This file contains the exercises, hints, and solutions for Chapter 2 of the book "Introduction to the Design and Analysis of Algorithms," 3rd edition, by A. Levitin. The problems that might be challenging for at least some students are marked by ▷; those that might be difficult for a majority of students are marked by ►.

Exercises 2.1

1. For each of the following algorithms, indicate (i) a natural size metric for its inputs, (ii) its basic operation, and (iii) whether the basic operation count can be different for inputs of the same size:

   a. computing the sum of n numbers

   b. computing $n!$

   c. finding the largest element in a list of n numbers

   d. Euclid's algorithm

   e. sieve of Eratosthenes

   f. pen-and-pencil algorithm for multiplying two n-digit decimal integers

2. a. Consider the definition-based algorithm for adding two $n \times n$ matrices. What is its basic operation? How many times is it performed as a function of the matrix order n? As a function of the total number of elements in the input matrices?

   b. Answer the same questions for the definition-based algorithm for matrix multiplication.

3. Consider a variation of sequential search that scans a list to return the number of occurrences of a given search key in the list. Will its efficiency differ from the efficiency of classic sequential search?

4. a. *Glove selection* There are 22 gloves in a drawer: 5 pairs of red gloves, 4 pairs of yellow, and 2 pairs of green. You select the gloves in the dark and can check them only after a selection has been made. What is the smallest number of gloves you need to select to have at least one matching pair in the best case? in the worst case?

   b. *Missing socks* Imagine that after washing 5 distinct pairs of socks, you discover that two socks are missing. Of course, you would like to have the largest number of complete pairs remaining. Thus, you are left with 4 complete pairs in the best-case scenario and with 3 complete pairs in the worst case. Assuming that the probability of disappearance for each

of the 10 socks is the same, find the probability of the best-case scenario; the probability of the worst-case scenario; the number of pairs you should expect in the average case.

5. a.▷ Prove formula (2.1) for the number of bits in the binary representation of a positive integer.

   b.▷ Prove the alternative formula for the number of bits in the binary representation of a positive integer n:

   $$b = \lceil \log_2(n+1) \rceil.$$

   c. What would be the analogous formulas for the number of decimal digits?

   d. Explain why, within the accepted analysis framework, it does not matter whether we use binary or decimal digits in measuring n's size.

6. Suggest how any sorting algorithm can be augmented in a way to make the best-case count of its key comparisons equal to just $n-1$ (n is a list's size, of course). Do you think it would be a worthwhile addition to any sorting algorithm?

7. Gaussian elimination, the classic algorithm for solving systems of n linear equations in n unknowns, requires about $\frac{1}{3}n^3$ multiplications, which is the algorithm's basic operation.

   a. How much longer should you expect Gaussian elimination to work on a system of 1000 equations versus a system of 500 equations?

   b. You are considering buying a computer that is 1000 times faster than the one you currently have. By what factor will the faster computer increase the sizes of systems solvable in the same amount of time as on the old computer?

8. For each of the following functions, indicate how much the function's value will change if its argument is increased fourfold.

   a. $\log_2 n$ b. $\sqrt{n}$ c. n d. n^2 e. n^3 f. 2^n

9. Indicate whether the first function of each of the following pairs has a smaller, same, or larger order of growth (to within a constant multiple) than the second function.

a. $n(n+1)$ and $2000n^2$

b. $100n^2$ and $0.01n^3$

c. $\log_2 n$ and $\ln n$

d. $\log_2^2 n$ and $\log_2 n^2$

e. 2^{n-1} and 2^n

f. $(n-1)!$ and $n!$

10. *Invention of chess* a. According to a well-known legend, the game of chess was invented many centuries ago in northwestern India by a certain sage. When he took his invention to his king, the king liked the game so much that he offered the inventor any reward he wanted. The inventor asked for some grain to be obtained as follows: just a single grain of wheat was to be placed on the first square of the chess board, two on the second, four on the third, eight on the fourth, and so on, until all 64 squares had been filled. If it took just 1 second to count each grain, how long would it take to count all the grain due to him?

    b. How long would it take if instead of doubling the number of grains for each square of the chessboard, the inventor asked for adding two grains?

Hints to Exercises 2.1

1. The questions are indeed as straightforward as they appear, though some of them may have alternative answers. Also, keep in mind the caveat about measuring an integer's size.

2. a. The sum of two matrices is defined as the matrix whose elements are the sums of the corresponding elements of the matrices given.

   b. Matrix multiplication requires two operations: multiplication and addition. Which of the two would you consider basic and why?

3. Will the algorithm's efficiency vary on different inputs of the same size?

4. a. Gloves are not socks: they can be right-handed and left-handed.

   b. You have only two qualitatively different outcomes possible. Count the number of ways to get each of the two.

5. a. First, prove first that if a positive decimal integer n has b digits in its binary representation, then

   $$2^{b-1} \le n < 2^b.$$

   Then, take logarithms to base 2 of the terms in this inequality.

   b. The proof is similar to the proof of formula (2.1).

   c. The formula will be the same, with just one small adjustment to account for the different radix.

   d. How can we switch from one logarithm base to another?

6. Insert a verification of whether the problem is already solved.

7. A similar question was investigated in the section.

8. Use either the difference between or the ratio of $f(4n)$ and $f(n)$, whichever is more convenient for getting a compact answer. If it is possible, try to get an answer that does not depend on n.

9. If necessary, simplify the functions in question to single out terms defining their orders of growth to within a constant multiple. (We will discuss formal methods for answering such questions in the next section; however, these questions can be answered without knowledge of such methods.)

10. a. Use the formula $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

    b. Use the formula for the sum of the first n odd numbers or the formula for the sum of arithmetic progression.

Solutions to Exercises 2.1

1. The answers are as follows.

   a. (i) n; (ii) addition of two numbers; (iii) no

   b. (i) the magnitude of n, i.e., the number of bits in its binary representation; (ii) multiplication of two integers; (iii) no

   c. (i) n; (ii) comparison of two numbers; (iii) no (for the standard list scanning algorithm)

   d. (i) either the magnitude of the larger of two input numbers, or the magnitude of the smaller of two input numbers, or the sum of the magnitudes of two input numbers; (ii) modulo division; (iii) yes

   e. (i) the magnitude of n, i.e., the number of bits in its binary representation; (ii) elimination of a number from the list of remaining candidates to be prime; (iii) no

   f. (i) n; (ii) multiplication of two digits; (iii) no

2. a. Addition of two numbers. It's performed n^2 times (once for each of n^2 elements in the matrix being computed). .Since the total number of elements in two given matrices is $N = 2n^2$, the total number of additions can also be expressed as $n^2 = N/2$.

   b. Since on most computers multiplication takes longer than addition, multiplication is a better choice for being considered the basic operation of the standard algorithm for matrix multiplication. Each of n^2 elements of the product of two n-by-n matrices is computed as the scalar (dot) product of two vectors of size n, which requires n multiplications. The total number of multiplications is $n \cdot n^2 = n^3 = (N/2)^{3/2}$.

3. This algorithm will always make n key comparisons on every input of size n, whereas this number may vary between n and 1 for the classic version of sequential search.

4. a. The best-case number is, obviously, two. The worst-case number is twelve: one more than the number of gloves of one handedness.

   b. There are just two possible outcomes here: the two missing socks make a pair (the best case) and the two missing stocks do not make a pair (the worst case). The total number of different outcomes (the ways

to choose the missing socks) is $\binom{10}{2} = 45$. The number of best-case ones is 5; hence its probability is $\frac{5}{45} = \frac{1}{9}$. The number of worst-case ones is $45 - 5 = 40$; hence its probability is $\frac{40}{45} = \frac{8}{9}$. On average, you should expect $4 \cdot \frac{1}{9} + 3 \cdot \frac{8}{9} = \frac{28}{9} = 3\frac{1}{9}$ matching pairs.

5. a. The smallest positive integer that has b binary digits in its binary expansion is $1\underbrace{0...0}_{b-1}$, which is 2^{b-1}; the largest positive integer that has b binary digits in its binary expansion is $1\underbrace{1...1}_{b-1}$, which is $2^{b-1}+2^{b-2}+...+1 = 2^b - 1$. Thus,

$$2^{b-1} \le n < 2^b.$$

Hence

$$\log_2 2^{b-1} \le \log_2 n < \log_2 2^b$$

or

$$b - 1 \le \log_2 n < b.$$

These inequalities imply that $b - 1$ is the largest integer not exceeding $\log_2 n$. In other words, using the definition of the floor function, we conclude that

$$b - 1 = \lfloor \log_2 n \rfloor \text{ or } b = \lfloor \log_2 n \rfloor + 1.$$

b. If $n > 0$ has b bits in its binary representation, then, as shown in part a,

$$2^{b-1} \le n < 2^b.$$

Hence

$$2^{b-1} < n + 1 \le 2^b$$

and therefore

$$\log_2 2^{b-1} < \log_2(n+1) \le \log_2 2^b$$

or

$$b - 1 < \log_2(n+1) \le b.$$

These inequalities imply that b is the smallest integer not smaller than $\log_2(n+1)$. In other words, using the definition of the ceiling function, we conclude that

$$b = \lceil \log_2(n+1) \rceil.$$

c. $B = \lfloor \log_{10} n \rfloor + 1 = \lceil \log_{10}(n+1) \rceil$.

d. $b = \lfloor \log_2 n \rfloor + 1 \approx \log_2 n = \log_2 10 \log_{10} n \approx (\log_2 10)B$, where $B =$

$\lfloor \log_{10} n \rfloor + 1$. That is, the two size metrics are about equal to within a constant multiple for large values of n.

6. Before applying a sorting algorithm, compare the adjacent elements of its input: if $a_i \leq a_{i+1}$ for every $i = 0, .., n-2$, stop. Generally, it is not a worthwhile addition because it slows down the algorithm on all but very special inputs. Note that some sorting algorithms (notably bubble sort and insertion sort, which are discussed in Sections 3.1 and 4.1, respectively) intrinsically incorporate this test in the body of the algorithm.

7. a. $\frac{T(2n)}{T(n)} \approx \frac{c_M \frac{1}{3}(2n)^3}{c_M \frac{1}{3} n^3} = 8$, where c_M is the time of one multiplication.

   b. We can estimate the running time for solving systems of order n on the old computer and that of order N on the new computer as $T_{old}(n) \approx c_M \frac{1}{3} n^3$ and $T_{new}(N) \approx 10^{-3} c_M \frac{1}{3} N^3$, respectively, where c_M is the time of one multiplication on the old computer. Replacing $T_{old}(n)$ and $T_{new}(N)$ by these estimates in the equation $T_{old}(n) = T_{new}(N)$ yields $c_M \frac{1}{3} n^3 \approx 10^{-3} c_M \frac{1}{3} N^3$ or $\frac{N}{n} \approx 10$.

8. a. $\log_2 4n - \log_2 n = (\log_2 4 + \log_2 n) - \log_2 n = 2$.

   b. $\frac{\sqrt{4n}}{\sqrt{n}} = 2$.

   c. $\frac{4n}{n} = 4$.

   d. $\frac{(4n)^2}{n^2} = 4^2$.

   e. $\frac{(4n)^3}{n^3} = 4^3$.

   f. $\frac{2^{4n}}{2^n} = 2^{3n} = (2^n)^3$.

9. a. $n(n+1) \approx n^2$ has the same order of growth (quadratic) as $2000n^2$ to within a constant multiple.

   b. $100n^2$ (quadratic) has a lower order of growth than $0.01n^3$ (cubic).

   c. Since changing a logarithm's base can be done by the formula

   $$\log_a n = \log_a b \log_b n,$$

   all logarithmic functions have the same order of growth to within a constant multiple.

d. $\log_2^2 n = \log_2 n \log_2 n$ and $\log_2 n^2 = 2 \log n$. Hence $\log_2^2 n$ has a higher order of growth than $\log_2 n^2$.

e. $2^{n-1} = \frac{1}{2}2^n$ has the same order of growth as 2^n to within a constant multiple.

f. $(n-1)!$ has a lower order of growth than $n! = (n-1)!n$.

10. a. The total number of grains due to the inventor is

$$\sum_{i=1}^{64} 2^{i-1} = \sum_{j=0}^{63} 2^j = 2^{64} - 1 \approx 1.8 \cdot 10^{19}.$$

(It is many times more than one can get by planting with grain the entire surface of the planet Earth.) If it took just one second to count each grain, the total amount of time needed to count all these grains comes to about 585 billion years, over 100 times more than the estimated age of our planet.

b. Here, the total amount of grains would have been equal to

$$1 + 3 + ... + (2 \cdot 64 - 1) = 64^2.$$

With the same speed of counting one grain per second, he would have needed less than one hour and fourteen minutes to count his modest reward.

Exercises 2.2

1. Use the most appropriate notation among O, Θ, and Ω to indicate the time efficiency class of sequential search (see Section 2.1)

   a. in the worst case.

   b. in the best case.

   c. in the average case.

