4.4-3

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 4T(n/2 + 2) + n$. Use the substitution method to verify your answer.

4.4-4

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 2T(n-1) + 1$. Use the substitution method to verify your answer.

4.4-5

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n-1) + T(n/2) + n$. Use the substitution method to verify your answer.

4.4-6

Argue that the solution to the recurrence $T(n) = T(n/3) + T(2n/3) + cn$, where c is a constant, is $\Omega(n \lg n)$ by appealing to a recursion tree.

4.4-7

Draw the recursion tree for $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, where c is a constant, and provide a tight asymptotic bound on its solution. Verify your bound by the substitution method.

4.4-8

Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(n-a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants.

4.4-9

Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant.

4.5 The master method for solving recurrences

The master method provides a "cookbook" method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n) , \tag{4.20}$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. To use the master method, you will need to memorize three cases, but then you will be able to solve many recurrences quite easily, often without pencil and paper.

Problems

4-1 Recurrence examples

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \le 2$. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 2T(n/2) + n^4$.

b. $T(n) = T(7n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n-2) + n^2$.

4-2 Parameter-passing costs

Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time $= \Theta(1)$.
2. An array is passed by copying. Time $= \Theta(N)$, where N is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time $= \Theta(q - p + 1)$ if the subarray $A[p \mathinner{..} q]$ is passed.

a. Consider the recursive binary search algorithm for finding a number in a sorted array (see Exercise 2.3-5). Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let N be the size of the original problem and n be the size of a subproblem.

b. Redo part (a) for the MERGE-SORT algorithm from Section 2.3.1.

b. Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

c. Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

4-6 Monge arrays

An $m \times n$ array A of real numbers is a ***Monge array*** if for all i, j, k, and l such that $1 \le i < k \le m$ and $1 \le j < l \le n$, we have

$$A[i,j] + A[k,l] \le A[i,l] + A[k,j] .$$

In other words, whenever we pick two rows and two columns of a Monge array and consider the four elements at the intersections of the rows and the columns, the sum of the upper-left and lower-right elements is less than or equal to the sum of the lower-left and upper-right elements. For example, the following array is Monge:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. Prove that an array is Monge if and only if for all $i = 1, 2, \ldots, m-1$ and $j = 1, 2, \ldots, n-1$, we have

$$A[i,j] + A[i+1, j+1] \le A[i, j+1] + A[i+1, j] .$$

(*Hint:* For the "if" part, use induction separately on rows and columns.)

b. The following array is not Monge. Change one element in order to make it Monge. (*Hint:* Use part (a).)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c. Let $f(i)$ be the index of the column containing the leftmost minimum element of row i. Prove that $f(1) \le f(2) \le \cdots \le f(m)$ for any $m \times n$ Monge array.

d. Here is a description of a divide-and-conquer algorithm that computes the leftmost minimum element in each row of an $m \times n$ Monge array A:

> Construct a submatrix A' of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row of A'. Then compute the leftmost minimum in the odd-numbered rows of A.

Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum of the even-numbered rows is known) in $O(m + n)$ time.

e. Write the recurrence describing the running time of the algorithm described in part (d). Show that its solution is $O(m + n \log m)$.

Chapter notes

Divide-and-conquer as a technique for designing algorithms dates back to at least 1962 in an article by Karatsuba and Ofman [194]. It might have been used well before then, however; according to Heideman, Johnson, and Burrus [163], C. F. Gauss devised the first fast Fourier transform algorithm in 1805, and Gauss's formulation breaks the problem into smaller subproblems whose solutions are combined.

The maximum-subarray problem in Section 4.1 is a minor variation on a problem studied by Bentley [43, Chapter 7].

Strassen's algorithm [325] caused much excitement when it was published in 1969. Before then, few imagined the possibility of an algorithm asymptotically faster than the basic SQUARE-MATRIX-MULTIPLY procedure. The asymptotic upper bound for matrix multiplication has been improved since then. The most asymptotically efficient algorithm for multiplying $n \times n$ matrices to date, due to Coppersmith and Winograd [78], has a running time of $O(n^{2.376})$. The best lower bound known is just the obvious $\Omega(n^2)$ bound (obvious because we must fill in n^2 elements of the product matrix).

From a practical point of view, Strassen's algorithm is often not the method of choice for matrix multiplication, for four reasons:

1. The constant factor hidden in the $\Theta(n^{\lg 7})$ running time of Strassen's algorithm is larger than the constant factor in the $\Theta(n^3)$-time SQUARE-MATRIX-MULTIPLY procedure.

2. When the matrices are sparse, methods tailored for sparse matrices are faster.