Chapter 5

Divide and Conquer

Divide and conquer refers to a class of algorithmic techniques in which one breaks the input into several parts, solves the problem in each part recursively, and then combines the solutions to these subproblems into an overall solution. In many cases, it can be a simple and powerful method.

Analyzing the running time of a divide and conquer algorithm generally involves solving a *recurrence relation* that bounds the running time recursively in terms of the running time on smaller instances. We begin the chapter with a general discussion of recurrence relations, illustrating how they arise in the analysis and describing methods for working out upper bounds from them.

We then illustrate the use of divide and conquer with applications to a number of different domains: computing a distance function on different rankings of a set of objects; finding the closest pair of points in the plane; multiplying two integers; and smoothing a noisy signal. Divide and conquer will also come up in subsequent chapters, since it is a method that often works well when combined with other algorithm design techniques. For example, in Chapter 6 we will see it combined with dynamic programming to produce a space-efficient solution to a fundamental sequence comparison problem, and in Chapter 13 we will see it combined with randomization to yield a simple and efficient algorithm for computing the median of a set of numbers.

One thing to note about many settings in which divide and conquer is applied, including these, is that the natural brute-force algorithm may already be polynomial time, and the divide and conquer strategy is serving to reduce the running time to a lower polynomial. This is in contrast to most of the problems in the previous chapters, for example, where brute force was exponential and the goal in designing a more sophisticated algorithm was to achieve *any* kind of polynomial running time. For example, we discussed in

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.

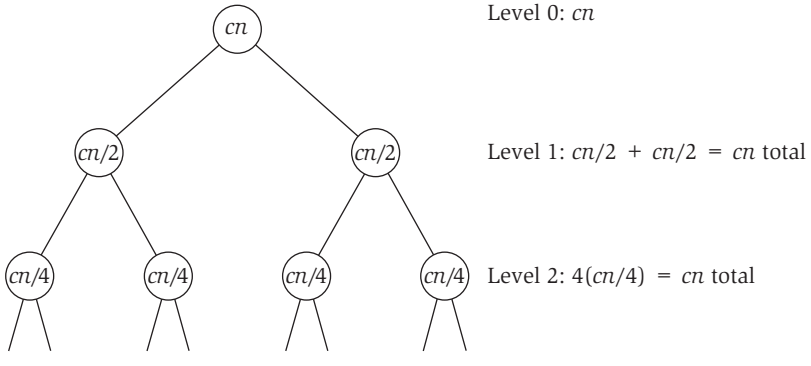

Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

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

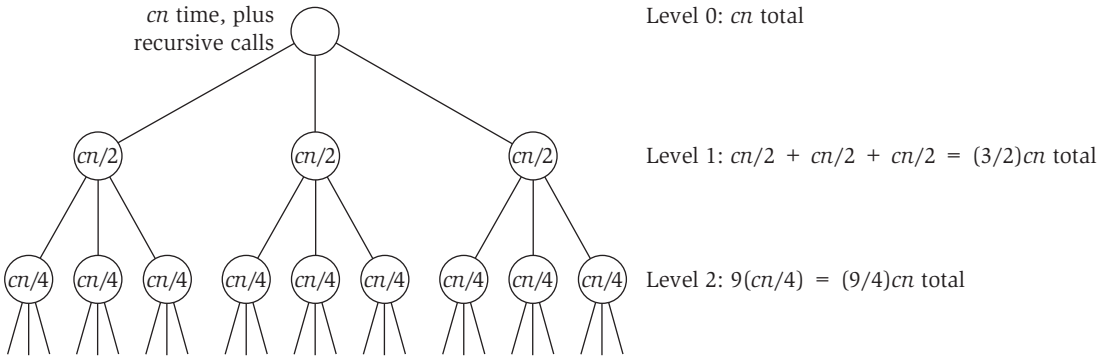

Figure 5.2 Unrolling the recurrence $T(n) \leq 3T(n/2) + O(n)$.

- *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

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

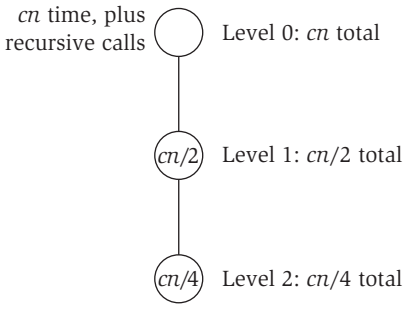

Figure 5.3 Unrolling the recurrence $T(n) \leq T(n/2) + O(n)$.

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 $a_1, \ldots, a_n$.

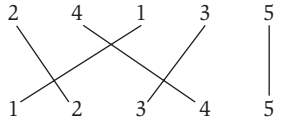

Figure 5.4 Counting the number of inversions in the sequence $2, 4, 1, 3, 5$. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the input list and the ascending list—in other words, an inversion.

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

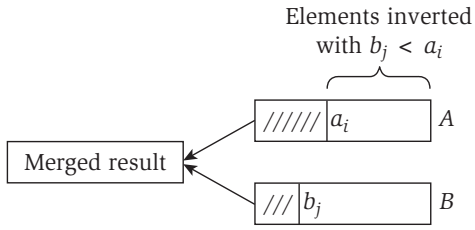

Figure 5.5 Merging two sorted lists while also counting the number of inversions between them.

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.

The Problem

The problem we consider is very simple to state: Given n points in the plane, find the pair that is closest together.

The problem was considered by M. I. Shamos and D. Hoey in the early 1970s, as part of their project to work out efficient algorithms for basic computational primitives in geometry. These algorithms formed the foundations of the then-fledgling field of *computational geometry*, and they have found their way into areas such as graphics, computer vision, geographic information systems, and molecular modeling. And although the closest-pair problem is one of the most natural algorithmic problems in geometry, it is surprisingly hard to find an efficient algorithm for it. It is immediately clear that there is an $O(n^2)$ solution—compute the distance between each pair of points and take the minimum—and so Shamos and Hoey asked whether an algorithm asymptotically faster than quadratic could be found. It took quite a long time before they resolved this question, and the $O(n \log n)$ algorithm we give below is essentially the one they discovered. In fact, when we return to this problem in Chapter 13, we will see that it is possible to further improve the running time to $O(n)$ using randomization.

Designing the Algorithm

We begin with a bit of notation. Let us denote the set of points by $P = \{p_1, \ldots, p_n\}$, where p_i has coordinates (x_i, y_i); and for two points $p_i, p_j \in P$, we use $d(p_i, p_j)$ to denote the standard Euclidean distance between them. Our goal is to find a pair of points p_i, p_j that minimizes $d(p_i, p_j)$.

We will assume that no two points in P have the same x-coordinate or the same y-coordinate. This makes the discussion cleaner; and it's easy to eliminate this assumption either by initially applying a rotation to the points that makes it true, or by slightly extending the algorithm we develop here.

It's instructive to consider the one-dimensional version of this problem for a minute, since it is much simpler and the contrasts are revealing. How would we find the closest pair of points on a line? We'd first sort them, in $O(n \log n)$ time, and then we'd walk through the sorted list, computing the distance from each point to the one that comes after it. It is easy to see that one of these distances must be the minimum one.

