Chapter 2 that the natural brute-force algorithm for finding the closest pair among n points in the plane would simply measure all $\Theta(n^2)$ distances, for a (polynomial) running time of $\Theta(n^2)$. Using divide and conquer, we will improve the running time to $O(n \log n)$. At a high level, then, the overall theme of this chapter is the same as what we've been seeing earlier: that improving on brute-force search is a fundamental conceptual hurdle in solving a problem efficiently, and the design of sophisticated algorithms can achieve this. The difference is simply that the distinction between brute-force search and an improved solution here will not always be the distinction between exponential and polynomial.

5.1 A First Recurrence: The Mergesort Algorithm

To motivate the general approach to analyzing divide-and-conquer algorithms, we begin with the *Mergesort* Algorithm. We discussed the Mergesort Algorithm briefly in Chapter 2, when we surveyed common running times for algorithms. Mergesort sorts a given list of numbers by first dividing them into two equal halves, sorting each half separately by recursion, and then combining the results of these recursive calls—in the form of the two sorted halves—using the linear-time algorithm for merging sorted lists that we saw in Chapter 2.

To analyze the running time of Mergesort, we will abstract its behavior into the following template, which describes many common divide-and-conquer algorithms.

> *(†) 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 only linear time for the initial division and final recombining.*

In Mergesort, as in any algorithm that fits this style, we also need a base case for the recursion, typically having it "bottom out" on inputs of some constant size. In the case of Mergesort, we will assume that once the input has been reduced to size 2, we stop the recursion and sort the two elements by simply comparing them to each other.

Consider any algorithm that fits the pattern in (†), and let $T(n)$ denote its worst-case running time on input instances of size n. Supposing that n is even, the algorithm spends $O(n)$ time to divide the input into two pieces of size $n/2$ each; it then spends time $T(n/2)$ to solve each one (since $T(n/2)$ is the worst-case running time for an input of size $n/2$); and finally it spends $O(n)$ time to combine the solutions from the two recursive calls. Thus the running time $T(n)$ satisfies the following *recurrence relation*.

(5.1) *For some constant c,*

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

when $n > 2$, and

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

The structure of (5.1) is typical of what recurrences will look like: there's an inequality or equation that bounds $T(n)$ in terms of an expression involving $T(k)$ for smaller values k; and there is a base case that generally says that $T(n)$ is equal to a constant when n is a constant. Note that one can also write (5.1) more informally as $T(n) \leq 2T(n/2) + O(n)$, suppressing the constant c. However, it is generally useful to make c explicit when analyzing the recurrence.

To keep the exposition simpler, we will generally assume that parameters like n are even when needed. This is somewhat imprecise usage; without this assumption, the two recursive calls would be on problems of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, and the recurrence relation would say that

$$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn$$

for $n \geq 2$. Nevertheless, for all the recurrences we consider here (and for most that arise in practice), the asymptotic bounds are not affected by the decision to ignore all the floors and ceilings, and it makes the symbolic manipulation much cleaner.

Now (5.1) does not explicitly provide an asymptotic bound on the growth rate of the function T; rather, it specifies $T(n)$ implicitly in terms of its values on smaller inputs. To obtain an explicit bound, we need to solve the recurrence relation so that T appears only on the left-hand side of the inequality, not the right-hand side as well.

Recurrence solving is a task that has been incorporated into a number of standard computer algebra systems, and the solution to many standard recurrences can now be found by automated means. It is still useful, however, to understand the process of solving recurrences and to recognize which recurrences lead to good running times, since the design of an efficient divide-and-conquer algorithm is heavily intertwined with an understanding of how a recurrence relation determines a running time.

Approaches to Solving Recurrences

There are two basic ways one can go about solving a recurrence, each of which we describe in more detail below.

- The most intuitively natural way to search for a solution to a recurrence is to "unroll" the recursion, accounting for the running time across the first few levels, and identify a pattern that can be continued as the recursion expands. One then sums the running times over all levels of the recursion (i.e., until it "bottoms out" on subproblems of constant size) and thereby arrives at a total running time.
- A second way is to start with a guess for the solution, substitute it into the recurrence relation, and check that it works. Formally, one justifies this plugging-in using an argument by induction on n. There is a useful variant of this method in which one has a general form for the solution, but does not have exact values for all the parameters. By leaving these parameters unspecified in the substitution, one can often work them out as needed.