2. Use the informal definitions of O, Θ, and Ω to determine whether the following assertions are true or false.

   a. $n(n+1)/2 \in O(n^3)$ b. $n(n+1)/2 \in O(n^2)$

   c. $n(n+1)/2 \in \Theta(n^3)$ d. $n(n+1)/2 \in \Omega(n)$

3. For each of the following functions, indicate the class $\Theta(g(n))$ the function belongs to. (Use the simplest $g(n)$ possible in your answers.) Prove your assertions.
   a. $(n^2+1)^{10}$ b. $\sqrt{10n^2+7n+3}$

   c. $2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2}$ d. $2^{n+1} + 3^{n-1}$

   e. $\lfloor \log_2 n \rfloor$

4. a. Table 2.1 contains values of several functions that often arise in analysis of algorithms. These values certainly suggest that the functions

   $$\log n, \quad n, \quad n \log n, \quad n^2, \quad n^3, \quad 2^n, \quad n!$$

   are listed in increasing order of their order of growth. Do these values prove this fact with mathematical certainty?

   b. Prove that the functions are indeed listed in increasing order of their order of growth.

5. Order the following functions according to their order of growth (from the lowest to the highest):

   $(n-2)!, \quad 5 \lg(n+100)^{10}, \quad 2^{2n}, \quad 0.001n^4 + 3n^3 + 1, \quad \ln^2 n, \quad \sqrt[3]{n}, \quad 3^n.$

6. a. Prove that every polynomial of degree k, $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$ with $a_k > 0$, belongs to $\Theta(n^k)$.

   b. Prove that exponential functions a^n have different orders of growth for different values of base $a > 0$.

7. Prove (by using the definitions of the notations involved) or disprove (by giving a specific counterexample) the following assertions.

   a. If $t(n) \in O(g(n))$, then $g(n) \in \Omega(t(n))$.

   b. $\Theta(\alpha g(n)) = \Theta(g(n))$, where $\alpha > 0$.

   c. $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

   d.▷ For any two nonnegative functions $t(n)$ and $g(n)$ defined on the set of nonnegative integers, either $t(n) \in O(g(n))$, or $t(n) \in \Omega(g(n))$, or both.

8. ▷ Prove the section's theorem for

   a. Ω notation.

   b. Θ notation.

9. We mentioned in this section that one can check whether all elements of an array are distinct by a two-part algorithm based on the array's presorting.

   a. If the presorting is done by an algorithm with the time efficiency in $\Theta(n \log n)$, what will be the time efficiency class of the entire algorithm?

   b. If the sorting algorithm used for presorting needs an extra array of size n, what will be the space efficiency class of the entire algorithm?

10. The ***range*** of a finite nonempty set of n real numbers S is defined as the difference between the largest and smallest elements of S. For each representation of S given below, describe in English an algorithm to compute the range. Indicate the time efficiency classes of these algorithms using the most appropriate notation (O, Θ, or Ω).

    a. An unsorted array

    b. A sorted array

    c. A sorted singly linked list

    d. A binary search tree

11. *Lighter or heavier?* You have $n > 2$ identical-looking coins and a two-pan balance scale with no weights. One of the coins is a fake, but you do not know whether it is lighter or heavier than the genuine coins, which all weigh the same. Design a $\Theta(1)$ algorithm to determine whether the fake coin is lighter or heavier than the others.

12. ▷ *Door in a wall* You are facing a wall that stretches infinitely in both directions. There is a door in the wall, but you know neither how far away nor in which direction. You can see the door only when you are right next to it. Design an algorithm that enables you to reach the door by walking at most $O(n)$ steps where n is the (unknown to you) number of steps between your initial position and the door. [Par95]

Hints to Exercises 2.2

1. Use the corresponding counts of the algorithm's basic operation (see Section 2.1) and the definitions of O, Θ, and Ω.

2. Establish the order of growth of $n(n+1)/2$ first and then use the informal definitions of O, Θ, and Ω. (Similar examples were given in the section.)

3. Simplify the functions given to single out the terms defining their orders of growth.

4. a. Check carefully the pertinent definitions.

   b. Compute the ratio limits of every pair of consecutive functions on the list.

5. First simplify some of the functions. Then, use the list of functions in Table 2.2 to "anchor" each of the functions given. Prove their final placement by computing appropriate limits.

6. a. You can prove this assertion either by computing an appropriate limit or by applying mathematical induction.

   b. Compute $\lim_{n\to\infty} a_1^n/a_2^n$.

7. Prove the correctness of (a), (b), and (c) by using the appropriate definitions; construct a counterexample for (d) (e.g., by constructing two functions behaving differently for odd and even values of their arguments).

8. The proof of part (a) is similar to the one given for the theorem's assertion in Section 2.2. Of course, different inequalities need to be used to bound the sum from below.

9. Follow the analysis plan used in the text when the algorithm was mentioned for the first time.

10. You may use straightforward algorithms for all the four questions asked. Use the O notation for the time efficiency class of one of them, and the Θ notation for the three others.

11. The problem can be solved in two weighings.

12. You should walk intermittently left and right from your initial position until the door is reached.

Solutions to Exercises 2.2

1. a. Since $C_{worst}(n) = n$, $C_{worst}(n) \in \Theta(n)$.

   b. Since $C_{best}(n) = 1$, $C_{best}(1) \in \Theta(1)$.

   c. Since $C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p) = (1 - \frac{p}{2})n + \frac{p}{2}$ where $0 \le p \le 1$, $C_{avg}(n) \in \Theta(n)$.

2. $n(n+1)/2 \approx n^2/2$ is quadratic. Therefore

   a. $n(n+1)/2 \in O(n^3)$ is true. b. $n(n+1)/2 \in O(n^2)$ is true.

   c. $n(n+1)/2 \in \Theta(n^3)$ is false. d. $n(n+1)/2 \in \Omega(n)$ is true.

3. a. Informally, $(n^2+1)^{10} \approx (n^2)^{10} = n^{20} \in \Theta(n^{20})$ Formally,

   $$\lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{20}} = \lim_{n\to\infty} \frac{(n^2+1)^{10}}{(n^2)^{10}} = \lim_{n\to\infty} \left(\frac{n^2+1}{n^2}\right)^{10} == \lim_{n\to\infty} \left(1 + \frac{1}{n^2}\right)^{10} = 1.$$

   Hence $(n^2+1)^{10} \in \Theta(n^{20})$.

   Note: An alternative proof can be based on the binomial formula and the assertion of Exercise 6a.

   b. Informally, $\sqrt{10n^2+7n+3} \approx \sqrt{10n^2} = \sqrt{10}n \in \Theta(n)$. Formally,

   $$\lim_{n\to\infty} \frac{\sqrt{10n^2+7n+3}}{n} = \lim_{n\to\infty} \sqrt{\frac{10n^2+7n+3}{n^2}} = \lim_{n\to\infty} \sqrt{10 + \frac{7}{n} + \frac{3}{n^2}} = \sqrt{10}.$$

   Hence $\sqrt{10n^2+7n+3} \in \Theta(n)$.

   c. $2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2} = 2n2\lg(n+2) + (n+2)^2(\lg n - 1) \in \Theta(n \lg n) + \Theta(n^2 \lg n) = \Theta(n^2 \lg n)$.

   d. $2^{n+1} + 3^{n-1} = 2^n 2 + 3^n \frac{1}{3} \in \Theta(2^n) + \Theta(3^n) = \Theta(3^n)$.

   e. Informally, $\lfloor \log_2 n \rfloor \approx \log_2 n \in \Theta(\log n)$. Formally, by using the inequalities $x - 1 < \lfloor x \rfloor \le x$ (see Appendix A), we obtain an upper bound

   $$\lfloor \log_2 n \rfloor \le \log_2 n$$

   and a lower bound

   $$\lfloor \log_2 n \rfloor > \log_2 n - 1 \ge \log_2 n - \frac{1}{2}\log_2 n \text{ (for every } n \ge 4) = \frac{1}{2}\log_2 n.$$

   Hence $\lfloor \log_2 n \rfloor \in \Theta(\log_2 n) = \Theta(\log n)$.

4. a. The order of growth and the related notations O, Ω, and Θ deal with the asymptotic behavior of functions as n goes to infinity. Therefore no specific values of functions within a finite range of n's values, suggestive as they might be, can establish their orders of growth with mathematical certainty.

   b. $\lim_{n\to\infty} \frac{\log_2 n}{n} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(n)'} = \lim_{n\to\infty} \frac{\frac{1}{n}\log_2 e}{1} = \log_2 e \lim_{n\to\infty} \frac{1}{n} = 0.$

   $\lim_{n\to\infty} \frac{n}{n\log_2 n} = \lim_{n\to\infty} \frac{1}{\log_2 n} = 0.$

   $\lim_{n\to\infty} \frac{n\log_2 n}{n^2} = \lim_{n\to\infty} \frac{\log_2 n}{n} =$ (see the first limit of this exercise) $= 0.$

   $\lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0.$

   $\lim_{n\to\infty} \frac{n^3}{2^n} = \lim_{n\to\infty} \frac{(n^3)'}{(2^n)'} = \lim_{n\to\infty} \frac{3n^2}{2^n \ln 2} = \frac{3}{\ln 2} \lim_{n\to\infty} \frac{n^2}{2^n} = \frac{3}{\ln 2} \lim_{n\to\infty} \frac{(n^2)'}{(2^n)'}$
   $= \frac{3}{\ln 2} \lim_{n\to\infty} \frac{2n}{2^n \ln 2} = \frac{6}{\ln^2 2} \lim_{n\to\infty} \frac{n}{2^n} = \frac{6}{\ln^2 2} \lim_{n\to\infty} \frac{(n)'}{(2^n)'}$
   $= \frac{6}{\ln^2 2} \lim_{n\to\infty} \frac{1}{2^n \ln 2} = \frac{6}{\ln^3 2} \lim_{n\to\infty} \frac{1}{2^n} = 0.$

   $\lim_{n\to\infty} \frac{2^n}{n!} =$ (see Example 3 in the section) $0.$

5. $(n-2)! \in \Theta((n-2)!)$, $5\lg(n+100)^{10} = 50\lg(n+100) \in \Theta(\log n)$, $2^{2n} = (2^2)^n \in \Theta(4^n)$, $0.001n^4 + 3n^3 + 1 \in \Theta(n^4)$, $\ln^2 n \in \Theta(\log^2 n)$, $\sqrt[3]{n} \in \Theta(n^{\frac{1}{3}})$, $3^n \in \Theta(3^n)$. The list of these functions ordered in increasing order of growth looks as follows:

   $$5\lg(n+100)^{10}, \quad \ln^2 n, \quad \sqrt[3]{n}, \quad 0.001n^4 + 3n^3 + 1, \quad 3^n, \quad 2^{2n}, \quad (n-2)!$$

6. a. $\lim_{n\to\infty} \frac{p(n)}{n^k} = \lim_{n\to\infty} \frac{a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0}{n^k} = \lim_{n\to\infty} (a_k + \frac{a_{k-1}}{n} + \ldots + \frac{a_0}{n^k})$
   $= a_k > 0.$

   Hence $p(n) \in \Theta(n^k)$.

   b.

   $$\lim_{n\to\infty} \frac{a_1^n}{a_2^n} = \lim_{n\to\infty} \left(\frac{a_1}{a_2}\right)^n = \begin{cases} 0 & \text{if } a_1 < a_2 \quad \Leftrightarrow \quad a_1^n \in o(a_2^n) \\ 1 & \text{if } a_1 = a_2 \quad \Leftrightarrow \quad a_1^n \in \Theta(a_2^n) \\ \infty & \text{if } a_1 > a_2 \quad \Leftrightarrow \quad a_2^n \in o(a_1^n) \end{cases}$$

7. a. The assertion should be correct because it states that if the order of growth of $t(n)$ is smaller than or equal to the order of growth of $g(n)$, then

the order of growth of $g(n)$ is larger than or equal to the order of growth of $t(n)$. The formal proof is immediate, too:

$$t(n) \le cg(n) \quad \text{for all } n \ge n_0, \text{ where } c > 0,$$

implies

$$(\frac{1}{c})t(n) \le g(n) \quad \text{for all } n \ge n_0.$$

b. The assertion that $\Theta(\alpha g(n)) = \Theta(g(n))$ should be true because $\alpha g(n)$ and $g(n)$ differ just by a positive constant multiple and, hence, by the definition of Θ, must have the same order of growth. The formal proof has to show that $\Theta(\alpha g(n)) \subseteq \Theta(g(n))$ and $\Theta(g(n)) \subseteq \Theta(\alpha g(n))$. Let $f(n) \in \Theta(\alpha g(n))$; we'll show that $f(n) \in \Theta(g(n))$. Indeed,

$$f(n) \le c\alpha g(n) \quad \text{for all } n \ge n_0 \text{ (where } c > 0)$$

can be rewritten as

$$f(n) \le c_1 g(n) \quad \text{for all } n \ge n_0 \text{ (where } c_1 = c\alpha > 0),$$

i.e., $f(n) \in \Theta(g(n))$.

