Solutions to Exercises 1.1

1. Al-Khwarizmi (9th century C.E.) was a great Arabic scholar, most famous for his algebra textbook. In fact, the word "algebra" is derived from the Arabic title of this book while the word "algorithm" is derived from a translation of Al-Khwarizmi's last name (see, e.g., [KnuI, pp. 1-2], [Knu96, pp. 88-92, 114]).

2. This legal issue has yet to be settled. The current legal state of affairs distinguishes mathematical algorithms, which are not patentable, from other algorithms, which may be patentable if implemented as computer programs (e.g., [Cha00]).

3. n/a

4. A straightforward algorithm that does not rely on the availability of an approximate value of $\sqrt{n}$ can check the squares of consecutive positive integers until the first square exceeding n is encountered. The answer will be the number's immediate predecessor. Note: A much faster algorithm for solving this problem can be obtained by using Newton's method (see Sections 11.4 and 12.4).

5. Initialize the list of common elements to empty. Starting with the first elements of the lists given, repeat the following until one of the lists becomes empty. Compare the current elements of the two lists: if they are equal, add this element to the list of common elements and move to the next elements of both lists (if any); otherwise, move to the element following the smaller of the two involved in the comparison.
   The maximum number of comparisons, which is made by this algorithm on some lists with no common elements such as the first m positive odd numbers and the first n positive even numbers, is equal to $m + n - 1$.

6. a. $\gcd(31415, 14142) = \gcd(14142, 3131) = \gcd(3131, 1618) =$
   $\gcd(1618, 1513) = \gcd(1513, 105) = \gcd(1513, 105) = \gcd(105, 43) =$
   $\gcd(43, 19) = \gcd(19, 5) = \gcd(5, 4) = \gcd(4, 1) = \gcd(1, 0) = 1.$

   b. To answer the question, we need to compare the number of divisions the algorithms make on the input given. The number of divisions made by Euclid's algorithm is 11 (see part a). The number of divisions made by the consecutive integer checking algorithm on each of its 14142 iterations is either 1 and 2; hence the total number of multiplications is between $1 \cdot 14142$ and $2 \cdot 14142$. Therefore, Euclid's algorithm will be between $1 \cdot 14142/11 \approx 1300$ and $2 \cdot 14142/11 \approx 2600$ times faster.

7. Let us first prove that if d divides two integers u and v, it also divides both $u+v$ and $u-v$. By definition of division, there exist integers s and t such that $u = sd$ and $v = td$. Therefore

$$u \pm v = sd \pm td = (s \pm t)d,$$

i.e., d divides both $u+v$ and $u-v$.

Also note that if d divides u, it also divides any integer multiple ku of u. Indeed, since d divides u, $u = sd$. Hence

$$ku = k(sd) = (ks)d,$$

i.e., d divides ku.

Now we can prove the assertion in question. For any pair of positive integers m and n, if d divides both m and n, it also divides both n and $r = m \bmod n = m - qn$. Similarly, if d divides both n and $r = m \bmod n = m - qn$, it also divides both $m = r + qn$ and n. Thus, the two pairs (m, n) and (n, r) have the same finite nonempty set of common divisors, including the largest element in the set, i.e., $\gcd(m, n) = \gcd(n, r)$.

8. For any input pair m, n such that $0 \le m < n$, Euclid's algorithm simply swaps the numbers on the first iteration:

$$\gcd(m, n) = \gcd(n, m)$$

because $m \bmod n = m$ if $m < n$. Such a swap can happen only once since $\gcd(m, n) = \gcd(n, m \bmod n)$ implies that the first number of the new pair (n) will be greater than its second number $(m \bmod n)$ after every iteration of the algorithm.

9. a. For any input pair $m \ge n \ge 1$, in which m is a multiple of n, Euclid's algorithm makes exactly one division; it is the smallest number possible for two positive numbers.

b. The answer is 5 divisions, which is made by Euclid's algorithm in computing $\gcd(5, 8)$. It is not too time consuming to get this answer by examining the number of divisions made by the algorithm on all input pairs $1 < m < n \le 10$.
Note: A pertinent general result (see [KnuII, p. 360]) is that for any input pair m, n where $0 \le n < N$, the number of divisions required by Euclid's algorithm to compute $\gcd(m, n)$ is at most $\lfloor \log_\phi (3 - \phi) N) \rfloor$ where $\phi = (1 + \sqrt{5})/2$.

10. a. Here is a nonrecursive version:
Algorithm *Euclid2*(m, n)
//Computes gcd(m, n) by Euclid's algorithm based on subtractions
//Input: Two nonnegative integers m and n not both equal to 0
//Output: The greatest common divisor of m and n
while $n \neq 0$ **do**
    if $m < n$ $swap(m, n)$
        $m \leftarrow m - n$
return m

b. It is not too difficult to prove that the integers that can be written on the board are the integers generated by the subtraction version of Euclid's algorithm and only them. Although the order in which they appear on the board may vary, their total number always stays the same: It is equal to $m/\gcd(m, n)$, where m is the maximum of the initial numbers, which includes two integers of the initial pair. Hence, the total number of possible moves is $m/\gcd(m, n) - 2$. Consequently, if $m/\gcd(m, n)$ is odd, one should choose to go first; if it is even, one should choose to go second.

11. n/a

12. Since all the doors are initially closed, a door will be open after the last pass if and only if it is toggled an odd number of times. Door i $(1 \leq i \leq n)$ is toggled on pass j $(1 \leq j \leq n)$ if and only if j divides i. Hence, the total number of times door i is toggled is equal to the number of its divisors. Note that if j divides i, i.e. $i = jk$, then k divides i too. Hence all the divisors of i can be paired (e.g., for $i = 12$, such pairs are 1 and 12, 2 and 6, 3 and 4) unless i is a perfect square (e.g., for $i = 16$, 4 does not have another divisor to be matched with). This implies that i has an odd number of divisors if and only if it is a perfect square, i.e., $i = j^2$. Hence doors that are in the positions that are perfect squares and only such doors will be open after the last pass. The total number of such positions not exceeding n is equal to $\lfloor \sqrt{n} \rfloor$: these numbers are the squares of the positive integers between 1 and $\lfloor \sqrt{n} \rfloor$ inclusively.

Solutions to Exercises 1.2

1. Let `P`, `w`, `g`, and `c` stand for the peasant, wolf, goat, and cabbage head, respectively. The following is one of the two principal sequences that solve the problem:

	`P g`	`g`	`Pwg`	`w`	`Pw c`	`w c`	`Pwgc`
`Pwgc`	`w c`	`Pw c`	`c`	`P gc`	`g`	`P g`	

   Note: This problem is revisited later in the book (see Section 6.6).

2. Let 1, 2, 5, 10 be labels representing the men of the problem, f represent the flashlight's location, and the number in the parenthesis be the total amount of time elapsed. The following sequence of moves solves the problem:

	f,1,2	2	f,2,5,10	5,10	f,1,2,5,10
(0)	(2)	(3)	(13)	(15)	(17)
f,1,2,5,10	5,10	f,1,5,10	1	f,1,2	

3. a. The formula can be considered an algorithm if we assume that we know how to compute the square root of an arbitrary positive number.

   b. The difficulty here lies in computing $\sin A$. Since the formula says nothing about how it has to be computed, it should not be considered an algorithm. This is true even if we assume, as we did for the square root function, that we know how to compute the sine of a given angle. (There are several algorithms for doing this but only approximately, of course.) The problem is that the formula says nothing about how to compute angle A either.

   c. The formula says nothing about how to compute h_a.

4. **Algorithm** *Quadratic*(a, b, c)
   //The algorithm finds real roots of equation $ax^2 + bx + c = 0$
   //Input: Real coefficients a, b, c
   //Output: The real roots of the equation or a message about their absence
   if $a \neq 0$
      $D \leftarrow b * b - 4 * a * c$
      if $D > 0$
         $temp \leftarrow 2 * a$
         $x1 \leftarrow (-b + sqrt(D))/temp$
         $x2 \leftarrow (-b - sqrt(D))/temp$

return $x1, x2$
else if $D = 0$ **return** $-b/(2 * a)$
else return '*no real roots*'
else $//a = 0$
if $b \neq 0$ **return** $-c/b$
else $//a = b = 0$
if $c = 0$ **return** '*all real numbers*'
else return '*no real roots*'

Note: See a more realistic algorithm for this problem in Section 11.4.

5. a. Divide the given number n by 2: the remainder r_n (0 or 1) will be the next (from right to left) digit of the binary representation in question. Replace n by the quotient of the last division and repeat this operation until n becomes 0.

   b. **Algorithm** *Binary*(n)
   //The algorithm implements the standard method for finding
   //the binary expansion of a positive decimal integer
   //Input: A positive decimal integer n
   //Output: The list $b_k\ b_{k-1}...b_1\ b_0$ of n's binary digits
   $k \leftarrow 0$
   while $n \neq 0$
   $b_k \leftarrow n \bmod 2$
   $n \leftarrow \lfloor n/2 \rfloor$
   $k \leftarrow k + 1$

6. n/a

7. a. π, as an irrational number, can be computed only approximately.

   b. It is natural to consider, as an instance of this problem, computing π's value with a given level of accuracy, say, with n correct decimal digits. With this interpretation, the problem has infinitely many instances.

8. n/a

9. The following improved version considers the same pair of elements only once and avoids recomputing the same expression in the innermost loop:

Algorithm *MinDistance2*$(A[0..n-1])$
//Input: An array $A[0..n-1]$ of numbers
//Output: The minimum distance d between two of its elements

```
dmin ← ∞
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        temp ← |A[i] − A[j]|
        if temp < dmin
            dmin ← temp
return dmin
```

A faster algorithm is based on the idea of presorting (see Section 6.1).

10. Pólya's general four-point approach is:
    1. Understand the problem
    2. Devise a plan
    3. Implement the plan
    4. Look back/check

Solutions to Exercises 1.3

1. a. Sorting 60, 35, 81, 98, 14, 47 by comparison counting will work as follows:

Array $A[0..5]$		60	35	81	98	14	47
Initially	$Count[]$	0	0	0	0	0	0
After pass $i = 0$	$Count[]$	3	0	1	1	0	0
After pass $i = 1$	$Count[]$		1	2	2	0	1
After pass $i = 2$	$Count[]$			4	3	0	1
After pass $i = 3$	$Count[]$				5	0	1
After pass $i = 4$	$Count[]$					0	2
Final state	$Count[]$	3	1	4	5	0	2
Array $S[0..5]$		14	35	47	60	81	98

   b. The algorithm is not stable. Consider, as a counterexample, the result of its application to $1'$, $1''$.

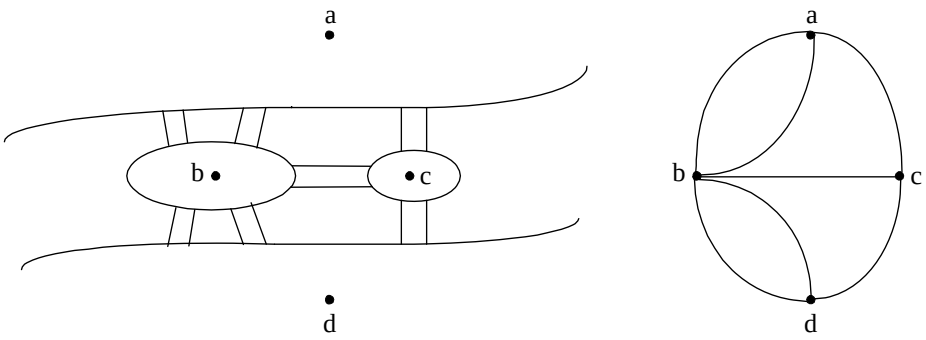
   c. The algorithm is not in place because it uses two extra arrays of size n: $Count$ and S.

2. Answers may vary but most students should be familiar with sequential search, binary search, binary tree search and, possibly, hashing from their introductory programming courses.

3. Align the pattern with the beginning of the text. Compare the corresponding characters of the pattern and the text left-to right until either all the pattern characters are matched (then stop—the search is successful) or the algorithm runs out of the text's characters (then stop—the search is unsuccessful) or a mismatching pair of characters is encountered. In the latter case, shift the pattern one position to the right and resume the comparisons.

4. a. If we represent each of the river's banks and each of the two islands by

vertices and the bridges by edges, we will get the following graph:

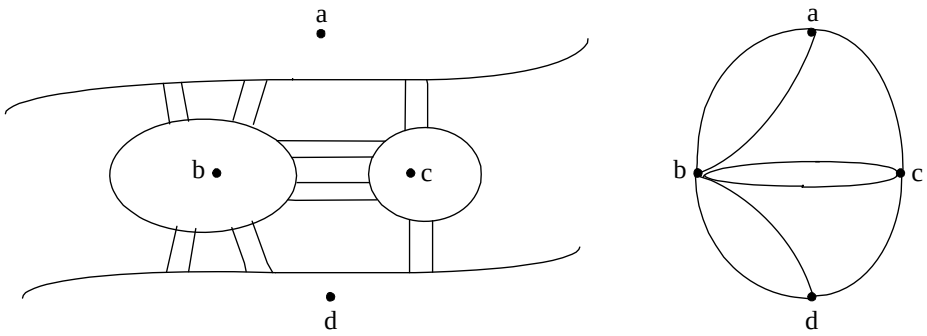


(This is, in fact, a multigraph, not a graph, because it has more than one edge between the same pair of vertices. But this doesn't matter for the issue at hand.) The question is whether there exists a path (i.e., a sequence of adjacent vertices) in this multigraph that traverses all the edges exactly once and returns to a starting vertex. Such paths are called *Eulerian circuits*; if a path traverses all the edges exactly once but does not return to its starting vertex, it is called an *Eulerian path.*

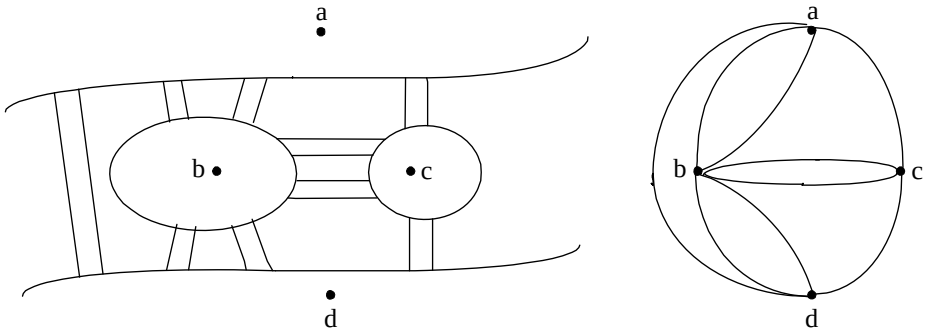
b. Euler proved that an Eulerian circuit exists in a connected (multi)graph if and only if all its vertices have even degrees, where the degree of a vertex is defined as the number of edges for which it is an endpoint. Also, an Eulerian path exists in a connected (multi)graph if and only if it has exactly two vertices of odd degrees; such a path must start at one of those two vertices and end at the other. Hence, for the multigraph of the puzzle, there exists neither an Eulerian circuit nor an Eulerian path because all its four vertices have odd degrees.

If we are to be satisfied with an Eulerian path, two of the multigraph's vertices must be made even. This can be accomplished by adding one new bridge connecting the same places as the existing bridges. For example, a new bridge between the two islands would make possible, among others,

the walk $a-b-c-a-b-d-c-b-d$.

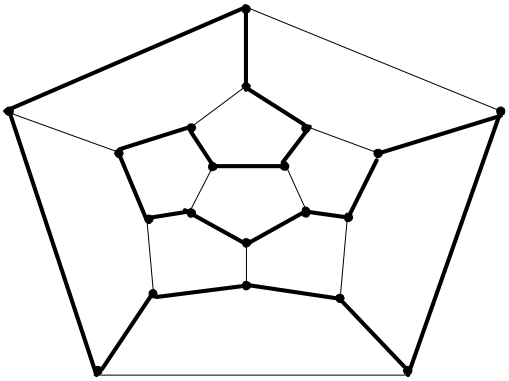


If we want a walk that returns to its starting point, all the vertices in the corresponding multigraph must be even. Since a new bridge/edge changes the parity of two vertices, at least two new bridges/edges will be needed. For example, here is one such "enhancement":



This would make possible $a-b-c-a-b-d-c-b-d-a$, among several other such walks.

5. A Hamiltonian circuit is marked on the graph below:



6. a. At least three "reasonable"criteria come to mind: the fastest trip, a trip with the smallest number of train stops, and a trip that requires the smallest number of train changes. Note that the first criterion requires the information about the expected traveling time between stations and the time needed for train changes whereas the other two criteria do not require such information.

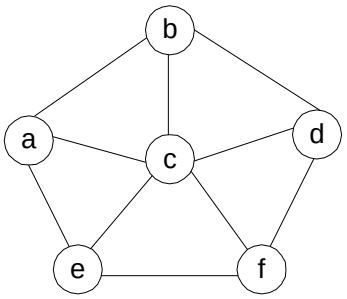
   b. A natural approach is to mimic subway plans by representing stations by vertices of a graph, with two vertices connected by an edge if there is a train line between the corresponding stations. If the time spent on changing a train is to be taken into account (e.g., because the station in question is on more than one line), the station should be represented by more then one vertex.

7. a. Find a permutation of n given cities for which the sum of the distances between consecutive cities in the permutation plus the distance between its last and first city is as small as possible.

   b. Partition all the graph's vertices into the smallest number of disjoint subsets so that there is no edge connecting vertices from the same subset.

8. a. Create a graph whose vertices represent the map's regions and the edges connect two vertices if and only if the corresponding regions have a common border (and therefore cannot be colored the same color). Here

is the graph for the map given:



Solving the graph coloring problem for this graph yields the map's coloring with the smallest number of colors possible.

b. Without loss of generality, we can assign colors 1 and 2 to vertices c and a, respectively. This forces the following color assignment to the remaining vertices: 3 to b, 2 to d, 3 to f, 4 to e. Thus, the smallest number of colors needed for this map is four.

Note: It's a well-known fact that *any* map can be colored in four colors or less. This problem—known as the Four-Color Problem—has remained unresolved for more than a century until 1976 when it was finally solved by the American mathematicians K. Appel and W. Haken by a combination of mathematical arguments and extensive computer use.

9. If $n = 2$, the answer is always "yes"; so, we may assume that $n \geq 3$. Select three points P_1, P_2, and P_3 from the set given. Write an equation of the perpendicular bisector l_1 of the line segment with the endpoints at P_1 and P_2, which is the locus of points equidistant from P_1 and P_2. Write an equation of the perpendicular bisector l_2 of the line segment with the endpoints at P_2 and P_3, which is the locus of points equidistant from P_2 and P_3. Find the coordinates (x, y) of the intersection point P of the lines l_1 and l_2 by solving the system of two equations in two unknowns x and y. (If the system has no solutions, return "no": such a circumference does not exist.) Compute the distances (or much better yet the distance squares!) from P to each of the points P_i, $i = 3, 4, ..., n$ and check whether all of them are the same: if they are, return "yes," otherwise, return "no".

10. n/a

Solutions to Exercises 1.4

1. a. Replace the ith element with the last element and decrease the array size by 1.

   b. Replace the ith element with a special symbol that cannot be a value of the array's element (e.g., 0 for an array of positive numbers) to mark the ith position as empty. (This method is sometimes called the "lazy deletion".)

2. a. Use binary search (see Section 4.4 if you are not familiar with this algorithm).

   b. When searching in a sorted linked list, stop as soon as an element greater than or equal to the search key is encountered.

3. a.

$push(a)$		$push(b)$		pop		$push(c)$		$push(d)$		pop	
	a		b				c		d		
			a		a		a		c		c
									a		a

   b.

$enqueue(a)$	$enqueue(b)$	$dequeue$	$enqueue(c)$	$enqueue(d)$	$dequeue$
a	ab	b	bc	bcd	cd

4. a. For the adjacency matrix representation:

   i. A graph is complete if and only if all the elements of its adjacency matrix except those on the main diagonal are equal to 1, i.e., $A[i, j] = 1$ for every $1 \le i, j \le n$, $i \ne j$.

   ii. A graph has a loop if and only if its adjacency matrix has an element equal to 1 on its main diagonal, i.e., $A[i, i] = 1$ for some $1 \le i \le n$.

   iii. An (undirected, without loops) graph has an isolated vertex if and only if its adjacency matrix has an all-zero row.

   b. For the adjacency list representation:

   i. A graph is complete if and only if each of its linked lists contains all the other vertices of the graph.

   ii. A graph has a loop if and only if one of its adjacency lists contains the

vertex defining the list.

iii. An (undirected, without loops) graph has an isolated vertex if and only if one of its adjacency lists is empty.

5. The first algorithm works as follows. Mark a vertex to serve as the root of the tree, make it the root of the tree to be constructed, and initialize a stack with this vertex. Repeat the following operation until the stack becomes empty: If there is an unmarked vertex adjacent to the vertex on the top to the stack, mark the former vertex, attach it as a child of the top's vertex in the tree, and push it onto the stack; otherwise, pop the vertex off the top of the stack.

   The second algorithm works as follows. Mark a vertex to serve as the root of the tree, make it the root of the tree to be constructed, and initialize a queue with this vertex. Repeat the following operations until the queue becomes empty: If there are unmarked vertices adjacent to the vertex at the front of the queue, mark all of them, attach them as children to the front vertex in the tree, and add them to the queue; then dequeue the queue.

6. Since the height is defined as the length of the longest simple path from the tree's root to its leaf, such a pass will include no more than n vertices, which is the total number of vertices in the tree. Hence, $h \le n - 1$.

   The binary tree of height h with the largest number of vertices is the full tree that has all its $h + 1$ levels filled with the largest number of vertices possible. The total number of vertices in such a tree is $\sum_{l=0}^{h} 2^l = 2^{h+1} - 1$. Hence, for any binary tree with n vertices and height h

   $$2^{h+1} - 1 \ge n.$$

   This implies that

   $$2^{h+1} \ge n + 1$$

   or, after taking binary logarithms of both hand sides and taking into account that $h + 1$ is an integer,

   $$h + 1 \ge \lceil \log_2(n + 1) \rceil.$$

   Since $\lceil \log_2(n + 1) \rceil = \lfloor \log_2 n \rfloor + 1$ (see Appendix A), we finally obtain

   $$h + 1 \ge \lfloor \log_2 n \rfloor + 1 \text{ or } h \ge \lfloor \log_2 n \rfloor.$$

7. a. Insertion can be implemented by adding the new item after the array's last element. Finding the largest element requires a standard scan

through the array to find its largest element. Deleting the largest element $A[i]$ can be implemented by exchanging it with the last element and decreasing the array's size by 1.

b. We will assume that the array $A[0..n-1]$ representing the priority queue is sorted in ascending order. Inserting a new item of value v can be done by scanning the sorted array, say, left to right until an element $A[j] \geq v$ or the end of the array is reached. (A faster algorithm for finding a place for inserting a new element is binary search discussed in Section 4.4.) In the former case, the new item is inserted before $A[j]$ by first moving $A[n-1], ..., A[j]$ one position to the right; in the latter case, the new item is simply appended after the last element of the array. Finding the largest element is done by simply returning the value of the last element of the sorted array. Deletion of the largest element is done by decreasing the array's size by one.

c. Insertion of a new element is done by using the standard algorithm for inserting a new element in a binary search tree: recursively, the new key is inserted in the left or right subtree depending on whether it is smaller or larger than the root's key. Finding the largest element will require finding the rightmost element in the binary tree by starting at the root and following the chain of the right children until a vertex with no right subtree is reached. The key of that vertex will be the largest element in question. Deleting it can be done by making the right pointer of its parent to point to the left child of the vertex being deleted;. if the rightmost vertex has no left child, this pointer is made "null". Finally, if the rightmost vertex has no parent, i.e., if it happens to be the root of the tree, its left child becomes the new root; if there is no left child, the tree becomes empty.

8. Use a bit vector, i.e., an array on n bits in which the ith bit is 1 if the ith element of the underlying set is currently in the dictionary and 0 otherwise. The search, insertion, and deletion operations will require checking or changing a single bit in this array.

9. Use: (a) a priority queue; (b) a queue; (c) a stack (and *reverse Polish notation*—a clever way of representing arithmetical expressions without parentheses, which is usually studied in a data structures course).

10. The most straightforward solution is to search for each successive letter of the first word in the second one. If the search is successful, delete the first occurrence of the letter in the second word, stop otherwise.

    Another solution is to sort the letters of each word and then compare

them in a simple parallel scan.

We can also generate and compare "letter vectors" of the given words: $V_w[i]$ = the number of occurrences of the alphabet's ith letter in the word w. Such a vector can be generated by initializing all its components to 0 and then scanning the word and incrementing appropriate letter counts in the vector.

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,

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

Hence

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

or

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

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,

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

Hence

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

and therefore

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

or

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

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.

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

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

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

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

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

   $\lim\limits_{n\to\infty} \frac{2^n}{n!} =$ (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. $\lim\limits_{n\to\infty} \frac{p(n)}{n^k} = \lim\limits_{n\to\infty} \frac{a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0}{n^k} = \lim\limits_{n\to\infty} (a_k + \frac{a_{k-1}}{n} + \ldots + \frac{a_0}{n^k})$
   $= a_k > 0.$

   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.)

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 $D(1000) = 2893$.

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

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

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

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

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

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

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:

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

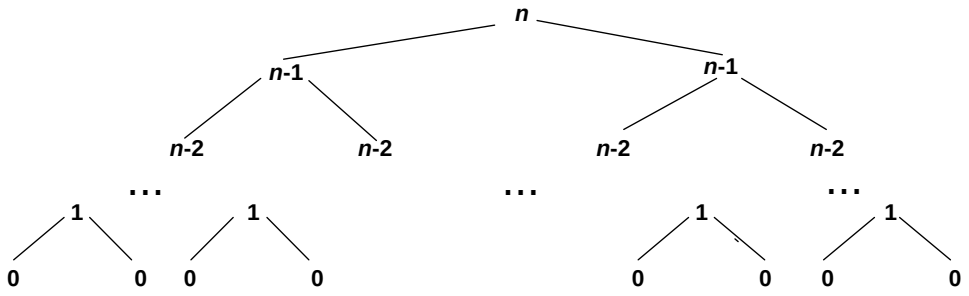
with the solution $A(n) = \lfloor \log_2 n \rfloor + 1$.

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.

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 $D(n)$:

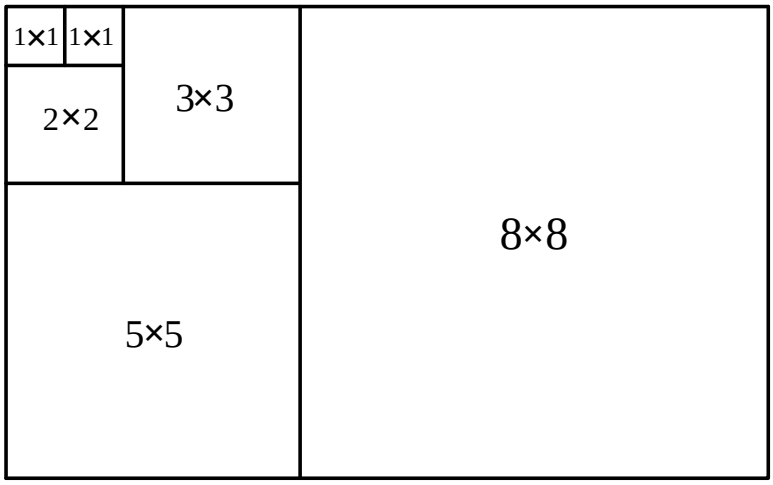
$$D(n) = D(n-1) + 1 \quad \text{for } n \geq 4, \quad D(3) = 1,$$

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

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

Solutions to Exercises 3.1

1. a. Euclid's algorithm and the standard algorithm for finding the binary representation of an integer are examples from the algorithms previously mentioned in this book. There are, of course, many more examples in its other chapters.

   b. Solving nonlinear equations or computing definite integrals are examples of problems that cannot be solved exactly (except for special instances) by any algorithm.

2. a. $M(n) = n \approx 2^b$ where $M(n)$ is the number of multiplications made by the brute-force algorithm in computing a^n and b is the number of bits in the n's binary representation. Hence, the efficiency is linear as a function of n and exponential as a function of b.

   b. Perform all the multiplications modulo m, i.e.,

   $$a^i \bmod m = (a^{i-1} \bmod m \cdot a \bmod m) \bmod m \quad \text{for } i = 1, ..., n.$$

3. Problem 4 (computes $\sum_1^n i^2$): yes

   Problem 5 (computes the range of an array's values): yes

   Problem 6 (checks whether a matrix is symmetric): yes

4. a. Here is a pseudocode of the most straightforward version:

   Algorithm *BruteForcePolynomialEvaluation*($P[0..n]$, x)
   //The algorithm computes the value of polynomial P at a given point x
   //by the "highest-to-lowest term" brute-force algorithm
   //Input: Array $P[0..n]$ of the coefficients of a polynomial of degree n,
   // stored from the lowest to the highest and a number x
   //Output: The value of the polynomial at the point x
   $p \leftarrow 0.0$
   for $i \leftarrow n$ **downto** 0 **do**
   $power \leftarrow 1$
   for $j \leftarrow 1$ **to** i **do**
   $power \leftarrow power * x$
   $p \leftarrow p + P[i] * power$
   return p

   We will measure the input's size by the polynomial's degree n. The basic operation of this algorithm is a multiplication of two numbers; the number of multiplications $M(n)$ depends on the polynomial's degree only.

Although it is not difficult to find the total number of multiplications in this algorithm, we can count just the number of multiplications in the algorithm's inner-most loop to find the algorithm's efficiency class:

$$M(n) = \sum_{i=0}^{n} \sum_{j=1}^{i} 1 = \sum_{i=0}^{n} i == \frac{n(n+1)}{2} \in \Theta(n^2).$$

b. The above algorithm is very inefficient: we recompute powers of x again and again as if there were no relationship among them. Thus, the obvious improvement is based on computing consecutive powers more efficiently. If we proceed from the highest term to the lowest, we could compute x^{i-1} by using x^i but this would require a division and hence a special treatment for $x = 0$. Alternatively, we can move from the lowest term to the highest and compute x^i by using x^{i-1}. Since the second alternative uses multiplications instead of divisions and does not require any special treatment for $x = 0$, it is both more efficient and cleaner. It leads to the following algorithm:

Algorithm *BetterBruteForcePolynomialEvaluation*($P[0..n]$, x)
//The algorithm computes the value of polynomial P at a given point x
//by the "lowest-to-highest term" algorithm
//Input: Array $P[0..n]$ of the coefficients of a polynomial of degree n,
// from the lowest to the highest, and a number x
//Output: The value of the polynomial at the point x
$p \leftarrow P[0]$; $power \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  $power \leftarrow power * x$
  $p \leftarrow p + P[i] * power$
return p

The number of multiplications here is

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

(while the number of additions is n), i.e., we have a linear algorithm.

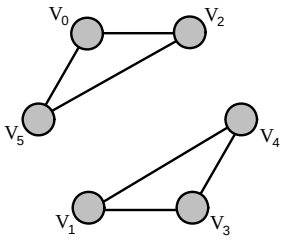
Note: Horner's Rule discussed in Section 6.5 needs only n multiplications (and n additions) to solve this problem.

c. No, because any algorithm for evaluating an arbitrary polynomial of degree n at an arbitrary point x must process all its $n + 1$ coefficients. (Note that even when $x = 1$, $p(x) = a_n + a_{n-1} + ... + a_1 + a_0$, which needs at least n additions to be computed correctly for arbitrary $a_n, a_{n-1}, ..., a_0$.)

5. For simplicity, we check each of the three topologies separately.

The adjacency matrix of a graph with the ring topology must be, of course, symmetric, and each of its rows must have exactly two 1's, both not on the main diagonal to avoid loops. These necessary conditions are not sufficient because they do not guarantee connectivity of the graph, as the following example demonstrates.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$



The following brute-force algorithm follows the 1's in a given matrix indicating two edges incident with a vertex: one for the edge entering it and the other leaving the vertex, making sure the cycle closes at the starting vertex only after visiting all the other vertices of the graph.

Start by scanning row 0 to verify that it has exactly two 1's in columns we denote j_1 and j_{n-1} so that $0 < j_1 < j_{n-1}$. If it is not the case, stop: the matrix is not the adjacency matrix of a graph with the ring topology. If it is the case, mark column 0 as that of a visited vertex and proceed to row j_1. ($0 - j_1$ is the first edge of the cycle being traversed.) Check that $A[j_1, 0] = 1$ and find the unmarked column $j_2 \neq j_1$ with the only other 1 in this row. If this is impossible to do, stop; otherwise, mark column j_1 as that of a visited vertex and proceed to row j_2. ($j_1 - j_2$ is the second edge of the cycle being traversed.) Continue in this fashion until the row corresponding to the only remaining unmarked vertex needs to be processed. This must be row j_{n-1}, where j_{n-1} was found on the first iteration of the algorithm. The two 1's in row j_{n-1} must be in columns j_{n-2} (of the vertex from which vertex j_{n-1} was reached) and 0 (to close the cycle).

The time efficiency of the algorithm is $O(n^2)$, because it checks all the elements of an $n \times n$ matrix in the worst case.

