This more general class of algorithms is obtained by considering divide-and-conquer algorithms that create recursive calls on q subproblems of size $n/2$ each and then combine the results in $O(n)$ time. This corresponds to the Mergesort recurrence (5.1) when $q = 2$ recursive calls are used, but other algorithms find it useful to spawn $q > 2$ recursive calls, or just a single ($q = 1$) recursive call. In fact, we will see the case $q > 2$ later in this chapter when we design algorithms for integer multiplication; and we will see a variant on the case $q = 1$ much later in the book, when we design a randomized algorithm for median finding in Chapter 13.

If $T(n)$ denotes the running time of an algorithm designed in this style, then $T(n)$ obeys the following recurrence relation, which directly generalizes (5.1) by replacing 2 with q:

(5.3) *For some constant c,*

$$T(n) \le qT(n/2) + cn$$

when $n > 2$, and

$$T(2) \le c.$$

$q = 1$ $O(n)$
$= 2$ $O(n \log n)$
> 2 $O(n^{\log_2 q})$

We now describe how to solve (5.3) by the methods we've seen above: unrolling, substitution, and partial substitution. We treat the cases $q > 2$ and $q = 1$ separately, since they are qualitatively different from each other—and different from the case $q = 2$ as well.

The Case of $q > 2$ Subproblems

We begin by unrolling (5.3) in the case $q > 2$, following the style we used earlier for (5.1). We will see that the punch line ends up being quite different.

- *Analyzing the first few levels:* We show an example of this for the case $q = 3$ in Figure 5.2. At the first level of recursion, we have a single problem of size n, which takes time at most cn plus the time spent in all subsequent recursive calls. At the next level, we have q problems, each of size $n/2$. Each of these takes time at most $cn/2$, for a total of at most $(q/2)cn$, again plus the time in subsequent recursive calls. The next level yields q^2 problems of size $n/4$ each, for a total time of $(q^2/4)cn$. Since $q > 2$, we see that the total work per level is *increasing* as we proceed through the recursion.
- *Identifying a pattern:* At an arbitrary level j, we have q^j distinct instances, each of size $n/2^j$. Thus the total work performed at level j is $q^j(cn/2^j) = (q/2)^j cn$.

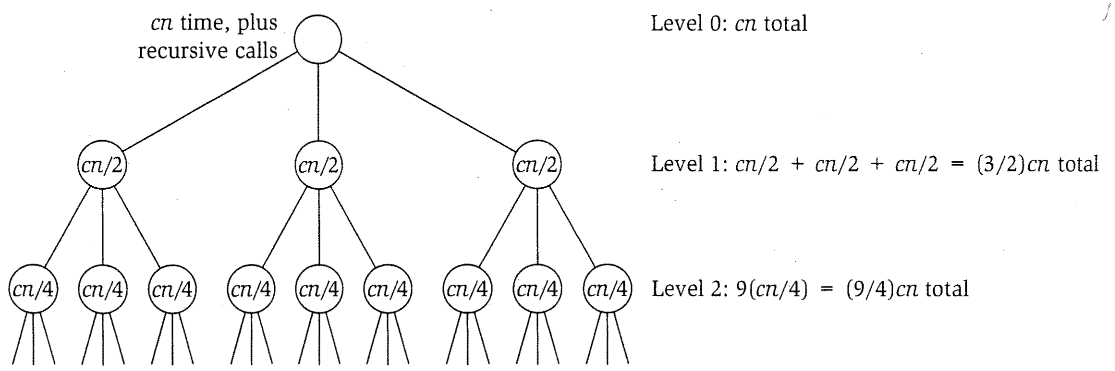
cn time, plus recursive calls
cn/2
cn/4
Level 0: cn total
Level 1: cn/2 + cn/2 + cn/2 = (3/2)cn total
Level 2: 9(cn/4) = (9/4)cn total

Figure 5.2 Unrolling the recurrence $T(n) \leq 3T(n/2) + O(n)$.