Let now $f(n) \in \Theta(g(n))$; we'll show that $f(n) \in \Theta(\alpha g(n))$ for $\alpha > 0$. Indeed, if $f(n) \in \Theta(g(n))$,

$$f(n) \le cg(n) \quad \text{for all } n \ge n_0 \text{ (where } c > 0)$$

and therefore

$$f(n) \le \frac{c}{\alpha} ag(n) = c_1 \alpha g(n) \quad \text{for all } n \ge n_0 \text{ (where } c_1 = \frac{c}{\alpha} > 0),$$

i.e., $f(n) \in \Theta(\alpha g(n))$.

c. The assertion is obviously correct (similar to the assertion that $a = b$ if and only if $a \le b$ and $a \ge b$). The formal proof should show that $\Theta(g(n)) \subseteq O(g(n)) \cap \Omega(g(n))$ and that $O(g(n)) \cap \Omega(g(n)) \subseteq \Theta(g(n))$, which immediately follow from the definitions of O, Ω, and Θ.

d. The assertion is false. The following pair of functions can serve as a counterexample

$$t(n) = \begin{cases} n & \text{if } n \text{ is even} \\ n^2 & \text{if } n \text{ is odd} \end{cases} \quad \text{and} \quad g(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ n & \text{if } n \text{ is odd} \end{cases}$$

8. a. We need to prove that if $t_1(n) \in \Omega(g_1(n))$ and $t_2(n) \in \Omega(g_2(n))$, then $t_1(n) + t_2(n) \in \Omega(\max\{g_1(n), g_2(n)\})$.

Proof Since $t_1(n) \in \Omega(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \geq c_1 g_1(n) \quad \text{for all } n \geq n_1.$$

Since $t_2(n) \in \Omega(g_2(n))$, there exist some positive constant c_2 and some nonnegative integer n_2 such that

$$t_2(n) \geq c_2 g_2(n) \quad \text{for all } n \geq n_2.$$

Let us denote $c = \min\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\geq c_1 g_1(n) + c_2 g_2(n) \\ &\geq c g_1(n) + c g_2(n) = c[g_1(n) + g_2(n)] \\ &\geq c \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence $t_1(n) + t_2(n) \in \Omega(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $\min\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

b. The proof follows immediately from the theorem proved in the text (the O part), the assertion proved in part (a) of this exercise (the Ω part), and the definition of Θ (see Exercise 7c).

9. a. Since the running time of the sorting part of the algorithm will still dominate the running time of the second, it's the former that will determine the time efficiency of the entire algorithm. Formally, it follows from equality

$$\Theta(n \log n) + O(n) = \Theta(n \log n),$$

whose validity is easy to prove in the same manner as that of the section's theorem.

b. Since the second part of the algorithm will use no extra space, the space efficiency class will be determined by that of the first (sorting) part. Therefore, it will be $\Theta(n)$.

10. a. Scan the array to find the maximum and minimum values among its elements and then compute the difference between them. The algorithm's time efficiency is $\Theta(n)$. Note: Although one can find both the maximum and minimum values in an n-element array with about $1.5n$ comparisons

(see the solutions to Problem 5 in Exercises in 2.3 and Problem 2 in Exercises 5.1), it doesn't change the linear efficiency class, of course.

b. For a sorted array, we can simply compute the difference between its first and last elements: $A[n-1] - A[0]$. The time efficiency class is obviously $\Theta(1)$.

c. The smallest element is in the first node of the list and hence its values can be obtained in constant time. The largest element is in the last node reachable only by the traversal of the entire list, which requires linear time. Computing the difference between the two values requires constant time. Hence, the time efficiency class is $\Theta(n)$.

d. The smallest (largest) element in a binary search tree is in the leftmost (rightmost) node. To reach it, one needs to start with the root and follow the chain of left-child (right-child) pointers until a node with the null left-child (right-child) pointer is reached. Depending on the structure of the tree, this chain of nodes can be between 1 and n nodes long. Hence, the time of reaching its last node will be in $O(n)$. The running time of the entire algorithm will also be linear: $O(n) + O(n) + \Theta(1) = O(n)$.

11. The puzzle can be solved in two weighings as follows. Start by taking aside one coin if n is odd and two coins if n is even. After that divide the remaining even number of coins into two equal-size groups and put them on the opposite pans of the scale. If they weigh the same, all these coins are genuine and the fake coin is among the coins set aside. So we can weigh the set aside group of one or two coins against the same number of genuine coins: if the former weighs less, the fake coin is lighter, otherwise, it is heavier. If the first weighing does not result in a balance, take the lighter group and, if the number of coins in it is odd, add to it one of the coins initially set aside (which must be genuine). Divide all these coins into two equal-size groups and weigh them. If they weigh the same, all these coins are genuine and therefore the fake coin is heavier; otherwise, they contain the fake, which is lighter.
Note: The puzzle provides a very rare example of a problem that can be solved in the same number of basic operations (namely, two weighings) irrespective of how large the problem's instance (here, the number of coins) is. Of course, had we considered putting one coin on the scale as the algorithm's basic operation, the algorithm's efficiency would have been in $\Theta(n)$ instead of $\Theta(1)$.

12. The key idea here is to walk intermittently right and left going each time exponentially farther from the initial position. A simple implementation of this idea is to do the following until the door is reached: For $i = 0, 1, ...,$ make 2^i steps to the right, return to the initial position, make 2^i steps to the left, and return to the initial position again. Let $2^{k-1} < n \leq 2^k$. The

number of steps this algorithm will need to find the door can be estimated above as follows:

$$\sum_{i=0}^{k-1} 4 \cdot 2^i + 3 \cdot 2^k = 4(2^k - 1) + 3 \cdot 2^k < 7 \cdot 2^k = 14 \cdot 2^{k-1} < 14n.$$

Hence the number of steps made by the algorithm is in $O(n)$. (Note: It is not difficult to improve the multiplicative constant with a better algorithm.)

Exercises 2.3

1. Compute the following sums.

   a. $1 + 3 + 5 + 7 + \cdots + 999$

   b. $2 + 4 + 8 + 16 + \cdots + 1024$

   c. $\sum_{i=3}^{n+1} 1$ d. $\sum_{i=3}^{n+1} i$ e. $\sum_{i=0}^{n-1} i(i+1)$

   f. $\sum_{j=1}^{n} 3^{j+1}$ g. $\sum_{i=1}^{n} \sum_{j=1}^{n} ij$ h. $\sum_{i=0}^{n-1} 1/i(i+1)$

2. Find the order of growth of the following sums.

   a. $\sum_{i=0}^{n-1} (i^2+1)^2$ b. $\sum_{i=2}^{n-1} \lg i^2$

   c. $\sum_{i=1}^{n} (i+1)2^{i-1}$ d. $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} (i+j)$

   Use the $\Theta(g(n))$ notation with the simplest function $g(n)$ possible.

3. The sample variance of n measurements $x_1, x_2, ..., x_n$ can be computed as

$$\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1} \text{ where } \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

   or

$$\frac{\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2/n}{n-1}.$$

   Find and compare the number of divisions, multiplications, and additions/subtractions (additions and subtractions are usually bunched together) that are required for computing the variance according to each of these formulas.

4. Consider the following algorithm.

   Algorithm *Mystery(n)*
   //Input: A nonnegative integer n
   $S \leftarrow 0$
   for $i \leftarrow 1$ **to** n **do**
   $S \leftarrow S + i * i$
   return S

   a. What does this algorithm compute?

   b. What is its basic operation?

   c. How many times is the basic operation executed?

d. What is the efficiency class of this algorithm?

e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

5. Consider the following algorithm.

```
Algorithm Secret(A[0..n − 1])
//Input: An array A[0..n − 1] of n real numbers
minval ← A[0];   maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] < minval
        minval ← A[i]
    if A[i] > maxval
        maxval ← A[i]
return maxval − minval
```

Answer questions a–e of Problem 4 about this algorithm.

6. Consider the following algorithm.

```
Algorithm Enigma(A[0..n − 1, 0..n − 1])
//Input: A matrix A[0..n − 1, 0..n − 1] of real numbers
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i, j] ≠ A[j, i]
            return false
return true
```

Answer the questions a–e of Problem 4 about this algorithm.

7. Improve the implementation of the matrix multiplication algorithm (see Example 3) by reducing the number of additions made by the algorithm. What effect will this change have on the algorithm's efficiency?

8. Determine the asymptotic order of growth for the total number of times all the doors are toggled in the locker doors puzzle (Problem 12 in Exercises 1.1).

9. Prove the formula

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

either by mathematical induction or by following the insight of a 10-year-old schoolboy named Karl Friedrich Gauss (1777–1855) who grew up to become one of the greatest mathematicians of all times.

10. *Mental arithmetic* A 10×10 table is filled with repeating numbers on its diagonals as shown below. Calculate the total sum of the table's numbers in your head. (after [Cra07, Question 1.33])

1	2	3			•••			9	10
2	3						9	10	11
3						9	10	11	
					9	10	11		
				9	10	11			
⋮			9	10	11				⋮
		9	10	11					
	9	10	11						17
9	10	11						17	18
10	11				•••		17	18	19

11. Consider the following version of an important algorithm that we will study later in the book.

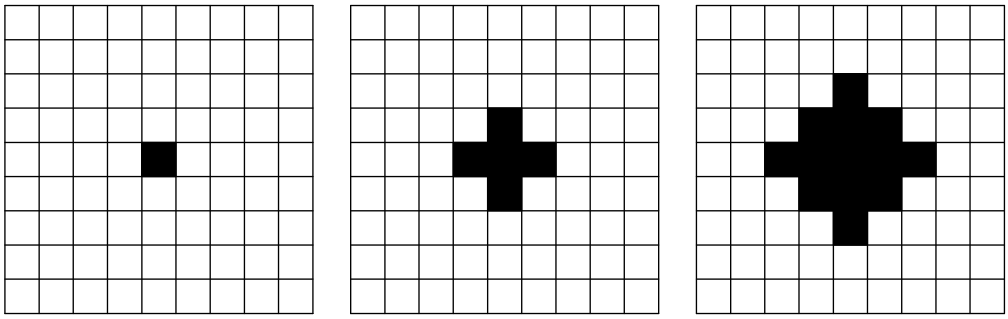
Algorithm $GE(A[0..n-1, 0..n])$
//Input: An $n \times (n+1)$ matrix $A[0..n-1, 0..n]$ of real numbers
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        for $k \leftarrow i$ **to** n **do**
            $A[j,k] \leftarrow A[j,k] - A[i,k] * A[j,i] \ / \ A[i,i]$
return A

a.▷ Find the time efficiency class of this algorithm.

b.▷ What glaring inefficiency does this pseudocode contain and how can it be eliminated to speed the algorithm up?

12. *von Neumann's neighborhood* How many one-by-one squares are generated by the algorithm that starts with a single square square and on each of its n iterations adds new squares all round the outside. How many one-by-one squares are generated on the nth iteration? [Gar99] (In the parlance of cellular automata theory, the answer is the number of cells in the von Neumann neighborhood of range n.) The results for $n = 0$, 1, and

2 are illustrated below.



$n = 0$ $n = 1$ $n = 2$

13. *Page numbering* Find the total number of decimal digits needed for numbering pages in a book of 1000 pages. Assume that the pages are numbered consecutively starting with 1.

Hints to Exercises 2.3

1. Use the common summation formulas and rules listed in Appendix A. You may need to perform some simple algebraic operations before applying them.

2. Find a sum among those in Appendix A that looks similar to the sum in question and try to transform the latter to the former. Note that you do not have to get a closed-end formula for a sum before establishing its order of growth.

3. Just follow the formulas in question.

4. a. Tracing the algorithm to get its output for a few small values of n (e.g., $n = 1$, 2, and 3) should help if you need it.

   b. We faced the same question for the examples discussed in the text. One of them is particularly pertinent here.

   c. Follow the plan outlined in the section.

   d. As a function of n, the answer should follow immediately from your answer to part (c). You may also want to give an answer as a function of the number of bits in the n's representation (why?).

   e. Have you not encountered this sum somewhere?

5. a. Tracing the algorithm to get its output for a few small values of n (e.g., $n = 1$, 2, and 3) should help if you need it.

   b. We faced the same question for the examples discussed in the section. One of them is particularly pertinent here.

   c. You can either follow the section's plan by setting up and computing a sum or answer the question directly. (Try to do both.)

   d. Your answer will immediately follow from the answer to part (c).

   e. Does the algorithm always have to make two comparisons on each iteration? This idea can be developed further to get a more significant improvement than the obvious one—try to do it for a four-element array and then generalize the insight. But can we hope to find an algorithm with a better than linear efficiency?

