

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.

$$(5.6) \quad \text{For some constant } c,$$

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

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

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

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 $(n/2)^2$, $(n/4)^2$, 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, \dots, 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 $a_1 < a_2 < \dots < a_n$, 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, \dots, a_n .

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).

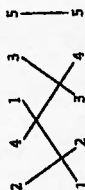


Figure 5.4 Counting the number of inversions in the sequence 2, 4, 1, 3, 5. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the input list and the ascending list—in other words, an inversion.

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, \dots, a_m and a_{m+1}, \dots, 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

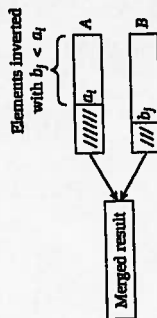


Figure 5.5 Merging two sorted lists while also counting the number of inversions between them.

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 list 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  $\lfloor n/2 \rfloor$  elements
       $B$  contains the remaining  $\lfloor n/2 \rfloor$  elements
     $(C_A, A) = \text{Sort-and-Count}(A)$ 
     $(C_B, B) = \text{Sort-and-Count}(B)$ 
     $(C, I) = \text{Merge-and-Count}(A, B)$ 
  Endif
  Return  $I = I_A + I_B + I$ , 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 $P = \{p_1, \dots, p_n\}$, 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 $\lceil n/2 \rceil$ 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

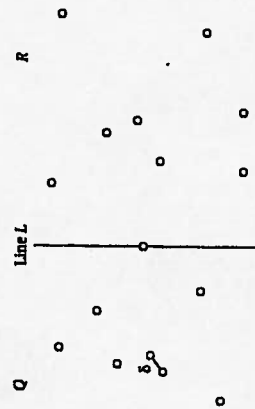


Figure 5.6 The first level of recursion: The point set P is divided evenly into Q and R by the line L , and the closest pair is found on each side recursively.

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)
   $(q_0^x, r_0^x) = \text{Closest-Pair-Rec}(P_x, P_x)$ 
   $(q_0^y, r_0^y) = \text{Closest-Pair-Rec}(P_y, P_y)$ 
  If  $|P| \leq 3$  then
    find closest pair by measuring all pairwise distances
  Endif

  Construct  $Q_x, Q_y, R_x, R_y$  ( $O(n)$  time)
   $(q_0^x, q_1^x) = \text{Closest-Pair-Rec}(Q_x, Q_x)$ 
   $(q_0^y, r_1^y) = \text{Closest-Pair-Rec}(R_x, R_y)$ 

   $\delta = \min(d(q_0^x, q_1^x), d(q_0^y, r_1^y))$ 
   $x^* = \text{maximum } x\text{-coordinate of a point in set } Q$ 
   $L = \{(x, y) : x = x^*\}$ 
   $S = \text{points in } P \text{ within distance } \delta \text{ of } L$ 

  Construct  $S_y$  ( $O(n)$  time)
  For each point  $s \in S_y$ , compute distances 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  $d(s, s') < \delta$ , then
    Return  $(s, s')$ 
  Else if  $d(q_0^x, q_1^x) < d(q_0^y, r_1^y)$  then
    Return  $(q_0^x, q_1^x)$ 

```

```

Else
  Return  $(q_0^y, r_1^y)$ 
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