We now discuss each of these approaches, using the recurrence in (5.1) as an example.

Unrolling the Mergesort Recurrence

Let's start with the first approach to solving the recurrence in (5.1). The basic argument is depicted in Figure 5.1.

- *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 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 $cn/2$, for a total of at most cn, 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 $cn/4$, for a total of at most cn.

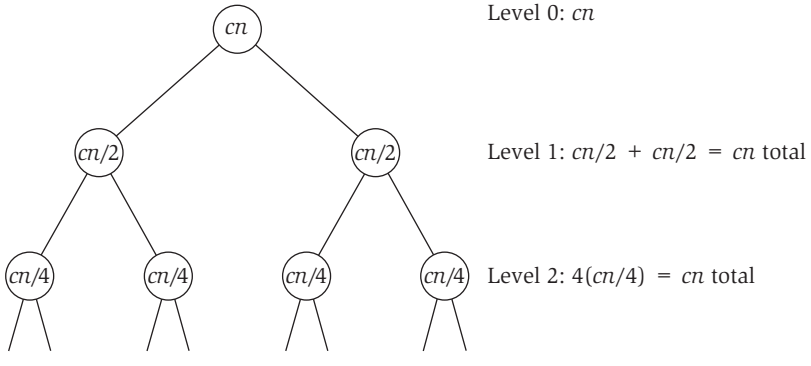

Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

- *Identifying a pattern:* What's going on in general? At level j of the recursion, the number of subproblems has doubled j times, so there are now a total of 2^j. Each has correspondingly shrunk in size by a factor of two j times, and so each has size $n/2^j$, and hence each takes time at most $cn/2^j$. Thus level j contributes a total of at most $2^j(cn/2^j) = cn$ to the total running time.
- *Summing over all levels of recursion:* We've found that the recurrence in (5.1) has the property that the same upper bound of cn applies to total amount of work performed at each level. The number of times the input must be halved in order to reduce its size from n to 2 is $\log_2 n$. So summing the cn work over $\log n$ levels of recursion, we get a total running time of $O(n \log n)$.

We summarize this in the following claim.

(5.2) *Any function $T(\cdot)$ satisfying (5.1) is bounded by $O(n \log n)$, when $n > 1$.*

Substituting a Solution into the Mergesort Recurrence

The argument establishing (5.2) can be used to determine that the function $T(n)$ is bounded by $O(n \log n)$. If, on the other hand, we have a guess for the running time that we want to verify, we can do so by plugging it into the recurrence as follows.

Suppose we believe that $T(n) \leq cn \log_2 n$ for all $n \geq 2$, and we want to check whether this is indeed true. This clearly holds for $n = 2$, since in this case $cn \log_2 n = 2c$, and (5.1) explicitly tells us that $T(2) \leq c$. Now suppose, by induction, that $T(m) \leq cm \log_2 m$ for all values of m less than n, and we want to establish this for $T(n)$. We do this by writing the recurrence for $T(n)$ and plugging in the inequality $T(n/2) \leq c(n/2) \log_2(n/2)$. We then simplify the resulting expression by noticing that $\log_2(n/2) = (\log_2 n) - 1$. Here is the full calculation.

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &\leq 2c(n/2) \log_2(n/2) + cn \\ &= cn[(\log_2 n) - 1] + cn \\ &= (cn \log_2 n) - cn + cn \\ &= cn \log_2 n. \end{aligned}$$

This establishes the bound we want for $T(n)$, assuming it holds for smaller values $m < n$, and thus it completes the induction argument.

An Approach Using Partial Substitution

There is a somewhat weaker kind of substitution one can do, in which one guesses the overall form of the solution without pinning down the exact values of all the constants and other parameters at the outset.