In two dimensions, we could try sorting the points by their y-coordinate (or x-coordinate) and hoping that the two closest points were near one another in the order of this sorted list. But it is easy to construct examples in which they are very far apart, preventing us from adapting our one-dimensional approach.

Instead, our plan will be to apply the style of divide and conquer used in Mergesort: we find the closest pair among the points in the "left half" of

P and the closest pair among the points in the "right half" of P; and then we use this information to get the overall solution in linear time. If we develop an algorithm with this structure, then the solution of our basic recurrence from (5.1) will give us an $O(n \log n)$ running time.

It is the last, "combining" phase of the algorithm that's tricky: the distances that have not been considered by either of our recursive calls are precisely those that occur between a point in the left half and a point in the right half; there are $\Omega(n^2)$ such distances, yet we need to find the smallest one in $O(n)$ time after the recursive calls return. If we can do this, our solution will be complete: it will be the smallest of the values computed in the recursive calls and this minimum "left-to-right" distance.

Setting Up the Recursion Let's get a few easy things out of the way first. It will be very useful if every recursive call, on a set $P' \subseteq P$, begins with two lists: a list P'_x in which all the points in P' have been sorted by increasing x-coordinate, and a list P'_y in which all the points in P' have been sorted by increasing y-coordinate. We can ensure that this remains true throughout the algorithm as follows.

First, before any of the recursion begins, we sort all the points in P by x-coordinate and again by y-coordinate, producing lists P_x and P_y. Attached to each entry in each list is a record of the position of that point in both lists.

The first level of recursion will work as follows, with all further levels working in a completely analogous way. We define Q to be the set of points in the first $\lceil n/2 \rceil$ positions of the list P_x (the "left half") and R to be the set of points in the final $\lfloor n/2 \rfloor$ positions of the list P_x (the "right half"). See Figure 5.6. By a single pass through each of P_x and P_y, in $O(n)$ time, we can create the

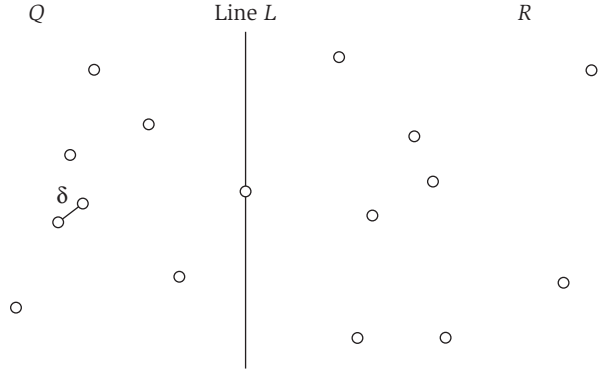

Figure 5.6 The first level of recursion: The point set P is divided evenly into Q and R by the line L, and the closest pair is found on each side recursively.

following four lists: Q_x, consisting of the points in Q sorted by increasing x-coordinate; Q_y, consisting of the points in Q sorted by increasing y-coordinate; and analogous lists R_x and R_y. For each entry of each of these lists, as before, we record the position of the point in both lists it belongs to.

We now recursively determine a closest pair of points in Q (with access to the lists Q_x and Q_y). Suppose that q_0^* and q_1^* are (correctly) returned as a closest pair of points in Q. Similarly, we determine a closest pair of points in R, obtaining r_0^* and r_1^*.

Combining the Solutions The general machinery of divide and conquer has gotten us this far, without our really having delved into the structure of the closest-pair problem. But it still leaves us with the problem that we saw looming originally: How do we use the solutions to the two subproblems as part of a linear-time "combining" operation?

Let δ be the minimum of $d(q_0^*,q_1^*)$ and $d(r_0^*,r_1^*)$. The real question is: Are there points $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$? If not, then we have already found the closest pair in one of our recursive calls. But if there are, then the closest such q and r form the closest pair in P.

Let x^* denote the x-coordinate of the rightmost point in Q, and let L denote the vertical line described by the equation $x = x^*$. This line L "separates" Q from R. Here is a simple fact.

(5.8) *If there exists $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$, then each of q and r lies within a distance δ of L.*

Proof. Suppose such q and r exist; we write $q = (q_x, q_y)$ and $r = (r_x, r_y)$. By the definition of x^*, we know that $q_x \le x^* \le r_x$. Then we have

$$x^* - q_x \le r_x - q_x \le d(q, r) < \delta$$

and

$$r_x - x^* \le r_x - q_x \le d(q, r) < \delta,$$

so each of q and r has an x-coordinate within δ of x^* and hence lies within distance δ of the line L. ■

So if we want to find a close q and r, we can restrict our search to the narrow band consisting only of points in P within δ of L. Let $S \subseteq P$ denote this set, and let S_y denote the list consisting of the points in S sorted by increasing y-coordinate. By a single pass through the list P_y, we can construct S_y in $O(n)$ time.

We can restate (5.8) as follows, in terms of the set S.

(5.9) *There exist $q \in Q$ and $r \in R$ for which $d(q, r) < \delta$ if and only if there exist $s, s' \in S$ for which $d(s, s') < \delta$.*

It's worth noticing at this point that S might in fact be the whole set P, in which case (5.8) and (5.9) really seem to buy us nothing. But this is actually far from true, as the following amazing fact shows.

(5.10) *If $s, s' \in S$ have the property that $d(s, s') < \delta$, then s and s' are within 15 positions of each other in the sorted list S_y.*

Proof. Consider the subset Z of the plane consisting of all points within distance δ of L. We partition Z into *boxes*: squares with horizontal and vertical sides of length $\delta/2$. One *row* of Z will consist of four boxes whose horizontal sides have the same y-coordinates. This collection of boxes is depicted in Figure 5.7.

Suppose two points of S lie in the same box. Since all points in this box lie on the same side of L, these two points either both belong to Q or both belong to R. But any two points in the same box are within distance $\delta \cdot \sqrt{2}/2 < \delta$, which contradicts our definition of δ as the minimum distance between any pair of points in Q or in R. Thus each box contains at most one point of S.

Now suppose that $s, s' \in S$ have the property that $d(s, s') < \delta$, and that they are at least 16 positions apart in S_y. Assume without loss of generality that s has the smaller y-coordinate. Then, since there can be at most one point per box, there are at least three rows of Z lying between s and s'. But any two points in Z separated by at least three rows must be a distance of at least $3\delta/2$ apart—a contradiction. ■

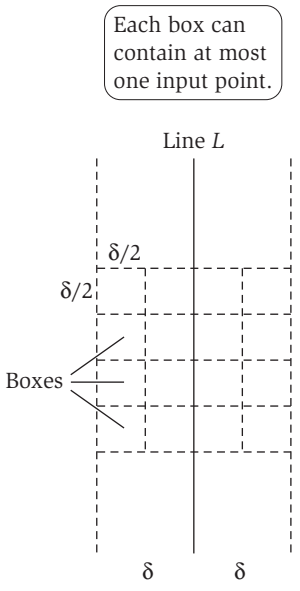

Figure 5.7 The portion of the plane close to the dividing line L, as analyzed in the proof of (5.10).