6. a. Elements $A[i, j]$ and $A[j, i]$ are symmetric with respect to the main diagonal of the matrix.

   b. There is just one candidate here.

c. You may investigate the worst case only.

d. Your answer will immediately follow from the answer to part (c).

e. Compare the problem the algorithm solves with the way it does this.

7. Computing a sum of n numbers can be done with $n - 1$ additions. How many does the algorithm make in computing each element of the product matrix?

8. Set up a sum for the number of times all the doors and toggled and find its asymptotic order of growth by using some properties from Appendix A.

9. For the general step of the proof by induction, use the formula

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n.$$

The young Gauss computed the sum $1 + 2 + \cdots + 99 + 100$ by noticing that it can be computed as the sum of 50 pairs, each with the same sum.

10. There are at least two different ways to solve this problem, which comes from a collection of Wall Street interview questions.

11. a. Setting up a sum should pose no difficulties. Using the standard summation formulas and rules will require more effort than in the previous examples, however.

b. Optimize the algorithm's innermost loop.

12. Set up a sum for the number of squares after n iterations of the algorithm and then simplify it to get a closed-form answer.

13. To derive a formula expressing the total number of digits as a function of the number of pages n, where $1 \leq n \leq 1000$, it's convenient to partition the function's domain into several natural intervals.

Solutions to Exercises 2.3

1. a. $1+3+5+7+...+999 = \sum_{i=1}^{500}(2i\text{-}1) = \sum_{i=1}^{500} 2i\text{-}\sum_{i=1}^{500} 1 = 2\frac{500*501}{2}\text{-}500 = 250,000.$

(Or by using the formula for the sum of odd integers: $\sum_{i=1}^{500}(2i\text{-}1) = 500^2 = 250,000$.
Or by using the formula for the sum of the arithmetic progression with $a_1 = 1$, $a_n = 999$, and $n = 500$: $\frac{(a_1+a_n)n}{2} = \frac{(1+999)500}{2} = 250,000$.)

b. $2+4+8+16+...+1,024 = \sum_{i=1}^{10} 2^i = \sum_{i=0}^{10} 2^i - 1 = (2^{11}-1)-1 = 2,046.$
(Or by using the formula for the sum of the geometric series with $a = 2$, $q = 2$, and $n = 9$: $a\frac{q^{n+1}-1}{q-1} = 2\frac{2^{10}-1}{2-1} = 2,046$.)

c. $\sum_{i=3}^{n+1} 1 = (n+1) - 3 + 1 = n - 1.$

d. $\sum_{i=3}^{n+1} i = \sum_{i=0}^{n+1} i - \sum_{i=0}^{2} i = \frac{(n+1)(n+2)}{2} - 3 = \frac{n^2+3n-4}{2}.$

e. $\sum_{i=0}^{n-1} i(i+1) = \sum_{i=0}^{n-1}(i^2+i) = \sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i = \frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2}$

$= \frac{(n^2-1)n}{3}.$

f. $\sum_{j=1}^{n} 3^{j+1} = 3\sum_{j=1}^{n} 3^j = 3[\sum_{j=0}^{n} 3^j - 1] = 3[\frac{3^{n+1}-1}{3-1} - 1] = \frac{3^{n+2}-9}{2}.$

g. $\sum_{i=1}^{n}\sum_{j=1}^{n} ij = \sum_{i=1}^{n} i\sum_{j=1}^{n} j = \sum_{i=1}^{n} i\frac{n(n+1)}{2} = \frac{n(n+1)}{2}\sum_{i=1}^{n} i = \frac{n(n+1)}{2}\frac{n(n+1)}{2}$

$= \frac{n^2(n+1)^2}{4}.$

h. $\sum_{i=1}^{n} 1/i(i+1) = \sum_{i=1}^{n}(\frac{1}{i} - \frac{1}{i+1})$

$= (\frac{1}{1} - \frac{1}{2}) + (\frac{1}{2} - \frac{1}{3}) + ... + (\frac{1}{n-1} - \frac{1}{n}) + (\frac{1}{n} - \frac{1}{n+1}) = 1 - \frac{1}{n+1} = \frac{n}{n+1}.$
(This is a special case of the so-called *telescoping series*—see Appendix A—$\sum_{i=l}^{u}(a_i - a_{i-1}) = a_u - a_{l-1}$.)

2. a. $\sum_{i=0}^{n-1}(i^2+1)^2 = \sum_{i=0}^{n-1}(i^4+2i^2+1) = \sum_{i=0}^{n-1} i^4 + 2\sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} 1$
$\in \Theta(n^5) + \Theta(n^3) + \Theta(n) = \Theta(n^5)$ (or just $\sum_{i=0}^{n-1}(i^2+1)^2 \approx \sum_{i=0}^{n-1} i^4 \in \Theta(n^5)$).

b. $\sum_{i=2}^{n-1} \log_2 i^2 = \sum_{i=2}^{n-1} 2\log_2 i = 2\sum_{i=2}^{n-1} \log_2 i = 2\sum_{i=1}^{n} \log_2 i - 2\log_2 n$
$\in 2\Theta(n\log n) - \Theta(\log n) = \Theta(n\log n).$

c. $\sum_{i=1}^{n}(i+1)2^{i-1} = \sum_{i=1}^{n} i2^{i-1} + \sum_{i=1}^{n} 2^{i-1} = \frac{1}{2}\sum_{i=1}^{n} i2^i + \sum_{j=0}^{n-1} 2^j$
$\in \Theta(n2^n) + \Theta(2^n) = \Theta(n2^n)$ (or $\sum_{i=1}^{n}(i+1)2^{i-1} \approx \frac{1}{2}\sum_{i=1}^{n} i2^i \in \Theta(n2^n)$).

d. $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}(i+j) = \sum_{i=0}^{n-1}[\sum_{j=0}^{i-1} i + \sum_{j=0}^{i-1} j] = \sum_{i=0}^{n-1}[i^2 + \frac{(i-1)i)}{2}] = \sum_{i=0}^{n-1}[\frac{3}{2}i^2 - \frac{1}{2}i]$
$= \frac{3}{2}\sum_{i=0}^{n-1} i^2 - \frac{1}{2}\sum_{i=0}^{n-1} i \in \Theta(n^3) - \Theta(n^2) = \Theta(n^3)$.

3. For the first formula: $D(n) = 2$, $M(n) = n$, $A(n) + S(n) = [(n-1) + (n-1)] + (n+1) = 3n - 1$.

   For the second formula: $D(n) = 2$, $M(n) = n + 1$, $A(n) + S(n) = [(n-1) + (n-1)] + 2 = 2n$.

4. a. Computes $S(n) = \sum_{i=1}^{n} i^2$.

   b. Multiplication (or, if multiplication and addition are assumed to take the same amount of time, either of the two).

   c. $C(n) = \sum_{i=1}^{n} 1 = n$.

   d. $C(n) = n \in \Theta(n)$. Since the number of bits $b = \lfloor \log_2 n \rfloor + 1 \approx \log_2 n$ and hence $n \approx 2^b$, $C(n) \approx 2^b \in \Theta(2^b)$.

   e. Use the formula $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ to compute the sum in $\Theta(1)$ time (which assumes that the time of arithmetic operations stay constant irrespective of the size of the operations' operands).

5. a. Computes the range, i.e., the difference between the array's largest and smallest elements.

   b. An element comparison.

   c. $C(n) = \sum_{i=1}^{n-1} 2 = 2(n-1)$.

   d. $\Theta(n)$.

   e. An obvious improvement for some inputs (but not for the worst case) is to replace the two if-statements by the following one:

   if $A[i] < minval$ $minval \leftarrow A[i]$

else if $A[i] > maxval$ $maxval \leftarrow A[i]$.

Another improvement, both more subtle and substantial, is based on the observation that it is more efficient to update the minimum and maximum values seen so far not for each element but for a pair of two consecutive elements. If two such elements are compared with each other first, the updates will require only two more comparisons for the total of three comparisons per pair. Note that the same improvement can be obtained by a divide-and-conquer algorithm (see Problem 2 in Exercises 5.1).

6. a. The algorithm returns "true" if its input matrix is symmetric and "false" if it is not.

   b. Comparison of two matrix elements.

   c. $C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1)$
   $= \sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + ... + 1 = \frac{(n-1)n}{2}$.

   d. Quadratic: $C_{worst}(n) \in \Theta(n^2)$ (or $C(n) \in O(n^2)$).

   e. The algorithm is optimal because any algorithm that solves this problem must, in the worst case, compare $(n-1)n/2$ elements in the upper-triangular part of the matrix with their symmetric counterparts in the lower-triangular part, which is all this algorithm does.

7. Replace the body of the j loop by the following fragment:

   $C[i, j] \leftarrow A[i, 0] * B[0, j]$
   for $k \leftarrow 1$ **to** $n-1$ **do**
   $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$

   This will decrease the number of additions from n^3 to $n^3 - n^2$, but the number of multiplications will still be n^3. The algorithm's efficiency class will remain cubic.

8. Let $T(n)$ be the total number of times all the doors are toggled. The problem statement implies that

$$T(n) = \sum_{i=1}^{n} \lfloor n/i \rfloor.$$

   Since $x - 1 < \lfloor x \rfloor \leq x$ and $\sum_{i=1}^{n} 1/i \approx \ln n + \gamma$, where $\gamma = 0.5772...$(see

Appendix A),

$$T(n) \leq \sum_{i=1}^{n} n/i = n \sum_{i=1}^{n} 1/i \approx n(\ln n + \gamma) \in \Theta(n \log n).$$

Similarly,

$$T(n) > \sum_{i=1}^{n} (n/i - 1) = n \sum_{i=1}^{n} 1/i - \sum_{i=1}^{n} 1 \approx n(\ln n + \gamma) - n \in \Theta(n \log n).$$

This implies that $T(n) \in \Theta(n \log n)$.

Note: Alternatively, we could use the formula for approximating sums by definite integrals (see Appendix A):

$$T(n) \leq \sum_{i=1}^{n} n/i = n(1 + \sum_{i=2}^{n} 1/i) \leq n(1 + \int_{1}^{n} \frac{1}{x} dx) = n(1 + \ln n) \in \Theta(n \log n)$$

and

$$T(n) > \sum_{i=1}^{n} (n/i - 1) = n \sum_{i=1}^{n} 1/i - \sum_{i=1}^{n} 1 \geq n \int_{1}^{n+1} \frac{1}{x} dx - n = n \ln(n+1) - n \in \Theta(n \log n).$$

9. Here is a proof by mathematical induction that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for every positive integer n.
(i) Basis step: For $n = 1$, $\sum_{i=1}^{n} i = \sum_{i=1}^{1} i = 1$ and $\frac{n(n+1)}{2}\Big|_{n=1} = \frac{1(1+1)}{2} = 1$.
(ii) Inductive step: Assume that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for a positive integer n.
We need to show that then $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$. This is obtained as follows:
$\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{n(n+1)+2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$.

The young Gauss computed the sum

$$1 + 2 + \cdots + 99 + 100$$

by noticing that it can be computed as the sum of 50 pairs, each with the sum 101:

$$1 + 100 = 2 + 99 = ... = 50 + 51 = 101.$$

Hence the entire sum is equal to $50 \cdot 101 = 5,050$. (The well-known historic anecdote claims that his teacher gave this assignment to a class to keep

the class busy.) The Gauss idea can be easily generalized to an arbitrary n by adding

$$S(n) = 1 + 2 + \cdots + (n-1) + n$$

and

$$S(n) = n + (n-1) + \cdots + 2 + 1$$

to obtain

$$2S(n) = (n+1)n \text{ and hence } S(n) = \frac{n(n+1)}{2}.$$

10. The object here is to compute (in one's head) the sum of the numbers in the table below:

1	2	3			•••			9	10
2	3						9	10	11
3						9	10	11	
					9	10	11		
				9	10	11			
⋮			9	10	11				⋮
		9	10	11					
	9	10	11						17
9	10	11						17	18
10	11				•••		17	18	19

The first method is based on the observation that the sum of any two numbers in the squares symmetric with respect to the diagonal connecting the lower left and upper right corners is equal to 20: 1+19, 2+18, 2+18, and so on. So, since there are (10·10-10)/2 = 45 such pairs (we subtracted the number of the squares on that diagonal from the total number of squares), the sum of the numbers outside that diagonal is equal to 20·45 = 900. With 10·10 = 100 on the diagonal, the total sum is equal to 900 + 100 = 1000.

The second method computes the sum row by row (or column by column). The sum in the first row is equal to 10·11/2 = 55 according to formula (S2). The sum of the numbers in second row is 55 + 10 since each of the numbers is larger by 1 than their counterparts in the row above. The same is true for all the other rows as well. Hence the total sum is equal to 55 + (55+10) + (55+20) + ...+ (55+90) = 55·10 + (10+20+...+90) =

55·10 + 10·(1+2+...+9) = 55·10 + 10·45 =1000.