Specifically, suppose we believe that $T(n) = O(n \log n)$, but we're not sure of the constant inside the $O(\cdot)$ notation. We can use the substitution method even without being sure of this constant, as follows. We first write $T(n) \leq kn \log_b n$ for some constant k and base b that we'll determine later. (Actually, the base and the constant we'll end up needing are related to each other, since we saw in Chapter 2 that one can change the base of the logarithm by simply changing the multiplicative constant in front.)

Now we'd like to know whether there is any choice of k and b that will work in an inductive argument. So we try out one level of the induction as follows.

$$T(n) \leq 2T(n/2) + cn \leq 2k(n/2) \log_b(n/2) + cn.$$

It's now very tempting to choose the base $b = 2$ for the logarithm, since we see that this will let us apply the simplification $\log_2(n/2) = (\log_2 n) - 1$. Proceeding with this choice, we have

$$\begin{aligned} T(n) &\leq 2k(n/2) \log_2(n/2) + cn \\ &= 2k(n/2)[(\log_2 n) - 1] + cn \\ &= kn[(\log_2 n) - 1] + cn \\ &= (kn \log_2 n) - kn + cn. \end{aligned}$$

Finally, we ask: Is there a choice of k that will cause this last expression to be bounded by $kn \log_2 n$? The answer is clearly yes; we just need to choose any k that is at least as large as c, and we get

$$T(n) \leq (kn \log_2 n) - kn + cn \leq kn \log_2 n,$$

which completes the induction.

Thus the substitution method can actually be useful in working out the exact constants when one has some guess of the general form of the solution.

5.2 Further Recurrence Relations

We've just worked out the solution to a recurrence relation, (5.1), that will come up in the design of several divide-and-conquer algorithms later in this chapter. As a way to explore this issue further, we now consider a class of recurrence relations that generalizes (5.1), and show how to solve the recurrences in this class. Other members of this class will arise in the design of algorithms both in this and in later chapters.

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,*

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

when $n > 2$, and

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

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 $q^j(cn/2^j) = (q/2)^j cn$.

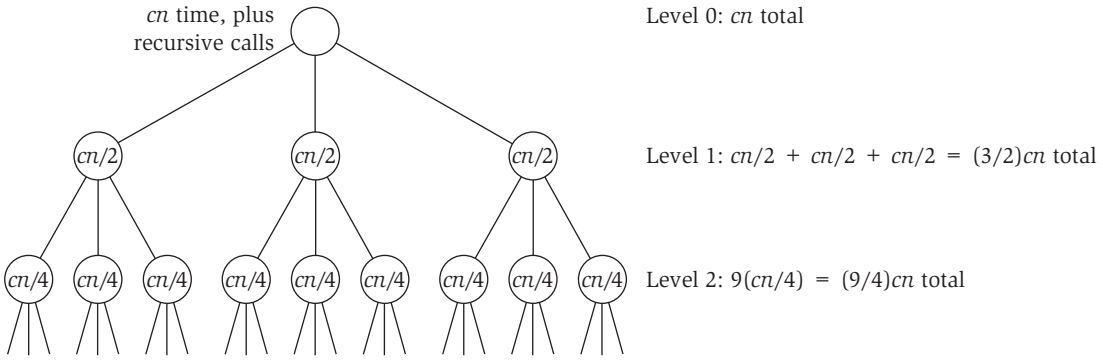

Figure 5.2 Unrolling the recurrence $T(n) \leq 3T(n/2) + O(n)$.

- *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 $T(n) \leq kn^d$ 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

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

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

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

This is remarkably close to something that works: if we choose d so that $q/2^d = 1$, then we have $T(n) \leq kn^d + cn$, 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 $T(n) \leq kn^d - \ell n$, 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

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

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:

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

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

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

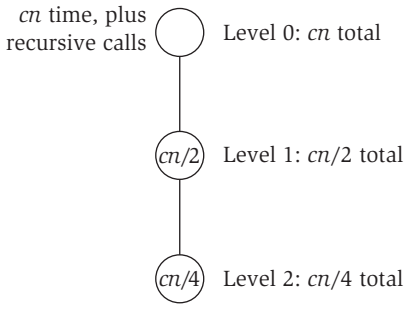

Figure 5.3 Unrolling the recurrence $T(n) \leq T(n/2) + O(n)$.