We note that the value of 15 can be reduced; but for our purposes at the moment, the important thing is that it is an absolute constant.

In view of (5.10), we can conclude the algorithm as follows. We make one pass through S_y, and for each $s \in S_y$, we compute its distance to each of the next 15 points in S_y. Statement (5.10) implies that in doing so, we will have computed the distance of each pair of points in S (if any) that are at distance less than δ from each other. So having done this, we can compare the smallest such distance to δ, and we can report one of two things: (i) the closest pair of points in S, if their distance is less than δ; or (ii) the (correct) conclusion that no pairs of points in S are within δ of each other. In case (i), this pair is the closest pair in P; in case (ii), the closest pair found by our recursive calls is the closest pair in P.

Note the resemblance between this procedure and the algorithm we rejected at the very beginning, which tried to make one pass through P in order

of y-coordinate. The reason such an approach works now is due to the extra knowledge (the value of δ) we've gained from the recursive calls, and the special structure of the set S.

This concludes the description of the "combining" part of the algorithm, since by (5.9) we have now determined whether the minimum distance between a point in Q and a point in R is less than δ, and if so, we have found the closest such pair.

A complete description of the algorithm and its proof of correctness are implicitly contained in the discussion so far, but for the sake of concreteness, we now summarize both.

Summary of the Algorithm A high-level description of the algorithm is the following, using the notation we have developed above.

```
Closest-Pair(P)
  Construct P_x and P_y  (O(n log n) time)
  (p_0*, p_1*) = Closest-Pair-Rec(P_x, P_y)

Closest-Pair-Rec(P_x, P_y)
  If |P| ≤ 3 then
    find closest pair by measuring all pairwise distances
  Endif

  Construct Q_x, Q_y, R_x, R_y (O(n) time)
  (q_0*, q_1*) = Closest-Pair-Rec(Q_x, Q_y)
  (r_0*, r_1*) = Closest-Pair-Rec(R_x, R_y)

  δ = min(d(q_0*, q_1*), d(r_0*, r_1*))
  x* = maximum x-coordinate of a point in set Q
  L = {(x, y) : x = x*}
  S = points in P within distance δ of L.

  Construct S_y (O(n) time)
  For each point s ∈ S_y, compute distance from s
     to each of next 15 points in S_y
     Let s, s' be pair achieving minimum of these distances
     (O(n) time)

  If d(s, s') < δ then
     Return (s, s')
  Else if d(q_0*, q_1*) < d(r_0*, r_1*) then
     Return (q_0*, q_1*)
```

```
Else
   Return (r*_0, r*_1)
Endif
```

Analyzing the Algorithm

We first prove that the algorithm produces a correct answer, using the facts we've established in the process of designing it.

(5.11) *The algorithm correctly outputs a closest pair of points in P.*

Proof. As we've noted, all the components of the proof have already been worked out, so here we just summarize how they fit together.

We prove the correctness by induction on the size of P, the case of $|P| \leq 3$ being clear. For a given P, the closest pair in the recursive calls is computed correctly by induction. By (5.10) and (5.9), the remainder of the algorithm correctly determines whether any pair of points in S is at distance less than δ, and if so returns the closest such pair. Now the closest pair in P either has both elements in one of Q or R, or it has one element in each. In the former case, the closest pair is correctly found by the recursive call; in the latter case, this pair is at distance less than δ, and it is correctly found by the remainder of the algorithm. ■

We now bound the running time as well, using (5.2).

(5.12) *The running time of the algorithm is $O(n \log n)$.*

Proof. The initial sorting of P by x- and y-coordinate takes time $O(n \log n)$. The running time of the remainder of the algorithm satisfies the recurrence (5.1), and hence is $O(n \log n)$ by (5.2). ■

5.5 Integer Multiplication

We now discuss a different application of divide and conquer, in which the "default" quadratic algorithm is improved by means of a different recurrence. The analysis of the faster algorithm will exploit one of the recurrences considered in Section 5.2, in which more than two recursive calls are spawned at each level.

The Problem

The problem we consider is an extremely basic one: the multiplication of two integers. In a sense, this problem is so basic that one may not initially think of it

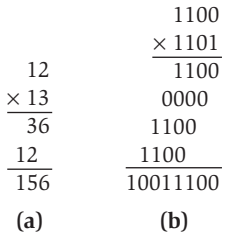

Figure 5.8 The elementary-school algorithm for multiplying two integers, in (a) decimal and (b) binary representation.

even as an algorithmic question. But, in fact, elementary schoolers are taught a concrete (and quite efficient) algorithm to multiply two n-digit numbers x and y. You first compute a "partial product" by multiplying each digit of y separately by x, and then you add up all the partial products. (Figure 5.8 should help you recall this algorithm. In elementary school we always see this done in base-10, but it works exactly the same way in base-2 as well.) Counting a single operation on a pair of bits as one primitive step in this computation, it takes $O(n)$ time to compute each partial product, and $O(n)$ time to combine it in with the running sum of all partial products so far. Since there are n partial products, this is a total running time of $O(n^2)$.

If you haven't thought about this much since elementary school, there's something initially striking about the prospect of improving on this algorithm. Aren't all those partial products "necessary" in some way? But, in fact, it is possible to improve on $O(n^2)$ time using a different, recursive way of performing the multiplication.

Designing the Algorithm

The improved algorithm is based on a more clever way to break up the product into partial sums. Let's assume we're in base-2 (it doesn't really matter), and start by writing x as $x_1 \cdot 2^{n/2} + x_0$. In other words, x_1 corresponds to the "high-order" $n/2$ bits, and x_0 corresponds to the "low-order" $n/2$ bits. Similarly, we write $y = y_1 \cdot 2^{n/2} + y_0$. Thus, we have

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0. \end{aligned} \tag{5.1}$$

Equation (5.1) reduces the problem of solving a single n-bit instance (multiplying the two n-bit numbers x and y) to the problem of solving four $n/2$-bit instances (computing the products x_1y_1, x_1y_0, x_0y_1, and x_0y_0). So we have a first candidate for a divide-and-conquer solution: recursively compute the results for these four $n/2$-bit instances, and then combine them using Equation

(5.1). The combining of the solution requires a constant number of additions of $O(n)$-bit numbers, so it takes time $O(n)$; thus, the running time $T(n)$ is bounded by the recurrence

$$T(n) \leq 4T(n/2) + cn$$

for a constant c. Is this good enough to give us a subquadratic running time?

We can work out the answer by observing that this is just the case $q = 4$ of the class of recurrences in (5.3). As we saw earlier in the chapter, the solution to this is $T(n) \leq O(n^{\log_2 q}) = O(n^2)$.

So, in fact, our divide-and-conquer algorithm with four-way branching was just a complicated way to get back to quadratic time! If we want to do better using a strategy that reduces the problem to instances on $n/2$ bits, we should try to get away with only *three* recursive calls. This will lead to the case $q = 3$ of (5.3), which we saw had the solution $T(n) \leq O(n^{\log_2 q}) = O(n^{1.59})$.