- *Summing over all levels of recursion:* As before, there are $\log_2 n$ levels of recursion, and the total amount of work performed is the sum over all these:

$$T(n) \leq \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j cn = cn \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j.$$

This is a geometric sum, consisting of powers of $r = q/2$. We can use the formula for a geometric sum when $r > 1$, which gives us the formula

$$T(n) \leq cn \left(\frac{r^{\log_2 n} - 1}{r - 1}\right) \leq cn \left(\frac{r^{\log_2 n}}{r - 1}\right).$$

Since we're aiming for an asymptotic upper bound, it is useful to figure out what's simply a constant; we can pull out the factor of $r - 1$ from the denominator, and write the last expression as

$$T(n) \leq \left(\frac{c}{r-1}\right) n r^{\log_2 n}.$$

Finally, we need to figure out what $r^{\log_2 n}$ is. Here we use a very handy identity, which says that, for any $a > 1$ and $b > 1$, we have $a^{\log b} = b^{\log a}$. Thus

$$r^{\log_2 n} = n^{\log_2 r} = n^{\log_2 (q/2)} = n^{(\log_2 q) - 1}.$$

Thus we have

$$T(n) \leq \left(\frac{c}{r-1}\right) n \cdot n^{(\log_2 q)-1} \leq \left(\frac{c}{r-1}\right) n^{\log_2 q} = O(n^{\log_2 q}).$$

We sum this up as follows.

(5.4) *Any function $T(\cdot)$ satisfying (5.3) with $q > 2$ is bounded by $O(n^{\log_2 q})$.*

So we find that the running time is more than linear, since $\log_2 q > 1$, but still polynomial in n. Plugging in specific values of q, the running time is $O(n^{\log_2 3}) = O(n^{1.59})$ when $q = 3$; and the running time is $O(n^{\log_2 4}) = O(n^2)$ when $q = 4$. This increase in running time as q increases makes sense, of course, since the recursive calls generate more work for larger values of q.

Applying Partial Substitution The appearance of $\log_2 q$ in the exponent followed naturally from our solution to (5.3), but it's not necessarily an expression one would have guessed at the outset. We now consider how an approach based on partial substitution into the recurrence yields a different way of discovering this exponent.

Suppose we guess that the solution to (5.3), when $q > 2$, has the form $T(n) \le kn^d$ for some constants $k > 0$ and $d > 1$. This is quite a general guess, since we haven't even tried specifying the exponent d of the polynomial. Now let's try starting the inductive argument and seeing what constraints we need on k and d. We have

$$T(n) \le qT(n/2) + cn,$$

and applying the inductive hypothesis to $T(n/2)$, this expands to

$$\begin{aligned} T(n) &\le qk\left(\frac{n}{2}\right)^d + cn \\ &= \frac{q}{2^d}kn^d + cn. \end{aligned}$$

This is remarkably close to something that works: if we choose d so that $q/2^d = 1$, then we have $T(n) \le kn^d + cn$, which is almost right except for the extra term cn. So let's deal with these two issues: first, how to choose d so we get $q/2^d = 1$; and second, how to get rid of the cn term.

Choosing d is easy: we want $2^d = q$, and so $d = \log_2 q$. Thus we see that the exponent $\log_2 q$ appears very naturally once we decide to discover which value of d works when substituted into the recurrence.

But we still have to get rid of the cn term. To do this, we change the form of our guess for $T(n)$ so as to explicitly subtract it off. Suppose we try the form $T(n) \le kn^d - \ell n$, where we've now decided that $d = \log_2 q$ but we haven't fixed the constants k or ℓ. Applying the new formula to $T(n/2)$, this expands to

$$\begin{aligned}T(n) &\le qk\left(\frac{n}{2}\right)^d - q\ell\left(\frac{n}{2}\right) + cn\\ &= \frac{q}{2^d}kn^d - \frac{q\ell}{2}n + cn\\ &= kn^d - \frac{q\ell}{2}n + cn\\ &= kn^d - (\frac{q\ell}{2} - c)n.\end{aligned}$$

This now works completely, if we simply choose ℓ so that $(\frac{q\ell}{2} - c) = \ell$: in other words, $\ell = 2c/(q-2)$. This completes the inductive step for n. We also need to handle the base case $n = 2$, and this we do using the fact that the value of k has not yet been fixed: we choose k large enough so that the formula is a valid upper bound for the case $n = 2$.