Note: It is not difficult to prove that a graph has the ring topology if and only if all its vertices have degree 2 while having no loops, and it is connected. Hence the problem can also be solved by the obvious checking of the first condition and checking the graph's connectivity by one of the two standard algorithms for doing that: depth-first search or breadth-first search, which are discussed in Section 3.5 of the book. The above algorithm mimics, in fact, the first of these alternatives.

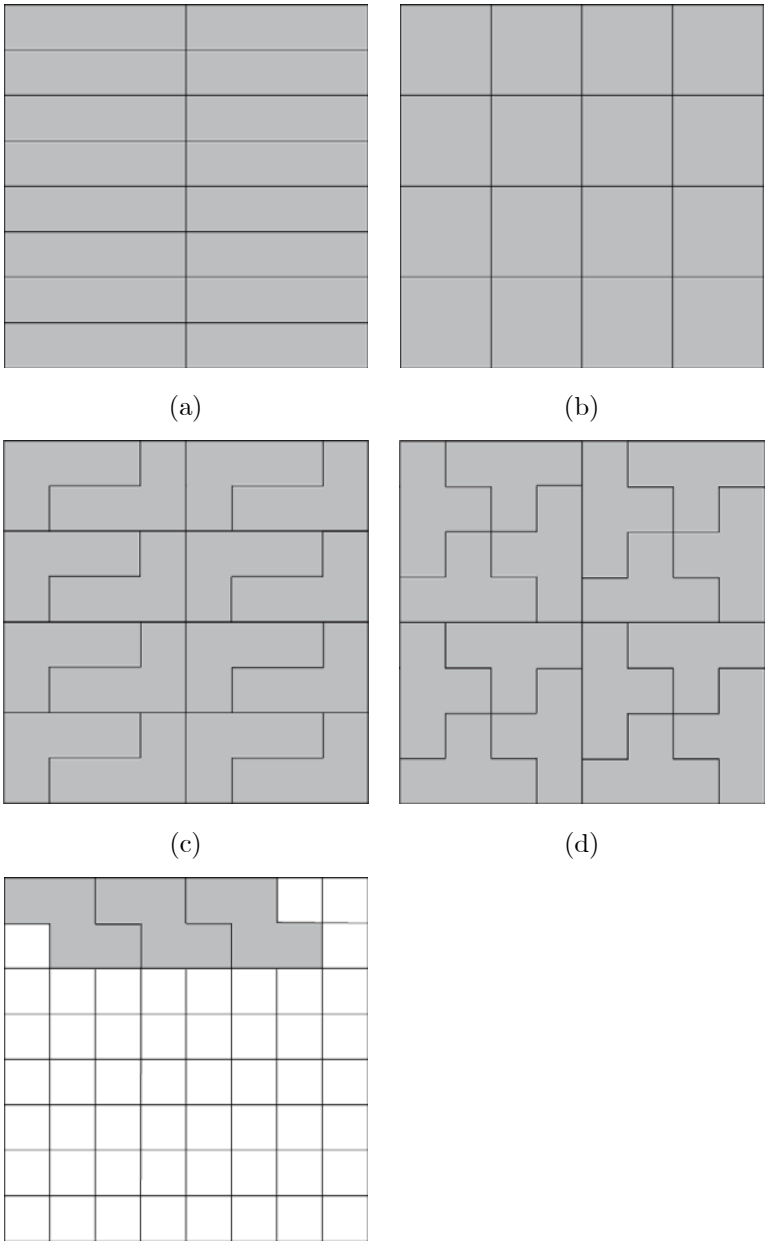
The adjacency matrix of a graph with the star topology contains only 0's except some row j_0 and column j_0 with all 1's except $A[j_0, j_0]$, the element on the main diagonal. Hence a brute-force algorithm can start by scanning row 0

to check whether it contains all 1's except in column 0 or it contains a single 1 in some column $j_0 > 0$. In the former case, every subsequent row $i = 1, 2, ..., n-1$ must contain a single 1 in column 0 (i.e., $A[i, 0] = 1$ and $A[i, j] = 0$ for $j = 1, ..., n-1$). In the latter case, every subsequent row $i = 1, 2, ..., n-1$ except $i = j_0$ is checked for the same properties as row 0 (i.e., $A[i, j_0] = 1$ and $A[i, j] = 0$ for $j = 0, 1, ..., n-1, j \neq j_0$), whereas row j_0 must contain only 1's except the element on its main diagonal (i.e., $A[j_0, , j_0] = 0$ and $A[j_0, j] = 1$ for $j = 0, 1, ..., n-1, j \neq j_0$). Obviously, the algorithm's time efficiency is also $O(n^2)$.

Finally, the adjacency matrix of a graph with the fully connected mesh topology contains only 1's except 0's on its main diagonal. Checking this property by scanning rows (or columns) of the matrix requires $O(n^2)$ time as well.

6. Tilings of an 8×8 board with straight tetrominoes, square tetrominoes, L-tetrominoes, and T-tetrominoes are shown below. It is impossible to cover an 8×8 board with Z-tetrominoes. Indeed, putting such a tile to cover a corner of the board makes it necessary to continue putting two more tiles along the boarder with no possibility to cover the two remaining squares

in the first row.



(a) (b)

(c) (d)

(e)

Tiling a chessboard with (a) straight tetrominoes; (b) square tetrominoes; (c) L-tetrominoes; (d) T-tetrominoes; (e) Z-tetrominoes (failed)

Note: Tiling with tetrominoes is discussed, among other types of polyominoes, by their inventor S. W. Golomb in his monograph *Polyominoes: Puzzles, Patterns, Problems, and Packings*. Revised and expanded second edition, Princeton University Press, Princeton, NJ, 1994.

7. a. Number the coin stacks from 1 to n. Starting with the first stack, repeat the following. If the current stack is not the last one, take any coin from the stack and weigh it: if it weighs 11 grams, this stack contains the fake coins and algorithm stops; if the coin weighs 10 grams, proceed to the next stack. If the last stack is reached, it must be the one with the fake coins and none of its coins need to be weighed. In the worst case (the stack of the fake coins is the last one), the algorithm makes $n-1$ weighings, which puts it in the $\Theta(n)$ class.

   b. The problem can be solved in one weighing. Number the coin stacks from 1 to n. Take one coin from the first stack, two coins from the second, and so on until all n coins are taken from the last stack. Weigh all these coins together. The difference between this weight and $10n(n+1)/2 = 5n(n+1)$, the weight of $(1+2+...+n) = n(n+1)/2$ genuine coins, indicates the number of the fake coins weighed, which is equal to the number of the stack with the fake coins. For example, if $n = 10$ and the selected coins weigh 553 grams, 3 coins are fake and hence it is the third stack that contains the fake coins.

   Note: Finding a stack of fake coins in one weighing is a well-known puzzle, which has been included in many collections of brain teasers.

8.
$$
\begin{array}{|ccccccc}
E & X & \mathbf{A} & M & P & L & E \\
A & X & \mathbf{E} & M & P & L & E \\
A & E & X & M & P & L & \mathbf{E} \\
A & E & E & M & P & \mathbf{L} & X \\
A & E & E & L & P & \mathbf{M} & X \\
A & E & E & L & M & \mathbf{P} & X \\
A & E & E & L & M & P & X
\end{array}
$$

9. Selection sort is not stable: In the process of exchanging elements that are not adjacent to each other, the algorithm can reverse an ordering of equal elements. The list $2'$, $2''$, 1 is such an example.

10. Yes. Both operations—finding the smallest element and swapping it—can be done as efficiently with a linked list as with an array.

11. E, X, A, M, P, L, E

$$
\begin{array}{llllllllllll}
E & \overset{?}{\leftrightarrow} & X & \overset{?}{\leftrightarrow} & A & & M & & P & & L & & E \\
E & & A & & X & \overset{?}{\leftrightarrow} & M & & P & & L & & E \\
E & & A & & M & & X & \overset{?}{\leftrightarrow} & P & & L & & E \\
E & & A & & M & & P & & X & \overset{?}{\leftrightarrow} & L & & E \\
E & & A & & M & & P & & L & & X & \overset{?}{\leftrightarrow} & E \\
E & & A & & M & & P & & L & & E & & |X \\
E & \overset{?}{\leftrightarrow} & A & & M & & P & & L & & E & & \\
A & & E & \overset{?}{\leftrightarrow} & M & \overset{?}{\leftrightarrow} & P & \overset{?}{\leftrightarrow} & L & & E & & \\
A & & E & & M & & L & & P & \overset{?}{\leftrightarrow} & E & & \\
A & & E & & M & & L & & E & & |P & & \\
A & \overset{?}{\leftrightarrow} & E & \overset{?}{\leftrightarrow} & M & \overset{?}{\leftrightarrow} & L & & E & & & & \\
A & & E & & L & & M & \overset{?}{\leftrightarrow} & E & & & & \\
A & & E & & L & & E & & |M & & & & \\
A & \overset{?}{\leftrightarrow} & E & \overset{?}{\leftrightarrow} & L & \overset{?}{\leftrightarrow} & E & & & & & & \\
A & & E & & E & & |L & & & & & & \\
A & \overset{?}{\leftrightarrow} & E & \overset{?}{\leftrightarrow} & E & \overset{?}{\leftrightarrow} & L & & & & & &
\end{array}
$$

The algorithm can be stopped here (see the next question).

12. a. Pass i $(0 \le i \le n-2)$ of bubble sort can be represented by the following diagram:

$$A_0, ..., A_j \overset{?}{\leftrightarrow} A_{j+1}, ..., A_{n-i-1} \le | \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

If there are no swaps during this pass, then

$$A_0 \le A_1 \le \cdots \le A_j \le A_{j+1} \le \cdots \le A_{n-i-1},$$

with the larger (more accurately, not smaller) elements in positions $n-i$ through $n-1$ being sorted during the previous iterations.

b. Here is a pseudocode for the improved version of bubble sort:

Algorithm *BetterBubbleSort*$(A[0..n-1])$
//The algorithm sorts array $A[0..n-1]$ by improved bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
count $\leftarrow n-1$ //number of adjacent pairs to be compared
sflag $\leftarrow$ **true** //swap flag
while *sflag* **do**

```
sflag ← false
for j ← 0 to count−1 do
    if A[j + 1] < A[j]
        swap A[j] and A[j + 1]
        sflag ← true
count ← count −1
```

c. The worst-case inputs will be strictly decreasing arrays. For them, the improved version will make the same comparisons as the original version, which was shown in the text to be quadratic.

13. Bubble sort is stable. It follows from the fact that it swaps adjacent elements only, provided $A[j+1] < A[j]$.

14. Here is a simple and efficient (in fact, optimal) algorithm for this problem: Starting with the first and ending with the last light disk, swap it with each of the i ($1 \le i \le n$) dark disks to the left of it. The ith iteration of the algorithm can be illustrated by the following diagram, in which 1s and 0s correspond to the dark and light disks, respectively.

$$\underbrace{00..0}_{i-1}\underbrace{11..1}_{i-1}\mathbf{1}0\mathbf{1}0..10 \quad \Rightarrow \quad \underbrace{00..0}_{i}\mathbf{0}\underbrace{11..11}_{i}0..10$$

The total number of swaps made is equal to $\sum_{i=1}^{n} i = n(n+1)/2$.

The problem can also be solved by mimicking the swaps made by bubble sort in sorting the array of 1's and 0's representing the dark and light disks, respectively.

Solutions to Exercises 3.2

1. a. $C_{worst}(n) = n + 1$.

   b. $C_{avg}(n) = \frac{(2-p)(n+1)}{2}$. In the manner almost identical to the analysis in Section 2.1, we obtain

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + (n+1) \cdot (1-p) \\ &= \frac{p}{n}[1 + 2 + ... + i + \cdots + n] + (n+1)(1-p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + (n+1)(1-p) = \frac{(2-p)(n+1)}{2}. \end{aligned}$$

2. The expression

$$\frac{p(n+1)}{2} + n(1-p) = p\frac{n+1}{2} + n - np = n - p(n - \frac{n+1}{2}) = n - \frac{n-1}{2}p$$

   is a linear function of p. Since the p's coefficient is negative for $n > 1$, the function is strictly decreasing on the interval $0 \le p \le 1$ from n to $(n+1)/2$. Hence $p = 0$ and $p = 1$ are its maximum and minimum points, respectively, on this interval. (Of course, this is the answer we should expect: The average number of comparisons should be the largest when the probability of a successful search is 0, and it should be the smallest when the probability of a successful search is 1.)

3. Drop the first gadget from floors $\lceil\sqrt{n}\rceil$, $2\lceil\sqrt{n}\rceil$, and so on until either the floor $i\lceil\sqrt{n}\rceil$ a drop from which makes the gadget malfunction is reached or no such floor in this sequence is encountered before the top of the building is reached. In the former case, the floor to be found is higher than $(i-1)\lceil\sqrt{n}\rceil$ and lower than $i\lceil\sqrt{n}\rceil$. So, drop the second gadget from floors $(i-1)\lceil\sqrt{n}\rceil + 1$, $(i-1)\lceil\sqrt{n}\rceil + 2$, and so on until the first floor a drop from which makes the gadget malfunction is reached. The floor immediately preceding that floor is the floor in question. If no drop in the first-pass sequence resulted in the gadget's failure, the floor in question is higher than $i\lceil\sqrt{n}\rceil$, the last tried floor of that sequence. Hence, continue the successive examination of floors $i\lceil\sqrt{n}\rceil + 1$, $i\lceil\sqrt{n}\rceil + 2$, and so on until either a failure is registered or the last floor is reached. The number of times the two gadgets are dropped doesn't exceed $\lceil\sqrt{n}\rceil + \lceil\sqrt{n}\rceil$, which puts it in $O(\sqrt{n})$.

4. 43 comparisons.
   The algorithm will make $47 - 6 + 1 = 42$ trials: In the first one, the G of the pattern will be aligned against the first T of the text; in the last one, it will be aligned against the last space. On each but one trial, the algorithm will make one unsuccessful comparison; on one trial—when the G of the pattern is aligned against the G of the text —it will make two

comparisons. Thus, the total number of character comparisons will be $41 \cdot 1 + 1 \cdot 2 = 43$.

5. a. For the pattern `00001`, the algorithm will make four successful and one unsuccessful comparison on each of its trials and then shift the pattern one position to the right:

```
0 0 0 0 0 0                                   0 0 0 0 0
0 0 0 0 1
  0 0 0 0 1
                  etc.
                                              0 0 0 0 1
```

The total number of character comparisons will be $C = 5 \cdot 996 = 4980$.

b. For the pattern `10000`, the algorithm will make one unsuccessful comparison on each of its trials and then shift the pattern one position to the right:

```
0 0 0 0 0 0                                   0 0 0 0 0
1 0 0 0 0
  1 0 0 0 0
                  etc.
                                              1 0 0 0 0
```

The total number of character comparisons will be $C = 1 \cdot 996 = 996$.

c. For the pattern `01010`, the algorithm will make one successful and one unsuccessful comparison on each of its trials and then shift the pattern one position to the right:

```
0 0 0 0 0 0                                   0 0 0 0 0
0 1 0 1 0
  0 1 0 1 0
                  etc.
                                              0 1 0 1 0
```

The total number of character comparisons will be $C = 2 \cdot 996 = 1,992$.

6. The text composed of n zeros and the pattern $\underbrace{0 \ldots 0}_{m-1}1$ is an example of the worst-case input. The algorithm will make $m(n - m + 1)$ character comparisons on such input.

7. Comparing pairs of the pattern and text characters righ-to-left can allow farther pattern shifts after a mismatch. This is the main insight the two string matching algorithms discussed in Section 7.2 are based on. (As a specific example, consider searching for the pattern 11111 in the text of one thousand zeros.)

8. a. Note that the number of desired substrings that starts with an A at a given position i $(0 \le i < n-1)$ in the text is equal to the number of B's to the right of that position. This leads to the following simple algorithm:

   Initialize the count of the desired substrings to 0. Scan the text left to right doing the following for every character except the last one: If an A is encountered, count the number of all the B's following it and add this number to the count of desired substrings. After the scan ends, return the last value of the count.

   For the worst case of the text composed of n A's, the total number of character comparisons is

$$n + (n-1) + \cdots + 2 = n(n+1)/2 - 1 \in \Theta(n^2).$$

   b. Note that the number of desired substrings that ends with a B at a given position i $(0 < i \le n-1)$ in the text is equal to the number of A's to the left of that position. This leads to the following algorithm:

   Initialize the count of the desired substrings and the count of A's encountered to 0. Scan the text left to right until the text is exhausted and do the following. If an A is encountered, increment the A's count; if a B is encountered, add the current value of the A's count to the desired substring count. After the text is exhausted, return the last value of the desired substring count.

   Since the algorithm makes a single pass through a given text spending constant time on each of its characters, the algorithm is linear.

9. n/a

10. n/a

11. n/a

Solutions to Exercises 3.3

1. If we take into account only the arithmetical operations involved into computing the Euclidean distance between two points versus computing its square, we will end up with the following estimate of the time ratio (both for the algorithm's innermost loop and the entire algorithm): $(2 + 3 + 10)/(2 + 3) = 3$.

If we also take into account the comparison and assignment, we get the time ratio estimate as $(2 + 3 + 10 + 2)/(2 + 3 + 2) \approx 2.4$.

2. Sort the numbers in ascending order, compute the differences between adjacent numbers in the sorted list, and find the smallest such difference. If sorting is done in $O(n \log n)$ time, the running time of the entire algorithm will be in
$$O(n \log n) + \Theta(n) + \Theta(n) = O(n \log n).$$

3. a. If we put the post office at location x_i, the average distance between it and all the points $x_1 < x_2 < ... < x_n$ is given by the formula $\frac{1}{n} \sum_{j=1}^{n} |x_j - x_i|$. Since the number of points n stays the same, we can ignore the multiple $\frac{1}{n}$ and minimize $\sum_{j=1}^{n} |x_j - x_i|$. We'll have to consider the cases of even and odd n separately.

Let n be even. Consider first the case of $n = 2$. The sum $|x_1 - x| + |x_2 - x|$ is equal to $x_2 - x_1$, the length of the interval with the endpoints at x_1 and x_2, for any point x of this interval (including the endpoints), and it is larger than $x_2 - x_1$ for any point x outside of this interval. This implies that for any even n, the sum

$$\sum_{j=1}^{n} |x_j - x| = [|x_1 - x| + |x_n - x|] + [|x_2 - x| + |x_{n-1} - x|] + \cdots + [|x_{n/2} - x| + |x_{n/2+1} - x|]$$

is minimized when x belongs to each of the intervals $[x_1, x_n] \supset [x_2, x_{n-1}] \supset ... \supset [x_{n/2}, x_{n/2+1}]$. If x must be one of the points given, either $x_{n/2}$ or $x_{n/2+1}$ solves the problem.

Let $n > 1$ be odd. Then, the sum $\sum_{j=1}^{n} |x_j - x|$ is minimized when $x = x_{\lceil n/2 \rceil}$, the point for which the number of the given points to the left of it is equal to the number of the given points to the right of it.

Note that the point $x_{\lceil n/2 \rceil}$—the $\lceil n/2 \rceil$th smallest called the *median*—solves the problem for even n's as well. For a sorted list implemented as an array, the median can be found in $\Theta(1)$ time by simply returning the $\lceil n/2 \rceil$th element of the array. (Section 5.6 provides a more general discussion of algorithms for computing the median.)

b. Assuming that the points $x_1, x_2, ..., x_n$ are given in increasing order, the answer is the point x_i that is the closest to $m = (x_1 + x_n)/2$, the middle point between x_1 and x_n. (The middle point would be the obvious solution if the

post-post office didn't have to be at one of the given locations.) Indeed, if we put the post office at any location x_i to the left of m, the longest distance from a village to the post office would be $x_n - x_i$; this distance is minimal for the rightmost among such points. If we put the post office at any location x_i to the right of m, the longest distance from a village to the post office would be $x_i - x_1$; this distance is minimal for the leftmost among such points.

Algorithm *PostOffice1*(P)
//Input: List P of n $(n \geq 2)$ points $x_1, x_2, ..., x_n$ in increasing order
//Output: Point x_i that minimizes $\max\limits_{1 \leq j \leq n} |x_j - x_i|$ among all $x_1, x_2, ..., x_n$
$m \leftarrow (x_1 + x_n)/2$
$i \leftarrow 1$
while $x_i < m$ **do**
$\quad i \leftarrow i + 1$
if $x_i - x_1 < x_n - x_{i-1}$
$\quad$ **return** x_i
else return x_{i-1}

The time efficiency of this algorithm is $O(n)$.

4. a. For $d_M(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$, we have the following:

(i) $d_M(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2| \geq 0$ and $d_M(p_1, p_2) = 0$ if and only if both $x_1 = x_2$ and $y_1 = y_2$, i.e., P_1 and P_2 coincide.

(ii) $d_M(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2| = |x_2 - x_1| + |y_2 - y_1|$
$= d_M(p_2, p_1)$.

(iii) $d_M(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$
$= |(x_1 - x_3) + (x_3 - x_2)| + |(y_1 - y_3) + (y_3 - y_2)|$
$\leq |x_1 - x_3| + |x_3 - x_2| + |y_1 - y_3| + |y_3 - y_2| = d(p_1, p_3) + d(p_3, p_2)$.

b. For the Manhattan distance, the points in question are defined by the equation

$$|x - 0| + |y - 0| = 1, \text{ i.e., } |x| + |y| = 1.$$

The graph of this equation is the boundary of the square with its vertices at $(1, 0)$, $(0, 1)$, $(-1, 0)$, and $(-1, -1)$.

For the Euclidean distance, the points in question are defined by the equation

$$\sqrt{(x-0)^2 + (y-0)^2} = 1, \text{ i.e., } x^2 + y^2 = 1.$$

The graph of this equation is the circumference of radius 1 and the center at $(0, 0)$.

c. False. Consider points $p_1(0, 0)$, $p_2(1, 0)$, and, say, $p_3(\frac{1}{2}, \frac{3}{4})$. Then

$$d_E(p_1, p_2) = 1 \text{ and } d_E(p_3, p_1) = d_E(p_3, p_2) = \sqrt{\left(\frac{1}{2}\right)^2 + \left(\frac{3}{4}\right)^2} < 1.$$

Therefore, for the Euclidean distance, the two closest points are either p_1 and p_3 or p_2 and p_3. For the Manhattan distance, we have

$$d_M(p_1, p_2) = 1 \text{ and } d_M(p_3, p_1) = d_M(p_3, p_2) = \frac{1}{2} + \frac{3}{4} = \frac{5}{4} > 1.$$

Therefore, for the Manhattan distance, the two closest points are p_1 and p_2.

5. a. Since the first two axioms of a metric is obviously satisfied for the Hamming distance, only the third one—the triangle inequality—needs a proof. It can be obtained by mathematical induction on the string length m. If $m = 1$, the inequality $d_H(S_1, S_2) \le d_H(S_1, S_3) + d_H(S_3, S_2)$ holds for any one-character strings S_1, S_2, and S_3: if $S_1 = S_2$, the left-hand side is equal to 0; if $S_1 \neq S_2$, the left-hand side is equal to 1 and the right-hand side is greater than or equal to 1 because S_3 cannot be the same as both S_1 and S_2. For the inductive step, assume that the triangle inequality holds for any three strings of length m and consider three arbitrary strings $S_i = S_i^{'}c_i$, where $i = 1, 2, 3$ and $S_i^{'}$'s are strings of length m and c_i's are their last characters. Then

$$\begin{aligned} d_H(S_1, S_2) &= d_H(S_1^{'}, S_2^{'}) + d_H(c_1, c_2) \\ &\le d_H(S_1^{'}, S_3^{'}) + d_H(S_3^{'}, S_2^{'}) + d_H(c_1, c_3) + d_H(c_3, c_2) \\ &= [d_H(S_1^{'}, S_3^{'}) + d_H(c_1, c_3)] + [d_H(S_3^{'}, S_2^{'}) + d_H(c_3, c_2)] \\ &= d_H(S_1, S_3) + d_H(S_3, S_2). \end{aligned}$$

b. Since the basic operation of the algorithm is comparing two characters in the strings of length m, the worst-case time efficiency class will be $\Theta(mn^2)$.

6. We'll prove by induction that there will always remain at least one person not hit by a pie. The basis step is easy: If $n = 3$, the two persons with the smallest pairwise distance between them throw at each other, while the third person throws at one of them (whoever is closer). Therefore, this third person remains "unharmed".
For the inductive step, assume that the assertion is true for odd $n \ge 3$, and consider $n + 2$ persons. Again, the two persons with the smallest pairwise distance between them (the closest pair) throw at each other.

Consider two possible cases as follows. If the remaining n persons all throw at one another, at least one of them remains "unharmed" by the inductive assumption. If at least one of the remaining n persons throws at one of the closest pair, among the remaining $n-1$ persons, at most $n-1$ pies are thrown at one another, and hence at least one person must remain "unharmed" because there is not enough pies to hit everybody in that group. This completes the proof.

Note: The problem is from the paper by L. Carmony titled "Odd pie fights," *Mathematics Teacher*, vol. 72, no. 1, 1979, 61–64.

7. The number of squarings will be

$$\begin{aligned} C(n,k) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{s=1}^{k} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} k = k \sum_{i=1}^{n-1} (n-i) \\ &= k[(n-1) + (n-2) + \cdots + 1] = \frac{k(n-1)n}{2} \in \Theta(kn^2). \end{aligned}$$

8. a. The convex hull of a line segment is the line segment itself; its extreme points are the endpoints of the segment.

   b. The convex hull of a square is the square itself; its extreme points are the four vertices of the square.

   c. The convex hull of the boundary of a square is the region comprised of the points within that boundary and on the boundary itself; its extreme points are the four vertices of the square.

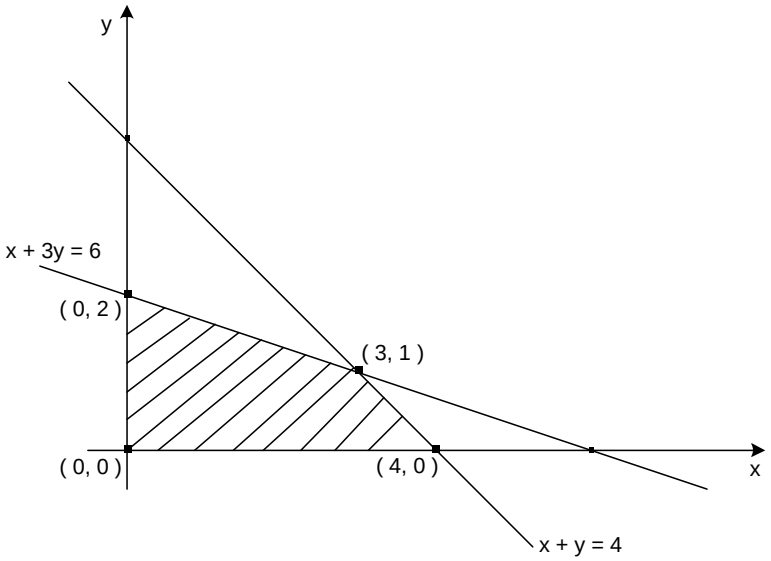
   d. The convex hull of a straight line is the straight line itself. It doesn't have any extreme points.

9. Find the point with the smallest x coordinate; if there are several such points, find the one with the smallest y coordinate among them. Similarly, find the point with the largest x coordinate; if there are several such points, find the one with the largest y coordinate among them. (Note that it's more efficient to look for the smallest and largest x coordinates on the same pass through the list of the points given. While it does not change the linear efficiency class of the algorithm, it can reduce the total number of comparisons to about $1.5n$.)

10. If there are other points of a given set on the straight line through p_i and p_j (while all the other points of the set lie on the same side of the line), a line segment of the convex hull's boundary will have its end points at the two farthest set points on the line. All the other points on the line can be eliminated from further processing.

11. n/a

12. a. Here is a sketch of the feasible region in question:



b. The extreme points are: $(0, 0)$, (4,0), (3,1), and (0,2).

c.

Extreme point	Value of $3x + 5y$
(0,0)	0
(4,0)	12
(3,1)	14
(0,2)	10

So, the optimal solution is (3, 1), with the maximum value of $3x + 5y$ equal to 14. (Note: This instance of the linear programming problem is discussed further in Section 10.1.)

Solutions to Exercises 3.4

1. a. $\Theta(n!)$
   For each tour (a sequence of $n+1$ cities), one needs n additions to compute the tour's length. Hence, the total number of additions $A(n)$ will be n times the total number of tours considered, i.e., $n*\frac{1}{2}(n-1)! = \frac{1}{2}n! \in \Theta(n!)$.

   b. (i) $n_{\max} = 16$; (ii) $n_{\max} = 17$; (iii) $n_{\max} = 19$; (iv) $n_{\max} = 21$.
   Given the answer to part a, we have to find the largest value of n such that
   $$\frac{1}{2}n!10^{-10} \leq t$$
   where t is the time available (in seconds). Thus, for $t = 1\text{hr} = 3.6*10^3\text{sec}$, we get the inequality
   $$n! \leq 2 * 10^{10}t = 7.2 * 10^{13}.$$
   The largest value of n for which this inequality holds is 16 (since $16! \approx 2.1 * 10^{13}$ and $17! \approx 3.6 * 10^{14}$).

   For the other given values of t, the answers can be obtained in the same manner.

2. The problem of finding a Hamiltonian circuit is very similar to the traveling salesman problem. Generate permutations of n vertices that start and end with, say, the first vertex, and check whether every pair of successive vertices in a current permutation are connected by an edge. If it's the case, the current permutation represents a Hamiltonian circuit, otherwise, a next permutation needs to be generated.

3. A connected graph has a Eulerian circuit if and only if all its vertices have even degrees. An algorithm should check this condition until either an odd vertex is encountered (then a Eulerian circuit doesn't exist) or all the vertices turn out to be even (then a Eulerian circuit must exist). For a graph (with no loops) represented by its $n \times n$ adjacency matrix, the degree of a vertex is the number of ones in the vertex's row. Thus, computing its degree will take the $\Theta(n)$ time, checking whether it's even will take $\Theta(1)$ time, and it will be done between 1 and n times. Hence, the algorithm's efficiency will be in $O(n^2)$.

4. The following assignments were generated in the chapter's text:
   $$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$
1, 2, 3, 4	cost = 9+4+1+4 = 18
1, 2, 4, 3	cost = 9+4+8+9 = 30
1, 3, 2, 4	cost = 9+3+8+4 = 24
1, 3, 4, 2	cost = 9+3+8+6 = 26
1, 4, 2, 3	cost = 9+7+8+9 = 33
1, 4, 3, 2	cost = 9+7+1+6 = 23

   etc.

The remaining ones are

2, 1, 3, 4	cost = 2+6+1+4 = 13
2, 1, 4, 3	cost = 2+6+8+9 = 25
2, 3, 1, 4	cost = 2+3+5+4 = 14
2, 3, 4, 1	cost = 2+3+8+7 = 20
2, 4, 1, 3	cost = 2+7+5+9 = 23
2, 4, 3, 1	cost = 2+7+1+7 = 17
3, 1, 2, 4	cost = 7+6+8+4 = 25
3, 1, 4, 2	cost = 7+6+8+6 = 27
3, 2, 1, 4	cost = 7+4+5+4 = 20
3, 2, 4, 1	cost = 7+4+8+7 = 26
3, 4, 1, 2	cost = 7+7+5+6 = 25
3, 4, 2, 1	cost = 7+7+8+7 = 29
4, 1, 2, 3	cost = 8+6+8+9 = 31
4, 1, 3, 2	cost = 8+6+1+6 = 21
4, 2, 1, 3	cost = 8+4+5+9 = 26
4, 2, 3, 1	cost = 8+4+1+7 = 20
4, 3, 1, 2	cost = 8+3+5+6 = 22
4, 3, 2, 1	cost = 8+3+8+7 = 26

The optimal solution is: Person 1 to Job 2, Person 2 to Job 1, Person 3 to Job 3, and Person 4 to Job 4, with the total (minimal) cost of the assignment being 13.

5. Here is a very simple example:

$$\begin{bmatrix} 1 & 2 \\ 2 & 9 \end{bmatrix}$$

6. Start by computing the sum S of the numbers given. If S is odd, stop because the problem doesn't have a solution. If S is even, generate the subsets until either a subset whose elements' sum is $S/2$ is encountered or no more subsets are left. Note that it will suffice to generate only subsets with no more than $n/2$ elements.

7. Generate a subset of k vertices and check whether every pair of vertices in the subset is connected by an edge. If it's true, stop (the subset is a clique); otherwise, generate the next subset.

8. Generate a permutation of the elements given and check whether they are ordered as required by comparing values of its consecutive elements. If they are, stop; otherwise, generate the next permutation. Since the

number of permutations of n items is equal to $n!$ and checking a permutation requires up to $n-1$ comparisons, the algorithm's efficiency class is in $O(n!(n-1)) = O((n+1)!)$.

9. The number of different positions of eight queens on the 8×8 board is equal to

   a. $C(64, 8) = 4,426,165,368$ if no two queens are on the same square.

   b. $8^8 = 16,777,216$ if no two queens are in the same row.

   c. $8! = 40,320$ if no two queens are in the same row or in the same column.