Recall that our goal is to compute the expression $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$ in Equation (5.1). It turns out there is a simple trick that lets us determine all of the terms in this expression using just three recursive calls. The trick is to consider the result of the single multiplication $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$. This has the four products above added together, at the cost of a single recursive multiplication. If we now also determine x_1y_1 and x_0y_0 by recursion, then we get the outermost terms explicitly, and we get the middle term by subtracting x_1y_1 and x_0y_0 away from $(x_1 + x_0)(y_1 + y_0)$.

Thus, in full, our algorithm is

```
Recursive-Multiply(x,y):
  Write x = x_1 · 2^{n/2} + x_0
        y = y_1 · 2^{n/2} + y_0
  Compute x_1 + x_0 and y_1 + y_0
  p = Recursive-Multiply(x_1 + x_0, y_1 + y_0)
  x_1y_1 = Recursive-Multiply(x_1, y_1)
  x_0y_0 = Recursive-Multiply(x_0, y_0)
  Return x_1y_1 · 2^n + (p − x_1y_1 − x_0y_0) · 2^{n/2} + x_0y_0
```

Analyzing the Algorithm

We can determine the running time of this algorithm as follows. Given two n-bit numbers, it performs a constant number of additions on $O(n)$-bit numbers, in addition to the three recursive calls. Ignoring for now the issue that $x_1 + x_0$ and $y_1 + y_0$ may have $n/2 + 1$ bits (rather than just $n/2$), which turns out not to affect the asymptotic results, each of these recursive calls is on an instance of size $n/2$. Thus, in place of our four-way branching recursion, we now have

a three-way branching one, with a running time that satisfies

$$T(n) \leq 3T(n/2) + cn$$

for a constant c.

This is the case $q = 3$ of (5.3) that we were aiming for. Using the solution to that recurrence from earlier in the chapter, we have

(5.13) *The running time of* `Recursive-Multiply` *on two n-bit factors is* $O(n^{\log_2 3}) = O(n^{1.59})$.

5.6 Convolutions and the Fast Fourier Transform

As a final topic in this chapter, we show how our basic recurrence from (5.1) is used in the design of the *Fast Fourier Transform*, an algorithm with a wide range of applications.

The Problem

Given two vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$, there are a number of common ways of combining them. For example, one can compute the sum, producing the vector $a + b = (a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1})$; or one can compute the inner product, producing the real number $a \cdot b = a_0b_0 + a_1b_1 + \cdots + a_{n-1}b_{n-1}$. (For reasons that will emerge shortly, it is useful to write vectors in this section with coordinates that are indexed starting from 0 rather than 1.)

A means of combining vectors that is very important in applications, even if it doesn't always show up in introductory linear algebra courses, is the *convolution* $a * b$. The convolution of two vectors of length n (as a and b are) is a vector with $2n - 1$ coordinates, where coordinate k is equal to

$$\sum_{\substack{(i,j):i+j=k \\ i,j<n}} a_i b_j.$$

In other words,

$$a * b = (a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_1b_1 + a_2b_0, \ldots, a_{n-2}b_{n-1} + a_{n-1}b_{n-2}, a_{n-1}b_{n-1}).$$

This definition is a bit hard to absorb when you first see it. Another way to think about the convolution is to picture an $n \times n$ table whose (i, j) entry is a_ib_j, like this,

$$\begin{array}{ccccc} a_0b_0 & a_0b_1 & \dots & a_0b_{n-2} & a_0b_{n-1} \\ a_1b_0 & a_1b_1 & \dots & a_1b_{n-2} & a_1b_{n-1} \\ a_2b_0 & a_2b_1 & \dots & a_2b_{n-2} & a_2b_{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n-1}b_0 & a_{n-1}b_1 & \dots & a_{n-1}b_{n-2} & a_{n-1}b_{n-1} \end{array}$$

and then to compute the coordinates in the convolution vector by summing along the diagonals.

It's worth mentioning that, unlike the vector sum and inner product, the convolution can be easily generalized to vectors of different lengths, $a = (a_0, a_1, \dots, a_{m-1})$ and $b = (b_0, b_1, \dots, b_{n-1})$. In this more general case, we define $a * b$ to be a vector with $m + n - 1$ coordinates, where coordinate k is equal to

$$\sum_{\substack{(i,j):i+j=k \\ i<m, j<n}} a_i b_j.$$

We can picture this using the table of products a_ib_j as before; the table is now rectangular, but we still compute coordinates by summing along the diagonals. (From here on, we'll drop explicit mention of the condition $i < m, j < n$ in the summations for convolutions, since it will be clear from the context that we only compute the sum over terms that are defined.)

It's not just the definition of a convolution that is a bit hard to absorb at first; the motivation for the definition can also initially be a bit elusive. What are the circumstances where you'd want to compute the convolution of two vectors? In fact, the convolution comes up in a surprisingly wide variety of different contexts. To illustrate this, we mention the following examples here.

- A first example (which also proves that the convolution is something that we all saw implicitly in high school) is polynomial multiplication. Any polynomial $A(x) = a_0 + a_1x + a_2x^2 + \cdots a_{m-1}x^{m-1}$ can be represented just as naturally using its vector of coefficients, $a = (a_0, a_1, \dots, a_{m-1})$. Now, given two polynomials $A(x) = a_0 + a_1x + a_2x^2 + \cdots a_{m-1}x^{m-1}$ and $B(x) = b_0 + b_1x + b_2x^2 + \cdots b_{n-1}x^{n-1}$, consider the polynomial $C(x) = A(x)B(x)$ that is equal to their product. In this polynomial $C(x)$, the coefficient on the x^k term is equal to

  $$c_k = \sum_{(i,j):i+j=k} a_i b_j.$$

  In other words, the coefficient vector c of $C(x)$ is the convolution of the coefficient vectors of $A(x)$ and $B(x)$.
- Arguably the most important application of convolutions in practice is for *signal processing*. This is a topic that could fill an entire course, so

we'll just give a simple example here to suggest one way in which the convolution arises.

Suppose we have a vector $a = (a_0, a_1, \ldots, a_{m-1})$ which represents a sequence of measurements, such as a temperature or a stock price, sampled at m consecutive points in time. Sequences like this are often very noisy due to measurement error or random fluctuations, and so a common operation is to "smooth" the measurements by averaging each value a_i with a weighted sum of its neighbors within k steps to the left and right in the sequence, the weights decaying quickly as one moves away from a_i. For example, in *Gaussian smoothing*, one replaces a_i with

$$a'_i = \frac{1}{Z} \sum_{j=i-k}^{i+k} a_j e^{-(j-i)^2},$$

for some "width" parameter k, and with Z chosen simply to normalize the weights in the average to add up to 1. (There are some issues with boundary conditions—what do we do when $i - k < 0$ or $i + k > m$?—but we could deal with these, for example, by discarding the first and last k entries from the smoothed signal, or by scaling them differently to make up for the missing terms.)

