

In Section 2.3.1, we saw how merge sort serves as an example of the divide-and-conquer paradigm. Recall that in divide-and-conquer, we solve a problem recursively, applying three steps at each level of the recursion:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

When the subproblems are large enough to solve recursively, we call that the *recursive case*. Once the subproblems become small enough that we no longer recurse, we say that the recursion “bottoms out” and that we have gotten down to the *base case*. Sometimes, in addition to subproblems that are smaller instances of the same problem, we have to solve subproblems that are not quite the same as the original problem. We consider solving such subproblems as part of the combine step.

In this chapter, we shall see more algorithms based on divide-and-conquer. The first one solves the maximum-subarray problem: it takes as input an array of numbers, and it determines the contiguous subarray whose values have the greatest sum. Then we shall see two divide-and-conquer algorithms for multiplying $n \times n$ matrices. One runs in $\Theta(n^3)$ time, which is no better than the straightforward method of multiplying square matrices. But the other, Strassen’s algorithm, runs in $O(n^{2.81})$ time, which beats the straightforward method asymptotically.

Recurrences

Recurrences go hand in hand with the divide-and-conquer paradigm, because they give us a natural way to characterize the running times of divide-and-conquer algorithms. A *recurrence* is an equation or inequality that describes a function in terms

of its value on smaller inputs. For example, in Section 2.3.2 we described the worst-case running time $T(n)$ of the MERGE-SORT procedure by the recurrence

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

whose solution we claimed to be $T(n) = \Theta(n \lg n)$.

Recurrences can take many forms. For example, a recursive algorithm might divide subproblems into unequal sizes, such as a 2/3-to-1/3 split. If the divide and combine steps take linear time, such an algorithm would give rise to the recurrence $T(n) = T(2n/3) + T(n/3) + \Theta(n)$.

Subproblems are not necessarily constrained to being a constant fraction of the original problem size. For example, a recursive version of linear search (see Exercise 2.1-3) would create just one subproblem containing only one element fewer than the original problem. Each recursive call would take constant time plus the time for the recursive calls it makes, yielding the recurrence $T(n) = T(n-1) + \Theta(1)$.

This chapter offers three methods for solving recurrences—that is, for obtaining asymptotic “ Θ ” or “ O ” bounds on the solution:

- In the ***substitution method***, we guess a bound and then use mathematical induction to prove our guess correct.
- The ***recursion-tree method*** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- The ***master method*** provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n), \quad (4.2)$$

where $a \geq 1$, $b > 1$, and $f(n)$ is a given function. Such recurrences arise frequently. A recurrence of the form in equation (4.2) characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is $1/b$ the size of the original problem, and in which the divide and combine steps together take $f(n)$ time.

To use the master method, you will need to memorize three cases, but once you do that, you will easily be able to determine asymptotic bounds for many simple recurrences. We will use the master method to determine the running times of the divide-and-conquer algorithms for the maximum-subarray problem and for matrix multiplication, as well as for other algorithms based on divide-and-conquer elsewhere in this book.

Occasionally, we shall see recurrences that are not equalities but rather inequalities, such as $T(n) \leq 2T(n/2) + \Theta(n)$. Because such a recurrence states only an upper bound on $T(n)$, we will couch its solution using O -notation rather than Θ -notation. Similarly, if the inequality were reversed to $T(n) \geq 2T(n/2) + \Theta(n)$, then because the recurrence gives only a lower bound on $T(n)$, we would use Ω -notation in its solution.

Technicalities in recurrences

In practice, we neglect certain technical details when we state and solve recurrences. For example, if we call MERGE-SORT on n elements when n is odd, we end up with subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Neither size is actually $n/2$, because $n/2$ is not an integer when n is odd. Technically, the recurrence describing the worst-case running time of MERGE-SORT is really

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1. \end{cases} \quad (4.3)$$