10. a. Let s be the sum of the numbers in each row of an $n \times n$ magic square. Let us add all the numbers in rows 1 through n. We will get the following equality:

$$sn = 1+2+\cdots+n^2, \text{ i.e., } sn = \frac{n^2(n^2+1)}{2}, \text{ which implies } s = \frac{n(n^2+1)}{2}.$$

    b. Number positions in an $n \times n$ matrix from 1 through n^2. Generate a permutation of the numbers 1 through n^2, put them in the corresponding positions of the matrix, and check the magic-square equality (proved in part (a)) for every row, every column, and each of the two main diagonals of the matrix.

    c. n/a

    d. n/a

11. a. Since the letter-digit correspondence must be one-to-one and there are only ten distinct decimal digits, the exhaustive search needs to check $P(10, k) = 10!/(10-k)!$ possible substitutions, where k is the number of distinct letters in the input. (The requirement that the first letter of a word cannot represent 0 can be used to reduce this number further.) Thus a program should run in a quite reasonable amount of time on today's computers. Note that rather than checking two cases—with and without a "1-carry"—for each of the decimal positions, the program can check just one equality, which stems from the definition of the decimal number system. For Dudeney's alphametic, for example, this equality is 1000(S+M) + 100(E+O) + 10(N+R) + (D+E) = 10000M + 1000O + 100N + 10E + Y

    b. Here is a "computerless" solution to this classic problem. First, notice that M must be 1. (Since both S and M are not larger than 9, their

sum, even if increased by 1 because of the carry from the hundred column, must be less than 20.) We will have to rely on some further insights into specifics of the problem. The leftmost digits of the addends imply one of the two possibilities: either S + M = 10 + O (if there was no carry from the hundred column) or 1 + S + M = 10 + O (if there was such a carry). First, let us pursue the former of the two possibilities. Since M = 1, S $\leq$ 9 and O $\geq$ 0, the equation S + 1 = 10 + O has only one solution: S = 9 and O = 0. This leaves us with

$$\begin{array}{r} \text{E N D} \\ +\ \text{O R E} \\ \hline \text{N E Y} \end{array}$$

Since we deal here with the case of no carry from the hundreds and E and N must be distinct, the only possibility is a carry from the tens: 1 + E = N and either N + R = 10 + E (if there was no carry from the rightmost column) or 1 + N + R = 10 + E (if there was such a carry). The first combination leads to a contradiction: Substituting 1 + E for N into N + R = 10 + E, we obtain R = 9, which is incompatible with the same digit already represented by S. The second combination of 1 + E = N and 1 + N + R = 10 + E implies, after substituting the first of these equations into the second one, R = 8. Note that the only remaining digit values still unassigned are 2, 3, 4, 5, 6, and 7. Finally, for the righmost column, we have the equation D + E = 10 + Y. But 10 + Y $\geq$ 12, because the smallest unassigned digit value is 2 while D + E $\leq$ 12 because the two largest unassigned digit values are 6 and 7 and E < N. Thus, D + E = 10 + Y = 12. Hence Y = 2 and D + E = 12. The only pair of still unassigned digit values that add up to 12, 5 and 7, must be assigned to E and D, respectively, since doing this the other way (E = 7, D = 5) would imply N = E + 1 = 8, which is already represented by R. Thus, we found the following solution to the puzzle:

$$\begin{array}{r} 9\ 5\ 6\ 7 \\ +\ 1\ 0\ 8\ 5 \\ \hline 1\ 0\ 6\ 5\ 2 \end{array}$$

Is this the only solution? To answer this question, we should pursue the carry possibility from the hundred column to the thousand column (see above). Then 1 + S + M = 10 + O or, since M =1, S = 8 + O. But S $\leq$ 9, while 8 + O $\geq$ 10 since O $\geq$ 2. Hence the last equation has no solutions in our domain. This proves that the puzzle has no other solutions.

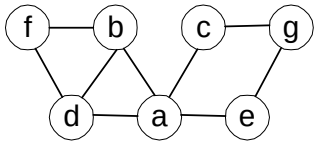
Solutions to Exercises 3.5

1. a. Here are the adjacency matrix and adjacency lists for the graph in question:

	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	1	0	0	1	0	1	0
c	1	0	0	0	0	0	1
d	1	1	0	0	0	1	0
e	1	0	0	0	0	0	1
f	0	1	0	1	0	0	0
g	0	0	1	0	1	0	0

$a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$
$b \rightarrow a \rightarrow d \rightarrow f$
$c \rightarrow a \rightarrow g$
$d \rightarrow a \rightarrow b \rightarrow f$
$e \rightarrow a \rightarrow g$
$f \rightarrow b \rightarrow d$
$g \rightarrow c \rightarrow e$

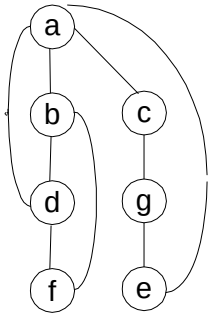
b. See below: (i) the graph; (ii) the traversal's stack (the first subscript number indicates the order in which the vertex was visited, i.e., pushed onto the stack, the second one indicates the order in which it became a dead-end, i.e., popped off the stack); (iii) the DFS tree (with the tree edges shown with solid lines and the back edges shown with dashed lines).



(i)

$f_{4,1}$ $e_{7,4}$
$d_{3,2}$ $g_{6,5}$
$b_{2,3}$ $c_{5,6}$
$a_{1,7}$

(ii)



(iii)

2. The time efficiency of DFS is $\Theta(|V|^2)$ for the adjacency matrix representation and $\Theta(|V| + |E|)$ for the adjacency lists representation, respectively. If $|E| \in O(|V|)$, the former remains $\Theta(|V|^2)$ while the latter becomes $\Theta(|V|)$. Hence, for sparse graphs, the adjacency lists version of DFS is more efficient than the adjacency matrix version.

3. a. The number of DFS trees is equal to the number of connected components of the graph. Hence, it will be the same for all DFS traversals of the graph.

b. For a connected (undirected) graph with $|V|$ vertices, the number of

tree edges $|E^{(tree)}|$ in a DFS tree will be $|V|-1$ and, hence, the number of back edges $|E^{(back)}|$ will be the total number of edges minus the number of tree edges: $|E|-(|V|-1) = |E|-|V|+1$. Therefore, it will be independent from a particular DFS traversal of the same graph. This observation can be extended to an arbitrary graph with $|C|$ connected components by applying this reasoning to each of its connected components:

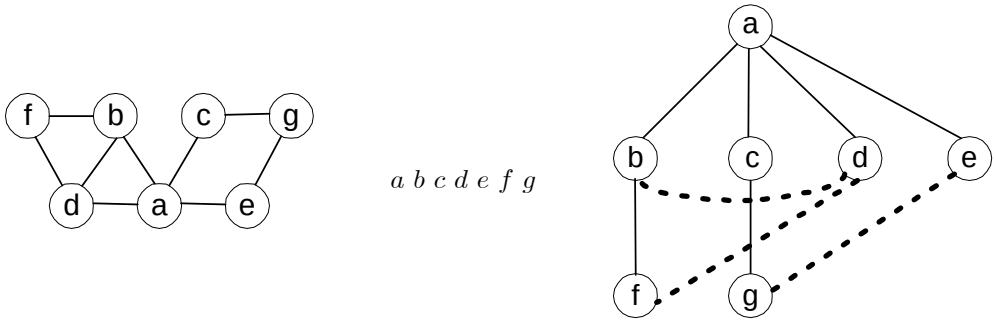
$$|E^{(tree)}| = \sum_{c=1}^{|C|} |E_c^{(tree)}| = \sum_{c=1}^{|C|} (|V_c| - 1) = \sum_{c=1}^{|C|} |V_c| - \sum_{c=1}^{|C|} 1 = |V| - |C|$$

and

$$|E^{(back)}| = |E| - |E^{(tree)}| = |E| - (|V| - |C|) = |E| - |V| + |C|,$$

where $|E_c^{(tree)}|$ and $|V_c|$ are the numbers of tree edges and vertices in the cth connected component, respectively.

4. Here is the result of the BFS traversal of the graph of Problem 1:

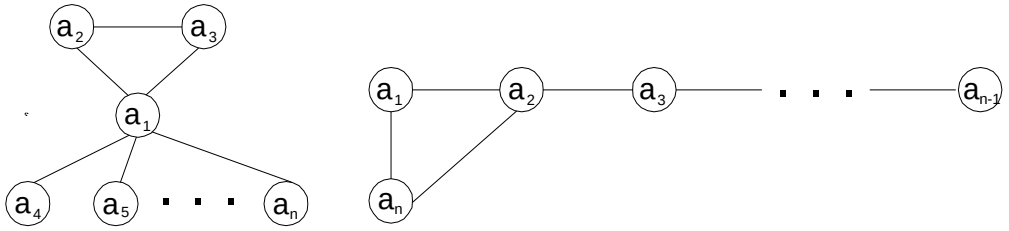


(i) the graph; (ii) the traversal's queue; (iii) the tree (the tree and cross edges are shown with solid and dotted lines, respectively).

5. We'll prove the assertion in question by contradiction. Assume that a BFS tree of some undirected graph has a cross edge connecting two vertices u and v such that $level[u] \geq level[v] + 2$. But $level[u] = d[u]$ and $level[v] = d[v]$, where $d[u]$ and $d[v]$ are the lengths of the minimum-edge paths from the root to vertices u and v, respectively. Hence, we have $d[u] \geq d[v] + 2$. The last inequality contradicts the fact that $d[u]$ is the length of the minimum-edge path from the root to vertex u because the minimum-edge path of length $d[v]$ from the root to vertex v followed by edge (v, u) has fewer edges than $d[u]$.

6. a. A graph has a cycle if and only if its BFS forest has a cross edge.

   b. Both traversals, DFS and BFS, can be used for checking a graph's acyclicity. For some graphs, a DFS traversal discovers a back edge in its DFS forest sooner than a BFS traversal discovers a cross edge (see example (i) below); for others the exactly opposite is the case (see example (ii) below).



(i) (ii)

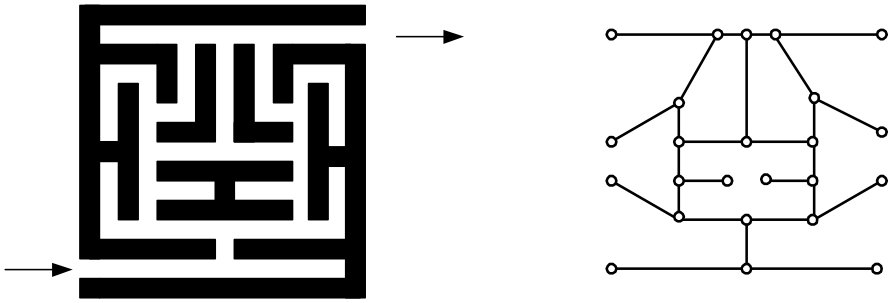
7. Start a DFS (or BFS) traversal at an arbitrary vertex and mark the visited vertices with 1. By the time the traversal's stack (queue) becomes empty, all the vertices in the same connected component as the starting vertex, and only they, will have been marked with 1. If there are unvisited vertices left, restart the traversal at one of them and mark all the vertices being visited with 2, and so on until no unvisited vertices are left.

8. a. Let F be a DFS forest of a graph. It is not difficult to see that F is 2-colorable if and only if there is no back edge connecting two vertices both on odd levels or both on even levels. It is this property that a DFS traversal needs to verify. Note that a DFS traversal can mark vertices as even or odd when it reaches them for the first time.

   b. Similarly to part (a), a graph is 2-colorable if and only if its BFS forest has no cross edge connecting vertices on the same level. Use a BFS traversal to check whether or not such a cross edge exists.

9. n/a

10. a. Here is the maze and a graph representing it:



b. DFS is much more convenient for going through a maze than BFS. When DFS moves to a next vertex, it is connected to a current vertex by an edge (i.e., "close nearby" in the physical maze), which is not generally the case for BFS. In fact, DFS can be considered a generalization of an ancient right-hand rule for maze traversal: go through the maze in such a way so that your right hand is always touching a wall.

11. The sequence shown in the figure below solves the puzzle in six steps, which is the minimum.

Step#	8-pint jug	5-pint jug	3-pint jug
	8	0	0
1	3	5	0
2	3	2	3
3	6	2	0
4	6	0	2
5	1	5	2
6	1	4	3

Solution to the Three Jugs puzzle

Although the solution can be obtained by trial and error, there is a systematic way of getting to it. We can represent a state of the jars by a triple of nonnegative integers indicating the amount of water in the 3-pint, 5-pint, and 8-pint jugs, respectively. Thus, we start with the triple 008. We will consider all legal transformations from a current state of the jugs to new possible states in the BFS manner. We initialize the queue with the initial state triple 008 and repeat the following until a desired state—a triple containing a 4—is encountered for the first time. For the state at the front of the queue, label all the *new*

states reachable from it by the triple of the front state, add them to the queue, and then delete the front state from the queue. After a desired state is reached for the first time, follow the labels backwards to get the shortest sequence of transformations that solve the puzzle.

The application of this algorithm to the puzzle's data yields the following sequence of the queue states, where the aforementioned labels are shown as subscripts the first time the new triples are added to the queue:

$$008 \mid 305_{008}, 053_{008} \mid 053,\ 035_{305}, 350_{305} \mid 035,\ 350,\ 323_{053} \mid 350,\ 323,\ 332_{035} \mid 323,\ 332 \mid$$
$$332,\ 026_{323} \mid 026,\ 152_{332} \mid 152,\ 206_{026} \mid 206,\ 107_{152} \mid 107,\ 251_{206} \mid 251,\ 017_{107} \mid 017,\ 341_{251}$$

Tracing the labels from that of 341 backwards, we get the following transformation sequence that solves the puzzle in the minimum number of six steps:

$$008 \to 053 \to 323 \to 026 \to 206 \to 251 \to 341.$$

Solutions to Exercises 4.1

1. First, the two boys take the boat to the other side, after which one of them returns with the boat. Then a soldier takes the boat to the other side and stays there while the other boy returns the boat. These four trips reduce the problem's instance of size n (measured by the number of soldiers to be ferried) to the instance of size $n-1$. Thus, if this four-trip procedure is repeated the total of n times, the problem will be solved after the total of $4n$ trips.

2. a. Assuming that the glasses are numbered left to right from 1 to $2n$, pour soda from glass 2 into glass $2n-1$. This makes the first and last pair of glasses alternate in the required pattern and hence reduces the problem to the same problem with $2(n-2)$ middle glasses. If n is even, the number of times this operation needs to be repeated is equal to $n/2$; if n is odd, it is equal to $(n-1)/2$. The formula $\lfloor n/2 \rfloor$ provides a closed-form answer for both cases. Note that this can also be obtained by solving the recurrence $M(n) = M(n-2) + 1$ for $n > 2$, $M(2) = 1$, $M(1) = 0$, where $M(n)$ is the number of moves made by the decrease-by-two algorithm described above. Since any algorithm for this problem must move at least one filled glass for each of the $\lfloor n/2 \rfloor$ nonoverlapping pairs of the filled glasses, $\lfloor n/2 \rfloor$ is the least number of moves needed to solve the problem.

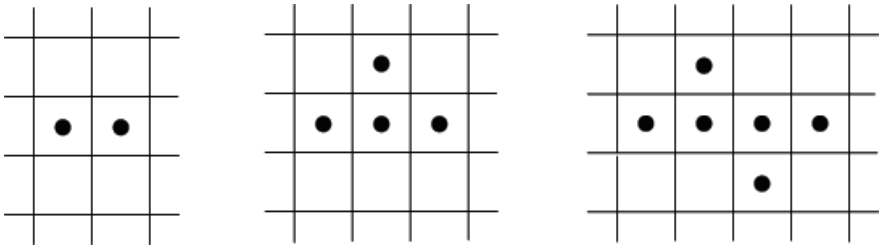
   For an alternative algorithm, see a more general version of the problem in part (b).

   Note: The problem was discussed in Martin Gardner's *aha!Insight*, Scientific American/W.H.Freeman, p. 7.

   b. In the final state of the glasses, all the glasses in the odd positions have to be filled and all the glasses in the even positions must be empty. If in the initial state of the puzzle there are k $(0 \le k \le n)$ full glasses in even positions, there are also k empty glasses in odd positions. To solve the puzzle in the minimum number of moves, it's necessary and sufficient to pour soda from the k glasses in the even positions into the k empty glasses in the odd positions. To find needed pairs of such glasses, one can simply scan the row of the glasses to find the next full glass in an even position and the next empty glass in an odd position.

3. For $n = 2$, an obvious solution is depicted in the figure below (the first figure) . Marking two cells adjacent to, say, the rightmost cell in this solution—one horizontally and the other vertically (say, up)—yields a solution for $n = 4$ (the middle figure). Repeating the same operation again but marking the vertical neighbor below rather than above the rightmost

cell, yields a solution for $n = 6$ (the right figure). In this manner, we can solve the puzzle for any even value of n.



Solutions to the *Marking Cells* puzzle for $n = 2$, $n = 4$, and $n = 6$

Note: The problem is from B. A. Kordemsky's *Mathematical Charmers*, Oniks, 2005 (in Russian).

4. Here is a general outline of a recursive algorithm that create list $L(n)$ of all the subsets of $\{a_1, ..., a_n\}$ (see a more detailed discussion in Section 4.3):

   if $n = 0$ **return** list $L(0)$ containing the empty set as its only element
   else create recursively list $L(n-1)$ of all the subsets of $\{a_1, ..., a_{n-1}\}$
   append a_n to each element of $L(n-1)$ to get list T
   return $L(n)$ obtained by concatenation of $L(n-1)$ and T

5. The line if **not** *Connected*$(A[0..n-2, 0..n-2])$ **return** 0 is incorrect. As a counter-example, consider a graph in which the first $n-1$ points have no edges between them but each has an edge connecting it to the nth vertex.

6. Initialize the desired list with any of the teams. For each of the other teams, scan the list to insert it before the first team it didn't loose or at the list's end if it lost to all the teams currently on the list. The efficeincy of the algorithm is $O(n^2)$, because in the worst case each of the teams will be inserted in the end of the list after checking $1+2+...+(n-1) = (n-1)n/2$ teams already on the list.

7. Sorting the list E,X,A, M, P, L, E in alphabetical order with insertion sort:

$$
\begin{array}{lllllll}
E & X & A & M & P & L & E \\
E & \mid \mathbf{X} & & & & & \\
E & X & \mid \mathbf{A} & & & & \\
A & E & X & \mid \mathbf{M} & & & \\
A & E & M & X & \mid \mathbf{P} & & \\
A & E & M & P & X & \mid \mathbf{L} & \\
A & E & L & M & P & X & \mid \mathbf{E} \\
A & E & E & L & M & P & X
\end{array}
$$

8. a. -∞ or, more generally, any value less than or equal to every element in the array.

   b. Yes, the efficiency class will stay the same. The number of key comparisons for strictly decreasing arrays (the worst-case input) will be

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

9. Yes, but we will have to scan the sorted part left to right while inserting $A[i]$ to get the same $O(n^2)$ efficiency as the array version.

10. The efficiency classes of both versions will be the same. The inner loop of *InsertionSort* consists of one key assignment and one index decrement; the inner loop of *InsertionSort2* consists of one key swap (i.e., three key assignments) and one index decrement. If we disregard the time spent on the index decrements, the ratio of the running times should be estimated as $3c_a/c_a = 3$; if we take into account the time spent on the index decrements, the ratio's estimate becomes $(3c_a + c_d)/(c_a + c_d)$,where c_a and c_d are the times of one key assignment and one index decrement, respectively.

11. a. The largest number of inversions for $A[i]$ $(0 \le i \le n-1)$ is $n-1-i$; this happens if $A[i]$ is greater than all the elements to the right of it. Therefore, the largest number of inversions for an entire array happens for a strictly decreasing array. This largest number is given by the sum:

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

    The smallest number of inversions for $A[i]$ $(0 \le i \le n-1)$ is 0; this happens if $A[i]$ is smaller than or equal to all the elements to the right of it. Therefore, the smallest number of inversions for an entire array will be 0 for nondecreasing arrays.

    b. Assuming that all elements are distinct and that inserting $A[i]$ in each of the $i+1$ possible positions among its predecessors is equally likely, we obtain the following for the expected number of key comparisons on the ith iteration of the algorithm's sentinel version:

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

    Hence for the average number of key comparisons, $C_{avg}(n)$,we have

$$C_{avg}(n) = \sum_{i=1}^{n-1} \frac{i+2}{2} = \frac{1}{2} \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 = \frac{1}{2} \frac{(n-1)n}{2} + n - 1 \approx \frac{n^2}{4}.$$

For the no-sentinel version, the number of key comparisons to insert $A[i]$ before and after $A[0]$ will be the same. Therefore the expected number of key comparisons on the ith iteration of the no-sentinel version is:

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

Hence, for the average number of key comparisons, $C_{avg}(n)$,we have

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

We have a closed-form formula for the first sum:

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

The second sum can be estimated as follows:

$$\sum_{i=1}^{n-1}\frac{i}{i+1} = \sum_{i=1}^{n-1}(1-\frac{1}{i+1}) = \sum_{i=1}^{n-1} 1 - \sum_{i=1}^{n-1}\frac{1}{i+1} = n-1-\sum_{j=2}^{n} j = n - H_n,$$

where $H_n = \sum_{j=1}^{n} 1/j \approx \ln n$ according to a well-known formula quoted in Appendix A. Hence, for the no-sentinel version of insertion sort too, we have

$$C_{avg}(n) \approx \frac{n^2-n}{4} + n - H_n \approx \frac{n^2}{4}.$$

12. a. Applying shellsort to the list S_1,H,E_1,L_1,L_2,S_2,O,R,T,I,S_3,U_1,S_4,E_2,F,U_2,L_3 with the step-sizes 13, 4, and 1 yields the following. (If a comparison causes

a swap, only the swap's result is shown.)

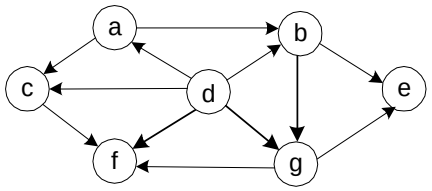
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S_1	H	E_1	L_1	L_2	S_2	O	R	T	I	S_3	U_1	S_4	E_2	F	U_2	L_3
E_2													S_1			
	F													H		
		E_1													U_2	
			L_1													L_3
E_2				L_2												
	F				S_2											
		E_1				O										
			L_1				R									
				L_2				T								
					I				S_2							
	F				I											
						O				S_3						
							R				U_1					
								S_4				T				
				L_2				S_4								
									S_2				S_1			
										H				S_3		
						H				O						
		E_1				H										
											U_1				U_2	
												L_3				T
								L_3				S_4				
				L_2				L_3								
E_2	F	E_1	L_1	L_2	I	H	R	L_3	S_2	O	U_1	S_4	S_1	S_3	U_2	T

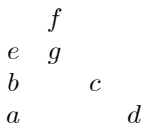
The final pass with the step-size 1—sorting the last array by insertion sort—is omitted from the solution because of its simplicity. Note that since relatively few elements in the last array are out of order as a result of the work done on the preceding passes of shellsort, insertion sort will need significantly fewer comparisons to finish the job than it would have needed if it were applied to the initial array.

b. Shellsort is not stable. As a counterexample for shellsort with the sequence of step-sizes 4 and 1, consider, say, the array 5, 1, 2, 3, 1. The first pass with the step-size of 4 will exchange 5 with the last 1, changing the relative ordering of the two 1's in the array. The second pass with the step-size of 1, which is insertion sort, will not make any exchanges because the array is already sorted.

Solutions to Exercises 4.2

1. a. The digraph and the stack of its DFS traversal that starts at vertex a are given below:





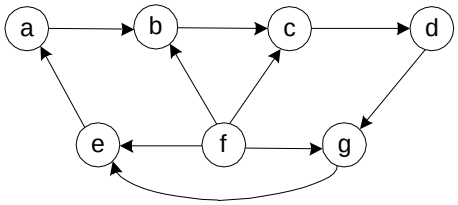
The vertices are popped off the stack in the following order:

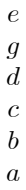
$$e\ f\ g\ b\ c\ a\ d.$$

The topological sorting order obtained by reversing the list above is

$$d\ a\ c\ b\ g\ f\ e.$$

b. The digraph below is not a dag. Its DFS traversal that starts at a encounters a back edge from e to a:





2. a. Let us prove by contradiction that if a digraph has a directed cycle, then the topological sorting problem does not have a solution. Assume that $v_{i_1}, ..., v_{i_n}$ is a solution to the topological sorting problem for a digraph with a directed cycle. Let v_{i_k} be the leftmost vertex of this cycle on the list $v_{i_1}, ..., v_{i_n}$. Since the cycle's edge entering v_{i_k} goes right to left, we have a contradiction that proves the assertion.

If a digraph has no directed cycles, a solution to the topological sorting problem is fetched by either of the two algorithms discussed in the section. (The correctness of the DFS-based algorithm was explained there; the correctness of the source removal algorithm stems from the assertion of Problem 6a.)

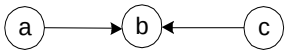
b. For a digraph with n vertices and no edges, any permutation of its

vertices solves the topological sorting problem. Hence, the answer to the question is $n!$.

3. a. Since reversing the order in which vertices have been popped off the DFS traversal stack is in $\Theta(|V|)$, the running time of the algorithm will be the same as that of DFS (except for the fact that it can stop before processing the entire digraph if a back edge is encountered). Hence, the running time of the DFS-based algorithm is in $O(|V|^2)$ for the adjacency matrix representation and in $O(|V|+|E|)$ for the adjacency lists representation.

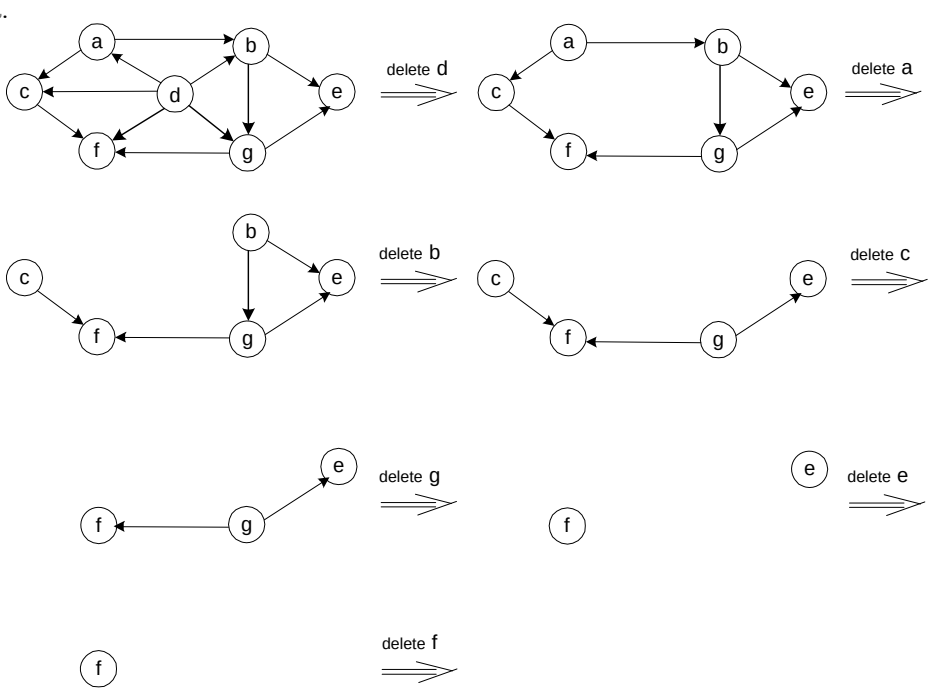
   b. Fill the array of length $|V|$ with vertices being popped off the DFS traversal stack right to left.

4. The answer is no. Here is a simple counterexample:



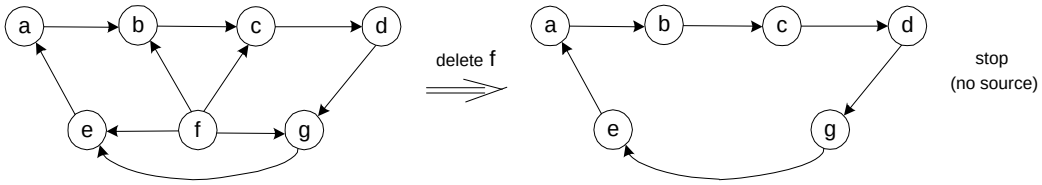
   The DFS traversal that starts at a pushes the vertices on the stack in the order a, b, c, and neither this ordering nor its reversal solves the topological sorting problem correctly.

5. a.



The topological ordering obtained is $d \quad a \quad b \quad c \quad g \quad e \quad f$.

b.



The topological sorting is impossible.

6. a. Assume that, on the contrary, there exists a dag with every vertex having an incoming edge. Reversing all its edges would yield a dag with every vertex having an outgoing edge. Then, starting at an arbitrary vertex and following a chain of such outgoing edges, we would get a directed cycle no later than after $|V|$ steps. This contradiction proves the assertion.

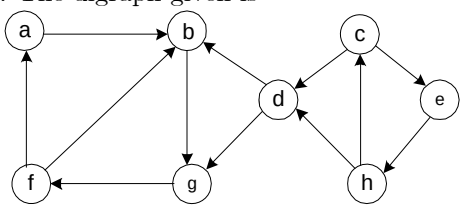
   b. A vertex of a dag is a source if and only if its column in the adjacency matrix contains only 0's. Looking for such a column is a $O(|V|^2)$ operation.

   c. A vertex of a dag is a source if and only if this vertex appears in none of the dag's adjacency lists. Looking for such a vertex is a $O(|V| + |E|)$ operation.

7. The answer to this well-known problem is yes (see, e.g., [KnuI], pp. 264-265).

8. n/a

9. a. The digraph given is

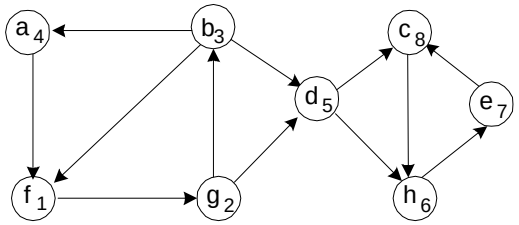


The stack of the first DFS traversal, with a as its starting vertex, will look as follows:

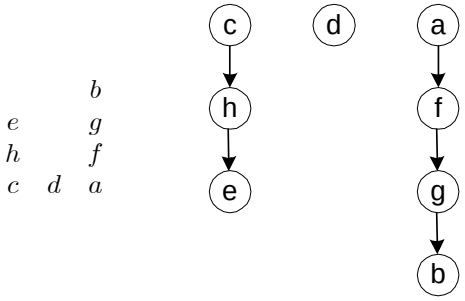
f_1
g_2 $\quad$ h_6
b_3 $\quad$ d_5 $\quad$ e_7
a_4 $\quad$ c_8

(The numbers indicate the order in which the vertices are popped off the stack.)

The digraph with the reversed edges is



The stack and the DFS trees (with only tree edges shown) of the DFS traversal of the second digraph will be as follows:



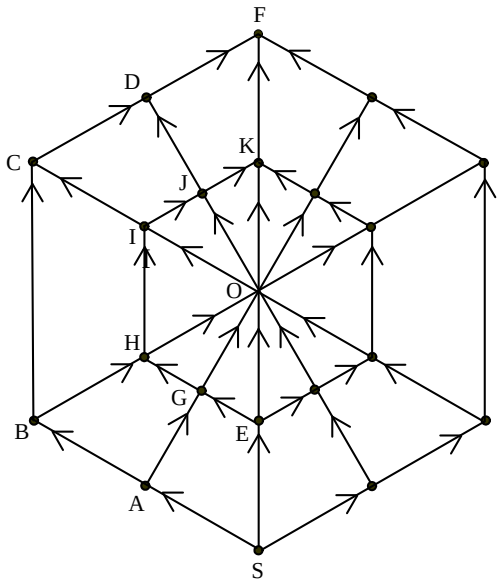
The strongly connected components of the given digraph are:

$$\{c, h, e\}, \quad \{d\}, \quad \{a, f, g, b\}.$$

b. If a graph is represented by its adjacency matrix, then the efficiency of the first DFS traversal will be in $\Theta(|V|^2)$. The efficiency of the edge-reversal step (set $B[j, i]$ to 1 in the adjacency matrix of the new digraph if $A[i, j] = 1$ in the adjacency matrix of the given digraph and to 0 otherwise) will also be in $\Theta(|V|^2)$. The time efficiency of the last DFS traversal of the new graph will be in $\Theta(|V|^2)$, too. Hence, the efficiency of the entire algorithm will be in $\Theta(|V|^2) + \Theta(|V|^2) + \Theta(|V|^2) = \Theta(|V|^2)$.

The answer for a graph represented by its adjacency lists will be, by similar reasoning (with a necessary adjustment for the middle step), in $\Theta(|V| + |E|)$.

10. The total number of directed paths from S to a vertex v of the spider's digraph can be obtained as the sum of the directed paths from S to all the vertices u for which there is a directed edge from u to v. Because of the digraph's symmetry with respect to the "straight" four-edge path from S to F, for each of the verices on this path including F, the sum of the paths can be simplified by doubling the number of paths entering the vertex from the left and disregarding the paths entering the vertex from the right. To be able to compute the path sums, we need to topologically sort the left part of the digraph; a result of this linear ordering is shown in the list below. Then the sums can be computed by processing the vertices in this linear order, doubling the countribution for every edge entering the vertex in question from the left. These numbers are shown in the list given.