Note that the first method uses the same trick Carl Gauss presumably used to find the sum of the first hundred integers (Problem 9 in Exercises 2.3). We also used this formula (twice, in fact) in the second solution to the problem.

11. a. The number of multiplications $M(n)$ and the number of divisions $D(n)$ made by the algorithm are given by the same sum:

$$\begin{aligned} M(n) &= D(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=i}^{n} 1 = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} (n-i+1) = \\ &= \sum_{i=0}^{n-2} (n-i+1)(n-1-(i+1)+1) = \sum_{i=0}^{n-2} (n-i+1)(n-i-1) \\ &= (n+1)(n-1) + n(n-2) + ... + 3*1 \\ &= \sum_{j=1}^{n-1} (j+2)j = \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j = \frac{(n-1)n(2n-1)}{6} + 2\frac{(n-1)n}{2} \\ &= \frac{n(n-1)(2n+5)}{6} \approx \frac{1}{3}n^3 \in \Theta(n^3). \end{aligned}$$

b. The inefficiency is the repeated evaluation of the ratio $A[j,i]$ / $A[i,i]$ in the algorithm's innermost loop, which, in fact, does not change with the loop variable k. Hence, this *loop invariant* can be computed just once before entering this loop: $temp \leftarrow A[j,i]$ / $A[i,i]$; the innermost loop is then changed to

$$A[j,k] \leftarrow A[j,k] - A[i,k] * temp.$$

This change eliminates the most expensive operation of the algorithm, the division, from its innermost loop. The running time gain obtained by this change can be estimated as follows:

$$\frac{T_{old}(n)}{T_{new}(n)} \approx \frac{c_M \frac{1}{3}n^3 + c_D \frac{1}{3}n^3}{c_M \frac{1}{3}n^3} = \frac{c_M + c_D}{c_M} = \frac{c_D}{c_M} + 1,$$

where c_D and c_M are the time for one division and one multiplication, respectively.

12. The answer can be obtained by a straightforward evaluation of the sum

$$2\sum_{i=1}^{n}(2i-1) + (2n+1) = 2n^2 + 2n + 1.$$

(One can also get the closed-form answer by noting that the cells on the alternating diagonals of the von Neumann neighborhood of range n compose two squares of sizes $n+1$ and n, respectively.)

13. Let $D(n)$ be the total number of decimal digits in the first n positive integers (book pages). The first nine numbers are one-digit, therefore $D(n) = n$ for $1 \leq n \leq 9$. The next 90 numbers from 10 to 99 inclusive are two-digits. Hence

$$D(n) = 9 + 2(n - 9) \text{ for } 10 \leq n \leq 99.$$

The maximal value of $D(n)$ for this range is $D(99) = 189$. Further, there are 900 three-digit decimals, which leads to the formula

$$D(n) = 189 + 3(n - 99) \text{ for } 100 \leq n \leq 999.$$

The maximal value of $D(n)$ for this range is $D(999) = 2889$. Adding four digits for page 1000, we obtain $D(1000) = 2893$.

Exercises 2.4

1. Solve the following recurrence relations.

   a. $x(n) = x(n-1) + 5 \quad \text{for } n > 1, \quad x(1) = 0$

   b. $x(n) = 3x(n-1) \quad \text{for } n > 1, \quad x(1) = 4$

   c. $x(n) = x(n-1) + n \quad \text{for } n > 0, \quad x(0) = 0$

   d. $x(n) = x(n/2) + n \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 2^k)$

   e. $x(n) = x(n/3) + 1 \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 3^k)$

2. Set up and solve a recurrence relation for the number of calls made by $F(n)$, the recursive algorithm for computing $n!$.

3. Consider the following recursive algorithm for computing the sum of the first n cubes: $S(n) = 1^3 + 2^3 + \cdots + n^3$.

   Algorithm $S(n)$
   //Input: A positive integer n
   //Output: The sum of the first n cubes
   if $n = 1$ **return** 1
   else return $S(n-1) + n * n * n$

   a. Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

   b. How does this algorithm compare with the straightforward nonrecursive algorithm for computing this function?

4. Consider the following recursive algorithm.

   Algorithm $Q(n)$
   //Input: A positive integer n
   if $n = 1$ **return** 1
   else return $Q(n-1) + 2 * n - 1$

   a. Set up a recurrence relation for this function's values and solve it to determine what this algorithm computes.

   b. Set up a recurrence relation for the number of multiplications made by this algorithm and solve it.

   c. Set up a recurrence relation for the number of additions/subtractions made by this algorithm and solve it.

5. *Tower of Hanoi* a. In the original version of the Tower of Hanoi puzzle, as it was published by Edouard Lucas, a French mathematician, in the 1890s, the world will end after 64 disks have been moved from a mystical Tower of Brahma. Estimate the number of years it will take if monks could move one disk per minute. (Assume that monks do not eat, sleep, or die.)

   b. How many moves are made by the ith largest disk ($1 \leq i \leq n$) in this algorithm?

   c. Find a nonrecursive algorithm for the Tower of Hanoi puzzle and implement it in the language of your choice.

6. $\triangleright$ *Restricted Tower of Hanoi* Consider the version of the Tower of Hanoi puzzle in which n disks have to be moved from peg A to peg C using peg B so that any move should either place a disk on peg B or move a disk from that peg. (Of course, the prohibition of placing a larger disk on top of a smaller one remains in place, too.) Design a recursive algorithm for this problem and find the number of moves made by it.

7. $\triangleright$ a. Prove that the exact number of additions made by the recursive algorithm *BinRec(*n) for an arbitrary positive integer n is $\lfloor \log_2 n \rfloor$.

   b. Set up a recurrence relation for the number of additions made by the nonrecursive version of this algorithm (see Section 2.3, Example 4) and solve it.

8. a. Design a recursive algorithm for computing 2^n for any nonnegative integer n that is based on the formula: $2^n = 2^{n-1} + 2^{n-1}$.

   b. Set up a recurrence relation for the number of additions made by the algorithm and solve it.

   c. Draw a tree of recursive calls for this algorithm and count the number of calls made by the algorithm.

   d. Is it a good algorithm for solving this problem?

9. Consider the following recursive algorithm.

Algorithm *Riddle*$(A[0..n-1])$
//Input: An array $A[0..n-1]$ of real numbers
if $n = 1$ **return** $A[0]$
else *temp* $\leftarrow$ *Riddle*$(A[0..n-2])$
    if *temp* $\leq A[n-1]$ **return** *temp*
    else return $A[n-1]$

a. What does this algorithm compute?

b. Set up a recurrence relation for the algorithm's basic operation count and solve it.

10. Consider the following algorithm to check whether a graph defined by its adjacency matrix is complete.

    Algorithm *GraphComplete*$(A[0..n-1, 0..n-1])$

    //Input: Adjacency matrix $A[0..n-1, 0..n-1])$ of an undirected graph G with $n \geq 1$ vertices
    //Output: 1 (true) if G is complete and 0 (false) otherwise
    if $n = 1$ **return** 1 //one-vertex graph is complete by definition
    else
        if **not** *GraphComplete*$(A[0..n-2, 0..n-2])$ **return** 0
        else for $j \leftarrow 0$ **to** $n-2$ **do**
            if $A[n-1, j] = 0$ **return** 0
        return 1

    What is the algorithm's efficiency class in the worst case?

11. The determinant of an $n \times n$ matrix

$$A = \begin{bmatrix} a_{0\,0} & \dots & a_{0\,n-1} \\ a_{1\,0} & \dots & a_{1\,n-1} \\ \vdots & & \\ a_{n-1\,0} & \dots & a_{n-1\,n-1} \end{bmatrix},$$

    denoted $\det A$, can be defined as a_{00} for $n = 1$ and, for $n > 1$, by the recursive formula

$$\det A = \sum_{j=0}^{n-1} s_j a_{0\,j} \det A_j,$$

    where s_j is $+1$ if j is even and -1 if j is odd, $a_{0\,j}$ is the element in row 0 and column j, and A_j is the $(n-1) \times (n-1)$ matrix obtained from matrix A by deleting its row 0 and column j.

    a.▷ Set up a recurrence relation for the number of multiplications made by the algorithm implementing this recursive definition.

    b.▷ Without solving the recurrence, what can you say about the solution's order of growth as compared to $n!$?

12. *von Neumann's neighborhood revisited* Find the number of cells in the von Neumann neighborhood of range n (Problem 12 in Exercises 2.3) by setting up and solving a recurrence relation.

13. *Frying hamburgers* There are n hamburgers to be fried on a small grill that can hold only two hamburgers at a time. Each hamburger has to be fried on both sides; frying one side of a hamburger takes one minute, regardless of whether one or two hamburgers are fried at the same time. Consider the following recursive algorithm for executing this task. If $n \leq 2$, fry the hamburger (or the two hamburgers together if $n = 2$) on each side. If $n > 2$, fry two hamburgers together on each side and then fry the remaining $n - 2$ hamburgers by the same algorithm.

    a. Set up and solve the recurrence for the amount of time this algorithm needs to fry n hamburgers.

    b. Explain why this algorithm does *not* fry the hamburgers in the minimum amount of time for all $n > 0$.

    c. Give a correct recursive algorithm that executes the task in the minimum amount of time for all $n > 0$ and find a closed-form formula for the minimum amount of time.

14. $\triangleright$ *Celebrity problem* A celebrity among a group of n people is a person who knows nobody but is known by everybody else. The task is to identify a celebrity by only asking questions to people of the form: "Do you know him/her?" Design an efficient algorithm to identify a celebrity or determine that the group has no such person. How many questions does your algorithm need in the worst case? [Man89]

Hints to Exercises 2.4

1. Each of these recurrences can be solved by the method of backward substitutions.

2. The recurrence relation in question is almost identical to the recurrence relation for the number of multiplications, which was set up and solved in the section.

3. a. The question is similar to that about the efficiency of the recursive algorithm for computing $n!$.

   b. Write a pseudocode for the nonrecursive algorithm and determine its efficiency.

4. a. Note that you are asked here about a recurrence for the function's values, not about a recurrence for the number of times its operation is executed. Just follow the pseudocode to set it up. It is easier to solve this recurrence by *forward* substitutions (see Appendix B).

   b. This question is very similar to one we have already discussed.

   c. You may want to include the substraction needed to decrease n.

5. a. Use the formula for the number of disk moves derived in the section.

   b. Solve the problem for 3 disks to investigate the number of moves made by each of the disks. Then generalize the observations and prove their validity for the general case of n disks.

6. The required algorithm and the method of its analysis are similar to those of the classic version of the puzzle. Because of the additional constraint, more than two smaller instances of the puzzle need to be solved here.

7. a. Consider separately the cases of even and odd values of n and show that for both of them $\lfloor \log_2 n \rfloor$ satisfies the recurrence relation and its initial condition.

   b. Just follow the algorithm's pseudocode.

8. a. Use the formula $2^n = 2^{n-1} + 2^{n-1}$ without simplifying it; do not forget to provide a condition for stopping your recursive calls.

   b. A similar algorithm was investigated in Section 2.4.

   c. A similar question was investigated in Section 2.4.

   d. A bad efficiency class of an algorithm by itself does not mean that

the algorithm is bad. For example, the classic algorithm for the Tower of Hanoi puzzle is optimal despite its exponential-time efficiency. Therefore, a claim that a particular algorithm is not good requires a reference to a better one.

9. a. Tracing the algorithm for $n = 1$ and $n = 2$ should help.

   b. It is very similar to one of the examples discussed in the section.

10. Get the basic operation count either by solving a recurrence relation or by computing directly the number of the adjacency matix elements the algorithm checks in the worst case.

11. a. Use the definition's formula to get the recurrence relation for the number of multiplications made by the algorithm.

    b. Investigate the right-hand side of the recurrence relation. Computing the first few values of $M(n)$ may be helpful, too.

12. You might want to use the neighborhood's symmetry to obtain a simple formula for the number of squares added to the neighborhood on the nth iteration of the algorithm.