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

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

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

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

of how you and they rate various things—it can recommend new things that these other people have liked. Another application arises in *meta-search tools* on the Web, which execute the same query on many different search engines and then try to synthesize the results by looking for similarities and differences among the various rankings that the search engines return.

A core issue in applications like this is the problem of comparing two rankings. You rank a set of n movies, and then a collaborative filtering system consults its database to look for other people who had "similar" rankings. But what's a good way to measure, numerically, how similar two people's rankings are? Clearly an identical ranking is very similar, and a completely reversed ranking is very different; we want something that interpolates through the middle region.

Let's consider comparing your ranking and a stranger's ranking of the same set of n movies. A natural method would be to label the movies from 1 to n according to your ranking, then order these labels according to the stranger's ranking, and see how many pairs are "out of order." More concretely, we will consider the following problem. We are given a sequence of n numbers $a_1, \ldots, a_n$; we will assume that all the numbers are distinct. We want to define a measure that tells us how far this list is from being in ascending order; the value of the measure should be 0 if $a_1 < a_2 < \ldots < a_n$, and should increase as the numbers become more scrambled.

A natural way to quantify this notion is by counting the number of *inversions*. We say that two indices $i < j$ form an inversion if $a_i > a_j$, that is, if the two elements a_i and a_j are "out of order." We will seek to determine the number of inversions in the sequence $a_1, \ldots, a_n$.

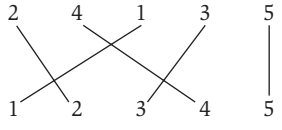

Figure 5.4 Counting the number of inversions in the sequence $2, 4, 1, 3, 5$. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the input list and the ascending list—in other words, an inversion.

Just to pin down this definition, consider an example in which the sequence is $2, 4, 1, 3, 5$. There are three inversions in this sequence: $(2, 1)$, $(4, 1)$, and $(4, 3)$. There is also an appealing geometric way to visualize the inversions, pictured in Figure 5.4: we draw the sequence of input numbers in the order they're provided, and below that in ascending order. We then draw a line segment between each number in the top list and its copy in the lower list. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the two lists—in other words, an inversion.

Note how the number of inversions is a measure that smoothly interpolates between complete agreement (when the sequence is in ascending order, then there are no inversions) and complete disagreement (if the sequence is in descending order, then every pair forms an inversion, and so there are $\binom{n}{2}$ of them).

Designing and Analyzing the Algorithm

What is the simplest algorithm to count inversions? Clearly, we could look at every pair of numbers (a_i, a_j) and determine whether they constitute an inversion; this would take $O(n^2)$ time.

We now show how to count the number of inversions much more quickly, in $O(n \log n)$ time. Note that since there can be a quadratic number of inversions, such an algorithm must be able to compute the total number without ever looking at each inversion individually. The basic idea is to follow the strategy (†) defined in Section 5.1. We set $m = \lceil n/2 \rceil$ and divide the list into the two pieces $a_1, \ldots, a_m$ and $a_{m+1}, \ldots, a_n$. We first count the number of inversions in each of these two halves separately. Then we count the number of inversions (a_i, a_j), where the two numbers belong to different halves; the trick is that we must do this part in $O(n)$ time, if we want to apply (5.2). Note that these first-half/second-half inversions have a particularly nice form: they are precisely the pairs (a_i, a_j), where a_i is in the first half, a_j is in the second half, and $a_i > a_j$.

To help with counting the number of inversions between the two halves, we will make the algorithm recursively sort the numbers in the two halves as well. Having the recursive step do a bit more work (sorting as well as counting inversions) will make the "combining" portion of the algorithm easier.

So the crucial routine in this process is `Merge-and-Count`. Suppose we have recursively sorted the first and second halves of the list and counted the inversions in each. We now have two sorted lists A and B, containing the first and second halves, respectively. We want to produce a single sorted list C from their union, while also counting the number of pairs (a, b) with $a \in A$, $b \in B$, and $a > b$. By our previous discussion, this is precisely what we will need for the "combining" step that computes the number of first-half/second-half inversions.