Boundary conditions represent another class of details that we typically ignore. Since the running time of an algorithm on a constant-sized input is a constant, the recurrences that arise from the running times of algorithms generally have $T(n) = \Theta(1)$ for sufficiently small n . Consequently, for convenience, we shall generally omit statements of the boundary conditions of recurrences and assume that $T(n)$ is constant for small n . For example, we normally state recurrence (4.1) as

$$T(n) = 2T(n/2) + \Theta(n), \quad (4.4)$$

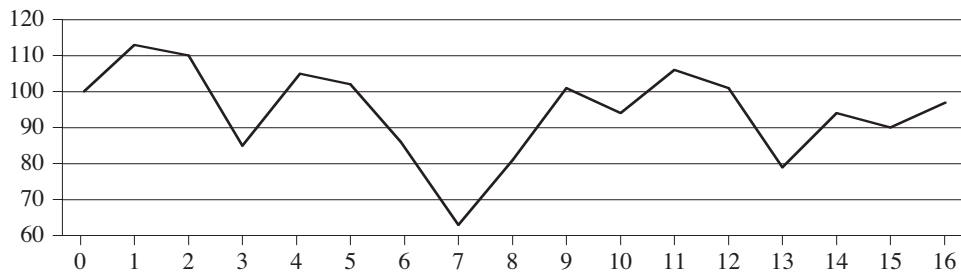
without explicitly giving values for small n . The reason is that although changing the value of $T(1)$ changes the exact solution to the recurrence, the solution typically doesn't change by more than a constant factor, and so the order of growth is unchanged.

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions. We forge ahead without these details and later determine whether or not they matter. They usually do not, but you should know when they do. Experience helps, and so do some theorems stating that these details do not affect the asymptotic bounds of many recurrences characterizing divide-and-conquer algorithms (see Theorem 4.1). In this chapter, however, we shall address some of these details and illustrate the fine points of recurrence solution methods.

4.1 The maximum-subarray problem

Suppose that you been offered the opportunity to invest in the Volatile Chemical Corporation. Like the chemicals the company produces, the stock price of the Volatile Chemical Corporation is rather volatile. You are allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day. To compensate for this restriction, you are allowed to learn what the price of the stock will be in the future. Your goal is to maximize your profit. Figure 4.1 shows the price of the stock over a 17-day period. You may buy the stock at any one time, starting after day 0, when the price is \$100 per share. Of course, you would want to “buy low, sell high”—buy at the lowest possible price and later on sell at the highest possible price—to maximize your profit. Unfortunately, you might not be able to buy at the lowest price and then sell at the highest price within a given period. In Figure 4.1, the lowest price occurs after day 7, which occurs after the highest price, after day 1.

You might think that you can always maximize profit by either buying at the lowest price or selling at the highest price. For example, in Figure 4.1, we would maximize profit by buying at the lowest price, after day 7. If this strategy always worked, then it would be easy to determine how to maximize profit: find the highest and lowest prices, and then work left from the highest price to find the lowest prior price, work right from the lowest price to find the highest later price, and take the pair with the greater difference. Figure 4.2 shows a simple counterexample,



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

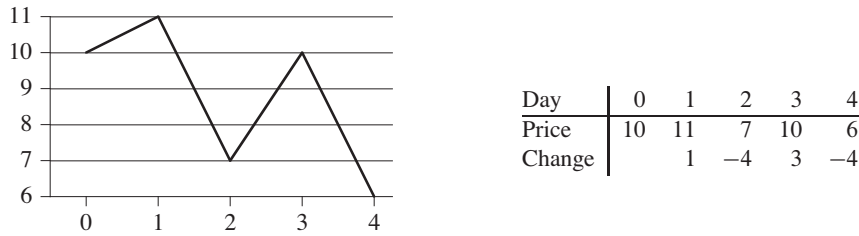


Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

(nC2) can be shown in two ways. First: Select 2 at a time from n possible elements. One of them will definitely be less than other and order of elements wouldn't matter. Second, begin from the left most, there are $(n-1)$ possible ways to select this number's pair. Select second-most left now $(n-2)$ possibilities for second selection, and so forth.

demonstrating that the maximum profit sometimes comes neither by buying at the lowest price nor by selling at the highest price.