S(1)–A(1)–B(1)–E(1)–G(1)–H(3)–O(11)–I(14)–C(15)–J(25)–D(40)–K(61)–F(141).

Thus, there are 141 paths from S to F.

Solutions to Exercises 4.3

1. Since $25! \approx 1.5 \cdot 10^{25}$, it would take an unrealistically long time to generate this number of permutations even on a supercomputer. On the other hand, $2^{25} \approx 3.3 \cdot 10^7$, which would take about 0.3 seconds to generate on a computer making one hundred million operations per second.

2. a. The permutations of $\{1, 2, 3, 4\}$ generated by the bottom-up minimal-change algorithm:

start	1			
insert 2 into 1 right to left	12	21		
insert 3 into 12 right to left	123	132	312	
insert 3 into 21 left to right	321	231	213	
insert 4 into 123 right to left	1234	1243	1423	4123
insert 4 into 132 left to right	4132	1432	1342	1324
insert 4 into 312 right to left	3124	3142	3412	4312
insert 4 into 321 left to right	4321	3421	3241	3214
insert 4 into 231 right to left	2314	2341	2431	4231
insert 4 into 213 left to right	4213	2413	2143	2134

b. The permutations of $\{1, 2, 3, 4\}$ generated by the Johnson-Trotter algorithm. (Read horizontally; the largest mobile element is shown in bold.)

$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{3}\,\overleftarrow{\mathbf{4}}$	$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{4}}\,\overleftarrow{3}$	$\overleftarrow{1}\,\overleftarrow{\mathbf{4}}\,\overleftarrow{2}\,\overleftarrow{3}$	$\overleftarrow{4}\,\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}}$
$\overrightarrow{\mathbf{4}}\,\overleftarrow{1}\,\overleftarrow{3}\,\overleftarrow{2}$	$\overleftarrow{1}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{3}\,\overleftarrow{2}$	$\overleftarrow{1}\,\overleftarrow{3}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{2}$	$\overleftarrow{1}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{4}$
$\overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{4}}$	$\overleftarrow{3}\,\overleftarrow{1}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{2}$	$\overleftarrow{3}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{1}\,\overleftarrow{2}$	$\overleftarrow{4}\,\overleftarrow{3}\,\overleftarrow{1}\,\overrightarrow{\mathbf{2}}$
$\overrightarrow{\mathbf{4}}\,\overrightarrow{3}\,\overleftarrow{2}\,\overleftarrow{1}$	$\overrightarrow{3}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{2}\,\overleftarrow{1}$	$\overrightarrow{3}\,\overleftarrow{2}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{1}$	$\overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1}\,\overleftarrow{4}$
$\overleftarrow{2}\,\overrightarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{4}}$	$\overleftarrow{2}\,\overrightarrow{3}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{1}$	$\overleftarrow{2}\,\overrightarrow{\mathbf{4}}\,\overrightarrow{3}\,\overleftarrow{1}$	$\overleftarrow{4}\,\overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1}$
$\overrightarrow{\mathbf{4}}\,\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}$	$\overleftarrow{2}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{1}\,\overrightarrow{3}$	$\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{\mathbf{4}}\,\overrightarrow{3}$	$\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}\,\overrightarrow{4}$

c. The permutations of $\{1, 2, 3, 4\}$ generated in lexicographic order. (Read horizontally.)

1234	1243	1324	1342	1423	1432
2134	2143	2314	2341	2413	2431
3124	3142	3214	3241	3412	3421
4123	4132	4213	4231	4312	4321

3. 1223, 1232, 1322, 2123, 2132, 2213, 2231, 2312, 2321, 3122, 3212, 3221.

4. a. For $n = 2$:
12 21

For $n = 3$ (read along the rows):
123 213
312 132
231 321

For $n = 4$ (read along the rows):

1234	2134	3124	1324	2314	3214
4231	2431	3421	4321	2341	3241
4132	1432	3412	4312	1342	3142
4123	1423	2413	4213	1243	2143

b. Let $C(n)$ be the number of times the algorithm writes a new permutation (on completion of the recursive call when $n = 1$). We have the following recurrence for $C(n)$:

$$C(n) = \sum_{i=1}^{n} C(n-1) \quad \text{or} \quad C(n) = nC(n-1) \text{ for } n > 1, \quad C(1) = 1.$$

Its solution (see Section 2.4) is $C(n) = n!$. The fact that all the permutations generated by the algorithm are distinct, can be proved by mathematical induction.

c. We have the following recurrence for the number of swaps $S(n)$:

$$S(n) = \sum_{i=1}^{n} (S(n-1)+1) \quad \text{or} \quad S(n) = nS(n-1)+n \quad \text{for } n > 1, \quad S(1) = 0.$$

Although it can be solved by backward substitution, this is easier to do after dividing both hand sides by $n!$

$$\frac{S(n)}{n!} = \frac{S(n-1)}{(n-1)!} + \frac{1}{(n-1)!} \quad \text{for } n > 1, \quad S(1) = 0$$

and substituting $T(n) = \frac{S(n)}{n!}$ to obtain the following recurrence:

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

Solving the last recurrence by backward substitutions yields

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

On returning to variable $S(n) = n!T(n)$, we obtain

$$S(n) = n! \sum_{i=1}^{n-1} \frac{1}{i!} \approx n!(e - 1 - \frac{1}{n!}) \in \Theta(n!).$$

5. Generate all the subsets of a four-element set $A = \{a_1, a_2, a_3, a_4\}$ bottom up:

n	subsets							
0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$
4	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$
	$\{a_4\}$	$\{a_1, a_4\}$	$\{a_2, a_4\}$	$\{a_1, a_2, a_4\}$	$\{a_3, a_4\}$	$\{a_1, a_3, a_4\}$	$\{a_2, a_3, a_4\}$	$\{a_1, a_2, a_3, a_4\}$

Generate all the subsets of a four-element set $A = \{a_1, a_2, a_3, a_4\}$ with bit vectors:

bit strings	0000	0001	0010	0011	0100	0101	0110	0111
subsets	$\varnothing$	$\{a_4\}$	$\{a_3\}$	$\{a_3, a_4\}$	$\{a_2\}$	$\{a_2, a_4\}$	$\{a_2, a_3\}$	$\{a_2, a_3, a_4\}$
bit strings	1000	1001	1010	1011	1100	1101	1110	1111
subsets	$\{a_1\}$	$\{a_1, a_4\}$	$\{a_1, a_3\}$	$\{a_1, a_3, a_4\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_4\}$	$\{a_1, a_2, a_3\}$	$\{a_1, a_2, a_3, a_4\}$

6. Establish the correspondence between subsets of $A = \{a_1, ..., a_n\}$ and bit strings $b_1...b_n$ of length n by associating bit i with the presence or absence of element a_{n-i+1} for $i = 1, ..., n$.

7. **Algorithm** *BitstringsRec*(n)
   //Generates recursively all the bit strings of a given length
   //Input: A positive integer n
   //Output: All bit strings of length n as contents of global array $B[0..n-1]$
   if $n = 0$
   print(B)
   else
   $B[n-1] \leftarrow 0$; *BitstringsRec*$(n-1)$
   $B[n-1] \leftarrow 1$; *BitstringsRec*$(n-1)$

8. **Algorithm** *BitstringsNonrec*(n)
   //Generates nonrecursively all the bit strings of a given length
   //Input: A positive integer n

```
//Output: All bit strings of length n as contents of global array B[0..n-1]
for i ← 0 to n − 1 do
    B[i] = 0
repeat
    print(B)
    k ← n − 1
    while k ≥ 0 and B[k] = 1
        k ← k − 1
    if k ≥ 0
        B[k] ← 1
        for i ← k + 1 to n − 1 do
            B[i] ← 0
until k = −1
```

9. a. The Gray code for $n = 3$ is given at the end of the section:

$$000 \quad 001 \quad 011 \quad 010 \quad 110 \quad 111 \quad 101 \quad 100.$$

Following the $BRGC(n)$ algorithm, we obtain the binary reflected Gray code for $n = 4$ as follows:

$L1$ 000 001 011 010 110 111 101 100

$L2$ 100 101 111 110 010 011 001 000

L 0000 0001 0011 0010 0110 0111 0101 0100 1100 1101 1111 1110 1010 1011 1001 1000

b. Tracing the nonrecursive algorithm to generate the binary reflexive Gray code of order 4 given in the problem's statement, we obtain the following.

i	0	1	2	3	4	5	6	7
i in binary	0	1	10	11	100	101	110	111
Gray code	0000	0001	0011	0010	0110	0111	0101	0100
i	8	9	10	11	12	13	14	15
i in binary	1000	1001	1010	1011	1100	1101	1110	1111
Gray code	1100	1101	1111	1110	1010	1011	1001	1000

10. Here is a recursive algorithm from "Problems on Algorithms" by Ian Parberry [Par95, p.120]:

call $Choose(1, k)$ where

Algorithm $Choose(i, k)$
//Generates all k-subsets of $\{i, i + 1, ..., n\}$ stored in global array $A[1..k]$
//in descending order of their components

```
if k = 0
    print(A)
else
    for j ← i to n − k + 1 do
        A[k] ← j
        Choose(j + 1, k − 1)
```

11. a. Number the disks from 1 to n in increasing order of their size. The disk movements will be represented by a tuple of n bits, in which the bits will be counted right to left so that the rightmost bit will represent the movements of the smallest disk and the leftmost bit will represent the movements of the largest disk. Initialize the tuple with all 0's. For each move in the puzzle's solution, flip the ith bit if the move involves the ith disk.

    b. Use the correspondence described in part a between bit strings of the binary reflected Gray code and the disk moves in the Tower of Hanoi puzzle with the following additional rule for situations when there is a choice of where to place a disk: When faced with a choice in placing a disk, always place an odd numbered disk on top of an even numbered disk; if an even numbered disk is not available, place the odd numbered disk on an empty peg. Similarly, place an even numbered disk on an odd disk, if available, or else on an empty peg.

12. The problem can be solved by the following recursive algorithm for pushing the buttons numbered from 1 to n. If $n = 1$ and the light bulb is not turned on, push button 1. If $n > 1$ and the light bulb is not turned on, push recursively the first $n - 1$ buttons. If this fails to turn the light bulb on, push button n and then push recursively the first $n - 1$ buttons again. The recurrence for the number of button pushes in the worst case is

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

    It is identical to the recurrence for the Tower of Hanoi puzzle discussed in Section 2.4, whose solution is $M(n) = 2^n - 1$.

    Alternatively, since a switch can be in one of the two states, it can be thought of as a bit in an n-bit string in which 0 and 1 represent, say, the initial and opposite states of the switch, respectively. The total number of such bit strings (switch configurations) is equal to 2^n; one of them represents an initial state, the remaining $2^n - 1$ bit strings contain the one that will turn on the light bulb. In the worst case, all these $2^n - 1$ switch combinations will have to be checked. To accomplish this with the minimum number of button pushes, every push must produce a new switch combination. In particular, we can take advantage of the binary reflected Gray code as follows. Number the switches from 1 to n right to left and

use the sequence of the Gray code's bit strings for guidance which buttons to push: if the next bit string differs from its immediate predecessor in the ith bit from the right, push button number i. For example, for $n = 4$, the Gray code is

0000	0001	0011	0010	0110	0111	0101	0100
1100	1101	1111	1110	1010	1011	1001	1000

and it guids to push the buttons in the following sequence:

121312141213121.

Solutions to Exercises 4.4

1. Since cutting several pieces of a given stick at the same time is allowed, we need to concern ourselves only with finding a cutting algorithm that reduces the size of the longest piece present to size 1. This implies that on each iteration an optimal algorithm must cut the longest piece—and simultaneously all the other pieces whose size is greater than 1—by half (or as close to this as possible). That is, it cuts every piece of size $l > 1$ into two pieces of lengths $\lceil l/2 \rceil$ and $\lfloor l/2 \rfloor$, respectively. The iterations stop after the longest—and, hence, all the other pieces of the stick—has length 1. The number of cuts (iterations) such an optimal algorithm makes for an n-unit stick is equal to $\lceil \log_2 n \rceil$, which is the least k such that $2^k \geq n$. More formally, the number of cut $C(n)$ can be obtained by solving the recurrence

$$C(n) = C(\lceil n/2 \rceil) + 1 \quad \text{for } n > 1, \quad C(1) = 0.$$

2. **Algorithm** *LogFloor*(n)
   //Input: A positive integer n
   //Output: Returns $\lfloor \log_2 n \rfloor$
   if $n = 1$ **return** 0
   else return *LogFloor*$(\lfloor \frac{n}{2} \rfloor) + 1$

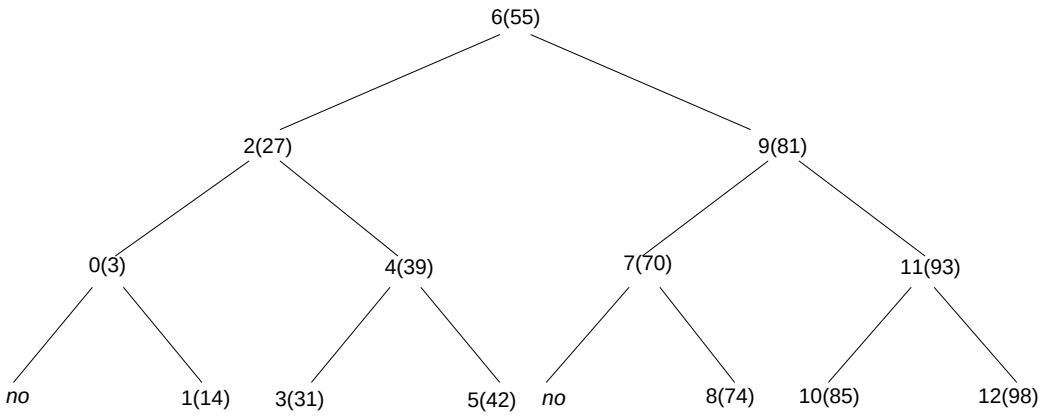
   The algorithm is almost identical to the algorithm for computing the number of binary digits, which was investigated in Section 2.4. The recurrence relation for the number of additions is

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

   Its solution is $A(n) = \lfloor \log_2 n \rfloor \in \Theta(\log n)$.

3. a. According to formula (4.5), $C_{worst}(13) = \lceil \log_2(13 + 1) \rceil = 4$.

   b. In the comparison tree below, the first number indicates the element's index, the second one is its value:



The searches for each of the elements on the last level of the tree, i.e., the elements in positions 1(14), 3(31), 5(42), 8(74), 10(85), and 12(98) will require the largest number of key comparisons.

c. $C_{avg}^{yes} = \frac{1}{13} \cdot 1 \cdot 1 + \frac{1}{13} \cdot 2 \cdot 2 + \frac{1}{13} \cdot 3 \cdot 4 + \frac{1}{13} \cdot 4 \cdot 6 = \frac{41}{13} \approx 3.2$.

d. $C_{avg}^{no} = \frac{1}{14} \cdot 3 \cdot 2 + \frac{1}{14} \cdot 4 \cdot 12 = \frac{54}{14} \approx 3.9$.

4. For the successful search, the ratio in question can be estimated as follows:

$$\frac{C_{avg}^{seq.}(n)}{C_{avg}^{bin.}(n)} \approx \frac{n/2}{\log_2 n} = (\text{for } n = 10^6)\frac{10^6/2}{\log_2 10^6} = \frac{1}{2 \cdot 6}\frac{10^6}{\log_2 10} \approx 25,000.$$

5. Unlike an array, where any element can be accessed in constant time, reaching the middle element in a linked list is a $\Theta(n)$ operation. Hence, though implementable in principle, binary search would be a horribly inefficient algorithm for searching in a (sorted) linked list.

6. a. Here is pseudocode of the algorithm in question.
Algorithm *TwoWayBinary Search*($A[0..n-1]$, K)
//Implements binary search with two-way comparisons
//Input: A sorted array $A[0..n-1]$ and a search key K
//Output: An index of the array's element equal to K
// or -1 if there is no such element.
$l \leftarrow 0$; $r \leftarrow n-1$
while $l < r$ **do**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  if $K \le A[m]$
    $r \leftarrow m$
  else $l \leftarrow m+1$
if $K = A[l]$ **return** l
else return -1

b. Algorithm *TwoWayBinarySearch* makes $\lceil \log_2 n \rceil + 1$ two-way comparisons in the worst case, which is obtained by solving the recurrence $C_w(n) = C_w(\lceil n/2 \rceil) + 1$ for $n > 1$, $C_w(1) = 1$. Also note that the best-case efficiency of this algorithm is not in $\Theta(1)$ but in $\Theta(\log n)$.

7. Apply a two-way comparison version of binary search using the picture numbering. That is, assuming that pictures are numbered from 1 to 42, start with a question such as "Is the picture's number > 21?". The largest number of questions that may be required is 6. (Because the search can be assumed successful, one less comparison needs to be made than in *TwoWayBinarySearch*, yielding here $\lceil \log_2 42 \rceil = 6$.)

8. a. The algorithm is based on the decrease-by-a constant factor (equal to 3) strategy.

   b. $C(n) = 2 + C(n/3)$ for $n = 3^k$ $(k > 0)$, $C(1) = 1$.

   c. $C(3^k) = 2 + C(3^{k-1})$ [sub. $C(3^{k-1}) = 2 + C(3^{k-2})$]
   $= 2 + [2 + C(3^{k-2})] = 2 \cdot 2 + C(3^{k-2}) =$ [sub. $C(3^{k-2}) = 2 + C(3^{k-3})$]
   $= 2 \cdot 2 + [2 + C(3^{k-3})] = 2 \cdot 3 + C(3^{k-3}) = ... = 2i + C(3^{k-i}) = ... =$
   $2k + C(3^{k-k}) = 2 \log_3 n + 1$.

   d. We have to compare this formula with the worst-case number of key comparisons in the binary search, which is about $\log_2 n + 1$. Since

$$2 \log_3 n + 1 = 2 \frac{\log_2 n}{\log_2 3} + 1 = \frac{2}{\log_2 3} \log_2 n + 1$$

   and $2/\log_2 3 > 1$, binary search has a smaller multiplicative constant and hence is more efficient (by about the factor of $2/\log_2 3$) in the worst case, although both algorithms belong to the same logarithmic class.

9. The problem can be solved by the decrease-by-half algorithm that is based on following observation. Compare the middle element $A[m]$ with $m + 1$: if $A[m] = m + 1$, the missing number is larger than $m + 1$ and therefore should be searched for in the second half of the array; otherwise (i.e., if $A[m] > m + 1$), it should be searched for in the first half of the array). Here is pseudocode for a nonrecursive version of the algorithm.

   Algorithm *MissingNumber*$(A[0..n-2])$
   //Input: An increasing array of $n - 1$ integers in the range from 1 to n
   //Output: An integer from 1 to n that is not in the array
   $l \leftarrow 0$; $r \leftarrow n - 2$
   while $l < r$ **do**
   $m \leftarrow \lfloor (l + r)/2 \rfloor$
   if $A[m] = m + 1$
   $l \leftarrow m + 1$
   else $r \leftarrow m - 1$
   if $A[l] = l + 1$ **return** $l + 2$
   else return $l + 1$

   Note: The algorithm computing the missing number as the difference between $n(n+1)/2$ and the sum of the array's elements is obviously linear and hence less efficient than the logarithmic algorithm given above. But it has an advantage of not requiring the array's elements be sorted.

10. a. If n is a multiple of 3 (i.e., $n \bmod 3 = 0$), we can divide the coins into three piles of $n/3$ coins each and weigh two of the piles. If $n = 3k + 1$

(i.e., $n \bmod 3 = 1$), we can divide the coins into the piles of sizes k, k, and $k+1$ or $k+1$, $k+1$, and $k-1$. (We will use the second option.) Finally, if $n = 3k+2$ (i.e., $n \bmod 3 = 2$), we will divide the coins into the piles of sizes $k+1$, $k+1$, and k. The following pseudocode assumes that there is exactly one fake coin among the coins given and that the fake coin is lighter than the other coins.

if $n = 1$ the coin is fake
else divide the coins into three piles of $\lceil n/3 \rceil$, $\lceil n/3 \rceil$, and $n - 2\lceil n/3 \rceil$ coins
    weigh the first two piles
    if they weigh the same
        discard all of them and continue with the coins of the third pile
    else continue with the lighter of the first two piles

b. The recurrence relation for the number of weighing $W(n)$ needed in the worst case is as follows:

$$W(n) = W(\lceil n/3 \rceil) + 1 \quad \text{for } n > 1, \quad W(1) = 0.$$

For $n = 3^k$, the recurrence becomes $W(3^k) = W(3^{k-1}) + 1$. Solving it by backward substitutions yields $W(3^k) = k = \log_3 n$.

c. The ratio of the numbers of weighings in the worst case can be approximated for large values of n by

$$\frac{\log_2 n}{\log_3 n} = \frac{\log_2 n}{\log_3 2 \log_2 n} = \log_2 3 \approx 1.6.$$

11. a. Compute $26 \cdot 47$ by the multiplication à la russe algorithm:

n	m	
26	47	
13	94	94
6	188	
3	376	376
1	752	752
		1,222

b. Multiplication à la russe does $\lfloor \log_2 n \rfloor$ iterations to compute $n \cdot m$ and $\lfloor \log_2 m \rfloor$ to compute $m \cdot n$.

12. a. Here are pseudocodes for the nonrecursive and recurisive implementations of multiplication à la russe

Algorithm *Russe*(n, m)
//Implements multiplication à la russe nonrecursively
//Input: Two positive integers n and m
//Output: The product of n and m
$p \leftarrow 0$
while $n \neq 1$ **do**
    if $n \bmod 2 = 1 \quad p \leftarrow p + m$
    $n \leftarrow \lfloor n/2 \rfloor$
    $m \leftarrow 2 * m$
return $p + m$

Algorithm *RusseRec*(n, m)
//Implements multiplication à la russe recursively
//Input: Two positive integers n and m
//Output: The product of n and m
if $n \bmod 2 = 0$ **return** *RusseRec*$(n/2, 2m)$
else if $n = 1$ **return** m
else return *RusseRec*$((n-1)/2, 2m) + m$

b. The time efficiency class of multiplication à la russe is $\Theta(\log n)$ where n is the first factor of the product. As a function of b, the number of binary digits of n, it is $\Theta(b)$.

13. Using the fact that $J(n)$ can be obtained by a one-bit left cyclic shift of n, we get the following for $n = 40$:

$$J(40) = J(101000_2) = 10001_2 = 17.$$

14. We can use the fact that $J(n)$ can be obtained by a one-bit left cyclic shift of n. If $n = 2^k$, where k is a nonnegative integer, then $J(2^k) = J(1\underbrace{0...0}_{k \text{ zeros}}) = 1$.

15. a. Using the initial condition $J(1) = 1$ and the recurrences $J(2k) = 2J(k) - 1$ and $J(2k+1) = 2J(k) + 1$ for even and odd values of n, respectively, we obtain the following values of $J(n)$ for $n = 1, 2, ..., 15$:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$J(n)$	1	1	3	1	3	5	7	1	3	5	7	9	11	13	15

b. On inspecting the values obtained in part (a), it is not difficult to observe that for the n's values between consecutive powers of 2, i.e., for

$2^k \le n < 2^{k+1}$ $(k = 0, 1, 2, 3)$ or $n = 2^k + i$ where $i = 0, 1, ..., 2^k - 1$, the corresponding values of $J(n)$ run the range of odd numbers from 1 to $2^{k+1} - 1$. This observation can be expressed by the formula

$$J(2^k + i) = 2i + 1 \quad \text{for } i = 0, 1, ..., 2^k - 1.$$

We'll prove that this formula solves the recurrences of the Josephus problem for any nonnegative integer k by induction on k. For the basis value $k = 0$, we have $J(2^0 + 0) = 2 \cdot 0 + 1 = 1$ as it should for the initial condition. Assuming that for a given nonnegative integer k and for every $i = 0, 1, ..., 2^k - 1$, $J(2^k + i) = 2i + 1$, we need to show that

$$J(2^{k+1} + i) = 2i + 1 \quad \text{for } i = 0, 1, ..., 2^{k+1} - 1.$$

If i is even, it can be represented as $2j$ where $j = 0, 1, ..., 2^k - 1$. Then we obtain

$$J(2^{k+1} + i) = J(2(2^k + j)) = 2J(2^k + j) - 1$$

and, using the induction's assumption, we can continue as follows

$$2J(2^k + j) - 1 = 2[2j + 1] - 1 = 2i + 1.$$

If i is odd, it can be expressed as $2j + 1$ where $0 \le j < 2^k$. Then we obtain

$$J(2^{k+1} + i) = J(2^{k+1} + 2j + 1) = J(2(2^k + j) + 1) = 2J(2^k + j) + 1$$

and, using the induction's assumption, we can continue as follows

$$2J(2^k + j) + 1 = 2[2j + 1] + 1 = 2i + 1.$$

c. Let $n = (b_k b_{k-1} ... b_0)_2$ where the first binary digit b_k is 1. In the n's representation used in part (b), $n = 2^k + i$, $i = (b_{k-1} ... b_0)_2$. Further, as proved in part (b),

$$J(n) = 2i + 1 = (b_{k-1} ... b_0 0)_2 + 1 = (b_{k-1} ... b_0 1)_2 = (b_{k-1} ... b_0 b_k)_2,$$

which is a one-bit left cyclic shift of $n = (b_k b_{k-1} ... b_0)_2$.

Note: The solutions to Problem 15 are from Graham, R.L., Knuth, D.E. and Patashnik, O. *Concrete Mathematics: a Foundation for Computer Science*, 2nd ed. Addison-Wesley, 1994.

Solutions to Exercises 4.5

1. a. Since the algorithm uses the formula $\gcd(m, n) = \gcd(n, m \bmod n)$, the size of the new pair will be $m \bmod n$. Hence it can be any integer between 0 and $n-1$. Thus, the size n can decrease by any number between 1 and n.

   b. Two consecutive iterations of Euclid's algorithm are performed according to the following formulas:

   $$\gcd(m, n) = \gcd(n, r) = \gcd(r, n \bmod r) \quad \text{where } r = m \bmod n.$$

   We need to show that $n \bmod r \le n/2$. Consider two cases: $r \le n/2$ and $n/2 < r < n$. If $r \le n/2$, then

   $$n \bmod r < r \le n/2.$$

   If $n/2 < r < n$, then

   $$n \bmod r = n - r < n/2,$$

   too.

2. Since $n = 7$, $k = \lceil 7/2 \rceil = 4$ and $k - 1 = 3$. Applying quickselect with the Lomuto partitioning to the list 9, 12, 5, 17, 20, 30, 8, we obtain the following partition

0	1	2	3	4	5	6
9	12	5	17	20	30	8
s		i				
9	5	12	17	20	30	8
	s					i
9	5	8	17	20	30	12
		s				
8	5	**9**	17	20	30	12

0	1	2	3	4	5	6
s	i					
9	12	5	17	20	30	8
s		i				
9	12	5	17	20	30	8
	s		i			
9	5	12	17	20	30	8
	s					i
9	5	12	17	20	30	8
		s				
9	5	8	17	20	30	12
8	5	**9**	17	20	30	12

   Since $s = 2 < k - 1$, we proceed with the right part of the list:

0	1	2	3	4	5	6
8	5	9	**17**	20	30	12
			s			i
			17	12	30	20
				s		i
			12	**17**	30	20

0	1	2	3	4	5	6
			s	i		
			17	20	30	12
			s			i
			17	20	30	12
				s		
			17	12	30	20
			12	**17**	30	20

Since $s = 4 > k - 1$, we proceed with the left part of the list, which has just one element 12, which is the median of the list

0	1	2	3	4	5	6
			s			
8	5	9	**12**	20	30	20

3. a. **Algorithm** *Quickselect*$(A[0..n-1], k)$
//Solves the selection problem by partition-based algorithm
//Input: An array $A[0..n-1]$ of orderable elements and integer k $(1 \le k \le n)$
//Output: The value of the kth smallest element in $A[0..n-1]$
$l \leftarrow 0$; $r \leftarrow n-1$
$A[n] \leftarrow \infty$ //append sentinel
while $l \le r$ **do**
    $p \leftarrow A[l]$ //the pivot
    $i \leftarrow l$; $j \leftarrow r+1$
    repeat
        repeat $i \leftarrow i+1$ **until** $A[i] \ge p$
        repeat $j \leftarrow j-1$ **until** $A[j] \le p$ **do**
        swap$(A[i], A[j])$
    until $i \ge j$
    swap$(A[i], A[j])$ //undo last swap
    swap$(A[l], A[j])$ //partition
    if $j > k-1$ $r \leftarrow j-1$
    else if $j < k-1$ $l \leftarrow j+1$
    else return $A[k-1]$

   b. call *QuickselectRec*$(A[0..n-1], k)$ where

Algorithm *QuickselectRec*$(A[l..r], k)$
//Solves the selection problem by recursive partition-based algorithm
//Input: A subarray $A[l..r]$ of orderable elements and
// integer k $(1 \le k \le r-l+1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow$ *Partition*$(A[l..r])$ //see Section 4.5; must return l if $l = r$
if $s > l+k-1$ *QuickselectRec*$(A[l..s-1], k)$
else if $s < l+k-1$ *QuickselectRec*$(A[s+1..r], k-1-s)$
else return $A[s]$

4. Using the standard form of an equation of the straight line through two given points, we obtain

$$y - A[l] = \frac{A[r] - A[l]}{r - l}(x - l).$$

Substituting a given value v for y and solving the resulting equation for x yields

$$x = l + \lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \rfloor$$

after the necessary round-off of the second term to guarantee index l to be an integer.

5. If $v = A[l]$ or $v = A[r]$, formula (4.4) will yield $x = l$ and $x = r$, respectively, and the search for v will stop successfully after comparing v with $A[x]$. If $A[l] < v < A[r]$,

$$0 < \frac{(v - A[l])(r - l)}{A[r] - A[l]} < r - l;$$

therefore

$$0 \le \lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \rfloor \le r - l - 1$$

and

$$l \le l + \lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \rfloor \le r - 1.$$

Hence, if interpolation search does not stop on its current iteration, it reduces the size of the array that remains to be investigated at least by one. Therefore, its worst-case efficiency is in $O(n)$. We want to show that it is, in fact, in $\Theta(n)$. Consider, for example, array $A[0..n-1]$ in which $A[0] = 0$ and $A[i] = n-1$ for $i = 1, 2, ..., n-1$. If we search for $v = n-1.5$ in this array by interpolation search, its kth iteration ($k = 1, 2, ..., n$) will have $l = 0$ and $r = n - k$. We will prove this assertion by mathematical induction on k. Indeed, for $k = 1$ we have $l = 0$ and $r = n - 1$. For the general case, assume that the assertion is correct for some iteration k ($1 \le k < n$) so that $l = 0$ and $r = n-k$. On this iteration, we will obtain the following by applying the algorithm's formula

$$x = 0 + \lfloor \frac{((n - 1.5) - 0)(n - k)}{(n - 1) - 0} \rfloor.$$

Since

$$\frac{(n - 1.5)(n - k)}{(n - 1)} = \frac{(n - 1)(n - k) - 0.5(n - k)}{(n - 1)} = (n-k) - 0.5\frac{(n - k)}{(n - 1)} < (n-k)$$

and

$$\frac{(n - 1.5)(n - k)}{(n - 1)} = (n-k) - 0.5\frac{(n - k)}{(n - 1)} > (n-k) - \frac{(n - k)}{(n - 1)} \ge (n-k) - 1,$$

$$x = \lfloor \frac{(n - 1.5)(n - k)}{(n - 1) - 0} \rfloor = (n - k) - 1 = n - (k + 1).$$

Therefore $A[x] = A[n-(k+1)] = n-1$ (unless $k = n-1$), implying that $l = 0$ and $r = n-(k+1)$ on the next $(k+1)$ iteration. (If $k = n-1$, the assertion holds true for the next and last iteration, too: $A[x] = A[0] = 0$, implying that $l = 0$ and $r = 0$.)

6. a. We can solve the inequality $\log_2 \log_2 n + 1 > 6$ as follows:

$$\begin{aligned} \log_2 \log_2 n + 1 &> 6 \\ \log_2 \log_2 n &> 5 \\ \log_2 n &> 2^5 \\ n &> 2^{32} \ (> 4 \cdot 10^9). \end{aligned}$$

b. Using the formula $\log_a n = \log_a e \ln n$, we can compute the limit as follows:

$$\begin{aligned} \lim_{n\to\infty} \frac{\log_a \log_a n}{\log_a n} &= \lim_{n\to\infty} \frac{\log_a e \ln(\log_a e \ln n)}{\log_a e \ln n} = \lim_{n\to\infty} \frac{\ln \log_a e + \ln \ln n}{\ln n} \\ &= \lim_{n\to\infty} \frac{\ln \log_a e}{\ln n} + \lim_{n\to\infty} \frac{\ln \ln n}{\ln n} = 0 + \lim_{n\to\infty} \frac{\ln \ln n}{\ln n}. \end{aligned}$$

The second limit can be computed by using L'Hôpital's rule:

$$\lim_{n\to\infty} \frac{\ln \ln n}{\ln n} = \lim_{n\to\infty} \frac{[\ln \ln n]'}{[\ln n]'} = \lim_{n\to\infty} \frac{(1/\ln n)(1/n)}{1/n} = \lim_{n\to\infty} (1/\ln n) = 0.$$

Hence, $\log \log n \in o(\log n)$.