This is closely related to the simpler problem we discussed in Chapter 2, which formed the corresponding "combining" step for Mergesort: there we had two sorted lists A and B, and we wanted to merge them into a single sorted list in $O(n)$ time. The difference here is that we want to do something extra: not only should we produce a single sorted list from A and B, but we should also count the number of "inverted pairs" (a, b) where $a \in A$, $b \in B$, and $a > b$.

It turns out that we will be able to do this in very much the same style that we used for merging. Our `Merge-and-Count` routine will walk through the sorted lists A and B, removing elements from the front and appending them to the sorted list C. In a given step, we have a *Current* pointer into each list, showing our current position. Suppose that these pointers are currently

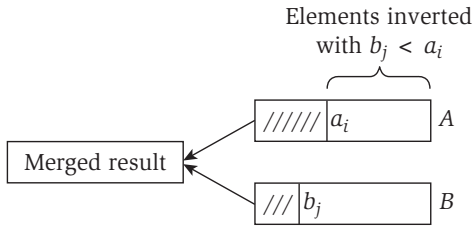

Figure 5.5 Merging two sorted lists while also counting the number of inversions between them.

at elements a_i and b_j. In one step, we compare the elements a_i and b_j being pointed to in each list, remove the smaller one from its list, and append it to the end of list C.

This takes care of merging. How do we also count the number of inversions? Because A and B are sorted, it is actually very easy to keep track of the number of inversions we encounter. Every time the element a_i is appended to C, no new inversions are encountered, since a_i is smaller than everything left in list B, and it comes before all of them. On the other hand, if b_j is appended to list C, then it is smaller than all the remaining items in A, and it comes after all of them, so we increase our count of the number of inversions by the number of elements remaining in A. This is the crucial idea: in constant time, we have accounted for a potentially large number of inversions. See Figure 5.5 for an illustration of this process.

To summarize, we have the following algorithm.

```
Merge-and-Count(A,B)
  Maintain a Current pointer into each list, initialized to
    point to the front elements
  Maintain a variable Count for the number of inversions,
    initialized to 0
  While both lists are nonempty:
    Let a_i and b_j be the elements pointed to by the Current pointer
    Append the smaller of these two to the output list
    If b_j is the smaller element then
      Increment Count by the number of elements remaining in A
    Endif
    Advance the Current pointer in the list from which the
      smaller element was selected.
  EndWhile
```

```
Once one list is empty, append the remainder of the other list
    to the output
Return Count and the merged list
```

The running time of Merge-and-Count can be bounded by the analogue of the argument we used for the original merging algorithm at the heart of Mergesort: each iteration of the While loop takes constant time, and in each iteration we add some element to the output that will never be seen again. Thus the number of iterations can be at most the sum of the initial lengths of A and B, and so the total running time is $O(n)$.

We use this Merge-and-Count routine in a recursive procedure that simultaneously sorts and counts the number of inversions in a list L.

```
Sort-and-Count(L)
  If the list has one element then
      there are no inversions
  Else
      Divide the list into two halves:
         A contains the first ⌈n/2⌉ elements
         B contains the remaining ⌊n/2⌋ elements
      (r_A, A) = Sort-and-Count(A)
      (r_B, B) = Sort-and-Count(B)
      (r, L) = Merge-and-Count(A, B)
   Endif
   Return r = r_A + r_B + r, and the sorted list L
```

Since our Merge-and-Count procedure takes $O(n)$ time, the running time $T(n)$ of the full Sort-and-Count procedure satisfies the recurrence (5.1). By (5.2), we have

(5.7) *The* Sort-and-Count *algorithm correctly sorts the input list and counts the number of inversions; it runs in $O(n \log n)$ time for a list with n elements.*

5.4 Finding the Closest Pair of Points

We now describe another problem that can be solved by an algorithm in the style we've been discussing; but finding the right way to "merge" the solutions to the two subproblems it generates requires quite a bit of ingenuity.