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 $A[1 .. j]$, extend the answer to find a maximum subarray ending at index $j + 1$ by using the following observation: a maximum subarray of $A[1 .. j + 1]$ is either a maximum subarray of $A[1 .. j]$ or a subarray $A[i .. j + 1]$, for some $1 \le i \le j + 1$. Determine a maximum subarray of the form $A[i .. j + 1]$ 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 $i, j = 1, 2, \ldots, n$, by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \; . \tag{4.8}$$

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 × 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 · 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

for loop of lines 4–7 computes each of the entries c_{ij}, for each column j. Line 5 initializes c_{ij} to 0 as we start computing the sum given in equation (4.8), and each iteration of the **for** loop of lines 6–7 adds in one more term of equation (4.8).

Because each of the triply-nested **for** loops runs exactly n iterations, and each execution of line 7 takes constant time, the SQUARE-MATRIX-MULTIPLY procedure takes $\Theta(n^3)$ time.

You might at first think that any matrix multiplication algorithm must take $\Omega(n^3)$ time, since the natural definition of matrix multiplication requires that many multiplications. You would be incorrect, however: we have a way to multiply matrices in $o(n^3)$ time. In this section, we shall see Strassen's remarkable recursive algorithm for multiplying $n \times n$ matrices. It runs in $\Theta(n^{\lg 7})$ time, which we shall show in Section 4.5. Since $\lg 7$ lies between 2.80 and 2.81, Strassen's algorithm runs in $O(n^{2.81})$ time, which is asymptotically better than the simple SQUARE-MATRIX-MULTIPLY procedure.

A simple divide-and-conquer algorithm

To keep things simple, when we use a divide-and-conquer algorithm to compute the matrix product $C = A \cdot B$, we assume that n is an exact power of 2 in each of the $n \times n$ matrices. We make this assumption because in each divide step, we will divide $n \times n$ matrices into four $n/2 \times n/2$ matrices, and by assuming that n is an exact power of 2, we are guaranteed that as long as $n \geq 2$, the dimension $n/2$ is an integer.

Suppose that we partition each of A, B, and C into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.11}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.12}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.13}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.14}$$

Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products. We can use these equations to create a straightforward, recursive, divide-and-conquer algorithm:

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

```
1   n = A.rows
2   let C be a new n × n matrix
3   if n == 1
4       c₁₁ = a₁₁ · b₁₁
5   else partition A, B, and C as in equations (4.9)
```
6   C_{11} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11})
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})
7   C_{12} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12})
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})
8   C_{21} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11})
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})
9   C_{22} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12})
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})
10   **return** C

This pseudocode glosses over one subtle but important implementation detail. How do we partition the matrices in line 5? If we were to create 12 new $n/2 \times n/2$ matrices, we would spend $\Theta(n^2)$ time copying entries. In fact, we can partition the matrices without copying entries. The trick is to use index calculations. We identify a submatrix by a range of row indices and a range of column indices of the original matrix. We end up representing a submatrix a little differently from how we represent the original matrix, which is the subtlety we are glossing over. The advantage is that, since we can specify submatrices by index calculations, executing line 5 takes only $\Theta(1)$ time (although we shall see that it makes no difference asymptotically to the overall running time whether we copy or partition in place).

Now, we derive a recurrence to characterize the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE. Let $T(n)$ be the time to multiply two $n \times n$ matrices using this procedure. In the base case, when $n = 1$, we perform just the one scalar multiplication in line 4, and so

$$T(1) = \Theta(1) . \tag{4.15}$$

The recursive case occurs when $n > 1$. As discussed, partitioning the matrices in line 5 takes $\Theta(1)$ time, using index calculations. In lines 6–9, we recursively call SQUARE-MATRIX-MULTIPLY-RECURSIVE a total of eight times. Because each recursive call multiplies two $n/2 \times n/2$ matrices, thereby contributing $T(n/2)$ to the overall running time, the time taken by all eight recursive calls is $8T(n/2)$. We also must account for the four matrix additions in lines 6–9. Each of these matrices contains $n^2/4$ entries, and so each of the four matrix additions takes $\Theta(n^2)$ time. Since the number of matrix additions is a constant, the total time spent adding ma-

trices in lines 6–9 is $\Theta(n^2)$. (Again, we use index calculations to place the results of the matrix additions into the correct positions of matrix C, with an overhead of $\Theta(1)$ time per entry.) The total time for the recursive case, therefore, is the sum of the partitioning time, the time for all the recursive calls, and the time to add the matrices resulting from the recursive calls:

$$
\begin{aligned}
T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\
&= 8T(n/2) + \Theta(n^2) .
\end{aligned}
\tag{4.16}
$$

Notice that if we implemented partitioning by copying matrices, which would cost $\Theta(n^2)$ time, the recurrence would not change, and hence the overall running time would increase by only a constant factor.