13. The minimum amount of time needed to fry three hamburgers is smaller than four minutes.

14. Solve first a simpler version in which a celebrity must be present.

Solutions to Exercises 2.4

1. a. $x(n) = x(n-1) + 5 \quad \text{for } n > 1, \quad x(1) = 0$

$$\begin{aligned} x(n) &= x(n-1) + 5 \\ &= [x(n-2) + 5] + 5 = x(n-2) + 5 \cdot 2 \\ &= [x(n-3) + 5] + 5 \cdot 2 = x(n-3) + 5 \cdot 3 \\ &= ... \\ &= x(n-i) + 5 \cdot i \\ &= ... \\ &= x(1) + 5 \cdot (n-1) = 5(n-1). \end{aligned}$$

Note: The solution can also be obtained by using the formula for the n term of the arithmetical progression:

$$x(n) = x(1) + d(n-1) = 0 + 5(n-1) = 5(n-1).$$

b. $x(n) = 3x(n-1) \quad \text{for } n > 1, \quad x(1) = 4$

$$\begin{aligned} x(n) &= 3x(n-1) \\ &= 3[3x(n-2)] = 3^2 x(n-2) \\ &= 3^2[3x(n-3)] = 3^3 x(n-3) \\ &= ... \\ &= 3^i x(n-i) \\ &= ... \\ &= 3^{n-1} x(1) = 4 \cdot 3^{n-1}. \end{aligned}$$

Note: The solution can also be obtained by using the formula for the n term of the geometric progression:

$$x(n) = x(1) q^{n-1} = 4 \cdot 3^{n-1}.$$

c. $x(n) = x(n-1) + n \quad \text{for } n > 0, \quad x(0) = 0$

$$\begin{aligned} x(n) &= x(n-1) + n \\ &= [x(n-2) + (n-1)] + n = x(n-2) + (n-1) + n \\ &= [x(n-3) + (n-2)] + (n-1) + n = x(n-3) + (n-2) + (n-1) + n \\ &= ... \\ &= x(n-i) + (n-i+1) + (n-i+2) + \cdots + n \\ &= ... \\ &= x(0) + 1 + 2 + \cdots + n = \frac{n(n+1)}{2}. \end{aligned}$$

d. $x(n) = x(n/2) + n \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 2^k)$

$$\begin{aligned}
x(2^k) &= x(2^{k-1}) + 2^k \\
&= [x(2^{k-2}) + 2^{k-1}] + 2^k = x(2^{k-2}) + 2^{k-1} + 2^k \\
&= [x(2^{k-3}) + 2^{k-2}] + 2^{k-1} + 2^k = x(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k \\
&= ... \\
&= x(2^{k-i}) + 2^{k-i+1} + 2^{k-i+2} + \cdots + 2^k \\
&= ... \\
&= x(2^{k-k}) + 2^1 + 2^2 + \cdots + 2^k = 1 + 2^1 + 2^2 + \cdots + 2^k \\
&= 2^{k+1} - 1 = 2 \cdot 2^k - 1 = 2n - 1.
\end{aligned}$$

e. $x(n) = x(n/3) + 1 \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 3^k)$

$$\begin{aligned}
x(3^k) &= x(3^{k-1}) + 1 \\
&= [x(3^{k-2}) + 1] + 1 = x(3^{k-2}) + 2 \\
&= [x(3^{k-3}) + 1] + 2 = x(3^{k-3}) + 3 \\
&= ... \\
&= x(3^{k-i}) + i \\
&= ... \\
&= x(3^{k-k}) + k = x(1) + k = 1 + \log_3 n.
\end{aligned}$$

2. $C(n) = C(n-1) + 1, \quad C(0) = 1$ (there is a call but no multiplications when $n = 0$).

$$\begin{aligned}
C(n) &= C(n-1) + 1 = [C(n-2) + 1] + 1 = C(n-2) + 2 = ... \\
&= C(n-i) + i = ... = C(0) + n = 1 + n.
\end{aligned}$$

3. a. Let $M(n)$ be the number of multiplications made by the algorithm. We have the following recurrence relation for it:

$$M(n) = M(n-1) + 2, \quad M(1) = 0.$$

We can solve it by backward substitutions:

$$\begin{aligned} M(n) &= M(n-1)+2 \\ &= [M(n-2)+2]+2 = M(n-2)+2+2 \\ &= [M(n-3)+2]+2+2 = M(n-3)+2+2+2 \\ &= \ldots \\ &= M(n-i)+2i \\ &= \ldots \\ &= M(1)+2(n-1) = 2(n-1). \end{aligned}$$

b. Here is a pseudocode for the nonrecursive option:

Algorithm *NonrecS*(n)
//Computes the sum of the first n cubes nonrecursively
//Input: A positive integer n
//Output: The sum of the first n cubes.
$S \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
    $S \leftarrow S + i * i * i$
return S

The number of multiplications made by this algorithm will be

$$\sum_{i=2}^{n} 2 = 2\sum_{i=2}^{n} 1 = 2(n-1).$$

This is exactly the same number as in the recursive version, but the nonrecursive version doesn't carry the time and space overhead associated with the recursion's stack.

4. a. $Q(n) = Q(n-1) + 2n - 1 \quad \text{for } n > 1, \quad Q(1) = 1.$

Computing the first few terms of the sequence yields the following:
$Q(2) = Q(1) + 2 \cdot 2 - 1 = 1 + 2 \cdot 2 - 1 = 4;$
$Q(3) = Q(2) + 2 \cdot 3 - 1 = 4 + 2 \cdot 3 - 1 = 9;$
$Q(4) = Q(3) + 2 \cdot 4 - 1 = 9 + 2 \cdot 4 - 1 = 16.$

Thus, it appears that $Q(n) = n^2$. We'll check this hypothesis by substituting this formula into the recurrence equation and the initial condition. The left hand side yields $Q(n) = n^2$. The right hand side yields

$$Q(n-1) + 2n - 1 = (n-1)^2 + 2n - 1 = n^2.$$

The initial condition is verified immediately: $Q(1) = 1^2 = 1$.

b. $M(n) = M(n-1) + 1$ for $n > 1$, $M(1) = 0$. Solving it by backward substitutions (it's almost identical to the factorial example—see Example 1 in the section) or by applying the formula for the nth term of an arithmetical progression yields $M(n) = n - 1$.

c. Let $C(n)$ be the number of additions and subtractions made by the algorithm. The recurrence for $C(n)$ is $C(n) = C(n-1) + 3$ for $n > 1$, $C(1) = 0$. Solving it by backward substitutions or by applying the formula for the nth term of an arithmetical progression yields $C(n) = 3(n-1)$.

Note: If we don't include in the count the subtractions needed to decrease n, the recurrence will be $C(n) = C(n-1) + 2$ for $n > 1$, $C(1) = 0$. Its solution is $C(n) = 2(n-1)$.

5. a. The number of moves is given by the formula: $M(n) = 2^n - 1$. Hence

$$\frac{2^{64} - 1}{60 \cdot 24 \cdot 365} \approx 3.5 \cdot 10^{13} \text{ years}$$

vs. the age of the Universe estimated to be about $13 \cdot 10^9$ years.

b. Observe that for every move of the ith disk, the algorithm first moves the tower of all the disks smaller than it to another peg (this requires one move of the $(i+1)$st disk) and then, after the move of the ith disk, this smaller tower is moved on the top of it (this again requires one move of the $(i+1)$st disk). Thus, for each move of the ith disk, the algorithm moves the $(i+1)$st disk exactly twice. Since for $i = 1$, the number of moves is equal to 1, we have the following recurrence for the number of moves made by the ith disk:

$$m(i+1) = 2m(i) \quad \text{for } 1 \le i < n, \quad m(1) = 1.$$

Its solution is $m(i) = 2^{i-1}$ for $i = 1, 2, ..., n$. (The easiest way to obtain this formula is to use the formula for the generic term of a geometric progression.) Note that the answer agrees nicely with the formula for the total number of moves:

$$M(n) = \sum_{i=1}^{n} m(i) = \sum_{i=1}^{n} 2^{i-1} = 1 + 2 + \cdots + 2^{n-1} = 2^n - 1.$$

6. If $n = 1$, move the single disk from peg A first to peg B and then from peg B to peg C. If $n > 1$, do the following:
transfer recursively the top $n - 1$ disks from peg A to peg C through peg B

move the disk from peg A to peg B
transfer recursively $n-1$ disks from peg C to peg A through peg B
move the disk from peg B to peg C
transfer recursively $n-1$ disks from peg A to peg C through peg B.

The recurrence relation for the number of moves $M(n)$ is

$$M(n) = 3M(n-1) + 2 \text{ for } n > 1, \quad M(1) = 2.$$

It can be solved by backward substitutions as follows

$$\begin{aligned} M(n) &= 3M(n-1) + 2 \\ &= 3[3M(n-2) + 2] + 2 = 3^2 M(n-2) + 3 \cdot 2 + 2 \\ &= 3^2[3M(n-3) + 2] + 3 \cdot 2 + 2 = 3^3 M(n-3) + 3^2 \cdot 2 + 3 \cdot 2 + 2 \\ &= ... \\ &= 3^i M(n-i) + 2(3^{i-1} + 3^{i-2} + \cdots + 1) = 3^i M(n-i) + 3^i - 1 \\ &= ... \\ &= 3^{n-1} M(1) + 3^{n-1} - 1 = 3^{n-1} \cdot 2 + 3^{n-1} - 1 = 3^n - 1. \end{aligned}$$

7. a. We'll verify by substitution that $A(n) = \lfloor \log_2 n \rfloor$ satisfies the recurrence for the number of additions

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for every } n > 1.$$

Let n be even, i.e., $n = 2k$.
The left-hand side is:
$A(n) = \lfloor \log_2 n \rfloor = \lfloor \log_2 2k \rfloor = \lfloor \log_2 2 + \log_2 k \rfloor = (1 + \lfloor \log_2 k \rfloor) = \lfloor \log_2 k \rfloor + 1$.
The right-hand side is:
$A(\lfloor n/2 \rfloor) + 1 = A(\lfloor 2k/2 \rfloor) + 1 = A(k) + 1 = \lfloor \log_2 k \rfloor + 1$.

Let n be odd, i.e., $n = 2k + 1$.
The left-hand side is:
$A(n) = \lfloor \log_2 n \rfloor = \lfloor \log_2 (2k+1) \rfloor =$ using $\lfloor \log_2 x \rfloor = \lceil \log_2 (x+1) \rceil - 1$
$\lceil \log_2 (2k+2) \rceil - 1 = \lceil \log_2 2(k+1) \rceil - 1$
$= \lceil \log_2 2 + \log_2 (k+1) \rceil - 1 = 1 + \lceil \log_2 (k+1) \rceil - 1 = \lfloor \log_2 k \rfloor + 1$.
The right-hand side is:
$A(\lfloor n/2 \rfloor) + 1 = A(\lfloor (2k+1)/2 \rfloor) + 1 = A(\lfloor k + 1/2 \rfloor) + 1 = A(k) + 1 = \lfloor \log_2 k \rfloor + 1$.

The initial condition is verified immediately: $A(1) = \lfloor \log_2 1 \rfloor = 0$.

b. The recurrence relation for the number of additions is identical to the one for the recursive version:

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad A(1) = 0,$$

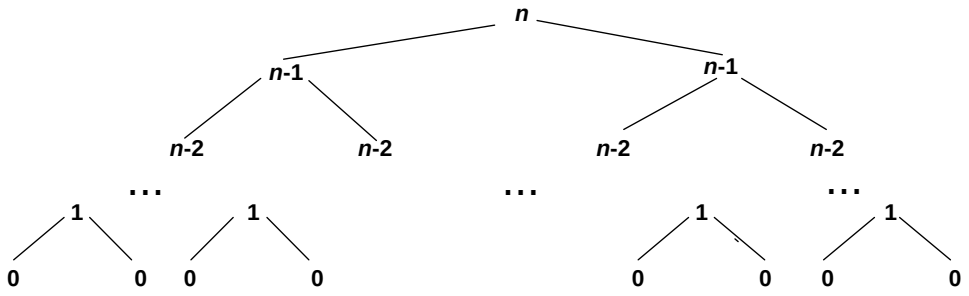
with the solution $A(n) = \lfloor \log_2 n \rfloor + 1$.

8. a. **Algorithm** *Power*(n)
//Computes 2^n recursively by the formula $2^n = 2^{n-1} + 2^{n-1}$
//Input: A nonnegative integer n
//Output: Returns 2^n
if $n = 0$ **return** 1
else return *Power*$(n-1)$ + *Power*$(n-1)$

b. $A(n) = 2A(n-1) + 1, \ \ A(0) = 0.$

$$\begin{aligned} A(n) &= 2A(n-1) + 1 \\ &= 2[2A(n-2) + 1] + 1 = 2^2 A(n-2) + 2 + 1 \\ &= 2^2[2A(n-3) + 1] + 2 + 1 = 2^3 A(n-3) + 2^2 + 2 + 1 \\ &= \ldots \\ &= 2^i A(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 1 \\ &= \ldots \\ &= 2^n A(0) + 2^{n-1} + 2^{n-2} + \cdots + 1 = 2^{n-1} + 2^{n-2} + \cdots + 1 = 2^n - 1. \end{aligned}$$

c. The tree of recursive calls for this algorithm looks as follows:



Note that it has one extra level compared to the similar tree for the Tower of Hanoi puzzle.

d. It's a very bad algorithm because it is vastly inferior to the algorithm that simply multiplies an accumulator by 2 n times, not to mention much more efficient algorithms discussed later in the book. Even if only additions are allowed, adding two 2^{n-1} times is better than this algorithm.