A brute-force solution

We can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date. A period of n days has $\binom{n}{2}$ such pairs of dates. Since $\binom{n}{2}$ is $\Theta(n^2)$, and the best we can hope for is to evaluate each pair of dates in constant time, this approach would take $\Omega(n^2)$ time. Can we do better?

A transformation

In order to design an algorithm with an $o(n^2)$ running time, we will look at the input in a slightly different way. We want to find a sequence of days over which the net change from the first day to the last is maximum. Instead of looking at the daily prices, let us instead consider the daily change in price, where the change on day i is the difference between the prices after day $i - 1$ and after day i . The table in Figure 4.1 shows these daily changes in the bottom row. If we treat this row as an array A , shown in Figure 4.3, we now want to find the nonempty, contiguous subarray of A whose values have the largest sum. We call this contiguous subarray the **maximum subarray**. For example, in the array of Figure 4.3, the maximum subarray of $A[1 \dots 16]$ is $A[8 \dots 11]$, with the sum 43. Thus, you would want to buy the stock just before day 8 (that is, after day 7) and sell it after day 11, earning a profit of \$43 per share.

At first glance, this transformation does not help. We still need to check $\binom{n-1}{2} = \Theta(n^2)$ subarrays for a period of n days. Exercise 4.1-2 asks you to show

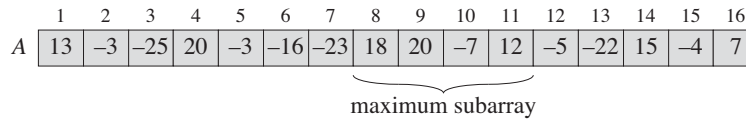


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8 \dots 11]$, with sum 43, has the greatest sum of any contiguous subarray of array A .

that although computing the cost of one subarray might take time proportional to the length of the subarray, when computing all $\Theta(n^2)$ subarray sums, we can organize the computation so that each subarray sum takes $O(1)$ time, given the values of previously computed subarray sums, so that the brute-force solution takes $\Theta(n^2)$ time.

So let us seek a more efficient solution to the maximum-subarray problem. When doing so, we will usually speak of “a” maximum subarray rather than “the” maximum subarray, since there could be more than one subarray that achieves the maximum sum.

The maximum-subarray problem is interesting only when the array contains some negative numbers. If all the array entries were nonnegative, then the maximum-subarray problem would present no challenge, since the entire array would give the greatest sum.

A solution using divide-and-conquer

Let’s think about how we might solve the maximum-subarray problem using the divide-and-conquer technique. Suppose we want to find a maximum subarray of the subarray $A[\text{low} \dots \text{high}]$. Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say mid , of the subarray, and consider the subarrays $A[\text{low} \dots \text{mid}]$ and $A[\text{mid} + 1 \dots \text{high}]$. As Figure 4.4(a) shows, any contiguous subarray $A[i \dots j]$ of $A[\text{low} \dots \text{high}]$ must lie in exactly one of the following places:

- entirely in the subarray $A[\text{low} \dots \text{mid}]$, so that $\text{low} \leq i \leq j \leq \text{mid}$,
- entirely in the subarray $A[\text{mid} + 1 \dots \text{high}]$, so that $\text{mid} < i \leq j \leq \text{high}$, or
- crossing the midpoint, so that $\text{low} \leq i \leq \text{mid} < j \leq \text{high}$.

Therefore, a maximum subarray of $A[\text{low} \dots \text{high}]$ must lie in exactly one of these places. In fact, a maximum subarray of $A[\text{low} \dots \text{high}]$ must have the greatest sum over all subarrays entirely in $A[\text{low} \dots \text{mid}]$, entirely in $A[\text{mid} + 1 \dots \text{high}]$, or crossing the midpoint. We can find maximum subarrays of $A[\text{low} \dots \text{mid}]$ and $A[\text{mid} + 1 \dots \text{high}]$ recursively, because these two subproblems are smaller instances of the problem of finding a maximum subarray. Thus, all that is left to do is find a