Combining equations (4.15) and (4.16) gives us the recurrence for the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE:

$$
T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 . \end{cases}
\tag{4.17}
$$

As we shall see from the master method in Section 4.5, recurrence (4.17) has the solution $T(n) = \Theta(n^3)$. Thus, this simple divide-and-conquer approach is no faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure.

Before we continue on to examining Strassen's algorithm, let us review where the components of equation (4.16) came from. Partitioning each $n \times n$ matrix by index calculation takes $\Theta(1)$ time, but we have two matrices to partition. Although you could say that partitioning the two matrices takes $\Theta(2)$ time, the constant of 2 is subsumed by the Θ-notation. Adding two matrices, each with, say, k entries, takes $\Theta(k)$ time. Since the matrices we add each have $n^2/4$ entries, you could say that adding each pair takes $\Theta(n^2/4)$ time. Again, however, the Θ-notation subsumes the constant factor of $1/4$, and we say that adding two $n^2/4 \times n^2/4$ matrices takes $\Theta(n^2)$ time. We have four such matrix additions, and once again, instead of saying that they take $\Theta(4n^2)$ time, we say that they take $\Theta(n^2)$ time. (Of course, you might observe that we could say that the four matrix additions take $\Theta(4n^2/4)$ time, and that $4n^2/4 = n^2$, but the point here is that Θ-notation subsumes constant factors, whatever they are.) Thus, we end up with two terms of $\Theta(n^2)$, which we can combine into one.

When we account for the eight recursive calls, however, we cannot just subsume the constant factor of 8. In other words, we must say that together they take $8T(n/2)$ time, rather than just $T(n/2)$ time. You can get a feel for why by looking back at the recursion tree in Figure 2.5, for recurrence (2.1) (which is identical to recurrence (4.7)), with the recursive case $T(n) = 2T(n/2) + \Theta(n)$. The factor of 2 determined how many children each tree node had, which in turn determined how many terms contributed to the sum at each level of the tree. If we were to ignore

the factor of 8 in equation (4.16) or the factor of 2 in recurrence (4.1), the recursion tree would just be linear, rather than "bushy," and each level would contribute only one term to the sum.

Bear in mind, therefore, that although asymptotic notation subsumes constant multiplicative factors, recursive notation such as $T(n/2)$ does not.

Strassen's method

The key to Strassen's method is to make the recursion tree slightly less bushy. That is, instead of performing eight recursive multiplications of $n/2 \times n/2$ matrices, it performs only seven. The cost of eliminating one matrix multiplication will be several new additions of $n/2 \times n/2$ matrices, but still only a constant number of additions. As before, the constant number of matrix additions will be subsumed by Θ-notation when we set up the recurrence equation to characterize the running time.

Strassen's method is not at all obvious. (This might be the biggest understatement in this book.) It has four steps:

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.

2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.

3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$.

4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

We shall see the details of steps 2–4 in a moment, but we already have enough information to set up a recurrence for the running time of Strassen's method. Let us assume that once the matrix size n gets down to 1, we perform a simple scalar multiplication, just as in line 4 of SQUARE-MATRIX-MULTIPLY-RECURSIVE. When $n > 1$, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires us to perform seven multiplications of $n/2 \times n/2$ matrices. Hence, we obtain the following recurrence for the running time $T(n)$ of Strassen's algorithm:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases} \tag{4.18}$$

We have traded off one matrix multiplication for a constant number of matrix additions. Once we understand recurrences and their solutions, we shall see that this tradeoff actually leads to a lower asymptotic running time. By the master method in Section 4.5, recurrence (4.18) has the solution $T(n) = \Theta(n^{\lg 7})$.

We now proceed to describe the details. In step 2, we create the following 10 matrices:

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} , \\
S_2 &= A_{11} + A_{12} , \\
S_3 &= A_{21} + A_{22} , \\
S_4 &= B_{21} - B_{11} , \\
S_5 &= A_{11} + A_{22} , \\
S_6 &= B_{11} + B_{22} , \\
S_7 &= A_{12} - A_{22} , \\
S_8 &= B_{21} + B_{22} , \\
S_9 &= A_{11} - A_{21} , \\
S_{10} &= B_{11} + B_{12} .
\end{aligned}
$$

Since we must add or subtract $n/2 \times n/2$ matrices 10 times, this step does indeed take $\Theta(n^2)$ time.