To see the connection with the convolution operation, we picture this smoothing operation as follows. We first define a "mask"

$$w = (w_{-k}, w_{-(k-1)}, \ldots, w_{-1}, w_0, w_1, \ldots, w_{k-1}, w_k)$$

consisting of the weights we want to use for averaging each point with its neighbors. (For example, $w = \frac{1}{Z}(e^{-k^2}, e^{-(k-1)^2}, \ldots, e^{-1}, 1, e^{-1}, \ldots, e^{-(k-1)^2}, e^{-k^2})$ in the Gaussian case above.) We then iteratively position this mask so it is centered at each possible point in the sequence a; and for each positioning, we compute the weighted average. In other words, we replace a_i with $a'_i = \sum_{s=-k}^{k} w_s a_{i+s}$.

This last expression is essentially a convolution; we just have to warp the notation a bit so that this becomes clear. Let's define $b = (b_0, b_1, \ldots, b_{2k})$ by setting $b_\ell = w_{k-\ell}$. Then it's not hard to check that with this definition we have the smoothed value

$$a'_i = \sum_{(j,\ell):j+\ell=i+k} a_j b_\ell.$$

In other words, the smoothed sequence is just the convolution of the original signal and the reverse of the mask (with some meaningless coordinates at the beginning and end).

- We mention one final application: the problem of combining histograms. Suppose we're studying a population of people, and we have the following two histograms: One shows the annual income of all the men in the population, and one shows the annual income of all the women. We'd now like to produce a new histogram, showing for each k the number of *pairs* (M, W) for which man M and woman W have a combined income of k.

  This is precisely a convolution. We can write the first histogram as a vector $a = (a_0, \ldots, a_{m-1})$, to indicate that there are a_i men with annual income equal to i. We can similarly write the second histogram as a vector $b = (b_0, \ldots, b_{n-1})$. Now, let c_k denote the number of pairs (m, w) with combined income k; this is the number of ways of choosing a man with income a_i and a woman with income b_j, for any pair (i, j) where $i + j = k$. In other words,

$$c_k = \sum_{(i,j):i+j=k} a_i b_j.$$

  so the combined histogram $c = (c_0, \ldots, c_{m+n-2})$ is simply the convolution of a and b.

  (Using terminology from probability that we will develop in Chapter 13, one can view this example as showing how convolution is the underlying means for computing the distribution of the sum of two independent random variables.)

Computing the Convolution Having now motivated the notion of convolution, let's discuss the problem of computing it efficiently. For simplicity, we will consider the case of equal length vectors (i.e., $m = n$), although everything we say carries over directly to the case of vectors of unequal lengths.

Computing the convolution is a more subtle question than it may first appear. The definition of convolution, after all, gives us a perfectly valid way to compute it: for each k, we just calculate the sum

$$\sum_{(i,j):i+j=k} a_i b_j$$

and use this as the value of the k^{th} coordinate. The trouble is that this direct way of computing the convolution involves calculating the product $a_i b_j$ for every pair (i, j) (in the process of distributing over the sums in the different terms) and this is $\Theta(n^2)$ arithmetic operations. Spending $O(n^2)$ time on computing the convolution seems natural, as the definition involves $O(n^2)$ multiplications $a_i b_j$. However, it's not inherently clear that we have to spend quadratic time to compute a convolution, since the input and output both only have size $O(n)$.

Could one design an algorithm that bypasses the quadratic-size definition of convolution and computes it in some smarter way?

In fact, quite surprisingly, this is possible. We now describe a method that computes the convolution of two vectors using only $O(n \log n)$ arithmetic operations. The crux of this method is a powerful technique known as the *Fast Fourier Transform* (FFT). The FFT has a wide range of further applications in analyzing sequences of numerical values; computing convolutions quickly, which we focus on here, is just one of these applications.

Designing and Analyzing the Algorithm

To break through the quadratic time barrier for convolutions, we are going to exploit the connection between the convolution and the multiplication of two polynomials, as illustrated in the first example discussed previously. But rather than use convolution as a primitive in polynomial multiplication, we are going to exploit this connection in the opposite direction.

Suppose we are given the vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$. We will view them as the polynomials $A(x) = a_0 + a_1x + a_2x^2 + \cdots a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1x + b_2x^2 + \cdots b_{n-1}x^{n-1}$, and we'll seek to compute their product $C(x) = A(x)B(x)$ in $O(n \log n)$ time. If $c = (c_0, c_1, \ldots, c_{2n-2})$ is the vector of coefficients of C, then we recall from our earlier discussion that c is exactly the convolution $a * b$, and so we can then read off the desired answer directly from the coefficients of $C(x)$.

Now, rather than multiplying A and B symbolically, we can treat them as functions of the variable x and multiply them as follows.

(i) First we choose $2n$ values $x_1, x_2, \ldots, x_{2n}$ and evaluate $A(x_j)$ and $B(x_j)$ for each of $j = 1, 2, \ldots, 2n$.

(ii) We can now compute $C(x_j)$ for each j very easily: $C(x_j)$ is simply the product of the two numbers $A(x_j)$ and $B(x_j)$.

(iii) Finally, we have to recover C from its values on $x_1, x_2, \ldots, x_{2n}$. Here we take advantage of a fundamental fact about polynomials: any polynomial of degree d can be reconstructed from its values on any set of $d + 1$ or more points. This is known as *polynomial interpolation*, and we'll discuss the mechanics of performing interpolation in more detail later. For the moment, we simply observe that since A and B each have degree at most $n - 1$, their product C has degree at most $2n - 2$, and so it can be reconstructed from the values $C(x_1), C(x_2), \ldots, C(x_{2n})$ that we computed in step (ii).

This approach to multiplying polynomials has some promising aspects and some problematic ones. First, the good news: step (ii) requires only

$O(n)$ arithmetic operations, since it simply involves the multiplication of $O(n)$ numbers. But the situation doesn't look as hopeful with steps (i) and (iii). In particular, evaluating the polynomials A and B on a single value takes $\Omega(n)$ operations, and our plan calls for performing $2n$ such evaluations. This seems to bring us back to quadratic time right away.

The key idea that will make this all work is to find a set of $2n$ values $x_1, x_2, \ldots, x_{2n}$ that are intimately related in some way, such that the work in evaluating A and B on all of them can be shared across different evaluations. A set for which this will turn out to work very well is the *complex roots of unity.*

The Complex Roots of Unity At this point, we're going to need to recall a few facts about complex numbers and their role as solutions to polynomial equations.

Recall that complex numbers can be viewed as lying in the "complex plane," with axes representing their real and imaginary parts. We can write a complex number using polar coordinates with respect to this plane as $re^{\theta i}$, where $e^{\pi i} = -1$ (and $e^{2\pi i} = 1$). Now, for a positive integer k, the polynomial equation $x^k = 1$ has k distinct complex roots, and it is easy to identify them. Each of the complex numbers $\omega_{j,k} = e^{2\pi ji/k}$ (for $j = 0, 1, 2, \ldots, k-1$) satisfies the equation, since

$$(e^{2\pi ji/k})^k = e^{2\pi ji} = (e^{2\pi i})^j = 1^j = 1,$$

and each of these numbers is distinct, so these are all the roots. We refer to these numbers as the k^{th} *roots of unity.* We can picture these roots as a set of k equally spaced points lying on the unit circle in the complex plane, as shown in Figure 5.9 for the case $k = 8$.