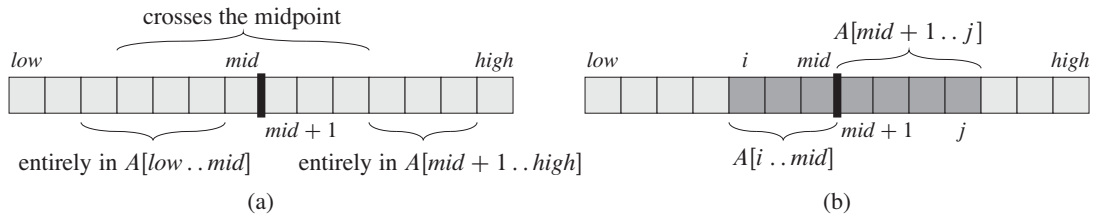


Figure 4.4 (a) Possible locations of subarrays of $A[low..high]$: entirely in $A[low..mid]$, entirely in $A[mid+1..high]$, or crossing the midpoint mid . (b) Any subarray of $A[low..high]$ crossing the midpoint comprises two subarrays $A[i..mid]$ and $A[mid+1..j]$, where $low \leq i \leq mid$ and $mid < j \leq high$.

maximum subarray that crosses the midpoint, and take a subarray with the largest sum of the three.

We can easily find a maximum subarray crossing the midpoint in time linear in the size of the subarray $A[low..high]$. This problem is *not* a smaller instance of our original problem, because it has the added restriction that the subarray it chooses must cross the midpoint. As Figure 4.4(b) shows, any subarray crossing the midpoint is itself made of two subarrays $A[i..mid]$ and $A[mid+1..j]$, where $low \leq i \leq mid$ and $mid < j \leq high$. Therefore, we just need to find maximum subarrays of the form $A[i..mid]$ and $A[mid+1..j]$ and then combine them. The procedure FIND-MAX-CROSSING-SUBARRAY takes as input the array A and the indices low , mid , and $high$, and it returns a tuple containing the indices demarcating a maximum subarray that crosses the midpoint, along with the sum of the values in a maximum subarray.

FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$)

```

1  left-sum =  $-\infty$ 
2  sum = 0
3  for i = mid downto low
4      sum = sum +  $A[i]$ 
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  right-sum =  $-\infty$ 
9  sum = 0
10 for j = mid + 1 to high
11     sum = sum +  $A[j]$ 
12     if sum > right-sum
13         right-sum = sum
14         max-right = j
15 return (max-left, max-right, left-sum + right-sum)
```

"max-left" and "max-right" are not initialized.

"max-left" moves from mid to low.

"max-left" and "max-right" only update if the newly added element increases the sum.

"max-right" moves from mid+1 to high.

This procedure works as follows. Lines 1–7 find a maximum subarray of the left half, $A[\text{low} \dots \text{mid}]$. Since this subarray must contain $A[\text{mid}]$, the **for** loop of lines 3–7 starts the index i at mid and works down to low , so that every subarray it considers is of the form $A[i \dots \text{mid}]$. Lines 1–2 initialize the variables left-sum , which holds the greatest sum found so far, and sum , holding the sum of the entries in $A[i \dots \text{mid}]$. Whenever we find, in line 5, a subarray $A[i \dots \text{mid}]$ with a sum of values greater than left-sum , we update left-sum to this subarray’s sum in line 6, and in line 7 we update the variable max-left to record this index i . Lines 8–14 work analogously for the right half, $A[\text{mid} + 1 \dots \text{high}]$. Here, the **for** loop of lines 10–14 starts the index j at $\text{mid} + 1$ and works up to high , so that every subarray it considers is of the form $A[\text{mid} + 1 \dots j]$. Finally, line 15 returns the indices max-left and max-right that demarcate a maximum subarray crossing the midpoint, along with the sum $\text{left-sum} + \text{right-sum}$ of the values in the subarray $A[\text{max-left} \dots \text{max-right}]$.