The Case of One Subproblem

We now consider the case of $q = 1$ in (5.3), since this illustrates an outcome of yet another flavor. While we won't see a direct application of the recurrence for $q = 1$ in this chapter, a variation on it comes up in Chapter 13, as we mentioned earlier.

We begin by unrolling the recurrence to try constructing a solution.

- *Analyzing the first few levels:* We show the first few levels of the recursion in Figure 5.3. At the first level of recursion, we have a single problem of size n, which takes time at most cn plus the time spent in all subsequent recursive calls. The next level has one problem of size $n/2$, which contributes $cn/2$, and the level after that has one problem of size $n/4$, which contributes $cn/4$. So we see that, unlike the previous case, the total work per level when $q = 1$ is actually *decreasing* as we proceed through the recursion.
- *Identifying a pattern:* At an arbitrary level j, we still have just one instance; it has size $n/2^j$ and contributes $cn/2^j$ to the running time.
- *Summing over all levels of recursion:* There are $\log_2 n$ levels of recursion, and the total amount of work performed is the sum over all these:

$$T(n) \le \sum_{j=0}^{\log_2 n - 1} \frac{cn}{2^j} = cn \sum_{j=0}^{\log_2 n - 1} \left(\frac{1}{2^j}\right).$$

This geometric sum is very easy to work out; even if we continued it to infinity, it would converge to 2. Thus we have

$$T(n) \le 2cn = O(n).$$

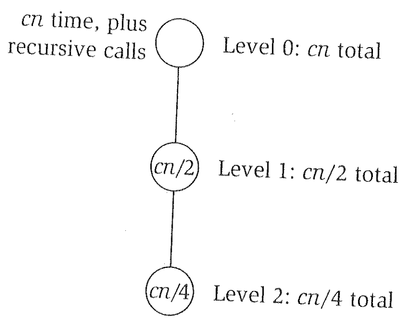

Figure 5.3 Unrolling the recurrence $T(n) \leq T(n/2) + O(n)$.

We sum this up as follows.

(5.5) *Any function $T(\cdot)$ satisfying (5.3) with $q = 1$ is bounded by $O(n)$.*

This is counterintuitive when you first see it. The algorithm is performing log n levels of recursion, but the overall running time is still linear in n. The point is that a geometric series with a decaying exponent is a powerful thing: fully half the work performed by the algorithm is being done at the top level of the recursion.

It is also useful to see how partial substitution into the recurrence works very well in this case. Suppose we guess, as before, that the form of the solution is $T(n) \leq kn^d$. We now try to establish this by induction using (5.3), assuming that the solution holds for the smaller value $n/2$:

$$\begin{aligned} T(n) &\leq T(n/2) + cn \\ &\leq k\left(\frac{n}{2}\right)^d + cn \\ &= \frac{k}{2^d}n^d + cn. \end{aligned}$$

If we now simply choose $d = 1$ and $k = 2c$, we have

$$T(n) \leq \frac{k}{2}n + cn = (\frac{k}{2} + c)n = kn,$$

which completes the induction.

The Effect of the Parameter q. It is worth reflecting briefly on the role of the parameter q in the class of recurrences $T(n) \leq qT(n/2) + O(n)$ defined by (5.3). When $q = 1$, the resulting running time is linear; when $q = 2$, it's $O(n \log n)$; and when $q > 2$, it's a polynomial bound with an exponent larger than 1 that grows with q. The reason for this range of different running times lies in where

$q=1$	$O(n)$
$=2$	$O(n \log n)$
>2	$O(n^{\log_2 q})$

most of the work is spent in the recursion: when $q = 1$, the total running time is dominated by the top level, whereas when $q > 2$ it's dominated by the work done on constant-size subproblems at the bottom of the recursion. Viewed this way, we can appreciate that the recurrence for $q = 2$ really represents a "knife-edge"—the amount of work done at each level is *exactly the same,* which is what yields the $O(n \log n)$ running time.