For our numbers $x_1, \ldots, x_{2n}$ on which to evaluate A and B, we will choose the $(2n)^{\text{th}}$ roots of unity. It's worth mentioning (although it's not necessary for understanding the algorithm) that the use of the complex roots of unity is the basis for the name *Fast Fourier Transform*: the representation of a degree-d

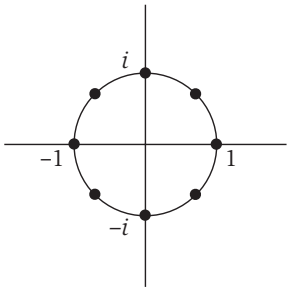

Figure 5.9 The 8^{th} roots of unity in the complex plane.

polynomial P by its values on the $(d+1)^{\text{st}}$ roots of unity is sometimes referred to as the *discrete Fourier transform* of P; and the heart of our procedure is a method for making this computation fast.

A Recursive Procedure for Polynomial Evaluation We want to design an algorithm for evaluating A on each of the $(2n)^{\text{th}}$ roots of unity recursively, so as to take advantage of the familiar recurrence from (5.1)—namely, $T(n) \leq 2T(n/2) + O(n)$ where $T(n)$ in this case denotes the number of operations required to evaluate a polynomial of degree $n-1$ on all the $(2n)^{\text{th}}$ roots of unity. For simplicity in describing this algorithm, we will assume that n is a power of 2.

How does one break the evaluation of a polynomial into two equal-sized subproblems? A useful trick is to define two polynomials, $A_{even}(x)$ and $A_{odd}(x)$, that consist of the even and odd coefficients of A, respectively. That is,

$$A_{even}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{(n-2)/2},$$

and

$$A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{(n-1)} x^{(n-2)/2}.$$

Simple algebra shows us that

$$A(x) = A_{even}(x^2) + x A_{odd}(x^2),$$

and so this gives us a way to compute $A(x)$ in a constant number of operations, given the evaluation of the two constituent polynomials that each have half the degree of A.

Now suppose that we evaluate each of A_{even} and A_{odd} on the n^{th} roots of unity. This is exactly a version of the problem we face with A and the $(2n)^{\text{th}}$ roots of unity, except that the input is half as large: the degree is $(n-2)/2$ rather than $n-1$, and we have n roots of unity rather than $2n$. Thus we can perform these evaluations in time $T(n/2)$ for each of A_{even} and A_{odd}, for a total time of $2T(n/2)$.

We're now very close to having a recursive algorithm that obeys (5.1) and gives us the running time we want; we just have to produce the evaluations of A on the $(2n)^{\text{th}}$ roots of unity using $O(n)$ additional operations. But this is easy, given the results from the recursive calls on A_{even} and A_{odd}. Consider one of these roots of unity $\omega_{j,2n} = e^{2\pi ji/2n}$. The quantity $\omega_{j,2n}^2$ is equal to $(e^{2\pi ji/2n})^2 = e^{2\pi ji/n}$, and hence $\omega_{j,2n}^2$ is an n^{th} root of unity. So when we go to compute

$$A(\omega_{j,2n}) = A_{even}(\omega_{j,2n}^2) + \omega_{j,2n} A_{odd}(\omega_{j,2n}^2),$$

we discover that both of the evaluations on the right-hand side have been performed in the recursive step, and so we can determine $A(\omega_{j,2n})$ using a

constant number of operations. Doing this for all $2n$ roots of unity is therefore $O(n)$ additional operations after the two recursive calls, and so the bound $T(n)$ on the number of operations indeed satisfies $T(n) \leq 2T(n/2) + O(n)$. We run the same procedure to evaluate the polynomial B on the $(2n)^{\text{th}}$ roots of unity as well, and this gives us the desired $O(n \log n)$ bound for step (i) of our algorithm outline.

Polynomial Interpolation We've now seen how to evaluate A and B on the set of all $(2n)^{\text{th}}$ roots of unity using $O(n \log n)$ operations and, as noted above, we can clearly compute the products $C(\omega_{j,n}) = A(\omega_{j,2n})B(\omega_{j,2n})$ in $O(n)$ more operations. Thus, to conclude the algorithm for multiplying A and B, we need to execute step (iii) in our earlier outline using $O(n \log n)$ operations, reconstructing C from its values on the $(2n)^{\text{th}}$ roots of unity.

In describing this part of the algorithm, it's worth keeping track of the following top-level point: it turns out that the reconstruction of C can be achieved simply by defining an appropriate polynomial (the polynomial D below) and evaluating it at the $(2n)^{\text{th}}$ roots of unity. This is exactly what we've just seen how to do using $O(n \log n)$ operations, so we do it again here, spending an additional $O(n \log n)$ operations and concluding the algorithms.

Consider a polynomial $C(x) = \sum_{s=0}^{2n-1} c_s x^s$ that we want to reconstruct from its values $C(\omega_{s,2n})$ at the $(2n)^{\text{th}}$ roots of unity. Define a new polynomial $D(x) = \sum_{s=0}^{2n-1} d_s x^s$, where $d_s = C(\omega_{s,2n})$. We now consider the values of $D(x)$ at the $(2n)^{\text{th}}$ roots of unity.

$$\begin{aligned} D(\omega_{j,2n}) &= \sum_{s=0}^{2n-1} C(\omega_{s,2n})\omega_{j,2n}^s \\ &= \sum_{s=0}^{2n-1} \Big(\sum_{t=0}^{2n-1} c_t \omega_{s,2n}^t\Big)\omega_{j,2n}^s \\ &= \sum_{t=0}^{2n-1} c_t \Big(\sum_{s=0}^{2n-1} \omega_{s,2n}^t \omega_{j,2n}^s\Big), \end{aligned}$$

by definition. Now recall that $\omega_{s,2n} = (e^{2\pi i/2n})^s$. Using this fact and extending the notation to $\omega_{s,2n} = (e^{2\pi i/2n})^s$ even when $s \geq 2n$, we get that

$$\begin{aligned} D(\omega_{j,2n}) &= \sum_{t=0}^{2n-1} c_t \Big(\sum_{s=0}^{2n-1} e^{(2\pi i)(st+js)/2n}\Big) \\ &= \sum_{t=0}^{2n-1} c_t \Big(\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s\Big). \end{aligned}$$

To analyze the last line, we use the fact that for any $(2n)^{\text{th}}$ root of unity $\omega \neq 1$, we have $\sum_{s=0}^{2n-1} \omega^s = 0$. This is simply because ω is by definition a root of $x^{2n} - 1 = 0$; since $x^{2n} - 1 = (x-1)(\sum_{t=0}^{2n-1} x^t)$ and $\omega \neq 1$, it follows that ω is also a root of $(\sum_{t=0}^{2n-1} x^t)$.