If the subarray $A[\text{low} \dots \text{high}]$ contains n entries (so that $n = \text{high} - \text{low} + 1$), we claim that the call $\text{FIND-MAX-CROSSING-SUBARRAY}(A, \text{low}, \text{mid}, \text{high})$ takes $\Theta(n)$ time. Since each iteration of each of the two **for** loops takes $\Theta(1)$ time, we just need to count up how many iterations there are altogether. The **for** loop of lines 3–7 makes $\text{mid} - \text{low} + 1$ iterations, and the **for** loop of lines 10–14 makes $\text{high} - \text{mid}$ iterations, and so the total number of iterations is

$$\begin{aligned} (\text{mid} - \text{low} + 1) + (\text{high} - \text{mid}) &= \text{high} - \text{low} + 1 \\ &= n. \end{aligned}$$

With a linear-time $\text{FIND-MAX-CROSSING-SUBARRAY}$ procedure in hand, we can write pseudocode for a divide-and-conquer algorithm to solve the maximum-subarray problem:

$\text{FIND-MAXIMUM-SUBARRAY}(A, \text{low}, \text{high})$

```

1  if  $\text{high} == \text{low}$ 
2      return  $(\text{low}, \text{high}, A[\text{low}])$            // base case: only one element
3  else  $\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor$ 
4       $(\text{left-low}, \text{left-high}, \text{left-sum}) =$ 
           $\text{FIND-MAXIMUM-SUBARRAY}(A, \text{low}, \text{mid})$ 
5       $(\text{right-low}, \text{right-high}, \text{right-sum}) =$ 
           $\text{FIND-MAXIMUM-SUBARRAY}(A, \text{mid} + 1, \text{high})$ 
6       $(\text{cross-low}, \text{cross-high}, \text{cross-sum}) =$ 
           $\text{FIND-MAX-CROSSING-SUBARRAY}(A, \text{low}, \text{mid}, \text{high})$ 
7      if  $\text{left-sum} \geq \text{right-sum}$  and  $\text{left-sum} \geq \text{cross-sum}$ 
8          return  $(\text{left-low}, \text{left-high}, \text{left-sum})$ 
9      elseif  $\text{right-sum} \geq \text{left-sum}$  and  $\text{right-sum} \geq \text{cross-sum}$ 
10         return  $(\text{right-low}, \text{right-high}, \text{right-sum})$ 
11     else return  $(\text{cross-low}, \text{cross-high}, \text{cross-sum})$ 
```


The initial call $\text{FIND-MAXIMUM-SUBARRAY}(A, 1, A.length)$ will find a maximum subarray of $A[1..n]$.

Similar to $\text{FIND-MAX-CROSSING-SUBARRAY}$, the recursive procedure $\text{FIND-MAXIMUM-SUBARRAY}$ returns a tuple containing the indices that demarcate a maximum subarray, along with the sum of the values in a maximum subarray. Line 1 tests for the base case, where the subarray has just one element. A subarray with just one element has only one subarray—itsself—and so line 2 returns a tuple with the starting and ending indices of just the one element, along with its value. Lines 3–11 handle the recursive case. Line 3 does the divide part, computing the index *mid* of the midpoint. Let's refer to the subarray $A[low..mid]$ as the **left subarray** and to $A[mid + 1..high]$ as the **right subarray**. Because we know that the subarray $A[low..high]$ contains at least two elements, each of the left and right subarrays must have at least one element. Lines 4 and 5 conquer by recursively finding maximum subarrays within the left and right subarrays, respectively. Lines 6–11 form the combine part. Line 6 finds a maximum subarray that crosses the midpoint. (Recall that because line 6 solves a subproblem that is not a smaller instance of the original problem, we consider it to be in the combine part.) Line 7 tests whether the left subarray contains a subarray with the maximum sum, and line 8 returns that maximum subarray. Otherwise, line 9 tests whether the right subarray contains a subarray with the maximum sum, and line 10 returns that maximum subarray. If neither the left nor right subarrays contain a subarray achieving the maximum sum, then a maximum subarray must cross the midpoint, and line 11 returns it.