9. a. The algorithm computes the value of the smallest element in a given array.

   b. The recurrence for the number of key comparisons is

$$C(n) = C(n-1) + 1 \quad \text{for } n > 1, \quad C(1) = 0.$$

   Solving it by backward substitutions yields $C(n) = n - 1$.

10. Let $C_w(n)$ be the number of times the adjacency matrix element is checked in the worst case (the graph is complete). We have the following recurrence for $C_w(n)$

$$C_w(n) = C_w(n-1) + n - 1 \quad \text{for } n > 1, \quad C(1) = 0.$$

    Solving the recurrence by backward substitutions yields the following:

$$\begin{aligned} C_w(n) &= C_w(n-1) + n - 1 \\ &= [C_w(n-2) + n - 2] + n - 1 \\ &= [C_w(n-3) + n - 3] + n - 2 + n - 1 \\ &= \ldots \\ &= C_w(n-i) + (n-i) + (n-i+1) + \cdots + (n-1) \\ &= \ldots \\ &= C_w(1) + 1 + 2 + \cdots + (n-1) = 0 + (n-1)n/2 = (n-1)n/2. \end{aligned}$$

    This result could also be obtained directly by observing that in the worst case the algorithm checks every element below the main diagonal of the adjacency matrix of a given graph.

11. a. Let $M(n)$ be the number of multiplications made by the algorithm based on the formula $\det A = \sum_{j=0}^{n-1} s_j a_{0j} \det A_j$. If we don't include multiplications by s_j, which are just ± 1, then

$$M(n) = \sum_{j=0}^{n-1} (M(n-1) + 1),$$

    i.e.,

$$M(n) = n(M(n-1) + 1) \quad \text{for } n > 1 \quad \text{and} \quad M(1) = 0.$$

    b. Since $M(n) = nM(n-1) + n$, the sequence $M(n)$ grows to infinity at least as fast as the factorial function defined by $F(n) = nF(n-1)$.

12. The number of squares added on the nth iteration to each of the four symmertic sides of the von Neumann neighborhood is equal to n. Hence we obtain the following recurrence for $S(n)$, the total number of squares in the neighborhood after the nth iteration:

$$S(n) = S(n-1) + 4n \quad \text{for } n > 0 \quad \text{and} \quad S(0) = 1.$$

Solving the recurrence by backward substitutions yields the following:

$$\begin{aligned}
S(n) &= S(n-1) + 4n \\
&= [S(n-2) + 4(n-1)] + 4n = S(n-2) + 4(n-1) + 4n \\
&= [S(n-3) + 4(n-2)] + 4(n-1) + 4n = S(n-3) + 4(n-2) + 4(n-1) + 4n \\
&= ... \\
&= S(n-i) + 4(n-i+1) + 4(n-i+2) + \cdots + 4n \\
&= ... \\
&= S(0) + 4 \cdot 1 + 4 \cdot 2 + \cdots + 4n = 1 + 4(1 + 2 + \cdots + n) \\
&= 1 + 4n(n+1)/2 = 2n^2 + 2n + 1.
\end{aligned}$$

13. a. Let $T(n)$ be the number of minutes needed to fry n hamburgers by the algorithm given. Then we have the following recurrence for $T(n)$:

$$T(n) = T(n-2) + 2 \ \text{ for } n > 2, \quad T(1) = 2, \ \ T(2) = 2.$$

Its solution is $T(n) = n$ for every even $n > 0$ and $T(n) = n + 1$ for every odd $n > 0$ can be obtained either by backward substitutitons or by applying the formula for the generic term of an arithmetical progression.

b. The algorithm fails to execute the task of frying n hamburgers in the minimum amount of time for any odd $n > 1$. In particular, it requires $T(3) = 4$ minutes to fry three hamburgers, whereas one can do this in 3 minutes: First, fry pancakes 1 and 2 on one side. Then fry pancake 1 on the second side together with pancake 3 on its first side. Finally, fry both pancakes 2 and 3 on the second side.

c. If $n \leq 2$, fry the hamburger (or the two hamburgers together if $n = 2$) on each side. If $n = 3$, fry the pancakes in 3 minutes as indicated in the answer to the part b question. If $n > 3$, fry two hamburgers together on each side and then fry the remaining $n - 2$ hamburgers by the same algorithm. The recurrence for the number of minutes needed to fry n hamburgers looks now as follows:

$$T(n) = T(n-2) + 2 \ \text{ for } n > 3, \quad T(1) = 2, \ \ T(2) = 2, \ \ T(3) = 3.$$

For every $n > 1$, this algorithm requires n minutes to do the job. This is the minimum time possible because n pancakes have $2n$ sides to be fried

and any algorithm can fry no more than two sides in one minute. The algorithm is also obviously optimal for the trivial case of $n = 1$, requiring two minutes to fry a single hamburger on both sides.

Note: The case of $n = 3$ is a well-known puzzle, which dates back at least to 1943. Its algorithmic version for an arbitrary n is included in *Algorithmic Puzzles* by A. Levitin and M. Levitin, Oxford University Press, 2011, Problem 16.

14. The problem can be solved by a recursive algorithm. Indeed, by asking just one question, we can eliminate the number of people who can be a celebrity by 1, solve the problem for the remaining group of $n - 1$ people recursively, and then verify the returned solution by asking no more than two questions. Here is a more detailed description of this algorithm:

    If $n = 1$, return that one person as a celebrity. If $n > 1$, proceed as follows:

Step 1 Select two people from the group given, say, A and B, and ask A whether A knows B. If A knows B, remove A from the remaining people who can be a celebrity; if A doesn't know B, remove B from this group.

Step 2 Solve the problem recursively for the remaining group of $n - 1$ people who can be a celebrity.

Step 3 If the solution returned in Step 2 indicates that there is no celebrity among the group of $n - 1$ people, the larger group of n people cannot contain a celebrity either. If Step 2 identified as a celebrity a person other than either A or B, say, C, ask whether C knows the person removed in Step 1 and, if the answer is no, whether the person removed in Step 1 knows C. If the answer to the second question is yes," return C as a celebrity and "no celebrity" otherwise. If Step 2 identified B as a celebrity, just ask whether B knows A: return B as a celebrity if the answer is no and "no celebrity" otherwise. If Step 2 identified A as a celebrity, ask whether B knows A: return A as a celebrity if the answer is yes and "no celebrity" otherwise.

The recurrence for $Q(n)$, the number of questions needed in the worst case, is as follows:

$$Q(n) = Q(n-1) + 3 \quad \text{for } n > 2, \quad Q(2) = 2, \quad Q(1) = 0.$$

Its solution is $Q(n) = 2 + 3(n-2)$ for $n > 1$ and $Q(1) = 0$.

Note: A discussion of this problem, including an implementation of this algorithm in a Pascal-like pseudocode, can be found in Udi Manber's *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, 1989.

Exercises 2.5

1. Find a Web site dedicated to applications of the Fibonacci numbers and study it.

2. *Fibonacci's rabbits problem* A man put a pair of rabbits in a place surrounded by a wall. How many pairs of rabbits will be there in a year if the initial pair of rabbits (male and female) are newborn, and all rabbit pairs are not fertile during their first month of life but thereafter give birth to one new male/female pair at the end of every month?

3. *Climbing stairs* Find the number of different ways to climb an n-stair staircase if each step is either or two stairs. For example, a 3-stair staircase can be climbed three ways: 1-1-1, 1-2, and 2-1.

4. How many even numbers are there among the first n Fibonacci numbers? Give a closed-form formula valid for every $n > 0$.

5. Check by direct substitutions that the function $\frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$ indeed satisfies recurrence (2.6) and initial conditions (2.7).

6. The maximum values of the Java primitive types `int` and `long` are $2^{31} - 1$ and $2^{63} - 1$, respectively. Find the smallest n for which the nth Fibonacci number is not going to fit in a memory allocated for
   a. the type `int`. b. the type `long`.

7. Consider the recursive definition-based algorithm for computing the nth Fibonacci number $F(n)$. Let $C(n)$ and $Z(n)$ be the number of times $F(1)$ and $F(0)$, respectively, are computed. Prove that
   a. $C(n) = F(n)$. b. $Z(n) = F(n-1)$.

8. Improve algorithm *Fib* of the text so that it requires only $\Theta(1)$ space.

9. Prove the equality

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \quad \text{for } n \geq 1.$$

10. $\triangleright$ How many modulo divisions are made by Euclid's algorithm on two consecutive Fibonacci numbers $F(n)$ and $F(n-1)$ as the algorithm's input?

11. *Dissecting a Fibonacci rectangle* Given a rectangle whose sides are two consecutive Fibonacci numbers, design an algorithm to dissect it into squares with no more than two of the squares be of the same size. What is the time efficiency class of your algorithm?

12. In the language of your choice, implement two algorithms for computing the last five digits of the nth Fibonacci number that are based on (a) the recursive definition-based algorithm $F(n)$; (b) the iterative definition-based algorithm $Fib(n)$. Perform an experiment to find the largest value of n for which your programs run under 1 minute on your computer.

Hints to Exercises 2.5

1. Use a search engine.
2. Set up an equation expressing the number of rabbits after n months in terms of the number of rabbits in some previous months.
3. There are several ways to solve this problem. The most elegant of them makes it possible to put the problem in this section.
4. Writing down the first, say, ten Fibonacci numbers makes the pattern obvious.
5. It is easier to substitute ϕ^n and $\hat{\phi}^n$ into the recurrence equation separately. Why will this suffice?
6. Use an approximate formula for $F(n)$ to find the smallest values of n to exceed the numbers given.
7. Set up the recurrence relations for $C(n)$ and $Z(n)$, with appropriate initial conditions, of course.
8. All the information needed on each iteration of the algorithm is the values of the last two consecutive Fibonacci numbers. Modify the algorithm to take advantage of this fact.
9. Prove it by mathematical induction.
10. Consider first a small example such as computing $\gcd(13, 8)$.
11. Take advantage of the special nature of the rectangle's dimensions.
12. The last k digits of an integer N can be obtained by computing $N \bmod 10^k$. Performing all operations of your algorithms modulo 10^k (see Appendix A) will enable you to circumvent the exponential growth of the Fibonacci numbers. Also note that Section 2.6 is devoted to a general discussion of the empirical analysis of algorithms.

Solutions to Exercises 2.5

1. n/a

2. Let $R(n)$ be the number of rabbit pairs at the end of month n. Clearly, $R(0) = 1$ and $R(1) = 1$. For every $n > 1$, the number of rabbit pairs, $R(n)$, is equal to the number of pairs at the end of month $n-1$, $R(n-1)$, plus the number of rabbit pairs born at the end of month n, which is according to the problem's assumptions is equal to $R(n-2)$, the number of rabbit pairs at the end of month $n-2$. Thus, we have the recurrence relation

$$R(n) = R(n-1) + R(n-2) \text{ for } n > 1, \quad R(0) = 1, \ R(1) = 1.$$

The following table gives the values of the first thirteen terms of the sequence, called the *Fibonacci numbers*, defined by this recurrence relation:

n	0	1	2	3	4	5	6	7	8	9	10	11	12
$R(n)$	1	1	2	3	5	8	13	21	34	55	89	144	233

Note that $R(n)$ differs slightly from the canonical Fibonacci sequence, which is defined by the same recurrence equation $F(n) = F(n-1) + F(n-2)$ but the different initial conditions, namely, $F(0) = 0$ and $F(1) = 1$. Obviously, $R(n) = F(n+1)$ for $n \geq 0$.

Note: The problem was included by Leonardo of Pisa (aka Fibonacci) in his 1202 book *Liber Abaci*, in which he advocated usage of the Hindu-Arabic numerals.

3. Let $W(n)$ be the number of different ways to climb an n-stair staircase. $W(n-1)$ of them start with a one-stair climb and $W(n-2)$ of them start with a two-stair climb. Thus,

$$W(n) = W(n-1) + W(n-2) \quad \text{for } n \geq 3, \quad W(1) = 1, \ \ W(2) = 2.$$

Solving this recurrence either "from scratch" or better yet noticing that the solution runs one step ahead of the canonical Fibonacci sequence $F(n)$, we obtain $W(n) = F(n+1)$ for $n \geq 1$.

4. Starting with $F(0) = 0$ and $F(1) = 1$ and the rule $F(n) = F(n-1) + F(n-2)$ for every subsequent element of the sequence, it's easy to see that the Fibonacci numbers form the following pattern

even, odd, odd, even, odd, odd, ...

Hence the number of even numbers among the first n Fibonacci numbers can be obtained by the formula $\lceil n/3 \rceil$.