7. a. Recursively, go to the right subtree until a node with the empty right subtree is reached; return the key of that node. We can consider this algorithm as a variable-size-decrease algorithm: after each step to the right, we obtain a smaller instance of the same problem (whether we measure a tree's size by its height or by the number of nodes).

b. The worst-case efficiency of the algorithm is linear; we should expect its average-case efficiency to be logarithmic (see the discussion in Section 4.5).

8. a. This is an important and well-known algorithm. Case 1: If a key to be deleted is in a leaf, make the pointer from its parent to the key's node null. (If it doesn't have a parent, i.e., it is the root of a single-node tree, make the tree empty.) Case 2: If a key to be deleted is in a node with a single child, make the pointer from its parent to the key's node to point to

that child. (If the node to be deleted is the root with a single child, make its child the new root.) Case 3: If a key K to be deleted is in a node with two children, its deletion can be done by the following three-stage procedure. First, find the smallest key K' in the right subtree of the K's node. (K' is the immediate successor of K in the inorder traversal of the given binary tree; it can be also found by making one step to the right from the K's node and then all the way to the left until a node with no left subtree is reached). Second, exchange K and K'. Third, delete K in its new node by using either Case 1 or Case 2, depending on whether that node is a leaf or has a single child.

This algorithm is not a variable-size-decrease algorithm because it does not work by reducing the problem to that of deleting a key from a smaller binary tree.

b. Consider, as an example of the worst case input, the task of deleting the root from the binary tree obtained by successive insertions of keys 2, 1, n, $n-1, ..., 3$. Since finding the smallest key in the right subtree requires following a chain of $n-2$ pointers, the worst-case efficiency of the deletion algorithm is in $\Theta(n)$. Since the average height of a binary tree constructed from n random keys is a logarithmic function (see Section 5.6), we should expect the average-case efficiency of the deletion algorithm be logarithmic as well.

9. Starting at an arbitrary vertex of the graph, traverse a sequence of its untraversed edges until no untraversed edge is available from the vertex arrived at. The traversed path will be a circuit C. If C includes all the edges of the graph, the problem is solved. If it doesn't, remove the circuit C from the graph. The remaining subgraph G' will be connected and have only vertices with even degrees. Find a vertex v that belongs to both C and G'. (Such a vertex will always exist.) Starting at vertex v, find recursively an Euler circuit of the subgraph G' and splice C into it to get an Euler circuit of the graph given.

10. If $n = 1$, Player 1 (the player to move first) loses by definition of the misere game because s/he has no choice but to take the last chip. If $2 \le n \le m+1$, Player 1 wins by taking $n-1$ chips to leave Player 2 with one chip. If $n = m+2 = 1+(m+1)$, Player 1 loses because any legal move puts Player 2 in a winning position. If $m+3 \le n \le 2m+2$ (i.e., $2+(m+1) \le n \le 2(m+1)$), Player 1 can win by taking $(n-1) \bmod (m+1)$ chips to leave Player 2 with $m+2$ chips, which is a losing position for the player to move next. Thus, an instance is a losing position for Player 1 if and only if $n \bmod (m+1) = 1$. Otherwise, Player 1 wins by taking $(n-1) \bmod (m+1)$ chips; any deviation from this winning strategy puts the opponent in a winning position. The formal proof of the solution's correctness is by strong induction.

11. The problem is equivalent to the game of Nim, with the piles represented by the rows and columns of the bar between the spoiled square and the bar's edges. Thus, the Nim's theory outlined in the section identifies both winning positions and winning moves in this game. According to this theory, an instance of Nim is a winning one (for the player to move next) if and only if its binary digital sum contains at least one 1. In such a position, a wining move can be found as follows. Scan left to right the binary digital sum of the bit strings representing the number of chips in the piles until the first 1 is encountered. Let j be the position of this 1. Select a bit string with a 1 in position j—this is the pile from which some chips will be taken in a winning move. To determine the number of chips to be left in that pile, scan its bit string starting at position j and flip its bits to make the new binary digital sum contain only 0's.

    Note: Under the name of *Yucky Chocolate*, the special case of this problem—with the spoiled square in the bar's corner—is discussed, for example, by Yan Stuart in "Math Hysteria: Fun and Games with Mathematics," Oxford University Press, 2004. For such instances, the player going first loses if $m = n$, i.e., the bar has the square shape, and wins if $m \neq n$. Here is a proof by strong induction, which doesn't involve binary representations of the pile sizes. If $m = n = 1$, the player moving first looses by the game's definition. Assuming that the assertion is true for every k-by-k square bar for all $k \leq n$, consider the $n+1$-by-$n+1$ bar. Any move (i.e., a break of the bar) creates a rectangular bar with one side of size $k \leq n$ and the other side's size remaining $n + 1$. The second player can always follow with a break creating a k-by-k square bar with a spoiled corner, which is a loosing instance by the inductive assumption. And if $m \neq n$, the first player can always "even" the bar by creating the square with the side's size $\min\{m, n\}$, putting the second player in a losing position.

12. Here is a decrease-and-conquer algorithm for this problem. Repeat the following until the problem is solved: Find the largest pancake that is out of order. (If there is none, the problem is solved.) If it is not on the top of the stack, slide the flipper under it and flip to put the largest pancake on the top. Slide the flipper under the first-from-the-bottom pancake that is not in its proper place and flip to increase the number of pancakes in their proper place at least by one.

    The number of flips needed by this algorithm in the worst case is $W(n) = 2n - 3$, where $n \geq 2$ is the number of pancakes. Here is a proof of this assertion by mathematical induction. For $n = 2$, the assertion is correct: the algorithm makes one flip for a two-pancake stack with a larger pancake on the top, and it makes no flips for a two-pancake stack with a larger pancake at the bottom. Assume now that the worst-case number of flips for some value of $n \geq 2$ is given by the formula $W(n) = 2n - 3$.

Consider an arbitrary stack of $n+1$ pancakes. With two flips or less, the algorithm puts the largest pancake at the bottom of the stack, where it doesn't participate in any further flips. Hence, the total number of flips needed for any stack of $n+1$ pancakes is bounded above by

$$2 + W(n) = 2 + (2n - 3) = 2(n + 1) - 3.$$

In fact, this upper bound is attained on the stack of $n+1$ pancakes constructed as follows: flip a worst-case stack of n pancakes upside down and insert a pancake larger than all the others between the top and the next-to-the-top pancakes. (On the new stack, the algorithm will make two flips to reduce the problem to flipping the worst-case stack of n pancakes.) This completes the proof of the fact that

$$W(n + 1) = 2(n + 1) - 3,$$

which, in turn, completes our mathematical induction proof.

Note: The Web site mentioned in the problem's statement contains, in addition to a visualization applet, an interesting discussion of the problem. (Among other facts, it mentions that the only research paper published by Bill Gates was devoted to this problem.)

13. Compare the search number with the last element in the first row. If they match, stop. If the search number is smaller than the matrix element, the former can't be in the last column of the matrix, whose elements can be eliminated from the search. If the search number is larger than the last element in the first row, the former can't be in the first row of the matrix, whose elements can be eliminated from the search. Repeat this step for the smaller matrix until either a match is found or the remaining matrix shrinks to the empty one. Since on each iteration the algorithm eliminates one row or one column of the matrix from a further consideration, its time efficiency class is $O(n)$.

Solutions to Exercises 5.1

1. a. Call **Algorithm** *MaxIndex*$(A, 0, n-1)$ where

Algorithm *MaxIndex*(A, l, r)
//Input: A portion of array $A[0..n-1]$ between indices l and r $(l \le r)$
//Output: The index of the largest element in $A[l..r]$
if $l = r$ **return** l
else $temp1 \leftarrow$ *MaxIndex*$(A, l, \lfloor (l+r)/2 \rfloor)$
    $temp2 \leftarrow$ *MaxIndex*$(A, \lfloor (l+r)/2 \rfloor + 1, r)$
    if $A[temp1] \ge A[temp2]$
        return $temp1$
    else return $temp2$

b. This algorithm returns the index of the leftmost largest element.

c. The recurrence for the number of element comparisons is

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

Solving it by backward substitutions for $n = 2^k$ yields the following:

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

We can verify that $C(n) = n - 1$ satisfies, in fact, the recurrence for every value of $n > 1$ by substituting it into the recurrence equation and considering separately the even $(n = 2i)$ and odd $(n = 2i + 1)$ cases. Let $n = 2i$, where $i > 0$. Then the left-hand side of the recurrence equation is $n - 1 = 2i - 1$. The right-hand side is

$$\begin{aligned}
C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + 1 &= C(\lceil 2i/2 \rceil) + C(\lfloor 2i/2 \rfloor) + 1 \\
&= 2C(i) + 1 = 2(i-1) + 1 = 2i - 1,
\end{aligned}$$

which is the same as the left-hand side.

Let $n = 2i + 1$, where $i > 0$. Then the left-hand side of the recurrence equation is $n - 1 = 2i$. The right-hand side is

$$\begin{aligned}
C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + 1 &= C(\lceil (2i+1)/2 \rceil) + C(\lfloor (2i+1)/2 \rfloor) + 1 \\
&= C(i+1) + C(i) + 1 = (i + 1 - 1) + (i - 1) + 1 = 2i,
\end{aligned}$$

which is the same as the left-hand side in this case, too.

d. A simple standard scan through the array in question requires the same number of key comparisons but avoids the overhead associated with recursive calls.

2. a. Call **Algorithm** *MinMax*$(A, 0, n-1,$ *minval, maxval*$)$ where

Algorithm *MinMax*$(A, l, r,$ *minval, maxval*$)$
//Finds the values of the smallest and largest elements in a given subarray
//Input: A portion of array $A[0..n-1]$ between indices l and r $(l \le r)$
//Output: The values of the smallest and largest elements in $A[l..r]$
//assigned to *minval* and *maxval,* respectively
if $r = l$
    $minval \leftarrow A[l]$; $maxval \leftarrow A[l]$
else if $r - l = 1$
        if $A[l] \le A[r]$
            $minval \leftarrow A[l]$; $maxval \leftarrow A[r]$
        else $minval \leftarrow A[r]$; $maxval \leftarrow A[l]$
else //$r - l > 1$
    MinMax$(A, l, \lfloor (l+r)/2 \rfloor,$ *minval, maxval*$)$
    MinMax$(A, \lfloor (l+r)/2 \rfloor + 1, r,$ *minval2, maxval2*$)$
     if *minval2* $<$ *minval*
        minval $\leftarrow$ *minval2*
     if *maxval2* $>$ *maxval*
        maxval $\leftarrow$ *maxval2*

b. Assuming for simplicity that $n = 2^k$, we obtain the following recurrence for the number of element comparisons $C(n)$:

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

Solving it by backward substitutions for $n = 2^k$, $k \ge 1$, yields the following:

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

c. This algorithm makes about 25% fewer comparisons—$1.5n$ compared to $2n$—than the brute-force algorithm. (Note that if we didn't stop recursive calls when $n = 2$, we would've lost this gain.) In fact, the algorithm is

optimal in terms of the number of comparisons made. As a practical matter, however, it might not be faster than the brute-force algorithm because of the recursion-related overhead. (As noted in the solution to Problem 5 of Exercises 2.3, a nonrecursive scan of a given array that maintains the minimum and maximum values seen so far and updates them not for each element but for a pair of two consecutive elements makes the same number of comparisons as the divide-and-conquer algorithm but doesn't have the recursion's overhead.)

3. a. The following divide-and-conquer algorithm for computing a^n is based on the formula $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil}$:

Algorithm *DivConqPower*(a, n)
//Computes a^n by a divide-and-conquer algorithm
//Input: A number a and a positive integer n
//Output: The value of a^n
if $n = 1$ **return** a
else return *DivConqPower*$(a, \lfloor n/2 \rfloor) *$ *DivConqPower*$(a, \lceil n/2 \rceil)$

b. The recurrence for the number of multiplications is

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

The solution to this recurrence (solved above for Problem 1) is $n - 1$.

c. Though the algorithm makes the same number of multiplications as the brute-force method, it has to be considered inferior to the latter because of the recursion overhead.

4. For the second case, where the solution's class is indicated as $\Theta(n^d \log n)$, the logarithm's base could change the function by a constant multiple only and, hence, is irrelevant. For the third case, where the solution's class is $\Theta(n^{\log_b a})$, the logarithm is in the function's exponent and, hence, must be indicated since functions n^α have different orders of growth for different values of α.

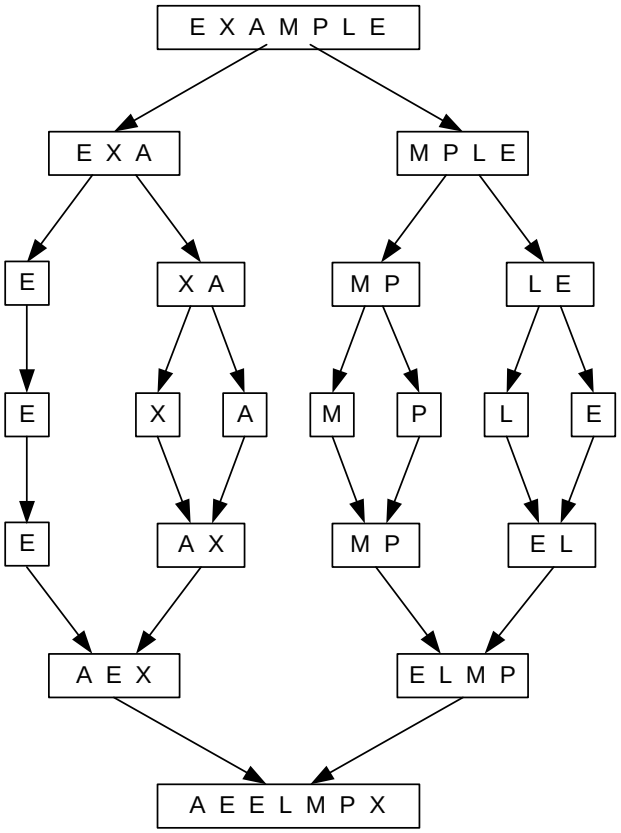
5. The applications of the Master Theorem yield the following.

a. $T(n) = 4T(n/2) + n$. Here, $a = 4$, $b = 2$, and $d = 1$. Since $a > b^d$, $T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$.

b. $T(n) = 4T(n/2) + n^2$. Here, $a = 4$, $b = 2$, and $d = 2$. Since $a = b^d$, $T(n) \in \Theta(n^2 \log n)$.

c. $T(n) = 4T(n/2) + n^3$. Here, $a = 4$, $b = 2$, and $d = 3$. Since $a < b^d$, $T(n) \in \Theta(n^3)$.

6. Here is a trace of mergesort applied to the input given:



7. Mergesort is stable, provided its implementation employs the comparison $\leq$ in merging. Indeed, assume that we have two elements of the same value in positions i and j, $i < j$, in a subarray before its two (sorted) halves are merged. If these two elements are in the same half of the subarray, their relative ordering will stay the same after the merging because the elements of the same half are processed by the merging operation in the FIFO fashion. Consider now the case when $A[i]$ is in the first half while $A[j]$ is in the second half. $A[j]$ is placed into the new array either after the first half becomes empty (and, hence, $A[i]$ has been already copied into the new array) or after being compared with some key $k > A[j]$ of the first half. In the latter case, since the first half is sorted before the merging

begins, $A[i] = A[j] < k$ cannot be among the unprocessed elements of the first half. Hence, by the time of this comparison, $A[i]$ has been already copied into the new array and therefore will precede $A[j]$ after the merging operation is completed.

8. a. The recurrence for the number of comparisons in the worst case, which was given in Section 5.1, is

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

Solving it by backward substitutions yields the following:

$$\begin{aligned}
C_w(2^k) &= 2C_w(2^{k-1}) + 2^k - 1 \\
&= 2[2C_w(2^{k-2}) + 2^{k-1} - 1] + 2^k - 1 = 2^2 C_w(2^{k-2}) + 2 \cdot 2^k - 2 - 1 \\
&= 2^2[2C_w(2^{k-3}) + 2^{k-2} - 1] + 2 \cdot 2^k - 2 - 1 = 2^3 C_w(2^{k-3}) + 3 \cdot 2^k - 2^2 - 2 - 1 \\
&= \ldots \\
&= 2^i C_w(2^{k-i}) + i2^k - 2^{i-1} - 2^{i-2} - \cdots - 1 \\
&= \ldots \\
&= 2^k C_w(2^{k-k}) + k2^k - 2^{k-1} - 2^{k-2} - \cdots - 1 = k2^k - (2^k - 1) = n \log n - n + 1.
\end{aligned}$$

b. The recurrence for the number of comparisons on best-case inputs (e.g.,

lists sorted in ascending or descending order) is

$$C_b(n) = 2C_b(n/2) + n/2 \quad \text{for } n > 1 \text{ (and } n = 2^k), \quad C_b(1) = 0.$$

Thus,

$$\begin{aligned}
C_b(2^k) &= 2C_b(2^{k-1}) + 2^{k-1} \\
&= 2[2C_b(2^{k-2}) + 2^{k-2}] + 2^{k-1} = 2^2 C_b(2^{k-2}) + 2^{k-1} + 2^{k-1} \\
&= 2^2[2C_b(2^{k-3}) + 2^{k-3}] + 2^{k-1} + 2^{k-1} = 2^3 C_b(2^{k-3}) + 2^{k-1} + 2^{k-1} + 2^{k-1} \\
&= \ldots \\
&= 2^i C_b(2^{k-i}) + i2^{k-1} \\
&= \ldots \\
&= 2^k C_b(2^{k-k}) + k2^{k-1} = k2^{k-1} = \frac{1}{2} n \log n.
\end{aligned}$$

c. If $n > 1$, the algorithm copies $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$ elements first and then makes n more moves during the merging stage. This yields the following recurrence for the number of moves $M(n)$:

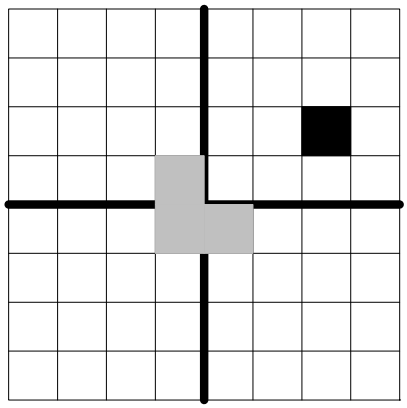
$$M(n) = 2M(n/2) + 2n \quad \text{for } n > 1, \quad M(1) = 0.$$

According to the Master Theorem, its solution is in $\Theta(n \log n)$—the same class established by the analysis of the number of key comparisons only.

5. Let *ModifiedMergesort* be a mergesort modified to return the number of inversions in its input array $A[0..n-1]$ in addition to sorting it. Obviously, for an array of size 1, *ModifiedMergesort*$(A[0])$ should return 0. Let i_{left} and i_{right} be the number of inversions returned by *ModifiedMergesort*$(A[0..mid-1])$ and *ModifiedMergesort*$(A[mid..n-1])$, respectively, where mid is the index of the middle element in the input array $A[0..n-1]$. The total number of inversions in $A[0..n-1]$ can then be computed as $i_{left} + i_{right} + i_{merge}$, where i_{merge}, the number of inversions involving elements from both halves of $A[0..n-1]$, is computed during the merging as follows. Let $A[i]$ and $A[j]$ be two elements from the left and right half of $A[0..n-1]$, respectively, that are compared during the merging. If $A[i] \leq A[j]$, we output $A[i]$ to the sorted list without incrementing i_{merge} because $A[i]$ cannot be a part of an inversion with any of the remaining elements in the second half, which are greater than or equal to $A[j]$. If, on the other hand, $A[i] > A[j]$, we output $A[j]$ and increment i_{merge} by $mid - i$, the number of remaining elements in the first half, because all those elements (and only they) form an inversion with $A[j]$.

10. n/a

11. If $n = 1$, each of the four possible 2×2 boards with a missing square can be covered in the obvious fashion by a single L-tromino. For $n > 1$, we can always place one L-tromino at the center of the $2^n \times 2^n$ chessboard with one missing square to reduce the problem to four subproblems of tiling $2^{n-1} \times 2^{n-1}$ boards, each with one missing square too. This L-tromino should cover three of the four central squares that are not in the quarter of the board with the missing square. An example is shown below.



Then each of the four smaller problems can be solved recursively until a trivial case of a 2×2 board with a missing square is reached.

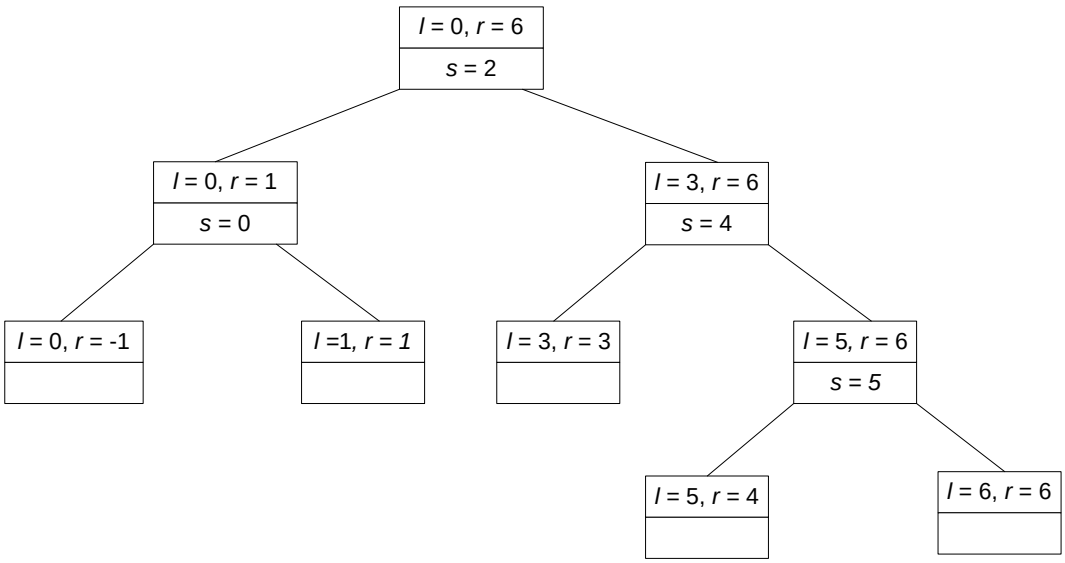
Solutions to Exercises 5.2

1. Applying the version of quicksort given in Section 5.2, we get the following:

0	1	2	3	4	5	6
	$\overset{i}{X}$					$\overset{j}{E}$
E	X	A	M	P	L	E
		$\overset{j}{A}$	$\overset{i}{M}$			
E	E	A	M	P	L	X
A	E	**E**	M	P	L	X
	$\overset{i\ j}{E}$					
A	E					
$\overset{j}{\mathbf{A}}$	$\overset{i}{E}$					
A	E					
	E					

		$\overset{i}{P}$		$\overset{j}{X}$
	M	P	L	X
		$\overset{i}{P}$	$\overset{j}{L}$	
	M	L	P	X
		$\overset{i}{L}$	$\overset{j}{P}$	
	M	L	P	X
		$\overset{j}{L}$	$\overset{i}{P}$	
	L	**M**	P	X
	L			

		$\overset{i\ j}{X}$
	P	X
	$\overset{j}{\mathbf{P}}$	$\overset{i}{X}$
	P	X
		X



2. a. Let $i = j$ be the coinciding values of the scanning indices. According to the rules for stopping the i (left-to-right) and j (right-to-left) scans, $A[i] \geq p$ and $A[j] \leq p$ where p is the pivot's value. Hence, $A[i] = A[j] = p$.

   b. Let i be the value of the left-to-right scanning index after it stopped. Since $A[i-1] \leq p$, the right-to-left scanning index will have to stop no later than reaching $i - 1$.

3. Consider how quicksort works on a two-element array of equal values $v_1 = v_2$:

$$\begin{array}{cc} 0 & 1 \\ & ij \\ \mathbf{v}_1 & v_2 \\ v_2 & \mathbf{v}_1 \end{array}$$

4. With the pivot being the leftmost element, the left-to-right scan will get out of bounds if and only if the pivot is larger than all the other elements. Appending a sentinel of value equal $A[0]$ (or larger than $A[0]$) after the array's last element will stop the index of the left-to-right scan of $A[0..n-1]$ from going beyond position n. A single sentinel will suffice by the following reason. In quicksort, when $Partition(A[l..r])$ is called for $r < n - 1$, all the elements to the right of position r are greater than or equal to all the elements in $A[l..r]$. Hence, $A[r + 1]$ will automatically play the role of a sentinel to stop index i from going beyond position $r + 1$.

5. a. Arrays composed of all equal elements constitute the best case because all the splits will happen in the middle of corresponding subarrays.

   b. Strictly decreasing arrays constitute the worst case because all the splits will yield one empty subarray. (Note that we need to show this to be the case on two consecutive iterations of the algorithm because the first iteration does not yield a decreasing array of size $n - 1$.)

6. The best case for both questions. For either a strictly increasing or strictly decreasing subarray, the median of the first, last, and middle values will be the median of the entire subarray. Using it as a pivot will split the subarray in the middle. This will cause the total number of key comparisons be the smallest.

7. a. The average case estimations for the number of comparisons made by quicksort and insertion sort are $2n \ln n$ and $n^2/4$, respectively. Hence,

$$\frac{T_{insert}(10^6)}{T_{quick}(10^6)} \approx \frac{c(10^6)^2/4}{c10^6 \ln 10^6} \approx 18,000.$$

b. On strictly increasing arrays of n elements, insertion sort makes $n-1$ comparisons while they are worst-case inputs for quicksort requiring $(n+1)(n+2)/2-3$ comparisons. Even with the median-of-three pivot selection, quicksort will make $n+1$ comparisons before the first partitioning of the array.

8. The following algorithm uses the partition idea similar to that of quicksort, although it's implemented somewhat differently. Namely, on each iteration the algorithm maintains three sections (possibly empty) in a given array: all the elements in $A[0..i-1]$ are negative, all the elements in $A[i..j]$ are unknown, and all the elements in $A[j+1..n]$ are nonnegative:

$A[0]$	...	$A[i-1]$	$A[i]$	...	$A[j]$	$A[j+1]$	...	$A[n-1]$
all are < 0			unknown			all are ≥ 0		

On each iteration, the algorithm shrinks the size of the unknown section by one element either from the left or from the right.

Algorithm *NegBeforePos*($A[0..n-1]$)
//Puts negative elements before positive (and zeros, if any) in an array
//Input: Array $A[0..n-1]$ of real numbers
//Output: Array $A[0..n-1]$ in which all its negative elements precede nonnegative
$i \leftarrow 0$; $j \leftarrow n-1$
while $i \leq j$ **do** //$i < j$ would suffice
  if $A[i] < 0$ //shrink the unknown section from the left
    $i \leftarrow i+1$
  else //shrink the unknown section from the right
    swap($A[i], A[j]$)
    $j \leftarrow j-1$

Note: If we want all the zero elements placed after all the negative elements but before all the positive ones, the problem becomes the Dutch flag problem (see Problem 9 in these exercises).

9. The following algorithm uses the partition idea similar to that of quicksort. (See also a simpler 2-color version of this problem in Problem 8 in these exercises.) On each iteration, the algorithm maintains four sections (possibly empty) in a given array: all the elements in $A[0..r-1]$ are filled with R's, all the elements in $A[r..w-1]$ are filled with W's, all the elements in $A[w..b]$ are unknown, and all the elements in $A[b+1..n-1]$ are filled with B's.

$A[0]$ ... $A[r-1]$	$A[r]$ ... $A[w-1]$	$A[w]$ ... $A[b]$	$A[b+1]$ ... $A[n-1]$
all are filled with R's	all are filled with W's	unknown	all are filled with B's

On each iteration, the algorithm shrinks the size of the unknown section by one element either from the left or from the right.

Algorithm *DutchFlag*$(A[0..n-1])$
//Sorts an array with values in a three-element set
//Input: An array $A[0..n-1]$ of characters from {'R', 'W', 'B'}
//Output: Array $A[0..n-1]$ in which all its R elements precede
// all its W elements that precede all its B elements
$r \leftarrow 0$; $w \leftarrow 0$; $b \leftarrow n-1$
while $w \leq b$ **do**
  if $A[w]$ = 'R'
    swap$(A[r], A[w])$; $r \leftarrow r+1$; $w \leftarrow w+1$
  else if $A[w]$ = 'W'
    $w \leftarrow w+1$
  else //$A[w]$ = 'B'
    swap$(A[w], A[b])$; $b \leftarrow b-1$

b. One can partition an array in three subarrays—the elements that are smaller than the pivot, the elements that are equal to the pivot, and the elements that are greater than the pivot—and then sort the first and last subarrays recursively.

10. n/a

11. Randomly select a nut and try each of the bolts for it to find the matching bolt and separate the bolts that are smaller and larger than the selected nut into two disjoint sets. Then try each of the unmatched nuts against the matched bolt to separate those that are larger from those that are smaller than the bolt. As a result, we've identified a matching pair and partitioned the remaining nuts and bolts into two smaller independent instances of the same problem. The average number of nut-bolt comparisons $C(n)$ is defined by the recurrence very similar to the one for quicksort in Section 5.2:

$$C(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(2n-1) + C(s) + C(n-1-s)], \quad C(1) = 0,\ C(0) = 0.$$

The solution to this recurrence can be shown to be in $\Theta(n \log n)$, similar to the solution for the average number of comparisons made by quicksort.

Note: See a $O(n \log n)$ deterministic algorithm for this problem in the paper by Janos Komlos, Yuan Ma and Endre Szemeredi "Matching Nuts and Bolts in O(nlogn) Time," SIAM J. Discrete Math. 11, No.3, 347-372 (1998).

Solutions to Exercises 5.3

1. **Algorithm** *Levels*(T)
   //Computes recursively the number of levels in a binary tree
   //Input: Binary tree T
   //Output: Number of levels in T
   if $T = \varnothing$ **return** 0
   else return $\max\{$*Levels*(T_L), *Levels*$(T_R)\} + 1$

   This is a $\Theta(n)$ algorithm, by the same reason *Height*(T) discussed in the section is.

2. The algorithm is incorrect because it returns 0 for any binary tree; in particular, it returns 0 instead of 1 for the one-node binary tree. Here is a corrected version:

   Algorithm *LeafCounter*(T)
   //Computes recursively the number of leaves in a binary tree
   //Input: A binary tree T
   //Output: The number of leaves in T
   if $T = \varnothing$ **return** 0 //empty tree
   else if $T_L = \varnothing$ **and** $T_R = \varnothing$ **return** 1 //one-node tree
   else return *LeafCounter*(T_L)+ *LeafCounter*(T_R) //general case

3. Starting at the root, traverse the binary tree by breadth-first search to find the level of each node. The largest of these numbers is, by the definition, the height of the tree.

4. Here is a proof of equality (5.2) by strong induction on the number of internal nodes $n \geq 0$. The basis step is true because for $n = 0$ we have the empty tree whose extended tree has 1 external node by definition. For the inductive step, let us assume that

   $$x = k + 1$$

   for any extended binary tree with $0 \leq k < n$ internal nodes. Let T be a binary tree with n internal nodes and let n_L and x_L be the numbers of internal and external nodes in the left subtree of T, respectively, and let n_R and x_R be the numbers of internal and external nodes in the right subtree of T, respectively. Since $n > 0$, T has a root, which is its internal node, and hence

   $$n = n_L + n_R + 1.$$

   Since both $n_L < n$ and $n_R < n$, we can use equality (4.5), assumed to be correct for the left and right subtree of T, to obtain the following:

   $$x = x_L + x_R = (n_L + 1) + (n_R + 1) = (n_L + n_R + 1) + 1 = n + 1,$$

which completes the proof.

5. a. Preorder: $a \quad b \quad d \quad e \quad c \quad f$

   b. Inorder: $d \quad b \quad e \quad a \quad c \quad f$

   c. Postorder: $d \quad e \quad b \quad f \quad c \quad a$

6. Here is pseudocode of the preorder traversal:

   Algorithm *Preorder*(T)
   //Implements the preorder traversal of a binary tree
   //Input: Binary tree T (with labeled vertices)
   //Output: Node labels listed in preorder
   if $T \neq \varnothing$
       print label of T's root
       Preorder(T_L) //T_L is the root's left subtree
       Preorder(T_R) //T_R is the root's right subtree

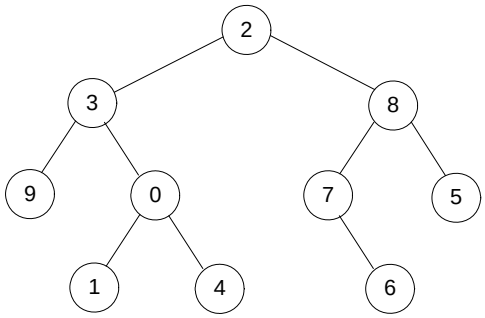
   The number of calls, $C(n)$, made by the algorithm is equal to the number of nodes, both internal and external, in the extended tree. Hence, according to the formula in the section,

   $$C(n) = 2n + 1.$$

7. The inorder traversal yields a sorted list of keys of a binary search tree. In order to prove it, we need to show that if $k_1 < k_2$ are two keys in a binary search tree then the inorder traversal visits the node containing k_1 before the node containing k_2. Let k_3 be the key at their nearest common ancestor. (Such a node is uniquely defined for any pair of nodes in a binary tree. If one of the two nodes at hand is an ancestor of the other, their nearest common ancestor coincides with the ancestor.) If the k_3's node differ from both k_1's node and k_2's node, the definition of a binary search tree implies that k_1 and k_2 are in the left and right subtrees of k_3, respectively. If k_3's node coincides with k_2's node (k_1's node), k_1's node (k_2's node) is in the left (right) subtree rooted at k_2's node (k_1's node). In each of these cases, the inorder traversal visits k_1's node before k_2's node.