Thus the only term of the last line's outer sum that is not equal to 0 is for c_t such that $\omega_{t+j,2n} = 1$; and this happens if $t+j$ is a multiple of $2n$, that is, if $t = 2n - j$. For this value, $\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s = \sum_{s=0}^{2n-1} 1 = 2n$. So we get that $D(\omega_{j,2n}) = 2nc_{2n-j}$. Evaluating the polynomial $D(x)$ at the $(2n)^{\text{th}}$ roots of unity thus gives us the coeffients of the polynomial $C(x)$ in reverse order (multiplied by $2n$ each). We sum this up as follows.

(5.14) *For any polynomial $C(x) = \sum_{s=0}^{2n-1} c_s x^s$, and corresponding polynomial $D(x) = \sum_{s=0}^{2n-1} C(\omega_{s,2n}) x^s$, we have that $c_s = \frac{1}{2n} D(\omega_{2n-s,2n})$.*

We can do all the evaluations of the values $D(\omega_{2n-s,2n})$ in $O(n \log n)$ operations using the divide-and-conquer approach developed for step (i).

And this wraps everything up: we reconstruct the polynomial C from its values on the $(2n)^{\text{th}}$ roots of unity, and then the coefficients of C are the coordinates in the convolution vector $c = a * b$ that we were originally seeking.

In summary, we have shown the following.

(5.15) *Using the Fast Fourier Transform to determine the product polynomial $C(x)$, we can compute the convolution of the original vectors a and b in $O(n \log n)$ time.*

Solved Exercises

Solved Exercise 1

Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], \ldots, A[n]$ is *unimodal*: For some index p between 1 and n, the values in the array entries increase up to position p in A and then decrease the remainder of the way until position n. (So if you were to draw a plot with the array position j on the x-axis and the value of the entry $A[j]$ on the y-axis, the plotted points would rise until x-value p, where they'd achieve their maximum, and then fall from there on.)

You'd like to find the "peak entry" p without having to read the entire array—in fact, by reading as few entries of A as possible. Show how to find the entry p by reading at most $O(\log n)$ entries of A.

Solution Let's start with a general discussion on how to achieve a running time of $O(\log n)$ and then come back to the specific problem here. If one needs to compute something using only $O(\log n)$ operations, a useful strategy that we discussed in Chapter 2 is to perform a constant amount of work, throw away half the input, and continue recursively on what's left. This was the idea, for example, behind the $O(\log n)$ running time for binary search.

We can view this as a divide-and-conquer approach: for some constant $c > 0$, we perform at most c operations and then continue recursively on an input of size at most $n/2$. As in the chapter, we will assume that the recursion "bottoms out" when $n = 2$, performing at most c operations to finish the computation. If $T(n)$ denotes the running time on an input of size n, then we have the recurrence

(5.16)

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

when $n > 2$*, and*

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

It is not hard to solve this recurrence by unrolling it, as follows.

- *Analyzing the first few levels:* At the first level of recursion, we have a single problem of size n, which takes time at most c plus the time spent in all subsequent recursive calls. The next level has one problem of size at most $n/2$, which contributes another c, and the level after that has one problem of size at most $n/4$, which contributes yet another c.
- *Identifying a pattern:* No matter how many levels we continue, each level will have just one problem: level j has a single problem of size at most $n/2^j$, which contributes c to the running time, independent of j.
- *Summing over all levels of recursion:* Each level of the recursion is contributing at most c operations, and it takes $\log_2 n$ levels of recursion to reduce n to 2. Thus the total running time is at most c times the number of levels of recursion, which is at most $c \log_2 n = O(\log n)$.

We can also do this by partial substitution. Suppose we guess that $T(n) \leq k \log_b n$, where we don't know k or b. Assuming that this holds for smaller values of n in an inductive argument, we would have

$$\begin{aligned} T(n) &\leq T(n/2) + c \\ &\leq k \log_b(n/2) + c \\ &= k \log_b n - k \log_b 2 + c. \end{aligned}$$

The first term on the right is exactly what we want, so we just need to choose k and b to negate the added c at the end. This we can do by setting $b = 2$ and $k = c$, so that $k \log_b 2 = c \log_2 2 = c$. Hence we end up with the solution $T(n) \leq c \log_2 n$, which is exactly what we got by unrolling the recurrence.

Finally, we should mention that one can get an $O(\log n)$ running time, by essentially the same reasoning, in the more general case when each level of the recursion throws away any constant fraction of the input, transforming an instance of size n to one of size at most an, for some constant $a < 1$. It now takes at most $\log_{1/a} n$ levels of recursion to reduce n down to a constant size, and each level of recursion involves at most c operations.

Now let's get back to the problem at hand. If we wanted to set ourselves up to use (5.16), we could probe the midpoint of the array and try to determine whether the "peak entry" p lies before or after this midpoint.

So suppose we look at the value $A[n/2]$. From this value alone, we can't tell whether p lies before or after $n/2$, since we need to know whether entry $n/2$ is sitting on an "up-slope" or on a "down-slope." So we also look at the values $A[n/2 - 1]$ and $A[n/2 + 1]$. There are now three possibilities.

- If $A[n/2 - 1] < A[n/2] < A[n/2 + 1]$, then entry $n/2$ must come strictly before p, and so we can continue recursively on entries $n/2 + 1$ through n.
- If $A[n/2 - 1] > A[n/2] > A[n/2 + 1]$, then entry $n/2$ must come strictly after p, and so we can continue recursively on entries 1 through $n/2 - 1$.
- Finally, if $A[n/2]$ is larger than both $A[n/2 - 1]$ and $A[n/2 + 1]$, we are done: the peak entry is in fact equal to $n/2$ in this case.

In all these cases, we perform at most three probes of the array A and reduce the problem to one of at most half the size. Thus we can apply (5.16) to conclude that the running time is $O(\log n)$.

Solved Exercise 2

You're consulting for a small computation-intensive investment company, and they have the following type of problem that they want to solve over and over. A typical instance of the problem is the following. They're doing a simulation in which they look at n consecutive days of a given stock, at some point in the past. Let's number the days $i = 1, 2, \ldots, n$; for each day i, they have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) Suppose during this time period, they wanted to buy 1,000 shares on some day and sell all these shares on some (later) day. They want to know: When should they have bought and when should they have sold in order to have made as much money as possible? (If

there was no way to make money during the n days, you should report this instead.)

For example, suppose $n = 3$, $p(1) = 9$, $p(2) = 1$, $p(3) = 5$. Then you should return *"buy on 2, sell on 3"* (buying on day 2 and selling on day 3 means they would have made \$4 per share, the maximum possible for that period).

Clearly, there's a simple algorithm that takes time $O(n^2)$: try all possible pairs of buy/sell days and see which makes them the most money. Your investment friends were hoping for something a little better.

Show how to find the correct numbers i and j in time $O(n \log n)$.

Solution We've seen a number of instances in this chapter where a brute-force search over pairs of elements can be reduced to $O(n \log n)$ by divide and conquer. Since we're faced with a similar issue here, let's think about how we might apply a divide-and-conquer strategy.

