying. Time = $\Theta(N)$, where N is the size of the array.

3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time = $\Theta(p - q + 1)$ if the subarray $A[p . . q]$ is passed.

a. Consider the recursive binary search algorithm for finding a number in a sorted array (see Exercise 1.3-5). Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let N be the size of the original problem and n be the size of a subproblem.

b. Redo part (a) for the MERGE-SORT algorithm from Section 1.3.1.

4-4 More recurrence examples

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 3T(n/2) + n \lg n$.

b. $T(n) = 3T(n/3 + 5) + n / 2$.

c. $T(n) = 2T(n/2) + n / \lg n$.

d. $T(n) = T(n - 1) + 1 / n$.

e. $T(n) = T(n - 1) + \lg n$.

f. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4-5 Sloppiness conditions

Often, we are able to bound a recurrence $T(n)$ at exact powers of an integral constant b. This problem gives sufficient conditions for us to extend the bound to all real $n > 0$.

a. Let $T(n)$ and $h(n)$ be monotonically increasing functions, and suppose that $T(n) < h(n)$ when n is an exact power of a constant $b > 1$. Moreover, suppose that $h(n)$ is "slowly growing" in the sense that $h(n) = O(h(n/b))$. Prove that $T(n) = O(h(n))$.

b. Suppose that we have the recurrence $T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n)$ is monotonically increasing. Suppose further that the initial conditions for the recurrence are given by $T(n) = g(n)$ for $n \leq n_0$, where $g(n)$ is monotonically increasing and $g(n_0) < aT(n_0/b) + f(n_0)$. Prove that $T(n)$ is monotonically increasing.

c. Simplify the proof of the master theorem for the case in which $f(n)$ is monotonically increasing and slowly growing. Use Lemma 4.4.

4-6 Fibonacci numbers

This problem develops properties of the Fibonacci numbers, which are defined by recurrence (2.13). We shall use the technique of generating functions to solve the Fibonacci recurrence. Define the ***generating function*** (or ***formal power series***) $\mathcal{F}$ as

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots .\end{aligned}$$

$$\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z).$$

a. Show that

b. Show that

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1 - \phi z)(1 - \hat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z}\right),\end{aligned}$$

where

$$\phi = \frac{1 + \sqrt{5}}{2} = 1.61803\ldots$$

and

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} = -0.61803\ldots .$$

c. Show that

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i .$$

d. Prove that $F_i = \phi^i/\sqrt{5}$ for $i > 0$, rounded to the nearest integer. *Hint*: $|\hat{\phi}| < 1$.)

e. Prove that $F_i + 2 \geq \Phi^i$ *for* $i \geq 0$.

4-7 VLSI chip testing

Professor Diogenes has *n* supposedly identical VLSI[1] chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the answer of a bad chip cannot be trusted. Thus, the four possible outcomes of a test are as follows:

Chip ***A*** says	Chip ***B*** says	Conclusion
B is good	*A* is good	both are good, or both are bad
B is good	*A* is bad	at least one is bad
B is bad	*A* is good	at least one is bad
B is bad	*A* is bad	at least one is bad

[1] VLSI stands for "very-large-scale integration," which is the integrated-circuit chip technology used to fabricate most microprocessors today.

a. Show that if more than *n*/2 chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

b. Consider the problem of finding a single good chip from among *n* chips, assuming that more than *n*/2 of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

c. Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than *n*/2 of the chips are good. Give and solve the recurrence that describes the number of tests.

Chapter notes

Recurrences were studied as early as 1202 by L. Fibonacci, for whom the Fibonacci numbers are named. A. De Moivre introduced the method of generating functions (see Problem 4-6) for solving recurrences. The master method is adapted from Bentley, Haken, and Saxe [26], which provides the extended method justified by Exercise 4.4-2. Knuth [121] and Liu [140] show how to solve linear recurrences using the method of generating functions. Purdom and Brown [164] contains an extended discussion of recurrence

solving.

Go to Chapter 5 Back to Table of Contents