Analyzing the divide-and-conquer algorithm

Next we set up a recurrence that describes the running time of the recursive $\text{FIND-MAXIMUM-SUBARRAY}$ procedure. As we did when we analyzed merge sort in Section 2.3.2, we make the simplifying assumption that the original problem size is a power of 2, so that all subproblem sizes are integers. We denote by $T(n)$ the running time of $\text{FIND-MAXIMUM-SUBARRAY}$ on a subarray of n elements. For starters, line 1 takes constant time. The base case, when $n = 1$, is easy: line 2 takes constant time, and so

$$T(1) = \Theta(1). \quad (4.5)$$

The recursive case occurs when $n > 1$. Lines 1 and 3 take constant time. Each of the subproblems solved in lines 4 and 5 is on a subarray of $n/2$ elements (our assumption that the original problem size is a power of 2 ensures that $n/2$ is an integer), and so we spend $T(n/2)$ time solving each of them. Because we have to solve two subproblems—for the left subarray and for the right subarray—the contribution to the running time from lines 4 and 5 comes to $2T(n/2)$. As we have

already seen, the call to `FIND-MAX-CROSSING-SUBARRAY` in line 6 takes $\Theta(n)$ time. Lines 7–11 take only $\Theta(1)$ time. For the recursive case, therefore, we have

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n). \end{aligned} \quad (4.6)$$

Combining equations (4.5) and (4.6) gives us a recurrence for the running time $T(n)$ of `FIND-MAXIMUM-SUBARRAY`:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \quad (4.7)$$

This recurrence is the same as recurrence (4.1) for merge sort. As we shall see from the master method in Section 4.5, this recurrence has the solution $T(n) = \Theta(n \lg n)$. You might also revisit the recursion tree in Figure 2.5 to understand why the solution should be $T(n) = \Theta(n \lg n)$.

Thus, we see that the divide-and-conquer method yields an algorithm that is asymptotically faster than the brute-force method. With merge sort and now the maximum-subarray problem, we begin to get an idea of how powerful the divide-and-conquer method can be. Sometimes it will yield the asymptotically fastest algorithm for a problem, and other times we can do even better. As Exercise 4.1-5 shows, there is in fact a linear-time algorithm for the maximum-subarray problem, and it does not use divide-and-conquer.

Exercises

4.1-1

What does `FIND-MAXIMUM-SUBARRAY` return when all elements of A are negative?

4.1-2

Write pseudocode for the brute-force method of solving the maximum-subarray problem. Your procedure should run in $\Theta(n^2)$ time.

4.1-3

Implement both the brute-force and recursive algorithms for the maximum-subarray problem on your own computer. What problem size n_0 gives the crossover point at which the recursive algorithm beats the brute-force algorithm? Then, change the base case of the recursive algorithm to use the brute-force algorithm whenever the problem size is less than n_0 . Does that change the crossover point?

4.1-4

Suppose we change the definition of the maximum-subarray problem to allow the result to be an empty subarray, where the sum of the values of an empty subar-

ray is 0. How would you change any of the algorithms that do not allow empty subarrays to permit an empty subarray to be the result?

4.1-5

Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of $A[1 \dots j]$, extend the answer to find a maximum subarray ending at index $j + 1$ by using the following observation: a maximum subarray of $A[1 \dots j + 1]$ is either a maximum subarray of $A[1 \dots j]$ or a subarray $A[i \dots j + 1]$, for some $1 \leq i \leq j + 1$. Determine a maximum subarray of the form $A[i \dots j + 1]$ in constant time based on knowing a maximum subarray ending at index j .

4.2 Strassen's algorithm for matrix multiplication

If you have seen matrices before, then you probably know how to multiply them. (Otherwise, you should read Section D.1 in Appendix D.) If $A = (a_{ij})$ and $B = (b_{ij})$ are square $n \times n$ matrices, then in the product $C = A \cdot B$, we define the entry c_{ij} , for $i, j = 1, 2, \dots, n$, by

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj} . \quad (4.8)$$

We must compute n^2 matrix entries, and each is the sum of n values. The following procedure takes $n \times n$ matrices A and B and multiplies them, returning their $n \times n$ product C . We assume that each matrix has an attribute *rows*, giving the number of rows in the matrix.

SQUARE-MATRIX-MULTIPLY(A, B)

```

1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8  return  $C$ 
```

The SQUARE-MATRIX-MULTIPLY procedure works as follows. The **for** loop of lines 3–7 computes the entries of each row i , and within a given row i , the