A Related Recurrence: $T(n) \leq 2T(n/2) + O(n^2)$

We conclude our discussion with one final recurrence relation; it is illustrative both as another application of a decaying geometric sum and as an interesting contrast with the recurrence (5.1) that characterized Mergesort. Moreover, we will see a close variant of it in Chapter 6, when we analyze a divide-and-conquer algorithm for solving the Sequence Alignment Problem using a small amount of working memory.

The recurrence is based on the following divide-and-conquer structure.

Divide the input into two pieces of equal size; solve the two subproblems on these pieces separately by recursion; and then combine the two results into an overall solution, spending quadratic time for the initial division and final recombining.

For our purposes here, we note that this style of algorithm has a running time $T(n)$ that satisfies the following recurrence.

(5.6) *For some constant c,*

$$T(n) \leq 2T(n/2) + cn^2$$

when $n > 2$, and

$$T(2) \leq c.$$

One's first reaction is to guess that the solution will be $T(n) = O(n^2 \log n)$, since it looks almost identical to (5.1) except that the amount of work per level is larger by a factor equal to the input size. In fact, this upper bound is correct (it would need a more careful argument than what's in the previous sentence), but it will turn out that we can also show a stronger upper bound.

We'll do this by unrolling the recurrence, following the standard template for doing this.

- *Analyzing the first few levels:* At the first level of recursion, we have a single problem of size n, which takes time at most cn^2 plus the time spent in all subsequent recursive calls. At the next level, we have two problems, each of size $n/2$. Each of these takes time at most $c(n/2)^2 = cn^2/4$, for a total of at most $cn^2/2$, again plus the time in subsequent recursive calls. At the third level, we have four problems each of size $n/4$, each taking time at most $c(n/4)^2 = cn^2/16$, for a total of at most $cn^2/4$. Already we see that something is different from our solution to the analogous recurrence (5.1); whereas the total amount of work per level remained the same in that case, here it's decreasing.

- *Identifying a pattern:* At an arbitrary level j of the recursion, there are 2^j subproblems, each of size $n/2^j$, and hence the total work at this level is bounded by $2^j c(\frac{n}{2^j})^2 = cn^2/2^j$.
- *Summing over all levels of recursion:* Having gotten this far in the calculation, we've arrived at almost exactly the same sum that we had for the case $q = 1$ in the previous recurrence. We have

$$T(n) \le \sum_{j=0}^{\log_2 n-1} \frac{cn^2}{2^j} = cn^2 \sum_{j=0}^{\log_2 n-1} \left(\frac{1}{2^j}\right) \le 2cn^2 = O(n^2),$$

where the second inequality follows from the fact that we have a convergent geometric sum.

In retrospect, our initial guess of $T(n) = O(n^2 \log n)$, based on the analogy to (5.1), was an overestimate because of how quickly n^2 decreases as we replace it with $(\frac{n}{2})^2$, $(\frac{n}{4})^2$, $(\frac{n}{8})^2$, and so forth in the unrolling of the recurrence. This means that we get a geometric sum, rather than one that grows by a fixed amount over all n levels (as in the solution to (5.1)).

5.3 Counting Inversions

We've spent some time discussing approaches to solving a number of common recurrences. The remainder of the chapter will illustrate the application of divide-and-conquer to problems from a number of different domains; we will use what we've seen in the previous sections to bound the running times of these algorithms. We begin by showing how a variant of the Mergesort technique can be used to solve a problem that is not directly related to sorting numbers.

The Problem

We will consider a problem that arises in the analysis of *rankings*, which are becoming important to a number of current applications. For example, a number of sites on the Web make use of a technique known as *collaborative filtering*, in which they try to match your preferences (for books, movies, restaurants) with those of other people out on the Internet. Once the Web site has identified people with "similar" tastes to yours—based on a comparison