5. On substituting ϕ^n into the left-hand side of the equation, we obtain $F(n) - F(n-1) - F(n-2) = \phi^n - \phi^{n-1} - \phi^{n-2} = \phi^{n-2}(\phi^2 - \phi - 1) = 0$ because ϕ is one of the roots of the characteristic equation $r^2 - r - 1 = 0$. The verification of $\hat{\phi}^n$ works out for the same reason. Since the equation $F(n) - F(n-1) - F(n-2) = 0$ is homogeneous and linear, any linear combination of its solutions ϕ^n and $\hat{\phi}^n$, i.e., any sequence of the form $\alpha\phi^n + \beta\hat{\phi}^n$ will also be a solution to $F(n) - F(n-1) - F(n-2) = 0$. In particular, it will be the case for the Fibonacci sequence $\frac{1}{\sqrt{5}}\phi^n - \frac{1}{\sqrt{5}}\hat{\phi}^n$. Both initial conditions are checked out in a quite straightforward manner (but, of course, not individually for ϕ^n and $\hat{\phi}^n$).

6. a. The question is to find the smallest value of n such that $F(n) > 2^{31} - 1$. Using the formula $F(n) = \frac{1}{\sqrt{5}}\phi^n$ rounded to the nearest integer, we get (approximately) the following inequality:

$$\frac{1}{\sqrt{5}}\phi^n > 2^{31} - 1 \quad \text{or} \quad \phi^n > \sqrt{5}(2^{31} - 1).$$

After taking natural logarithms of both hand sides, we obtain

$$n > \frac{\ln(\sqrt{5}(2^{31} - 1))}{\ln \phi} \approx 46.3.$$

Thus, the answer is $n = 47$.

b. Similarly, we have to find the smallest value of n such that $F(n) > 2^{63} - 1$. Thus,

$$\frac{1}{\sqrt{5}}\phi^n > 2^{63} - 1, \quad \text{or} \quad \phi^n > \sqrt{5}(2^{63} - 1)$$

or, after taking natural logarithms of both hand sides,

$$n > \frac{\ln(\sqrt{5}(2^{63} - 1))}{\ln \phi} \approx 92.4.$$

Thus, the answer is $n = 93$.

7. Since $F(n)$ is computed recursively by the formula $F(n) = F(n-1) + F(n-2)$, the recurrence equations for $C(n)$ and $Z(n)$ will be the same as the recurrence for $F(n)$. The initial conditions will be:

$$C(0) = 0, \quad C(1) = 1 \quad \text{and} \quad Z(0) = 1, \quad Z(1) = 0$$

for $C(n)$ and $Z(n)$, respectively. Therefore, since both the recurrence equation and the initial conditions for $C(n)$ and $F(n)$ are the same, $C(n) =$

$F(n)$. As to the assertion that $Z(n) = F(n-1)$, it is easy to see that it should be the case since the sequence $Z(n)$ looks as follows:

$$1,\ 0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ ...,$$

i.e., it is the same as the Fibonacci numbers shifted one position to the right. This can be formally proved by checking that the sequence $F(n-1)$ (in which $F(-1)$ is defined as 1) satisfies the recurrence relation

$$Z(n) = Z(n-1) + Z(n-2) \quad \text{for } n > 1 \quad \text{and} \quad Z(0) = 1, \quad Z(1) = 0.$$

It can also be proved either by mathematical induction or by deriving an explicit formula for $Z(n)$ and showing that this formula is the same as the value of the explicit formula for $F(n)$ with n replaced by $n-1$.

8. **Algorithm** *Fib2(n)*
//Computes the n-th Fibonacci number using just two variables
//Input: A nonnegative integer n
//Output: The n-th Fibonacci number
$u \leftarrow 0$; $v \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
    $v \leftarrow v + u$
    $u \leftarrow v - u$
if $n = 0$ **return** 0
else return v

9. (i) The validity of the equality for $n = 1$ follows immediately from the definition of the Fibonacci sequence.
(ii) Assume that

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \text{ for a positive integer } n.$$

We need to show that then

$$\begin{bmatrix} F(n) & F(n+1) \\ F(n+1) & F(n+2) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n+1}.$$

Indeed,

$$\begin{aligned} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n+1} &= \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \\ &= \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} F(n) & F(n+1) \\ F(n+1) & F(n+2) \end{bmatrix}. \end{aligned}$$

10. The principal observation here is the fact that Euclid's algorithm replaces two consecutive Fibonacci numbers as its input by another pair of consecutive Fibonacci numbers, namely:

$$\gcd(F(n), F(n-1)) = \gcd(F(n-1), F(n-2)) \quad \text{for every } n \geq 4.$$

Indeed, since $F(n-2) < F(n-1)$ for every $n \geq 4$,

$$F(n) = F(n-1) + F(n-2) < 2F(n-1).$$

Therefore for every $n \geq 4$, the quotient and remainder of division of $F(n)$ by $F(n-1)$ are 1 and $F(n) - F(n-1) = F(n-2)$, respectively. This is exactly what we asserted at the beginning of the solution. In turn, this leads to the following recurrence for the number of of divisions $D(n)$:

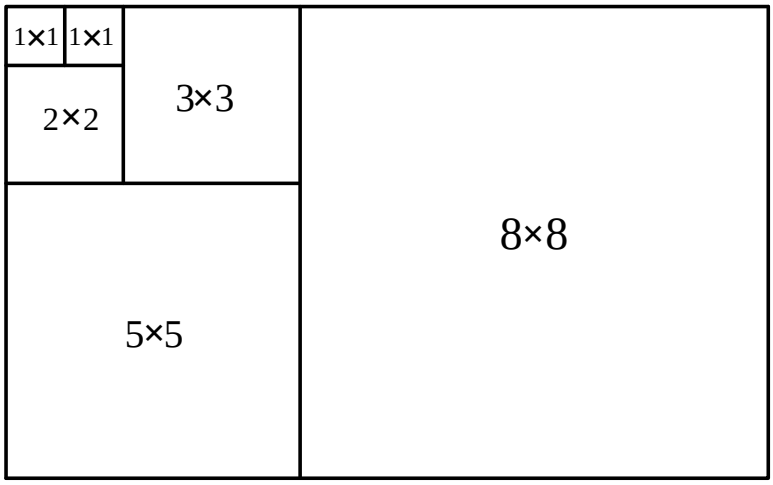
$$D(n) = D(n-1) + 1 \quad \text{for } n \geq 4, \quad D(3) = 1,$$

whose initial condition $D(3) = 1$ is obtained by tracing the algorithm on the input pair $F(3), F(2)$, i.e., 2,1. The solution to this recurrence is:

$$D(n) = n - 2 \quad \text{for every } n \geq 3.$$

(One can also easily find directly that $D(2) = 1$ and $D(1) = 0$.)

11. Given a rectangle with sides $F(n)$ and $F(n+1)$, the problem can be solved by the following recursive algorithm. If $n = 1$, the problem is already solved because the rectangle is a 1×1 square. If $n > 1$, dissect the rectangle into the $F(n) \times F(n)$ square and the rectangle with sides $F(n-1)$ and $F(n)$ and then dissect the latter by the same algorithm. The algorithm is illustrated below for the 8×13 square.



Since the algorithm dissects the rectangle with sides $F(n)$ and $F(n+1)$ into n squares—which can be formally obtained by solving the recurrence for the number of squares $S(n) = S(n-1)+1$, $S(1) = 1$—its time efficiency falls into the $\Theta(n)$ class.

12. n/a

Exercises 2.6

1. Consider the following well-known sorting algorithm (we shall study it more closely later in the book) with a counter inserted to count the number of key comparisons.

   Algorithm *SortAnalysis*$(A[0..n-1])$
   //Input: An array $A[0..n-1]$ of n orderable elements
   //Output: The total number of key comparisons made
   $count \leftarrow 0$
   for $i \leftarrow 1$ **to** $n-1$ **do**
       $v \leftarrow A[i]$
       $j \leftarrow i-1$
       while $j \geq 0$ **and** $A[j] > v$ **do**
           $count \leftarrow count + 1$
           $A[j+1] \leftarrow A[j]$
           $j \leftarrow j-1$
       $A[j+1] \leftarrow v$

   Is the comparison counter inserted in the right place? If you believe it is, prove it; if you believe it is not, make an appropriate correction.

2. a. Run the program of Problem 1, with a properly inserted counter (or counters) for the number of key comparisons, on 20 random arrays of sizes 1000, 1500, 2000, 2500,...,9000, 9500.

   b. Analyze the data obtained to form a hypothesis about the algorithm's average-case efficiency.

   c. Estimate the number of key comparisons one should expect for a randomly generated array of size 10,000 sorted by the same algorithm.

3. Repeat Problem 2 by measuring the program's running time in milliseconds.

4. Hypothesize a likely efficiency class of an algorithm based on the following empirical observations of its basic operation's count:

size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
count	11,966	24,303	39,992	53,010	67,272	78,692	91,274	113,063	129,799	140,538

5. What scale transformation will make a logarithmic scatterplot look like a linear one?

6. How can we distinguish a scatterplot for an algorithm in $\Theta(\lg \lg n)$ from a scatterplot for an algorithm in $\Theta(\lg n)$?

7. a. Find empirically the largest number of divisions made by Euclid's algorithm for computing $\gcd(m, n)$ for $1 \leq n \leq m \leq 100$.

   b. For each positive integer k, find empirically the smallest pair of integers $1 \leq n \leq m \leq 100$ for which Euclid's algorithm needs to make k divisions in order to find $\gcd(m, n)$.

8. The average-case efficiency of Euclid's algorithm on inputs of size n can be measured by the average number of divisions $D_{avg}(n)$ made by the algorithm in computing $\gcd(n, 1)$, $\gcd(n, 2)$, ..., $\gcd(n, n)$. For example,

$$D_{avg}(5) = \frac{1}{5}(1 + 2 + 3 + 2 + 1) = 1.8.$$

   Produce a scatterplot of $D_{avg}(n)$ and indicate a likely average-case efficiency class of the algorithm.

9. Run an experiment to ascertain the efficiency class of the sieve of Eratosthenes (see Section 1.1).

10. Run a timing experiment for the three algorithms for computing $\gcd(m, n)$ presented in Section 1.1.

Hints to Exercises 2.6

1. Does it return a correct comparison count for every array of size 2?
2. Debug your comparison counting and random input generating for small array sizes first.
3. On a reasonably fast desktop, you may well get zero time, at least for smaller sizes in your sample. Section 2.6 mentions a trick for overcoming this difficulty.
4. Check how fast the count values grow with doubling the size.
5. A similar question was discussed in the section.
6. Compare the values of the functions $\lg \lg n$ and $\lg n$ for $n = 2^k$.
7. Insert the division counter into a program implementing the algorithm and run it for the input pairs in the range indicated.
8. Get the empirical data for random values of n in a range of between, say, 10^2 and 10^4 or 10^5 and plot the data obtained. (You may want to use different scales for the axes of your coordinate system.)
9. n/a
10. n/a

Solutions to Exercises 2.6

1. It doesn't count the comparison $A[j] > v$ when the comparison fails (and, hence, the body of the while loop is not executed). If the language implies that the second comparison will always be executed even if the first clause of the conjunction fails, the count should be simply incremented by one either right before the **while** statement or right after the **while** statement's end. If the second clause of the conjunction is not executed after the first clause fails, we should add the line

   if $j \geq 0$ $\quad count \leftarrow count + 1$

   right after the **while** statement's end.

2. a. One should expect numbers very close to $n^2/4$ (the approximate theoretical number of key comparisons made by insertion sort on random arrays).

   b. The closeness of the ratios $C(n)/n^2$ to a constant suggests the $\Theta(n^2)$ average-case efficiency. The same conclusion can also be drawn by observing the four-fold increase in the number of key comparisons in response to doubling the array's size.

   c. $C(10,000)$ can be estimated either as $10,000^2/4$ or as $4C(5,000)$.

3. See the answers to Exercise 2. Note, however, that the timing data is inherently much less accurate and volatile than the counting data.

4. The data exhibits a behavior indicative of an $n \lg n$ algorithm.

5. If $M(n) \approx c \log n$, then the transformation $n = a^k$ $(a > 1)$ will yield $M(a^k) \approx (c \log a)k$.

6. The function $\lg \lg n$ grows much more slowly than the slow-growing function $\lg n$. Also, if we transform the plots by substitution $n = 2^k$, the plot of the former would look logarithmic while the plot of the latter would appear linear.

7. a. 9 (for $m = 89$ and $n = 55$)

   b. Two consecutive Fibonacci numbers—$m = F_{k+2}$, $n = F_{k+1}$—are the smallest pair of integers $m \geq n > 0$ that requires k comparisons for every $k \geq 2$. (This is a well-known theoretical fact established by G. Lamé (e.g., [KnuII].) For $k = 1$, the answer is F_{k+1} and F_k, which are both equal to 1.

8. The experiment should confirm the known theoretical result: the average-case efficiency of Euclid's algorithm is in $\Theta(\lg n)$. For a slightly different metric $T(n)$ investigated by D. Knuth, $T(n) \approx \frac{12 \ln 2}{\pi^2} \ln n \approx 0.843 \ln n$ (see [KnuII], Section 4.5.3).

9. n/a

10. n/a