8. a. The root's label is listed last in the postorder tree: hence, it is 2. The labels preceding 2 in the order list—9,3,1,0,4—form the inorder traversal list of the left subtree; the corresponding postorder list for the left subtree traversal is given by the first four labels of the postorder list: 9,1,4,0,3.

Similarly, for the right subtree, the inorder and postorder lists are, respectively, 7,6,8,5 and 6,7,5,8. Applying the same logic recursively to each of the subtrees yields the following binary tree:



b. There is no such example for $n = 2$. For $n = 3$, lists 0,1,2 (inorder) and 2,0,1 (postorder) provide one.

c. The problem can be solved by a recursive algorithm based on the following observation: There exists a binary tree with inorder traversal list $i_0, i_1, ..., i_{n-1}$ and postorder traversal list $p_0, p_1, ..., p_{n-1}$ if and only if $p_{n-1} = i_k$ (the root's label), the sets formed by the first k labels in both lists are the same: $\{i_0, i_1, ..., i_{k-1}\} = \{p_0, p_1, ..., p_{k-1}\}$ (the labels of the nodes in the left subtree) and the sets formed by the other $n - k - 1$ labels excluding the root are the same: $\{i_{k+1}, i_{k+2}, ..., i_{n-1}\} = \{p_k, p_{k+1}, ..., p_{n-2}\}$ (the labels of the nodes in the right subtree).

Algorithm *Tree*$(i_0, i_1, ..., i_{n-1}, p_0, p_1, ..., p_{n-1})$
//Construct recursively the binary tree based on the inorder and postorder traversal lists
//Input: Lists $i_0, i_1, ..., i_{n-1}$ and $p_0, p_1, ..., p_{n-1}$ of inorder and postorder traversals, respectively
//Output: Binary tree T, specified in preorder, whose inorder and postorder traversals yield the lists given or -1 if such a tree doesn't exist
Find element i_k in the inorder list that is equal to the last element p_{n-1} of the postorder list.
if the previous search was unsuccessful **return** -1
else *print*(i_k)
Tree$(i_0, i_1, ..., i_{k-1}, p_0, p_1, ..., p_{k-1})$
Tree$(i_{k+1}, i_{k+2}, ..., i_{n-1}, p_k, p_{k+1}, ..., p_{n-2})$

9. We can prove equality $E = I + 2n$, where E and I are, respectively,. the external and internal path lengths in an extended binary tree with n internal nodes by induction on n. The basis case, for $n = 0$, holds because

both E and I are equal to 0 in the extended tree of the empty binary tree. For the general case of induction, we assume that

$$E = I + 2k$$

for any extended binary tree with $0 \leq k < n$ internal nodes. To prove the equality for an extended binary tree T with n internal nodes, we are going to use this equality for T_L and T_R, the left and right subtrees of T. (Since $n > 0$, the root of the tree is an internal node, and hence the number of internal nodes in both the left and right subtree is less than n.) Thus,

$$E_L = I_L + 2n_L,$$

where E_L and I_L are external and internal paths, respectively, in the left subtree T_L, which has n_L internal and x_L external nodes, respectively. Similarly,

$$E_R = I_R + 2n_R,$$

where E_R and I_R are external and internal paths, respectively, in the right subtree T_R, which has n_R internal and x_R external nodes, respectively. Since the length of the simple path from the root of $T_L(T_R)$ to a node in $T_L(T_R)$ is one less than the length of the simple path from the root of T to that node, we have

$$\begin{aligned} E &= (E_L + x_L) + (E_R + x_R) \\ &= (I_L + 2n_L + x_L) + (I_R + 2n_R + x_R) \\ &= [(I_L + n_L) + (I_R + n_R)] + (n_L + n_R) + (x_L + x_R) \\ &= I + (n-1) + x, \end{aligned}$$

where x is the number of external nodes in T. Since $x = n+1$ (see Section 5.3), we finally obtain the desired equality:

$$E = I + (n-1) + x = I + 2n.$$

10. n/a

11. We can represent operations of any algorithm solving the problem by a full binary tree in which parental nodes represent breakable pieces and leaves represent 1×1 pieces of the original bar. The number of the latter is nm; and the number of the former, which is equal to the number of the bar breaks, is one less, i.e., $nm - 1$, according to equation (5.2) in Section 5.3. (Note: This elegant solution was suggested to the author by Simon Berkovich, one of the book's reviewers.)

    Alternatively, we can reason as follows: Since only one bar can be broken

at a time, any break increases the number of pieces by 1. Hence, $nm - 1$ breaks are needed to get from a single $n \times m$ piece to nm 1×1 pieces, which is obtained by *any* sequence of $nm - 1$ allowed breaks. (The same argument can be made more formally by mathematical induction.)

Solutions to Exercises 5.4

1. The smallest decimal n-digit positive integer is $\underbrace{10...0}_{n-1}$, i. e., 10^{n-1}. The product of two such numbers is $10^{n-1} \cdot 10^{n-1} = 10^{2n-2}$, which has $2n-1$ digits (1 followed by $2n-2$ zeros).

   The largest decimal n-digit integer is $\underbrace{9...9}_{n}$, i.e., $10^n - 1$. The product of two such numbers is $(10^n-1)(10^n-1) = 10^{2n} - 2 \cdot 10^n + 1$, which has $2n$ digits (because 10^{2n-1} and $10^{2n} - 1$ are the smallest and largest numbers with $2n$ digits, respectively, and $10^{2n} - 1 < 10^{2n} - 2 \cdot 10^n + 1 < 10^{2n} - 1$).

2. For $2101 * 1130$:

$$\begin{aligned} c_2 &= 21 * 11 \\ c_0 &= 01 * 30 \\ c_1 &= (21+01) * (11+30) - (c_2 + c_0) = 22 * 41 - 21 * 11 - 01 * 30. \end{aligned}$$

   For $21 * 11$:

$$\begin{aligned} c_2 &= 2 * 1 = 2 \\ c_0 &= 1 * 1 = 1 \\ c_1 &= (2+1) * (1+1) - (2+1) = 3 * 2 - 3 = 3. \\ \text{So, } 21 * 11 &= 2 \cdot 10^2 + 3 \cdot 10^1 + 1 = 231. \end{aligned}$$

   For $01 * 30$:

$$\begin{aligned} c_2 &= 0 * 3 = 0 \\ c_0 &= 1 * 0 = 0 \\ c_1 &= (0+1) * (3+0) - (0+0) = 1 * 3 - 0 = 3. \\ \text{So, } 01 * 30 &= 0 \cdot 10^2 + 3 \cdot 10^1 + 0 = 30. \end{aligned}$$

   For $22 * 41$:

$$\begin{aligned} c_2 &= 2 * 4 = 8 \\ c_0 &= 2 * 1 = 2 \\ c_1 &= (2+2) * (4+1) - (8+2) = 4 * 5 - 10 = 10. \\ \text{So, } 22 * 41 &= 8 \cdot 10^2 + 10 \cdot 10^1 + 2 = 902. \end{aligned}$$

   Hence

$$2101 * 1130 = 231 \cdot 10^4 + (902 - 231 - 30) \cdot 10^2 + 30 = 2,374,130.$$

3. a. Taking the base-b logarithms of both hand sides of the equality $a^{\log_b c} = c^{\log_b a}$ yields $\log_b c \log_b a = \log_b a \log_b c$. Since two numbers are equal if and only if their logarithms to the same base are equal, the equality in question is proved.

   b. It is easier to compare $n^{\log_2 3}$ with n^2 (the number of digit multiplications made by the classic algorithm) than $3^{\log_2 n}$ with n^2.

4. a. When working with decimal integers, multiplication by a power of 10 can be done by a shift.

   b. In the formula for c_1, the sum of two $n/2$-digit integers can have not $n/2$ digits, as it was assumed, but $n/2 + 1$.

5. Let a and b be two n-digit integers such that the product of each pair of their digits is a one-digit number. Then the result of the pen-and-pencil algorithm will look as follows:

$$\begin{array}{rccccccccc}
a: & & & & & a_{n-1} & ... & a_1 & a_0 \\
b: & & & & & b_{n-1} & ... & b_1 & b_0 \\
\hline
 & & & & & d_{n-1,0} & ... & d_{1,0} & d_{0,0} \\
 & & & & d_{n-1,1} & d_{n-2,1} & & d_{0,1} & \\
 & & d_{n-2,n-1} & ... & & & & & \\
 & d_{n-1,n-1} & d_{n-2,n-1} & ... & d_{1,n-1} & d_{0,n-1} & & & \\
\hline
\text{\# additions} & 0 & 1 & ... & n-2 & n-1 & ... & 1 & 0
\end{array}$$

   Hence, the total number of additions without carries will be

$$\begin{aligned}
& 1 + 2 + ... + (n-1) + (n-2) + ... + 1 \\
= \; & [1 + 2 + ... + (n-1)] + [(n-2) + ... + 1] \\
= \; & \frac{(n-1)n}{2} + \frac{(n-2)(n-1)}{2} = (n-1)^2.
\end{aligned}$$

6. $m_1 + m_4 - m_5 + m_7 =$
   $(a_{00}+a_{11})(b_{00}+b_{11})+a_{11}(b_{10}-b_{00})-(a_{00}+a_{01})b_{11}+(a_{01}-a_{11})(b_{10}+b_{11}) =$
   $a_{00}b_{00}+a_{11}b_{00}+a_{00}b_{11}+a_{11}b_{11}+a_{11}b_{10}-a_{11}b_{00}-a_{00}b_{11}-a_{01}b_{11}+a_{01}b_{10}-$
   $a_{11}b_{10} + a_{01}b_{11} - a_{11}b_{11}$
   $= a_{00}b_{00} + a_{01}b_{10}$

   $m_3+m_5 = a_{00}(b_{01}-b_{11})+(a_{00}+a_{01})b_{11} = a_{00}b_{01}-a_{00}b_{11}+a_{00}b_{11}+a_{01}b_{11} =$
   $a_{00}b_{01} + a_{01}b_{11}$

   $m_2+m_4 = (a_{10}+a_{11})b_{00}+a_{11}(b_{10}-b_{00}) = a_{10}b_{00}+a_{11}b_{00}+a_{11}b_{10}-a_{11}b_{00} =$

$a_{10}b_{00} + a_{11}b_{10}$

$m_1 + m_3 - m_2 + m_6 = (a_{00} + a_{11})(b_{00} + b_{11}) + a_{00}(b_{01} - b_{11}) - (a_{10} + a_{11})b_{00} + (a_{10} - a_{00})(b_{00} + b_{01}) =$
$a_{00}b_{00} + a_{11}b_{00} + a_{00}b_{11} + a_{11}b_{11} + a_{00}b_{01} - a_{00}b_{11} - a_{10}b_{00} - a_{11}b_{00} + a_{10}b_{00} - a_{00}b_{00} + a_{10}b_{01} - a_{00}b_{01}$
$= a_{10}b_{01} + a_{11}b_{11}$.

7. For the matrices given, Strassen's algorithm yields the following:

$$C = \left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

where

$$A_{00} = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix}, \quad A_{01} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}, \quad A_{10} = \begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix}, \quad A_{11} = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix},$$

$$B_{00} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}, \quad B_{01} = \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix}, \quad B_{10} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}, \quad B_{11} = \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}.$$

Therefore,

$$\begin{aligned}
M_1 &= (A_{00} + A_{11})(B_{00} + B_{11}) = \begin{bmatrix} 4 & 0 \\ 6 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix}, \\
M_2 &= (A_{10} + A_{11})B_{00} = \begin{bmatrix} 3 & 1 \\ 7 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix}, \\
M_3 &= A_{00}(B_{01} - B_{11}) = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ -5 & 4 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix}, \\
M_4 &= A_{11}(B_{10} - B_{00}) = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix}, \\
M_5 &= (A_{00} + A_{01})B_{11} = \begin{bmatrix} 3 & 1 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix}, \\
M_6 &= (A_{10} - A_{00})(B_{00} + B_{01}) = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 2 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix}, \\
M_7 &= (A_{01} - A_{11})(B_{10} + B_{11}) = \begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}.
\end{aligned}$$

Accordingly,

$$\begin{aligned}
C_{00} &= M_1 + M_4 - M_5 + M_7 \\
&= \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} - \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}, \\
C_{01} &= M_3 + M_5 \\
&= \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} + \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} = \begin{bmatrix} 7 & 3 \\ 1 & 9 \end{bmatrix}, \\
C_{10} &= M_2 + M_4 \\
&= \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 1 \\ 5 & 8 \end{bmatrix}, \\
C_{11} &= M_1 + M_3 - M_2 + M_6 \\
&= \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix} = \begin{bmatrix} 3 & 7 \\ 7 & 7 \end{bmatrix}.
\end{aligned}$$

That is,

$$C = \begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{bmatrix}.$$

8. For $n = 2^k$, the recurrence $A(n) = 7A(n/2)+18(n/2)^2$ for $n > 1$, $A(1) = 0$, becomes

$$A(2^k) = 7A(2^{k-1}) + \frac{9}{2}4^k \quad \text{for } k > 1, \quad A(1) = 0.$$

Solving it by backward substitutions yields the following:

$$\begin{aligned}
A(2^k) &= 7A(2^{k-1}) + \frac{9}{2}4^k \\
&= 7[7A(2^{k-2}) + \frac{9}{2}4^{k-1}] + \frac{9}{2}4^k = 7^2A(2^{k-2}) + 7 \cdot \frac{9}{2}4^{k-1} + \frac{9}{2}4^k \\
&= 7^2[7A(2^{k-3}) + \frac{9}{2}4^{k-2}] + 7 \cdot \frac{9}{2}4^{k-1} + \frac{9}{2}4^k \\
&= 7^3A(2^{k-3}) + 7^2 \cdot \frac{9}{2}4^{k-2} + 7 \cdot \frac{9}{2}4^{k-1} + \frac{9}{2}4^k \\
&= ... \\
&= 7^kA(2^{k-k}) + \frac{9}{2}\sum_{i=0}^{k-1} 7^i 4^{k-i} = 7^k \cdot 0 + \frac{9}{2}4^k \sum_{i=0}^{k-1} \left(\frac{7}{4}\right)^i \\
&= \frac{9}{2}4^k \frac{(7/4)^k - 1}{(7/4) - 1} = 6(7^k - 4^k).
\end{aligned}$$

Returning back to the variable $n = 2^k$, we obtain

$$A(n) = 6(7^{\log_2 n} - 4^{\log_2 n}) = 6(n^{\log_2 7} - n^2).$$

(Note that the number of additions in Strassen's algorithm has the same order of growth as the number of multiplications: $\Theta(n^s)$ where $s = n^{\log_2 7} \approx n^{2.807}$.)

9. The recurrence for the number of multiplications in Pan's algorithm is

$$M(n) = 143640M(n/70) \quad \text{for } n > 1, \qquad M(1) = 1.$$

Solving it for $n = 70^k$ or applying the Master Theorem yields $M(n) \in \Theta(n^p)$ where

$$p = \log_{70} 143640 = \frac{\ln 143640}{\ln 70} \approx 2.795.$$

This number is slightly smaller than the exponent of Strassen's algorithm

$$s = \log_2 7 = \frac{\ln 7}{\ln 2} \approx 2.807.$$

10. n/a

Solutions to Exercises 5.5

1. a. Assuming that the points are sorted in increasing order, we can find the closest pair (or, for simplicity, just the distance between two closest points) by comparing three distances: the distance between the two closest points in the first half of the sorted list, the distance between the two closest points in its second half, and the distance between the rightmost point in the first half and the leftmost point in the second half. Therefore, after sorting the numbers of a given array $P[0..n-1]$ in increasing order, we can call $ClosestNumbers(P[0..n-1])$, where

   Algorithm $ClosestNumbers(P[l..r])$
   //A divide-and-conquer alg. for the one-dimensional closest-pair problem
   //Input: A subarray $P[l..r]$ $(l \le r)$ of a given array $P[0..n-1]$
   // of real numbers sorted in nondecreasing order
   //Output: The distance between the closest pair of numbers
   if $r = l$ **return** ∞
   else if $r - l = 1$ **return** $P[r] - P[l]$
   else return $min\{ClosestNumbers(P[l..\lfloor (l+r)/2 \rfloor]),$
   $ClosestNumbers(P[(\lfloor (l+r)/2 \rfloor + 1..r]),$
   $P[\lfloor (l+r)/2 \rfloor + 1] - P[\lfloor (l+r)/2 \rfloor]\}$

   For $n = 2^k$, the recurrence for the running time $T(n)$ of this algorithm is

   $$T(n) = 2T(n/2) + c.$$

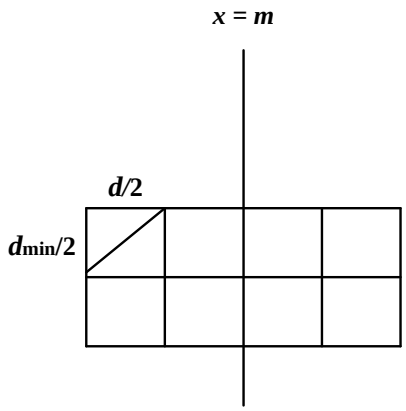
   Its solution, according to the Master Theorem, is in $\Theta(n^{\log_2 2}) = \Theta(n)$. If sorting of the input's numbers is done with a $\Theta(n \log n)$ algorithm such as mergesort, the overall running time will be in $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$.

   b. A simpler algorithm can sort the numbers given (e.g., by mergesort) and then compare the distances between the adjacent elements in the sorted list. The resulting algorithm has the same $\Theta(n \log n)$ efficiency but it is arguably simpler than the divide-and-conquer algorithms above.

   Note: In fact, any algorithm that solves this problem must be in $\Omega(n \log n)$ (see Problem 11 in Exercises 11.1).

2. Since both sides of each of the eight rectangles is no larger than $d/2$, the distance between any two of its points cannot exceed the length of its diagonal, which is equal to $\sqrt{(d/2)^2 + (d/2)^2} < d$. Hence each rectangle cannot contain two or more points with a distance greater than or equal

to d, and therefore all of them cannot contian more than eight such points.



3. $T(n) = 2T(n/2) + 2\frac{n}{2}\log_2\frac{n}{2}$ for $n > 2$ (and $n = 2^k$), $T(2) = 1$.

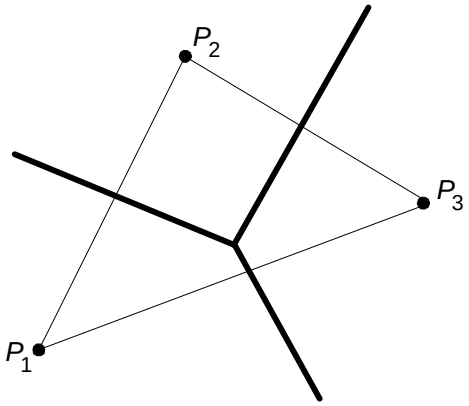
   Thus, $T(2^k) = 2T(2^{k-1}) + 2^k(k-1)$. Solving it by backward substitutions yields the following:

$$
\begin{aligned}
& T(2^k) = 2T(2^{k-1}) + 2^k(k-1) \\
= \;& 2[2T(2^{k-2}) + 2^{k-1}(k-2)] + 2^k(k-1) = 2^2T(2^{k-2}) + 2^k(k-2) + 2^k(k-1) \\
= \;& 2^2[2T(2^{k-3}) + 2^{k-2}(k\text{-}3)] + 2^k(k\text{-}2) + 2^k(k\text{-}1) = 2^3T(2^{k-3}) + 2^k(k\text{-}3) + 2^k(k\text{-}2) + 2^k(k\text{-}1) \\
& \ldots \\
= \;& 2^iT(2^{k-i}) + 2^k(k-i) + 2^k(k-i+1) + \cdots + 2^k(k-1) \\
& \ldots \\
= \;& 2^{k-1}T(2^1) + 2^k + 2^k2 + \cdots + 2^k(k-1) \\
= \;& 2^{k-1} + 2^k(1 + 2 + \cdots + (k-1)) = 2^{k-1} + 2^k\frac{(k-1)k}{2} \\
= \;& 2^{k-1}(1 + (k-1)k) = \frac{n}{2}(1 + (\log_2 n - 1)\log_2 n) \in \Theta(n\log^2 n).
\end{aligned}
$$

4. n/a

5. n/a

6. a. The Voronoi diagram of three points not on the same line is formed by the perpendicular bisectors of the sides of the triangle with vertices at p_1, p_2, and p_3:



(If p_1, p_2, and p_3 lie on the same line, with p_2 between p_1 and p_3, the Voronoi diagram is formed by the perpendicular bisectors of the segments with the endpoints at p_1 and p_2 and at p_2 and p_3.)

b. The Voronoi polygon of a set of points is made up of perpendicular bisectors; a point of their intersection has at least three of the set's points nearest to it.

7. Since all the points in question serve as the third vertex for triangles with the same base P_1P_n, the farthest point is the one that maximizes the area of such a triangle. The area of a triangle, in turn, can be computed as one half of the magnitude of the determinant

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3.$$

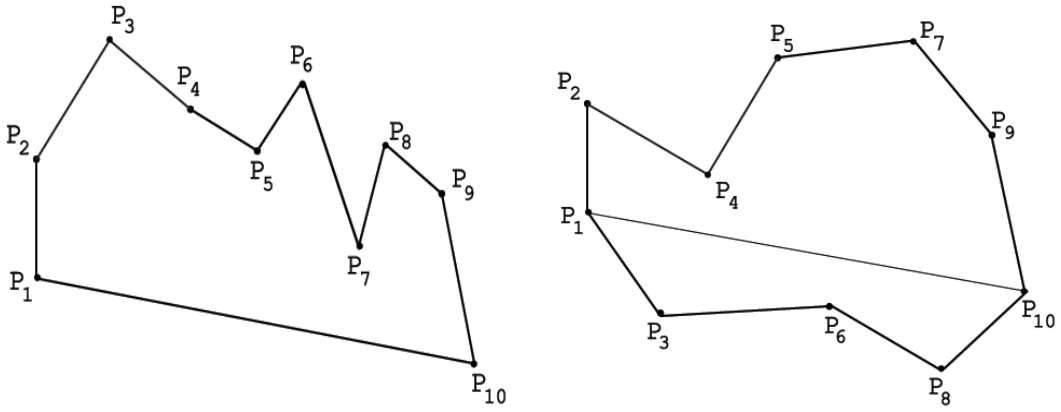
In other words, $P_{\max}$ is a point whose coordinates (x_3, y_3) maximize the absolute value of the above expression in which (x_1, y_1) and (x_2, y_2) are the coordinates of P_1 and P_n, respectively.

8. If all n points lie on the same line, both S_1 and S_2 will be empty and the convex hull (a line segment) will be found in linear time, assuming that the input points have been already sorted before the algorithm begins. Note: Any algorithm that finds the convex hull for a set of n points must be in $\Omega(n)$ because all n points must be processed before the convex hull is found.

9. Among many possible answers, one can take two endpoints of the horizontal diameter of some circumference as points p_1 and p_n and obtain the other points p_i, $i = 2, ..., n-1$, of the set in question by placing them successively in the middle of the circumference's upper arc between p_{i-1} and p_n.

10. n/a

11. We assume without loss of generality that the points are numbered left to right from 1 to 1000, with ties, if any, resolved by numbering a lower of the two points first. Consider the first ten points $p_1, ..., p_{10}$ and draw the straight line connecting p_1 and p_{10} There are two cases: either all the other eight points $p_2, ..., p_9$ lie on the same side of this line (see the left figure below) or they lie on both sides of the line (see the right figure below). In the former case, a decagon can be obtained by connecting the points $p_1, ..., p_{10}$ in this order, with the line connecting p_1 and p_{10} also serving as one of the decagon's sides. If the points $p_2, ..., p_9$ lie on both sides of the line connecting p_1 and p_{10}, a decagon is obtained by connecting consecutive pairs of the points on or above this line to form its upper boundary and connecting consecutive pairs of the points on or below this line to form its lower boundary.



All the 1990 remaining points would be to the right of the constructed decagon except, possibly, for the leftmost point p_{11} that can be on the same vertical line as p_{10}, the rightmost point of the constructed decagon. Hence, using the same method, we can construct a decagon with its vertices at the next ten points $p_{11}, ..., p_{20}$, and it will not intersect with the first decagon. Repeating this step for every consecutive ten points on the list solves the problem.

12. Find the upper and lower hulls of the set $\{a, b, p_1, ..., p_n\}$ (e.g., by quickhull), compute their lengths (by summing up the lengths of the line segments making up the polygonal chains), and return the smaller of the two.

Solutions to Exercises 6.1

1. a. Sort the array first and then scan it to find the smallest difference between two successive elements $A[i]$ and $A[i+1]$ $(0 \le i \le n-2)$.

   b. The time efficiency of the brute-force algorithm is in $\Theta(n^2)$ because the algorithm considers $n(n-1)/2$ pairs of the array's elements. (In the crude version given in Problem 9 of Exercises 1.2, the same pair is considered twice but this doesn't change the efficiency's order, of course.) If the presorting is done with a $O(n \log n)$ algorithm, the running time of the entire algorithm will be in

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

2. a. Initialize a list to contain elements of $C = A \cap B$ to empty. Compare every element a_i in A with successive elements of B: if $a_i = b_j$, add this value to the list C and proceed to the next element in A. (In fact, if $a_i = b_j$, b_j need not be compared with the remaining elements in A and may be deleted from B.) In the worst case of input sets with no common elements, the total number of element comparisons will be equal to nm, putting the algorithm's efficiency in $O(nm)$.

   b. First solution: Sort elements of one of the sets, say, A, stored in an array. Then use binary search to search for each element of B in the sorted array A: if a match is found, add this value to the list C. If sorting is done with a $O(n \log n)$ algorithm, the total running time will be in

   $$O(n \log n) + mO(\log n) = O((m+n) \log n).$$

   Note that the efficiency formula implies that it is more efficient to sort the smaller one of the two input sets.

   Second solution: Sort the lists representing sets A and B, respectively. Scan the lists in the mergesort-like manner but output only the values common to the two lists. If sorting is done with a $O(n \log n)$ algorithm, the total running time will be in

   $$O(n \log n) + O(m \log m) + O(n+m) = O(s \log s) \quad \text{where } s = \max\{n, m\}.$$

   Third solution: Combine the elements of both A and B in a single list and sort it. Then scan this sorted list by comparing pairs of its consecutive elements: if $L_i = L_{i+1}$, add this common value to the list C and

increment i by two. If sorting is done with an $n \log n$ algorithm, the total running time will be in

$$O((n+m)\log(n+m)) + \Theta(n+m) = O(s \log s) \quad \text{where } s = \max\{n, m\}.$$

3. a. Sort the list and return its first and last elements as the values of the smallest and largest elements, respectively. Assuming the efficiency of the sorting algorithm used is in $O(n \log n)$, the time efficiency of the entire algorithm will be in

$$O(n \log n) + \Theta(1) + \Theta(1) = O(n \log n).$$

b. The brute-force algorithm and the divide-and-conquer algorithm are both linear, and, hence, superior to the presorting-based algorithm.

4. Let k be the smallest number of searches needed for the sort–binary search algorithm to make fewer comparisons than k searches by sequential search (for average successful searches). Assuming that a sorting algorithm makes about $n \log n$ comparisons on the average and using the formulas for the average number of key comparisons for binary search (about $\log_2 n$) and sequential search (about $n/2$), we get the following inequality

$$n \log_2 n + k \log_2 n \le kn/2.$$

Thus, we need to find the smallest value of k so that

$$k \ge \frac{n \log_2 n}{n/2 - \log_2 n}.$$

Substituting $n = 10^3$ into the right-hand side yields $k_{\min} = 21$; substituting $n = 10^6$ yields $k_{\min} = 40$.

Note: For large values of n, we can simplify the last inequality by eliminating the relatively insignificant term $\log_2 n$ from the denominator of the right-hand side to obtain

$$k \ge \frac{n \log_2 n}{n/2} \quad \text{or} \quad k \ge 2 \log_2 n.$$

This inequality would yield the answers of 20 and 40 for $n = 10^3$ and $n = 10^6$, respectively.

5. a. The following algorithm will beat the brute-force comparisons of the telephone numbers on the bills and the checks: Using an efficient sorting algorithm, sort the bills and sort the checks. (In both cases, sorting has

to be done with respect to their telephone numbers, say, in increasing order.) Then do a merging-like scan of the two sorted lists by comparing the telephone numbers b_i and c_j on the current bill and check, respectively: if $b_i < c_j$, add b_i to the list of unpaid telephone numbers and increment i; if $b_i > c_j$, increment j; if $b_i = c_j$, increment both i and j. Stop as soon as one of the two lists becomes empty and append all the remaining telephone numbers on the bill list, if any, to the list of the unpaid ones.

The time efficiency of this algorithm will be in

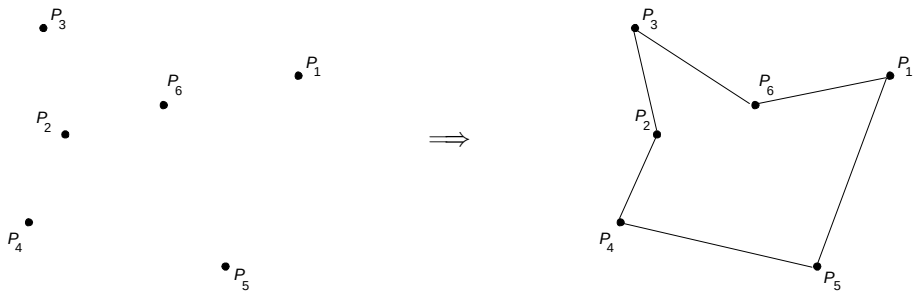
$$O(n \log n) + O(m \log m) + O(n + m) \underset{n \geq m}{=} O(n \log n).$$

This is superior to the $O(nm)$ efficiency of the brute-force algorithm (but inferior, for the average case, to solving this problem with hashing discussed in Section 7.3).

b. Initialize 50 state counters to zero. Scan the list of student records and, for a current student record, increment the corresponding state counter. The algorithm's time efficiency will be in $\Theta(n)$, which is superior to any algorithm that uses presorting of student records by a comparison-based algorithm.

6. a. The problem has a solution if and only if all the points don't lie on the same line. And, if a solution exists, it may not be unique.

b. Find the lowest point P^*, i.e., the one with the smallest y coordinate. in the set. (If there is a tie, take, say, the leftmost among them, i.e., the one with the smallest x coordinate.) For each of the other $n - 1$ points, compute its angle in the polar coordinate system with the origin at P^* and sort the points in increasing order of these angles, breaking ties in favor of a point closer to P^*. (Instead of the angles, you can use the line slopes with respect to the horizontal line through P^*.) Connect the points in the order generated, adding the last segment to return to P^*.



Finding the lowest point P^* is in $\Theta(n)$, computing the angles (slopes) is in $\Theta(n)$, sorting the points according to their angles can be done with a $O(n \log n)$ algorithm. The efficiency of the entire algorithm is in $O(n \log n)$.

7. Assume first that $s = 0$. Then $A[i] + A[j] = 0$ if and only if $A[i] = -A[j]$, i.e., these two elements have the same absolute value but opposite signs. We can check for presence of such elements in a given array in several different ways. If all the elements are known to be distinct, we can simply replace each element $A[i]$ by its absolute value $|A[i]|$ and solve the element uniqueness problem for the array of the absolute values in $O(n \log n)$ time with the presorting based algorithm. If a given array can have equal elements, we can modify our approach as follows: We can sort the array in nondecreasing order of their absolute values (e.g., -6, 3, -3, 1, 3 becomes 1, 3, -3, 3, -6), and then scan the array sorted in this fashion to check whether it contains a consecutive pair of elements with the same absolute value and opposite signs (e.g., 1, 3, -3, 3, -6 does). If such a pair of elements exists, the algorithm returns yes, otherwise, it returns no.

   The case of an arbitrary value of s is reduced to the case of $s = 0$ by the following substitution: $A[i]+A[j] = s$ if and only if $(A[i]-s/2)+(A[j]-s/2) = 0$. In other words, we can start the algorithm by subtracting $s/2$ from each element and then proceed as described above.

   (Note that we took advantage of the instance simplification idea twice: by reducing the problem's instance to one with $s = 0$ and by presorting the array.)

8. Sort all the a_i's and b_j's by a $O(n \log n)$ algorithm in a single nondecreasing list, treating b_j as if it were smaller than a_i in case of the tie $a_i = b_j$. Scan the list left to right computing the running difference D between the number of a_i's and b_j's seen so far. In other words, initialize D to 0 and then increment or decrement it by 1 depending on whether the next element on the list is a left endpoint a_i or a right endpoint b_j, respectively. The maximum value of D is the number in question.