In step 3, we recursively multiply $n/2 \times n/2$ matrices seven times to compute the following $n/2 \times n/2$ matrices, each of which is the sum or difference of products of A and B submatrices:

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} , \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} , \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} , \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} , \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} , \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} , \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} .
\end{aligned}
$$

Note that the only multiplications we need to perform are those in the middle column of the above equations. The right-hand column just shows what these products equal in terms of the original submatrices created in step 1.

Step 4 adds and subtracts the P_i matrices created in step 3 to construct the four $n/2 \times n/2$ submatrices of the product C. We start with

$$
C_{11} = P_5 + P_4 - P_2 + P_6 .
$$

Expanding out the right-hand side, with the expansion of each P_i on its own line and vertically aligning terms that cancel out, we see that C_{11} equals

$$A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}$$
$$- A_{22} \cdot B_{11} \qquad\qquad + A_{22} \cdot B_{21}$$
$$- A_{11} \cdot B_{22} \qquad\qquad\qquad\qquad - A_{12} \cdot B_{22}$$
$$- A_{22} \cdot B_{22} - A_{22} \cdot B_{21} + A_{12} \cdot B_{22} + A_{12} \cdot B_{21}$$
$$\overline{}$$
$$A_{11} \cdot B_{11} \qquad\qquad\qquad\qquad\qquad\qquad + A_{12} \cdot B_{21} \ ,$$

which corresponds to equation (4.11).

Similarly, we set

$$C_{12} = P_1 + P_2 \ ,$$

and so C_{12} equals

$$A_{11} \cdot B_{12} - A_{11} \cdot B_{22}$$
$$+ A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$$
$$\overline{}$$
$$A_{11} \cdot B_{12} \qquad\qquad + A_{12} \cdot B_{22} \ ,$$

corresponding to equation (4.12).

Setting

$$C_{21} = P_3 + P_4$$

makes C_{21} equal

$$A_{21} \cdot B_{11} + A_{22} \cdot B_{11}$$
$$- A_{22} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$\overline{}$$
$$A_{21} \cdot B_{11} \qquad\qquad + A_{22} \cdot B_{21} \ ,$$

corresponding to equation (4.13).

Finally, we set

$$C_{22} = P_5 + P_1 - P_3 - P_7 \ ,$$

so that C_{22} equals

$$A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}$$
$$- A_{11} \cdot B_{22} \qquad\qquad\qquad + A_{11} \cdot B_{12}$$
$$- A_{22} \cdot B_{11} \qquad\qquad\qquad - A_{21} \cdot B_{11}$$
$$- A_{11} \cdot B_{11} \qquad\qquad\qquad - A_{11} \cdot B_{12} + A_{21} \cdot B_{11} + A_{21} \cdot B_{12}$$
$$\overline{}$$
$$A_{22} \cdot B_{22} \qquad\qquad\qquad\qquad + A_{21} \cdot B_{12} \ ,$$

which corresponds to equation (4.14). Altogether, we add or subtract $n/2 \times n/2$ matrices eight times in step 4, and so this step indeed takes $\Theta(n^2)$ time.

Thus, we see that Strassen's algorithm, comprising steps 1–4, produces the correct matrix product and that recurrence (4.18) characterizes its running time. Since we shall see in Section 4.5 that this recurrence has the solution $T(n) = \Theta(n^{\lg 7})$, Strassen's method is asymptotically faster than the straightforward SQUARE-MATRIX-MULTIPLY procedure. The notes at the end of this chapter discuss some of the practical aspects of Strassen's algorithm.

Exercises

Note: Although Exercises 4.2-3, 4.2-4, and 4.2-5 are about variants on Strassen's algorithm, you should read Section 4.5 before trying to solve them.

4.2-1
Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}.$$

Show your work.

4.2-2
Write pseudocode for Strassen's algorithm.

4.2-3
How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\lg 7})$.

4.2-4
What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

4.2-5
V. Pan has discovered a way of multiplying 68×68 matrices using 132,464 multiplications, a way of multiplying 70×70 matrices using 143,640 multiplications, and a way of multiplying 72×72 matrices using 155,424 multiplications. Which method yields the best asymptotic running time when used in a divide-and-conquer matrix-multiplication algorithm? How does it compare to Strassen's algorithm?