A natural approach would be to consider the first $n/2$ days and the final $n/2$ days separately, solving the problem recursively on each of these two sets, and then figure out how to get an overall solution from this in $O(n)$ time. This would give us the usual recurrence $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$, and hence $O(n \log n)$ by (5.1).

Also, to make things easier, we'll make the usual assumption that n is a power of 2. This is no loss of generality: if n' is the next power of 2 greater than n, we can set $p(i) = p(n)$ for all i between n and n'. In this way, we do not change the answer, and we at most double the size of the input (which will not affect the $O()$ notation).

Now, let S be the set of days $1, \ldots, n/2$, and S' be the set of days $n/2 + 1, \ldots, n$. Our divide-and-conquer algorithm will be based on the following observation: either there is an optimal solution in which the investors are holding the stock at the end of day $n/2$, or there isn't. Now, if there isn't, then the optimal solution is the better of the optimal solutions on the sets S and S'. If there is an optimal solution in which they hold the stock at the end of day $n/2$, then the value of this solution is $p(j) - p(i)$ where $i \in S$ and $j \in S'$. But this value is maximized by simply choosing $i \in S$ which minimizes $p(i)$, and choosing $j \in S'$ which maximizes $p(j)$.

Thus our algorithm is to take the best of the following three possible solutions.

- The optimal solution on S.
- The optimal solution on S'.
- The maximum of $p(j) - p(i)$, over $i \in S$ and $j \in S'$.

The first two alternatives are computed in time $T(n/2)$, each by recursion, and the third alternative is computed by finding the minimum in S and the

maximum in S', which takes time $O(n)$. Thus the running time $T(n)$ satisfies

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n),$$

as desired.

We note that this is not the best running time achievable for this problem. In fact, one can find the optimal pair of days in $O(n)$ time using dynamic programming, the topic of the next chapter; at the end of that chapter, we will pose this question as Exercise 7.

Exercises

1. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values—so there are $2n$ values total—and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the n^{th} smallest value.

However, the only way you can access these values is through *queries* to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most $O(\log n)$ queries.

2. Recall the problem of finding the number of inversions. As in the text, we are given a sequence of n numbers $a_1, \ldots, a_n$, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i > a_j$.

We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$. Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

3. Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards

corresponding to it, and we'll say that two bank cards are *equivalent* if they correspond to the same account.

It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

4. You've been working with some physicists who need to study, as part of their experimental design, the interactions among large numbers of very small charged particles. Basically, their setup works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus we can model their structure as consisting of the points $\{1, 2, 3, \ldots, n\}$ on the real line; and at each of these points j, they have a particle with charge q_j. (Each charge can be either positive or negative.)

They want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational part is where they need your help. The total net force on particle j, by Coulomb's Law, is equal to

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

They've written the following simple program to compute F_j for all j:

```
For j = 1, 2, ..., n
  Initialize F_j to 0
  For i = 1, 2, ..., n
    If i < j then
      Add C q_i q_j / (j - i)^2 to F_j
    Else if i > j then
      Add -C q_i q_j / (j - i)^2 to F_j
    Endif
  Endfor
  Output F_j
Endfor
```

It's not hard to analyze the running time of this program: each invocation of the inner loop, over i, takes $O(n)$ time, and this inner loop is invoked $O(n)$ times total, so the overall running time is $O(n^2)$.

The trouble is, for the large values of n they're working with, the program takes several minutes to run. On the other hand, their experimental setup is optimized so that they can throw down n particles, perform the measurements, and be ready to handle n more particles within a few seconds. So they'd really like it if there were a way to compute all the forces F_j much more quickly, so as to keep up with the rate of the experiment.

Help them out by designing an algorithm that computes all the forces F_j in $O(n \log n)$ time.

5. *Hidden surface removal* is a problem in computer graphics that scarcely needs an introduction: when Woody is standing in front of Buzz, you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, . . . well, you get the idea.

The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n nonvertical lines in the plane, labeled $L_1, \ldots, L_n$, with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is *uppermost* at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at x_0: $a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line L_i is *visible* if there is some x-coordinate at which it is uppermost—intuitively, some portion of it can be seen if you look down from "$y = \infty$."

Give an algorithm that takes n lines as input and in $O(n \log n)$ time returns all of the ones that are visible. Figure 5.10 gives an example.

6. Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

You are given such a complete binary tree T, but the labeling is only specified in the following *implicit* way: for each node v, you can determine the value x_v by *probing* the node v. Show how to find a local minimum of T using only $O(\log n)$ *probes* to the nodes of T.

7. Suppose now that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of

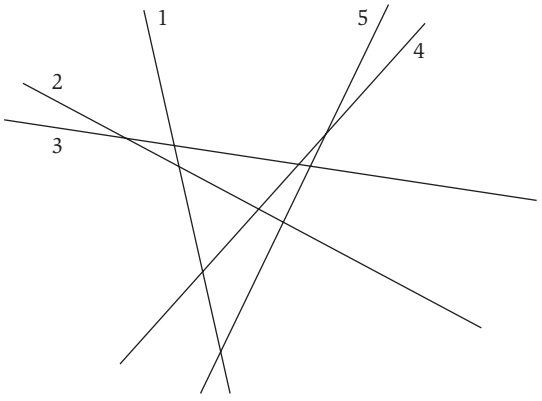

Figure 5.10 An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, ℓ) are joined by an edge if and only if $|i - k| + |j - \ell| = 1$.)

We use some of the terminology of the previous question. Again, each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Notes and Further Reading

The militaristic coinage "divide and conquer" was introduced somewhat after the technique itself. Knuth (1998) credits John von Neumann with one early explicit application of the approach, the development of the Mergesort Algorithm in 1945. Knuth (1997b) also provides further discussion of techniques for solving recurrences.

The algorithm for computing the closest pair of points in the plane is due to Michael Shamos, and is one of the earliest nontrivial algorithms in the field of computational geometry; the survey paper by Smid (1999) discusses a wide range of results on closest-point problems. A faster randomized algorithm for this problem will be discussed in Chapter 13. (Regarding the nonobviousness of the divide-and-conquer algorithm presented here, Smid also makes the interesting historical observation that researchers originally suspected quadratic time might be the best one could do for finding the closest pair of points in the plane.) More generally, the divide-and-conquer approach has proved very useful in computational geometry, and the books by Preparata and Shamos

(1985) and de Berg et al. (1997) give many further examples of this technique in the design of geometric algorithms.

The algorithm for multiplying two n-bit integers in subquadratic time is due to Karatsuba and Ofman (1962). Further background on asymptotically fast multiplication algorithms is given by Knuth (1997b). Of course, the number of bits in the input must be sufficiently large for any of these subquadratic methods to improve over the standard algorithm.

Press et al. (1988) provide further coverage of the Fast Fourier Transform, including background on its applications in signal processing and related areas.

Notes on the Exercises Exercise 7 is based on a result of Donna Llewellyn, Craig Tovey, and Michael Trick.