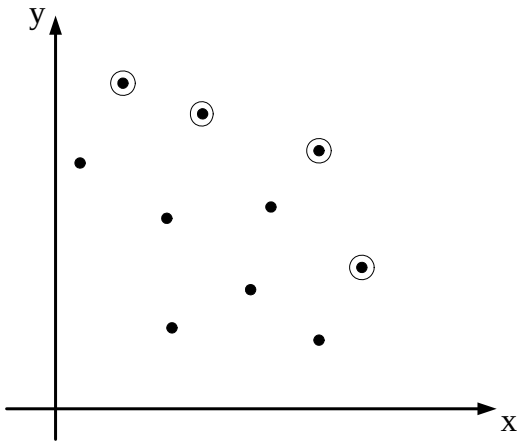
   Note 1: One can also implement this algorithm by sorting a_i's and b_j's separately and then computing the running difference D by merging-like processing of the two sorted lists.

   Note 2: This solution is suggested by D. Ginat in his paper "Algorithmic Pattern and the Case of the Sliding Delta," *SIGCSE Bulletin*, vol. 36, no. 2 (June 2004), pp. 29-33.

9. Start by sorting the list in increasing order. Then repeat the following $n-1$ times: If the first inequality sign is "<", place the first (smallest) number in the first box; otherwise, place there the last (largest) number. After that, delete the number from the list and the box it was put in. Finally, when just a single number remains, place it in the remaining box.

   Note: The problem was posted on a Web page of *The Math Circle* at www.themathcircle.org/researchproblems.php (accessed Oct. 4, 2010).

10. a. Sort the points in nondecreasing order of their x-coordinates resolving ties by listing first a point with a smaller y-coordinate. Then scan the sorted list right to left outputting the points with the strictly largest y-coordinate seen so far during this scan. All the outputted points and only them are the maximum points of the set.



   Let $(\bar{x}, \bar{y})$ be a point outputted by the algorithm. This means that it has a larger y-coordinate than any point previously encountered by the right-to-left scan, i.e., any point with a larger x coordinate. Hence, none of those points can dominate $(\bar{x}, \bar{y})$. No any point to be encountered later in the scan can dominate this point either: all of them have either a smaller x coordinate or the same x-coordinate but a smaller y-coordinate. Conversely, if a point is not outputted by the algorithm, it is dominated by another point in the set and hence is not a maximum point.

   Note: The algorithm is mentioned by Jon Bentley in his paper "Multidimensional divide-and-conquer," *Communications of the ACM*, vol. 23, no. 4 (April 1980), 214–229.

11. First, attach to every word in the file—as another field of the word's record, for example—its signature defined as the string of the word's letters in alphabetical order. Obviously, words belong to the same anagram set if and

only if they have the same signature. Sort the records in alphabetical order of their signatures. Scan the list to identify contiguous subsequences, of length greater than one, of records with the same signature.

Note: Jon Bentley describes a real-life application where a similar problem occurred in [Ben00], p.17, Problem 6.

Solutions to Exercises 6.2

1. a. Solve the following system by Gaussian elimination

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ 2x_1 + x_2 + x_3 &= 3 \\ x_1 - x_2 + 3x_3 &= 8 \end{aligned}$$

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 2 & 1 & 1 & 3 \\ 1 & -1 & 3 & 8 \end{bmatrix} \quad \begin{matrix} \text{row 2 - } \frac{2}{1}\text{row 1} \\ \text{row 3 - } \frac{1}{1}\text{row 1} \end{matrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 0 & -1 & -1 & -1 \\ 0 & -2 & 2 & 6 \end{bmatrix} \quad \text{row 3 - } \frac{-2}{-1}\text{row 2}$$

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 4 & 8 \end{bmatrix}$$

Then, by backward substitutions, we obtain the solution as follows:

$x_3 = 8/4 = 2,\ x_2 = (-1 + x_3)/(-1) = -1,$ and $x_1 = (2 - x_3 - x_2)/1 = 1.$

2. a. Repeating the elimination stage (or using its results obtained in Problem 1), we get the following matrices L and U:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 2 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & -1 \\ 0 & 0 & 4 \end{bmatrix}.$$

On substituting $y = Ux$ into $LUx = b$, the system $Ly = b$ needs to be solved first. Here, the augmented coefficient matrix is:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 2 & 1 & 0 & 3 \\ 1 & 2 & 1 & 8 \end{bmatrix}$$

Its solution is

$$y_1 = 2, \quad y_2 = 3 - 2y_1 = -1,\ y_3 = 8 - y_1 - 2y_2 = 8.$$

Solving now the system $Ux = y$, whose augmented coefficient matrix is

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 4 & 8 \end{bmatrix},$$

yields the following solution to the system given:

$$x_3 = 2, \quad x_2 = (-1 + x_3)/(-1) = -1, \; x_1 = 2 - x_3 - x_2 = 1.$$

b. The most fitting answer is the representation change technique.

3. Solving simultaneously the system with the three right-hand side vectors:

$$\left[\begin{array}{cccccc} 1 & 1 & 1 & 1 & 0 & 0 \\ 2 & 1 & 1 & 0 & 1 & 0 \\ 1 & -1 & 3 & 0 & 0 & 1 \end{array}\right] \begin{array}{l} \\ \text{row 2 - } \frac{2}{1}\text{row 1} \\ \text{row 3 - } \frac{1}{1}\text{row 1} \end{array}$$

$$\left[\begin{array}{cccccc} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & -1 & -1 & -2 & 1 & 0 \\ 0 & -2 & 2 & -1 & 0 & 1 \end{array}\right] \quad \text{row 3 - } \frac{-2}{-1}\text{row 1}$$

$$\left[\begin{array}{cccccc} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & -1 & -1 & -2 & 1 & 0 \\ 0 & 0 & 4 & 3 & -2 & 1 \end{array}\right]$$

Solving the system with the first right-hand side column

$$\left[\begin{array}{r} 1 \\ -2 \\ 3 \end{array}\right]$$

yields the following values of the first column of the inverse matrix:

$$\left[\begin{array}{r} -1 \\ \frac{5}{4} \\ \frac{3}{4} \end{array}\right].$$

Solving the system with the second right-hand side column

$$\left[\begin{array}{r} 0 \\ 1 \\ -2 \end{array}\right]$$

yields the following values of the second column of the inverse matrix:

$$\left[\begin{array}{r} 1 \\ -\frac{1}{2} \\ -\frac{1}{2} \end{array}\right].$$

Solving the system with the third right-hand side column

$$\left[\begin{array}{c} 0 \\ 0 \\ 1 \end{array}\right]$$

yields the following values of the third column of the inverse matrix:

$$\begin{bmatrix} 0 \\ -\frac{1}{4} \\ \frac{1}{4} \end{bmatrix}.$$

Thus, the inverse of the coefficient matrix is

$$\begin{bmatrix} -1 & 1 & 0 \\ \frac{5}{4} & -\frac{1}{2} & -\frac{1}{4} \\ \frac{3}{4} & -\frac{1}{2} & \frac{1}{4} \end{bmatrix},$$

which leads to the following solution to the original system

$$x = A^{-1}b = \begin{bmatrix} -1 & 1 & 0 \\ \frac{5}{4} & -\frac{1}{2} & -\frac{1}{4} \\ \frac{3}{4} & -\frac{1}{2} & \frac{1}{4} \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 8 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}.$$

4. In general, the fact that $f_1(n) \in \Theta(n^3)$, $f_2(n) \in \Theta(n^3)$, and $f_3(n) \in \Theta(n^3)$ does not necessarily imply that $f_1(n) - f_2(n) + f_3(n) \in \Theta(n^3)$, because the coefficients of the highest third-degree terms can cancel each other. As a specific example, consider $f_1(n) = n^3 + n$, $f_2(n) = 2n^3$, and $f_3(n) = n^3$. Each of this functions is in $\Theta(n^3)$, but $f_1(n) - f_2(n) + f_3(n) = n \in \Theta(n)$.

5. **Algorithm** *GaussBackSub*$(A[1..n, 1..n+1])$
   //Implements the backward substitution stage of Gaussian elimination
   //by solving a given system with an upper-triangular coefficient matrix
   //Input: Matrix $A[1..n, 1, ..n+1]$, with the first n columns in the upper-
   //triangular form
   //Output: A solution of the system of n linear equations in n unknowns
   //whose coefficient matrix and right-hand side are the first n columns
   //of A and its $(n+1)$st column, respectively
   for $i \leftarrow n$ **downto** 1 **do**
   $temp \leftarrow 0.0$
   for $j \leftarrow n$ **downto** $i + 1$
   $temp \leftarrow temp + A[i, j] * x[j]$
   $x[i] \leftarrow (A[i, n+1] - temp)/A[i, i]$
   return x

   The basic operation is multiplication of two numbers. The number of times it will be executed is given by the sum

$$\begin{aligned} M(n) &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n} (n - (i+1) + 1) = \sum_{i=1}^{n} (n - i) \\ &= (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2} \in \Theta(n^2). \end{aligned}$$

6. Let $D^{(G)}(n)$ and $M^{(G)}(n)$ be the numbers of divisions and multiplications made by *GaussElimination*, respectively. Using the count formula derived in Section 6.2, we obtain the following approximate counts:

$$D^{(G)}(n) = M^{(G)}(n) \approx \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1 \approx \frac{1}{3}n^3.$$

Let $D^{(BG)}(n)$ and $M^{(BG)}(n)$ be the numbers of divisions and multiplications made by *BetterGaussElimination*, respectively. We have the following approximations:

$$D^{(BG)}(n) \approx \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 \approx \frac{1}{2}n^2 \text{ and } M^{(BG)}(n) = M^{(G)}(n) \approx \frac{1}{3}n^3.$$

Let c_d and c_m be the times of one division and of one multiplication, respectively. We can estimate the ratio of the running times of the two algorithms as follows:

$$\begin{aligned} \frac{T^{(G)}(n)}{T^{(BG)}(n)} &\approx \frac{c_d D^{(G)}(n) + c_m M^{(G)}(n)}{c_d D^{(BG)}(n) + c_m M^{(BG)}(n)} \approx \frac{c_d \frac{1}{3}n^3 + c_m \frac{1}{3}n^3}{c_d \frac{1}{2}n^2 + c_m \frac{1}{3}n^3} \\ &\approx \frac{c_d \frac{1}{3}n^3 + c_m \frac{1}{3}n^3}{c_m \frac{1}{3}n^3} = \frac{c_d + c_m}{c_m} = \frac{c_d}{c_m} + 1 = 4. \end{aligned}$$

7. a. The elimination stage should yield a 2-by-2 upper-triangular matrix with nonzero coefficients on its main diagonal.

b. The elimination stage should yield a 2-by-2 matrix whose second row is $0 \;\; 0 \;\; \alpha$ where $\alpha \neq 0$.

c. The elimination stage should yield a 2-by-2 matrix whose second row is $0 \;\; 0 \;\; 0$.

8. a. Solve the following system by the Gauss-Jordan method

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ 2x_1 + x_2 + x_3 &= 3 \\ x_1 - x_2 + 3x_3 &= 8 \end{aligned}$$

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 2 & 1 & 1 & 3 \\ 1 & -1 & 3 & 8 \end{bmatrix} \begin{array}{l} \\ \text{row } 2 - \frac{2}{1}\text{row } 1 \\ \text{row } 3 - \frac{1}{1}\text{row } 1 \end{array} \begin{bmatrix} 1 & 1 & 1 & 2 \\ 0 & -1 & -1 & -1 \\ 0 & -2 & 2 & 6 \end{bmatrix} \begin{array}{l} \text{row } 1 - \frac{1}{-1}\text{row } 2 \\ \\ \text{row } 3 - \frac{-2}{-1}\text{row } 2 \end{array}$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 4 & 8 \end{bmatrix} \begin{array}{l} \text{row } 1 - \frac{0}{4}\text{row } 3 \\ \text{row } 2 - \frac{-1}{4}\text{row } 3 \\ \\ \end{array} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & 4 & 8 \end{bmatrix}$$

We obtain the solution by dividing the right hand side values by the corresponding elements of the diagonal matrix:

$$x_1 = 1/1 = 1, \quad x_2 = 1/-1 = -1, \quad x_3 = 8/4 = 2.$$

b. The Gauss-Jordan method is also an example of an algorithm based on the instance simplification idea. The two algorithms differ in the kind of a simpler instance to which they transfer a given system: Gaussian elimination transforms a system to an equivalent system with an upper-triangular coefficient matrix whereas the Gauss-Jordan method transforms it to a system with a diagonal matrix.

c. Here is a basic pseudocode for the Gauss-Jordan elimination:

Algorithm *GaussJordan*($A[1..n, 1..n]$, $b[1..n]$)
//Applies Gaussian-Jordan elimination to matrix A of a system's
//coefficients, augmented with vector b of the system's right-hand sides
//Input: Matrix $A[1..n, 1, ..n]$ and column-vector $b[1..n]$
//Output: An equivalent diagonal matrix in place of A with the
//corresponding right-hand side values in its $(n+1)$st column
for $i \leftarrow 1$ **to** n **do** $A[i, n+1] \leftarrow b[i]$ //augment the matrix
for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
        if $j \neq i$
            $temp \leftarrow A[j, i]$ / $A[i, i]$ //assumes $A[i, i] \neq 0$
            for $k \leftarrow i$ **to** $n+1$ **do**
                $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$

The number of multiplications made by the above algorithm can be computed as follows:

$$\begin{aligned}
M(n) &= \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} (n+1-i+1) = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} (n+2-i) \\
&= \sum_{i=1}^{n} (n+2-i)(n-1) = (n-1) \sum_{i=1}^{n} (n+2-i) \\
&= (n-1)[(n+1) + n + \cdots + 2] = (n-1)[(n+1) + n + \cdots + 1 - 1] \\
&= (n-1)[\frac{(n+1)(n+2)}{2} - 1] = \frac{(n-1)n(n+3)}{2} \approx \frac{1}{2} n^3.
\end{aligned}$$

The total number of multiplications made in both elimination and backward substitution stages of the Gaussian elimination method is equal to

$$\frac{n(n-1)(2n+5)}{6} + \frac{(n-1)n}{2} = \frac{(n-1)n(n+4)}{3} \approx \frac{1}{3} n^3,$$

which is about 1.5 smaller than in the Gauss-Jordan method.

Note: The Gauss-Jordan method has an important advantage over Gaussian elimination: being more uniform, it is more suitable for efficient implementation on a parallel computer.

9. Since the time needed for computing the determinant of the system's coefficient matrix is about the same as the time needed for solving the system (or detecting that the system does not have a unique solution) by Gaussian elimination, computing the determinant of the coefficient matrix to check whether it is equal to zero is not a good idea from the algorithmic point of view.

10. a. Solve the following system by Cramer's rule:

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ 2x_1 + x_2 + x_3 &= 3 \\ x_1 - x_2 + 3x_3 &= 8 \end{aligned}$$

$$|A| = \begin{vmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & -1 & 3 \end{vmatrix} = 1 \cdot 1 \cdot 3 + 1 \cdot 1 \cdot 1 + 2 \cdot (-1) \cdot 1 - 1 \cdot 1 \cdot 1 - 2 \cdot 1 \cdot 3 - (-1) \cdot 1 \cdot 1 = -4,$$

$$|A_1| = \begin{vmatrix} 2 & 1 & 1 \\ 3 & 1 & 1 \\ 8 & -1 & 3 \end{vmatrix} = 2 \cdot 1 \cdot 3 + 1 \cdot 1 \cdot 8 + 3 \cdot (-1) \cdot 1 - 8 \cdot 1 \cdot 1 - 3 \cdot 1 \cdot 3 - (-1) \cdot 1 \cdot 2 = -4,$$

$$|A_2| = \begin{vmatrix} 1 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 8 & 3 \end{vmatrix} = 1 \cdot 3 \cdot 3 + 2 \cdot 1 \cdot 1 + 2 \cdot 8 \cdot 1 - 1 \cdot 3 \cdot 1 - 2 \cdot 2 \cdot 3 - 8 \cdot 1 \cdot 1 = 4,$$

$$|A_3| = \begin{vmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 1 & -1 & 8 \end{vmatrix} = 1 \cdot 1 \cdot 8 + 1 \cdot 3 \cdot 1 + 2 \cdot (-1) \cdot 2 - 1 \cdot 1 \cdot 2 - 2 \cdot 1 \cdot 8 - (-1) \cdot 3 \cdot 1 = -8.$$

Hence,

$$x_1 = \frac{|A_1|}{|A|} = \frac{-4}{-4} = 1, \quad x_2 = \frac{|A_2|}{|A|} = \frac{4}{-4} = -1, \quad x_3 = \frac{|A_3|}{|A|} = \frac{-8}{-4} = 2.$$

b. Cramer's rule requires computing $n + 1$ distinct determinants. If each of them is computed by applying Gaussian elimination, it will take about $n+1$ times longer than solving the system by Gaussian elimination. (The time for the backward substitution stage was not accounted for in the preceding argument because of its quadratic efficiency vs. cubic efficiency of the elimination stage.)

11. a. Any feasible state of the board can be described by an $n \times n$ binary matrix, in which the element in the ith row and jth column is equal to 1 if and only if the corresponding panel is lit. Let S and F be such matrices representing the initial and final (all-zeros) boards, respectively. The impact of toggling the panel at (i, j) on a board represented by a binary matrix M can be interpreted as the modulo-2 matrix addition $M + A_{ij}$, where A_{ij} is the matrix in which the only entries equal to 1 are those that are in the (i, j) and adjacent to it positions. For example, if M is a 3-by-3 all-ones matrix representing a 3-by-3 board of all-lit panels, then the impact of turning off the (2,2) panel can be represented as

$$M + A_{22} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}.$$

Let x_{ij} is the number of times the (i, j) panel is toggled in a solution that transforms the board from a starting state S to a final state F. Since the ultimate impact of toggling this panel depends only on whether x_{ij} is even or odd, we can assume with no loss in generality that x_{ij} is either 0 or 1. Then a solution to the puzzle can be expressed by the matrix equation

$$S + \sum_{i,j=1}^{n} x_{ij} A_{ij} = F,$$

where all the operations are assumed to be performed modulo 2. Taking into account that $F = 0$, the all-zeros $n \times n$ matrix, the last equation is equivalent to

$$\sum_{i,j=1}^{n} x_{ij} A_{ij} = S.$$

(The last equation can also be interpreted as transforming the final all-zero board to the initial board S.)

Note: This solution follows Eric W. Weisstein et al. "Lights Out Puzzle" from MathWorld–A Wolfram Web Resource at http://mathworld.wolfram.com/LightsOutPuzzle.html

b. The system of linear equations for the instance in question (see part a) is

$$x_{11} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} + x_{12} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} + x_{21} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} + x_{22} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

or

$$\begin{aligned} 1 \cdot x_{11} + 1 \cdot x_{12} + 1 \cdot x_{21} + 0 \cdot x_{22} &= 1 \\ 1 \cdot x_{11} + 1 \cdot x_{12} + 0 \cdot x_{21} + 1 \cdot x_{22} &= 1 \\ 1 \cdot x_{11} + 0 \cdot x_{12} + 1 \cdot x_{21} + 1 \cdot x_{22} &= 1 \\ 0 \cdot x_{11} + 1 \cdot x_{12} + 1 \cdot x_{21} + 1 \cdot x_{22} &= 1. \end{aligned}$$

Solving this system in modulo-2 arithmetic by Gaussian elimination proceeds as follows:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

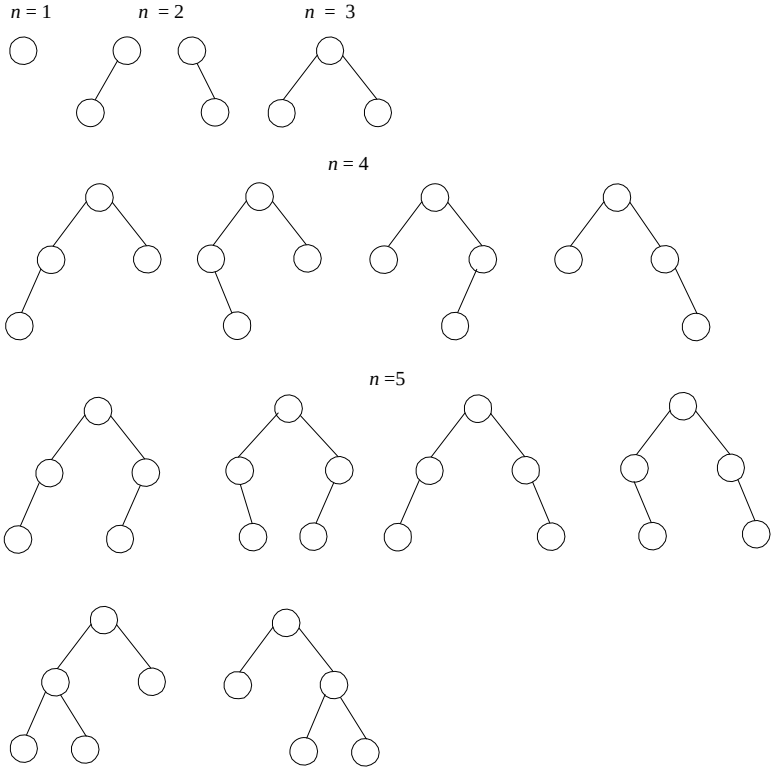
The backward substitutions yield the solution: $x_{11} = 1$, $x_{12} = 1$, $x_{21} = 1$, $x_{22} = 1$, i.e., each of the four panel switches should be toggled once (in any order).

c. The solution to this instance of the puzzle is $x_{11} = x_{13} = x_{22} = x_{31} = x_{33} = 1$ (with all the other components being 0).

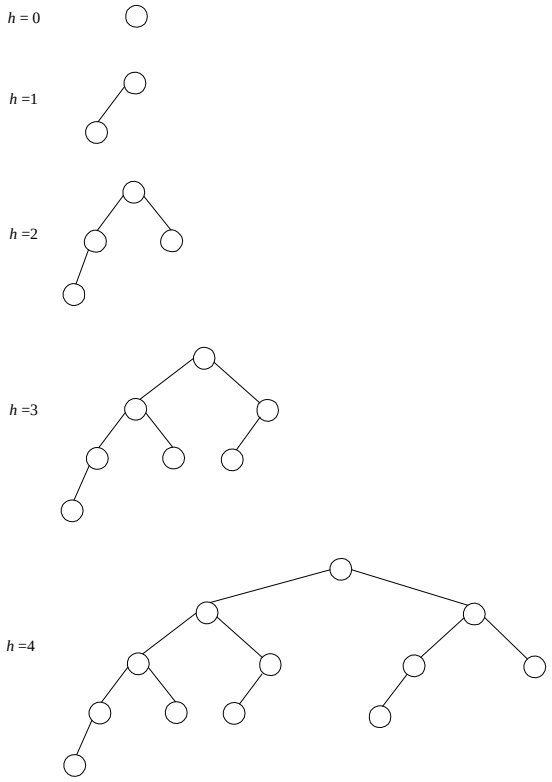
Solutions to Exercises 6.3

1. Only (a) is an AVL tree; (b) has a node (in fact, there are two of them: 4 and 6) that violates the balance requirement; (c) is not a binary search tree because 2 is in the right subtree of 3 (and 7 is in the left subtree of 6).

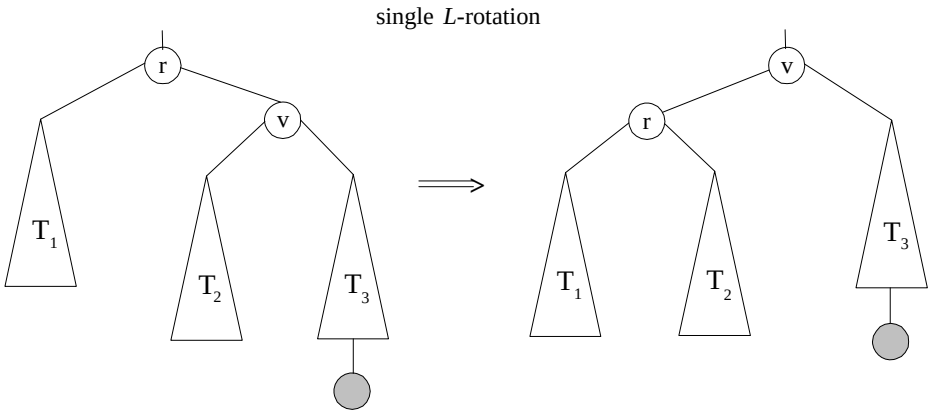
2. a . Here are all the binary trees with n nodes (for $n = 1, 2, 3, 4$, and 5) that satisfy the balance requirement of AVL trees.



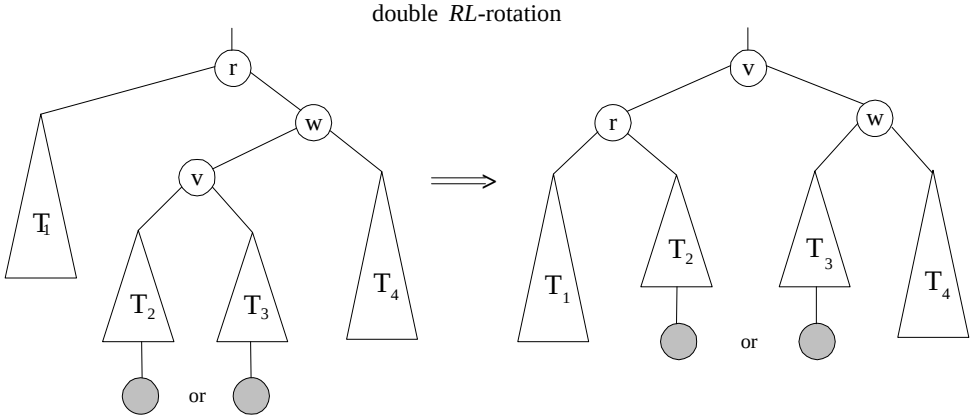
b. A minimal AVL tree (i.e., a tree with the smallest number of nodes) of height 4 must have its left and right subtrees being minimal AVL trees of heights 3 and 2. Following the same recursive logic further, we will find, as one of the possible examples, the following tree with 12 nodes built bottom up:



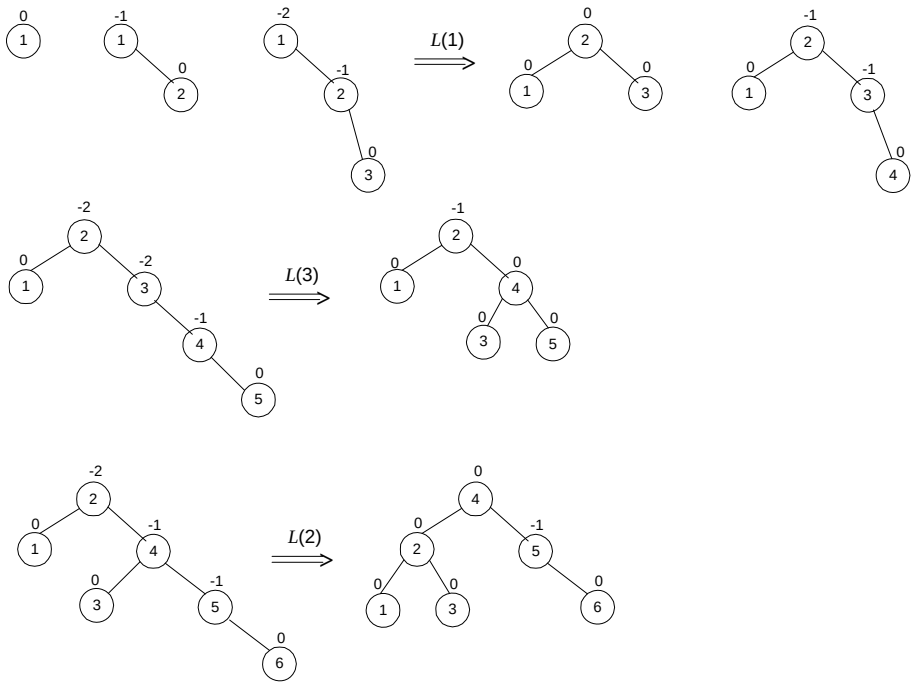
3. a. Here is a diagram of the single L-rotation in its general form:



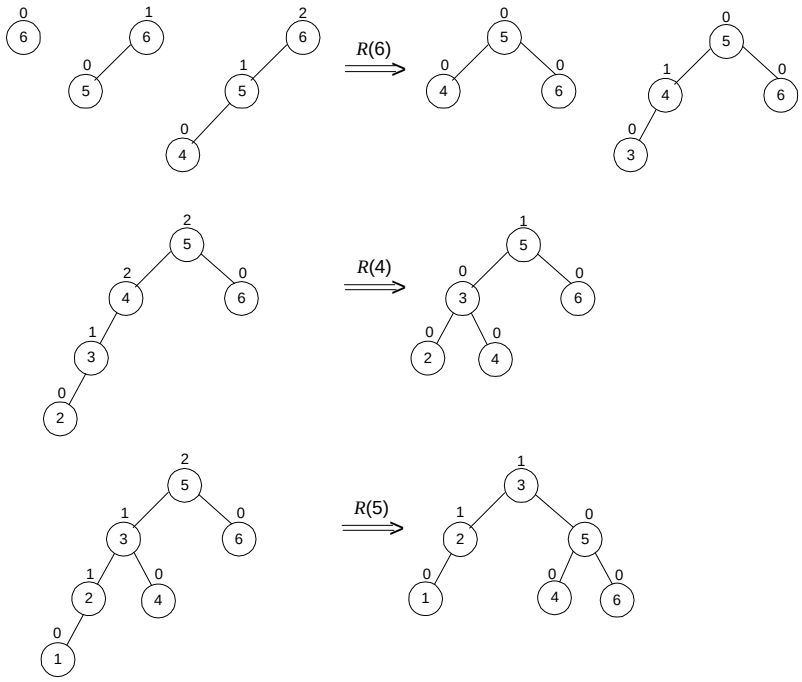
b. Here is a diagram of the double RL-rotation in its general form:



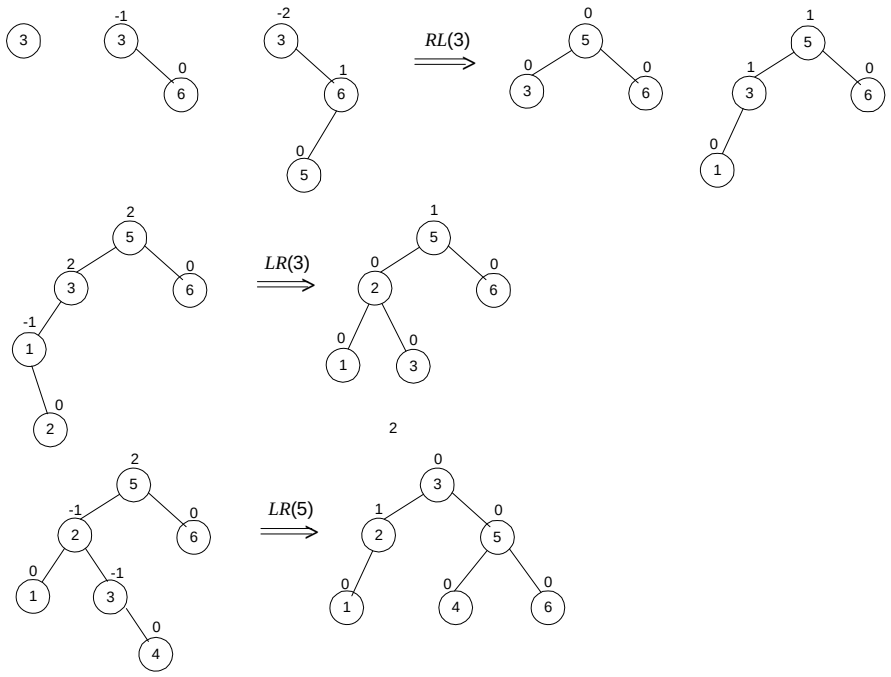
4. a. Construct an AVL tree for the list 1, 2, 3, 4, 5, 6.



b. Construct an AVL tree for the list 6, 5, 4, 3, 2, 1.



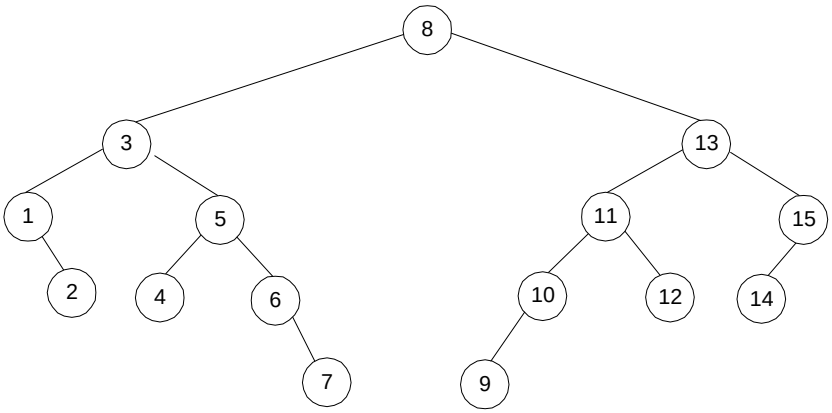
c. Construct an AVL tree for the list 3, 6, 5, 1, 2, 4.



5. a. The simple and efficient algorithm is based on the fact that the smallest and largest keys in a binary search tree are in the leftmost and rightmost nodes of the tree, respectively. Therefore, the smallest key can be found by starting at the root and following the chain of left pointers until a node with the null left pointer is reached: its key is the smallest one in the tree. Similarly, the largest key can be obtained by following the chain of the right pointers. Finally, the range is computed as the difference between the largest and smallest keys found.

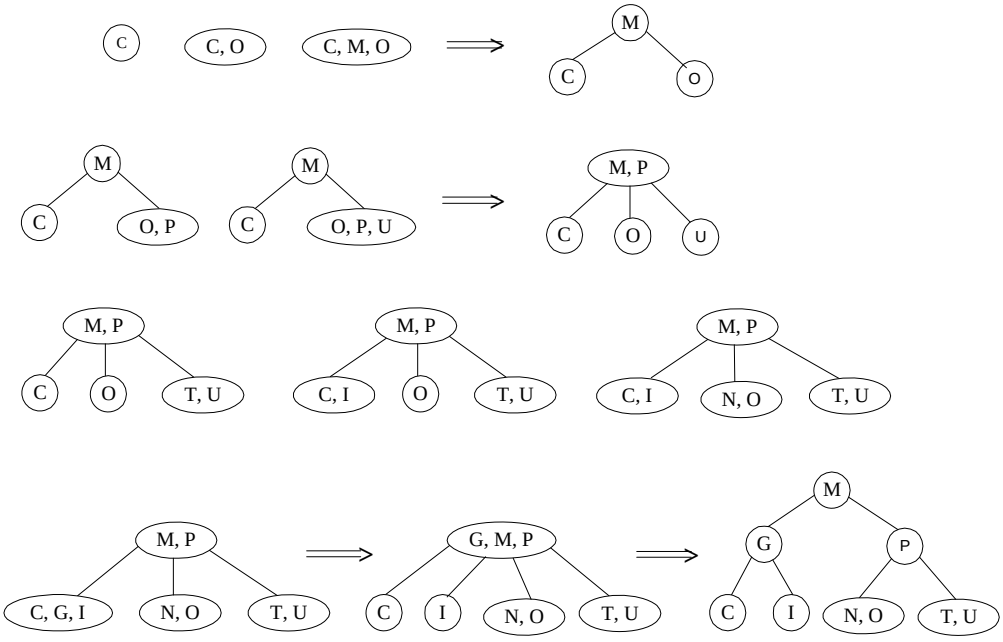
   In the worst case, the leftmost and rightmost nodes will be on the last level of the tree. Hence, the worst-case efficiency will be in $\Theta(\log n) + \Theta(\log n) + \Theta(1) = \Theta(\log n)$.

   b. False. Here is a counterexample in which neither the smallest nor the largest keys are on the last, or the next-to-last, level of an AVL tree:



6. n/a

7. a. Construct a 2-3 tree for the list C, O, M, P, U, T, I, N, G.



b. The largest number of key comparisons in a successful search will be in the searches for O and U; it will be equal to 4. The average number of key comparisons will be given by the following expression:

$\frac{1}{9}C(\mathrm{C}) + \frac{1}{9}C(\mathrm{O}) + \frac{1}{9}C(\mathrm{M}) + \frac{1}{9}C(\mathrm{P}) + \frac{1}{9}C(\mathrm{U}) + \frac{1}{9}C(\mathrm{T}) + \frac{1}{9}C(\mathrm{I}) + \frac{1}{9}C(\mathrm{N}) + \frac{1}{9}C(\mathrm{G})$

$= \frac{1}{9}\cdot 3 + \frac{1}{9}\cdot 4 + \frac{1}{9}\cdot 1 + \frac{1}{9}\cdot 2 + \frac{1}{9}\cdot 4 + \frac{1}{9}\cdot 3 + \frac{1}{9}\cdot 3 + \frac{1}{9}\cdot 3 + \frac{1}{9}\cdot 2 = \frac{25}{9} \approx 2.8.$

8. False. Consider the list B, A. Searching for B in the binary search tree requires 1 comparison while searching for B in the 2-3 tree requires 2 comparisons.

9. The smallest and largest keys in a 2-3 tree are the first key in the leftmost leaf and the second key in the rightmost leaf, respectively. So searching for them requires following the chain of the leftmost pointers from the root to the leaf and of the rightmost pointers from the root to the leaf. Since the height of a 2-3 tree is always in $\Theta(\log n)$, the time efficiency of the algorithm for all cases is in $\Theta(\log n) + \Theta(\log n) + \Theta(1) = \Theta(\log n)$.

10. n/a

Solutions to Exercises 6.4

1. a. Constructing a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm (a root of a subtree being heapified is shown in bold):

1	8	**6**	5	3	7	4	⇒	1	8	7	5	3	6	4
1	**8**	7	5	3	6	4								
1	8	7	5	3	6	4	⇒	8	5	7	1	3	6	4

   b. Constructing a heap for the list 1, 8, 6, 5, 3, 7, 4 by the top-down algorithm (a new element being inserted into a heap is shown in bold):

1														
1	**8**						⇒	8	1					
8	1	**6**												
8	1	6	**5**				⇒	8	5	6	1			
8	5	6	1	**3**										
8	5	6	1	3	**7**		⇒	8	5	7	1	3	6	
8	5	7	1	3	6	**4**								

   c. False. Although for the input to questions (a) and (b) the constructed heaps are the same, in general, it may not be the case. For example, for the input 1, 2, 3, the bottom-up algorithm yields 3, 2, 1 while the top-down algorithm yields 3, 1, 2.

2. For $i = 1, 2, ..., \lfloor n/2 \rfloor$, check whether

$$H[i] \geq \max\{H[2i],\ H[2i+1]\}.$$

   (Of course, if $2i + 1 > n$, just $H[i] \geq H[2i]$ needs to be satisfied.) If the inequality doesn't hold for some i, stop—the array is not a heap; if it holds for every $i = 1, 2, ..., \lfloor n/2 \rfloor$, the array is a heap.

   Since the algorithm makes up to $2\lfloor n/2 \rfloor$ key comparisons, its time efficiency is in $O(n)$.

3. a. A complete binary tree of height h with the minimum number of nodes has the maximum number of nodes on levels 0 through $h-1$ and one node on the last level. The total number of nodes in such a tree is

$$n_{\min}(h) = \sum_{i=0}^{h-1} 2^i + 1 = (2^h - 1) + 1 = 2^h.$$

   A complete binary tree of height h with the maximum number of nodes has the maximum number of nodes on levels 0 through h. The total

number of nodes in such a tree is

$$n_{\max}(h) = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

b. The results established in part (a) imply that for any heap with n nodes and height h

$$2^h \le n < 2^{h+1}.$$

Taking logarithms to base 2 yields

$$h \le \log_2 n < h + 1.$$

This means that h is the largest integer not exceeding $\log_2 n$, i.e., $h = \lfloor \log_2 n \rfloor$.

4. We are asked to prove that $\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1))$ where $n = 2^{h+1} - 1$.

For $n = 2^{h+1} - 1$, the right-hand side of the equality in question becomes

$$2(2^{h+1} - 1 - \log_2(2^{h+1} - 1 + 1)) = 2(2^{h+1} - 1 - (h+1)) = 2(2^{h+1} - h - 2).$$

Using the formula $\sum_{i=1}^{h-1} i2^i = (h-2)2^h + 2$ (see Appendix A), the left-hand side can be simplified as follows:

$$\begin{aligned}\sum_{i=0}^{h-1} 2(h-i)2^i &= 2\sum_{i=0}^{h-1}(h-i)2^i = 2[\sum_{i=0}^{h-1} h2^i - \sum_{i=0}^{h-1} i2^i] \\ &= 2[h(2^h - 1) - (h-2)2^h - 2] \\ &= 2(h2^h - h - h2^h + 2^{h+1} - 2) = 2(2^{h+1} - h - 2).\end{aligned}$$

5. a. The parental dominance requirement implies that we can always find the smallest element of a heap $H[1..n]$ among its leaf positions, i.e., among $H[\lfloor n/2 \rfloor + 1], ..H[n]$. (One can easily prove this assertion by contradiction.) Therefore, we can find the smallest element by simply scanning sequentially the second half of the array H. Deleting this element can be done by exchanging the found element with the last element $H[n]$, decreasing the heap's size by one, and then, if necessary, sifting up the former $H[n]$ from its new position until it is not larger than its parent.

The time efficiency of searching for the smallest element in the second half of the array is in $\Theta(n)$; the time efficiency of deleting it after it has

been found is in $\Theta(1) + \Theta(1) + O(\log n) = O(\log n)$.

b. Searching for v by sequential search in $H[1..n]$ takes care of the searching part of the question. Assuming that the first matching element is found in position i, the deletion of $H[i]$ can be done with the following three-part procedure (which is similar to the ones used for deleting the root and the smallest element): First, exchange $H[i]$ with $H[n]$; second, decrease n by 1; third, heapify the structure by sifting the former $H[n]$ either up or down depending on whether it is larger than its new parent or smaller than the larger of its new children, respectively.

The time efficiency of searching for an element of a given value is in $O(n)$; the time efficiency of deleting it after it has been found is in $\Theta(1) + \Theta(1) + O(\log n) = O(\log n)$.

6. The entries in the table are the efficiency classes for the tree operations of the priority queue ADT: find the value of the largest element, find and delete the largest element, and add a new element of value v:

	Unsorted array	Sorted array	Binary search tree	AVL tree	Heap
FindMax	$\Theta(n)$	$\Theta(1)$	$O(n)$	$O(\log n)$	$\Theta(1)$
DeleteMax	$\Theta(n)$	$\Theta(1)$	$O(n)$	$O(\log n)$	$O(\log n)$
Add(v)	$\Theta(1)$	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$

7. a. Sort 1, 2, 3, 4, 5 by heapsort

Heap Construction

1 **2** 3 4 5
1 5 3 4 2
1 5 3 4 2
5 4 3 1 2

Maximum Deletions

5 4 3 1 2
2 4 3 1 | **5**
4 2 3 1
1 2 3 | **4**
3 2 1
1 2 | **3**
2 1
1 | **2**
1

b. Sort 5, 4, 3, 2, 1 (in increasing order) by heapsort

Heap Construction

5	**4**	3	2	1
5	4	3	2	1

Maximum Deletions

5	4	3	2	1
1	4	3	2 |	**5**
4	2	3	1	
1	2	3 |	**4**	
3	2	1		
1	2 |	**3**		
2	1			
1 |	**2**			
1				

c. Sort S, O, R, T, I, N, G (in alphabetic order) by heapsort

Heap Construction

1	2	3	4	5	6	7
S	O	**R**	T	I	N	G
S	**O**	R	T	I	N	G
S	T	R	O	I	N	G
S	T	R	O	I	N	G
T	S	R	O	I	N	G

Maximum Deletions

1	2	3	4	5	6	7
T	S	R	O	I	N	G
G	S	R	O	I	N |	**T**
S	O	R	G	I	N	
N	O	R	G	I |	**S**	
R	O	N	G	I		
I	O	N	G |	**R**		
O	I	N	G			
G	I	N |	**O**			
N	I	G				
G	I |	**N**				
I	G					
G |	**I**					
G						

8. Heapsort is not stable. For example, it sorts $1'$, $1''$ into $1''$, $1'$.

9. If the heap is thought of as a tree, heapsort should be considered a representation-change algorithm; if the heap is thought of as an array with a special property, heapsort should be considered an instance-simplification algorithm.

10. The answer is selection sort. Note that selection sort is less efficient than heapsort because it uses the array, which is an inferior (to the heap) structure for implementing the priority queue.

11. n/a

12. a. After the bunch of spaghetti rods is put in a vertical position on a tabletop, repeatedly take the tallest rod among the remaining ones out until no more rods are left. This will sort the rods in decreasing order of their lengths.

    b. The method shares with heapsort its principal idea: represent the items to be sorted in a way that makes finding and deleting the largest item a simple task. From a more general perspective, the spaghetti sort is an example, albeit a rather exotic one, of a representation-change algorithm.

Solutions to Exercises 6.5

1. The total number of multiplications made by the algorithm can be computed as follows:

$$M(n) = \sum_{i=0}^{n}(\sum_{j=1}^{i} 1 + 1) = \sum_{i=0}^{n}(i+1) = \sum_{i=0}^{n} i + \sum_{i=0}^{n} 1$$
$$= \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2} \in \Theta(n^2).$$

The number of additions is obtained as

$$A(n) = \sum_{i=0}^{n} 1 = n + 1.$$

2. **Algorithm** *BetterBruteForcePolynomialEvaluation*($P[0..n]$, x)
//Computes the value of polynomial P at a given point x
//by the "lowest-to-highest term" algorithm
//Input: Array $P[0..n]$ of the coefficients of a polynomial of degree n,
// from the lowest to the highest and a number x
//Output: The value of the polynomial at the point x
$p \leftarrow P[0]$; $power \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  $power \leftarrow power * x$
  $p \leftarrow p + P[i] * power$
return p

The number of multiplications made by this algorithm is

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

The number of additions is

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

3. a. If only multiplications need to be taken into account, Horner's rule will be about twice as fast because it makes just n multiplications vs. $2n$ multiplications required by the other algorithm. If one addition takes about the same amount of time as one multiplication, then Horner's rule will be about $(2n + n)/(n + n) = 1.5$ times faster.

b. The answer is no, because Horner's rule doesn't use any extra memory.

4. a. Evaluate $p(x) = 3x^4 - x^3 + 2x + 5$ at $x = -2$.

coefficients	3	-1	0	2	5
$x = -2$	3	(-2)·3+(-1)= -7	(-2)·(-7)+0=14	(-2)·14+2= -26	(-2)·(-26)+5=57

b. The quotient and the remainder of the division of $3x^4 - x^3 + 2x + 5$ by $x + 2$ are $3x^3 - 7x^2 + 14x - 26$ and 57, respectively.

5. Applying Horner's rule to compute $p(2)$ where $p(x) = x^8 + x^7 + x^5 + x^2 + 1$ yields

coefficients	1	1	0	1	0	0	1	0	1
$x = 2$	1	3	6	13	26	52	105	210	421

Thus, $110100101_2 = 421_{10}$.

6. The long division by $x - c$ is done as illustrated below

$$
\begin{array}{r|l}
 & a_n x^{n-1} + \ldots \\
\hline
x - c & a_n x^n + a_{n-1} x^{n-1} \quad + \cdots + a_1 x + a_0 \\
 & {}^{-}a_n x^n - c a_n x^{n-1} \\
\hline
 & (ca_n + a_{n-1}) x^{n-1} + \cdots + a_1 x + a_0
\end{array}
$$

This clearly demonstrates that the first iteration—the one needed to get rid of the leading term $a_n x^n$—requires one multiplication (to get ca_n) and one addition (to add a_{n-1}). After this iteration is repeated $n - 1$ more times, the total number of multiplications and the total number of additions will be n each—exactly the same number of operations needed by Horner's rule. (In fact, it does exactly the same computations as Horner's algorithm would do in computing the value of the polynomial at $x = c$.) Thus, the long division, though much more cumbersome than the synthetic division for hand-and-pencil computations, is actually not less time efficient from the algorithmic point of view.

7. a. Compute a^{17} by the left-to-right binary exponentiation algorithm.
Here, $n = 17 = 10001_2$. So, we have the following table filled left-to-right:

binary digits of n	1	0	0	0	1
product accumulator	a	a^2	$(a^2)^2 = a^4$	$(a^4)^2 = a^8$	$(a^8)^2 \cdot a = a^{17}$

b. Algorithm *LeftRightBinaryExponentiation* will work correctly for $n = 0$ if the variable *product* is initialized to 1 (instead of a) and the loop starts with I (instead of $I - 1$).

8. Compute a^{17} by the right-to-left binary exponentiation algorithm.
Here, $n = 17 = 10001_2$. So, we have the following table filled right-to-left:

1	0	0	0	1	binary digits of n
a^{16}	a^8	a^4	a^2	a	terms a^{2^i}
$a \cdot a^{16} = a^{17}$				a	product accumulator

9. **Algorithm** *ImplicitBinaryExponentiation*(a, n)
//Computes a^n by the implicit right-to-left binary exponentiation
//Input: A number a and a nonnegative integer n
//Output: The value of a^n
$product \leftarrow 1$; $term \leftarrow a$
while $n \neq 0$ **do**
    $b \leftarrow n \bmod 2$; $n \leftarrow \lfloor n/2 \rfloor$
    if $b = 1$
        $product \leftarrow product * term$
    $term \leftarrow term * term$
return *product*

10. Since the polynomial's terms form a geometric series,

$$p(x) = x^n + x^{n-1} + \cdots + x + 1 = \begin{cases} \frac{x^{n+1}-1}{x-1} & \text{if } x \neq 1 \\ n+1 & \text{if } x = 1 \end{cases}$$

Its value can be computed faster than with Horner's rule by computing the right-hand side formula with an efficient exponentiation algorithm for evaluating x^{n+1}.

11. a. With Horner's rule, we can evaluate a polynomial in its coefficient form with n multiplications and n additions. The direct substitution of the x value in the factorized form requires the same number of operations, although these may be operations on complex numbers even for a polynomial with real coefficients.

b. Addition of two polynomials is incomparably simpler for polynomials in their coefficient forms, because, in general, knowing the roots of polynomials $p(x)$ and $q(x)$ helps little in deducing the root values of their sum $p(x) + q(x)$.

c. Multiplication of two polynomials is trivial when they are represented in their factorized form. Indeed, if

$$p(x) = a'_n(x - x'_1) \cdots (x - x'_n) \quad \text{and} \quad q(x) = a''_m(x - x''_1) \cdots (x - x''_m),$$

then

$$p(x)q(x) = a'_n a''_n (x - x'_1) \cdots (x - x'_n)(x - x''_1) \cdots (x - x''_m).$$

To multiply two polynomials in their coefficient form, we need to multiply out

$$p(x)q(x) = (a'_n x^n + \cdots + a'_0)(a''_m x^m + \cdots + a''_0)$$

and collect similar terms to get the product represented in the coefficient form as well.

12. For the general case of n points, *Lagrange's interpolation formula* looks as follows:

$$p(x) = \sum_{i=1}^{n} y_i \frac{(x - x_1) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n)}{(x_i - x_1) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n)}.$$

It's easy to see that when $x = x_i$, all the addends in the sum are equal to zero except the ith one that is equal to y_i.

Solutions to Exercises 6.6

1. a. Since

   $$\begin{aligned}\text{lcm}(m,n) \;=\;& \text{the product of the common prime factors of } m \text{ and } n\\ & \cdot \text{ the product of the prime factors of } m \text{ that are not in } n\\ & \cdot \text{ the product of the prime factors of } n \text{ that are not in } m\end{aligned}$$

   and

   $$\gcd(m,n) = \text{the product of the common prime factors of } m \text{ and } n,$$

   the product of $\text{lcm}(m,n)$ and $\gcd(m,n)$ is equal to

   $$\begin{aligned}&\text{the product of the common prime factors of } m \text{ and } n\\ &\cdot \text{ the product of the prime factors of } m \text{ that are not in } n\\ &\cdot \text{ the product of the prime factors of } n \text{ that are not in } m\\ &\cdot \text{ the product of the common prime factors of } m \text{ and } n.\end{aligned}$$

   Since the product of the first two terms is equal to m and the product of the last two terms is equal to n, we showed that $\text{lcm}(m,n) \cdot \gcd(m,n) = m \cdot n$, and, hence,

   $$\text{lcm}(m,n) = \frac{m \cdot n}{\gcd(m,n)}.$$

   b. If $\gcd(m,n)$ is computed in $O(\log n)$ time, $\text{lcm}(m,n)$ will also be computed in $O(\log n)$ time, because one extra multiplication and one extra division take only constant time.

2. Replace every key K_i of a given list by $-K_i$ and apply a max-heap construction algorithm to the new list. Then change the signs of all the keys again.

3. The induction basis: For $k = 1$, $A^1[i,j]$ is equal to 1 or 0 depending on whether there is an edge from vertex i to vertex j. In either case, it is also equal to the number of paths of length 1 from i to j. For the general step, assume that for a positive integer k, $A^k[i,j]$ is equal to the number of different paths of length k from vertex i to vertex j. Since $A^{k+1} = A^k A$, we have the following equality for the (i,j) element of A^{k+1}:

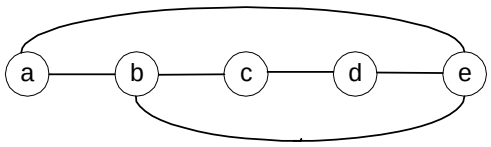
   $$A^{k+1}[i,j] = A^k[i,1]A[1,j] + \cdots + A^k[i,t]A[t,j] + \cdots + A^k[i,n]A[n,j],$$

   where $A^k[i,t]$ is equal to the number of different paths of length k from vertex i to vertex t according to the induction hypothesis and $A[t,j]$ is equal to 1 or 0 depending on whether there is an edge from vertex t to

vertex j for $t = 1, ..., n$. Further, any path of length $k + 1$ from vertex i to vertex j must be made up of a path of length k from vertex i to some intermediate vertex t and an edge from that t to vertex j. Since for different intermediate vertices t we get different paths, the formula above yields the total number of different paths of length $k + 1$ from i to j.

4. a. For the adjacency matrix A of a given graph, compute A^2 with an algorithm whose time efficiency is better than cubic (e.g., Strassen's matrix multiplication discussed in Section 5.4). Check whether there exists a nonzero element $A[i, j]$ in the adjacency matrix such that $A^2[i, j] > 0$: if there is, the graph contains a triangle subgraph, if there is not, the graph does not contain a triangle subgraph.

   b. The algorithm is incorrect because the condition is sufficient but not necessary for a graph to contain a cycle of length 3. Consider, as a counterexample, the DFS tree of the traversal that starts at vertex a of the following graph and resolves ties according to alphabetical order of vertices:



   It does not contain a back edge to a grandparent of a vertex, but the graph does have a cycle of length 3: $a - b - e - a$.

5. The problem can be reduced to the question about the convex hull of a given set of points: if the convex hull is a triangle, the answer is yes, otherwise, the answer is no. There are several algorithms for finding the convex hull for a set of points; quickhull, which was discussed in Section 5.5, is particularly appropriate for this application.

6. Let x be one of the numbers in question; hence, the other number is $n-x$. The problem can be posed as the problem of maximizing $f(x) = x(n-x)$ on the set of all integer values of x. Since the graph of $f(x) = x(n-x)$ is a parabola with the apex at $x = n/2$, the solution is $n/2$ if n is even and $\lfloor n/2 \rfloor$ (or $\lceil n/2 \rceil$) if n is odd. Hence, the numbers in question can be computed as $\lfloor n/2 \rfloor$ and $n - \lfloor n/2 \rfloor$, which works both for even and odd values of n. Assuming that one division by 2 takes a constant time irrespective of n's size, the algorithm's time efficiency is clearly in $\Theta(1)$.

7. Let x_{ij} be a 0-1 variable indicating an assignment of the ith person to the jth job (or, in terms of the cost matrix C, a selection of the matrix element from the ith row and the jth column). The assignment problem can then be posed as the following 0–1 linear programming problem:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \qquad \text{(the total assignment cost)} \\ \text{subject to} & \sum_{j=1}^{n} x_{ij} = 1 \text{ for } i = 1, ...n \text{ (person } i \text{ is assigned to one job)} \\ & \sum_{i=1}^{n} x_{ij} = 1 \text{ for } j = 1, ...n \text{ (job } j \text{ is assigned to one person)} \\ & x_{ij} \in \{0, 1\} \quad \text{for } i = 1, ..., n \text{ and } j = 1, ..., n \end{array}$$

8. We can exploit the specific features of the instance in question to solve it by the following reasoning. Since the expected return from cash is the smallest, the value of cash investment needs to be minimized. Hence, $z = 0.25(x + y)$ in an optimal solution. Substituting $z = 0.25(x + y)$ into $x + y + z = 100$, yields $x + y = 80$ and hence $z = 20$. Similarly, since the expected return on stocks is larger than that of bonds, the amount invested in stocks needs to be maximized. Hence, in an optimal allocation $x = y/3$. Substituting this into $x+y = 80$ yields $y = 60$ and $x = 20$. Thus, the optimal allocation is to put 20 million in stocks, 60 millions in bonds, and 20 million in cash.

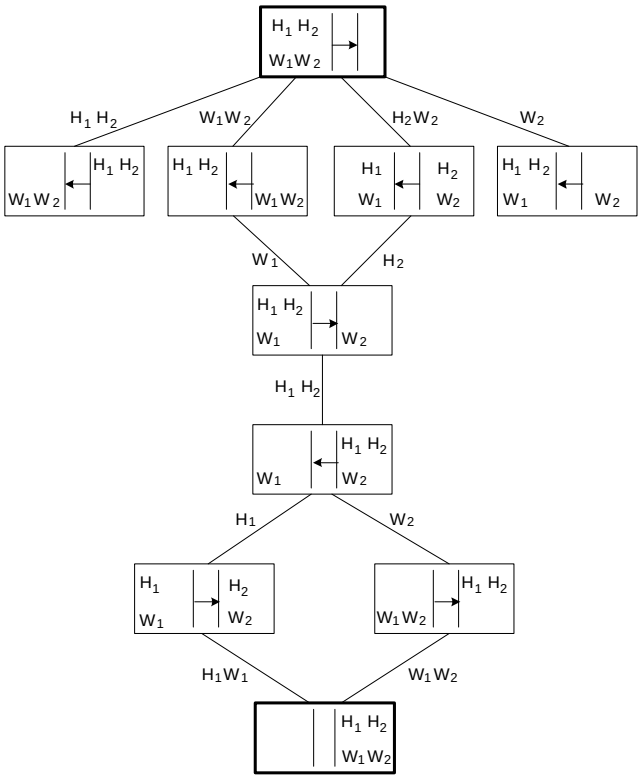
   Note: This method should not be construed as having a power beyond this particular instance. Generally speaking, we need to use general algorithms such as the simplex method for solving linear programming problems with three or more unknowns. A special technique applicable to instances with only two variables is discussed in Section 10.1 (see also the solution to Problem 12 in Exercises 3.3).

9. Create a new graph whose vertices represent the edges of the given graph and connect two vertices in the new graph by an edge if and only if these vertices represent two edges with a common endpoint in the original graph. A solution of the vertex-coloring problem for the new graph solves the edge-coloring problem for the original graph.

10. The problem is obviously equivalent to minimizing independently $\frac{1}{n}\sum_{i=1}^{n} |x_i - x|$ and $\frac{1}{n}\sum_{i=1}^{n} |y_i - y|$. Thus we have two instances of the same problem, whose solution is the median of the numbers defining the instance (see the solution to Problem 2a in Exercises 3.3). Thus, x and y can be found by computing the medians of $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_n$, respectively.

11. a. Here is a state-space graph for the two jealous husbands puzzle: H_i, W_i denote the husband and wife of couple i ($i = 1, 2$), respectively; the two bars | | denote the river; the arrow indicates the direction of the next trip, which is defined by the boat's location. (For the sake of simplicity, the

graph doesn't include crossings that differ by obvious index substitutions such as starting with the first couple H_1W_1 crossing the river instead of the second one H_2W_2.) The vertices corresponding to the initial and final states are shown in bold.



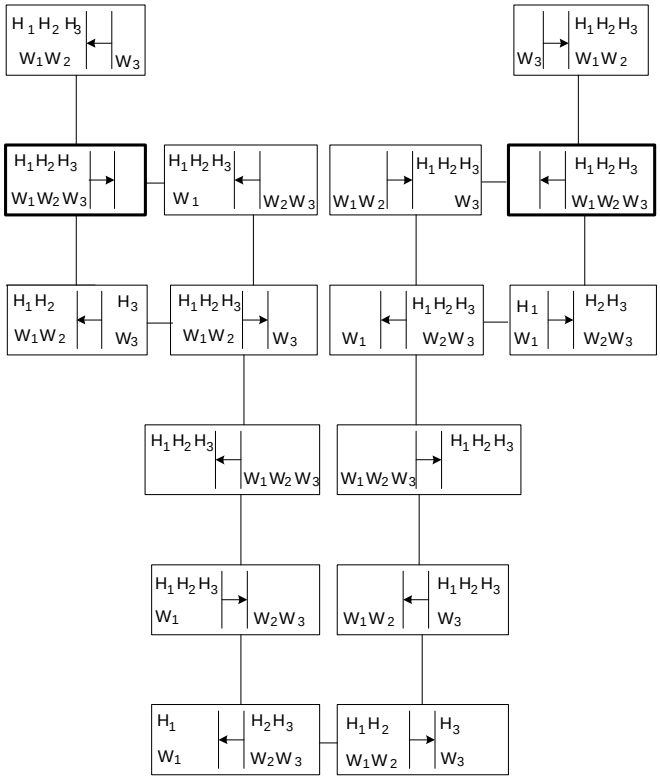
There are four simple paths from the initial-state vertex to the final-state vertex, each five edges long, in this graph. If specified by their edges, they are:

$$\begin{array}{lllll} W_1W_2 & W_1 & H_1H_2 & H_1 & H_1W_1 \\ W_1W_2 & W_1 & H_1H_2 & W_2 & W_1W_2 \\ H_2W_2 & H_2 & H_1H_2 & H_1 & H_1W_1 \\ H_2W_2 & H_2 & H_1H_2 & W_2 & W_1W_2 \end{array}$$

Hence, there are four (to within obvious symmetric substitutions) optimal solutions to this problem, each requiring five river crossings.

b. Here is a state-space graph for the three jealous husbands puzzle: H_i, W_i denote the husband and wife of couple i ($i = 1, 2, 3$), respectively; the two bars | | denote the river; the arrow indicates the possible direction

of the next trip, which is defined by the boat's location. (For the sake of simplicity, the graph doesn't include crossings that differ by obvious index substitutions such as starting with the first or second couple crossing the river instead of the third one H_3W_3.) The vertices corresponding to the initial and final states are shown in bold.



There are four simple paths from the initial-state vertex to the final-state vertex, each eleven edges long, in this graph. If specified by their edges, they are:

W_2W_3 W_2 W_1W_2 W_1 H_2H_3 H_2W_2 H_1H_2 W_3 W_2W_3 W_2 W_1W_2
W_2W_3 W_2 W_1W_2 W_1 H_2H_3 H_2W_2 H_1H_2 W_3 W_2W_3 H_1 H_1W_1
H_3W_3 H_3 W_1W_2 W_1 H_2H_3 H_2W_2 H_1H_2 W_3 W_2W_3 W_2 W_1W_2
H_3W_3 H_3 W_1W_2 W_1 H_2H_3 H_2W_2 H_1H_2 W_3 W_2W_3 H_1 H_1W_1

Hence, there are four (to within obvious symmetric substitutions) optimal solutions to this problem, each requiring eleven river crossings.

c. The problem doesn't have a solution for the number of couples $n \geq 4$. If we start with one or more extra (i.e., beyond 3) couples, no new qualitatively different states will result and after the first six river crossings (see the solution to part (b)), we will arrive at the state with $n-1$ couples

and the boat on the original bank and one couple on the other bank. The only allowed transition from that state will be going back to its predecessor by ferrying a married couple to the other side.

12. One can reduce the problem to the question about existence of an Eulerian circuit in a complete graph with $n+1$ vertices. Vertex i, $0 \leq i \leq n$, in this graph represents a possible number of spots on one of the two halves of an n-domino, and an edge between vertices i and j represents the domino with i and j spots on its halves. The doubles can be either eliminated until a ring including all the other dominoes is constructed and then inserted between any two dominoes with the same number of spots, or they can be represented by loops (edges connecting vertices to themselves). Obviously, an Eulerian circuit in such a graph would specify a ring of all n-dominoes and vice versa. By a well-known theorem—see also Problem 4 in Exercises 1.3—a connected graph has an Eulerian circuit if and only if all its vertices have even degrees. It is the case for the graph in question if and only if n is even. An algorithm for constructing an Eulerian circuit is the subject of Problem 9 in Exercises 4.5.

Solutions to Exercises 7.1

1. The following operations will exchange values of variables u and v:
   $u \leftarrow u + v$ //u holds $u + v$, v holds v
   $v \leftarrow u - v$ //u holds $u + v$, v holds u
   $u \leftarrow u - v$ //u holds v, v holds u

   Note: The same trick is applicable, in fact, to any binary data by employing the "exclusive or" (XOR) operation:
   $u \leftarrow u\text{XOR}v$
   $v \leftarrow u\text{XOR}v$
   $u \leftarrow u\text{XOR}v$

2. Yes, it will work correctly for arrays with equal elements.

3. Input: A: $b,\ c,\ d,\ c,\ b,\ a,\ a,\ b$

	a	b	c	d
Frequencies	2	3	2	1

	a	b	c	d
Distribution values	2	5	7	8

	$D[a..d]$			
$A[7] = b$	2	**5**	7	8
$A[6] = a$	**2**	4	7	8
$A[5] = a$	**1**	4	7	8
$A[4] = b$	0	**4**	7	8
$A[3] = c$	0	3	**7**	8
$A[2] = d$	0	3	6	**8**
$A[1] = c$	0	3	**6**	7
$A[0] = b$	0	**3**	5	7

$S[0..7]$							
				b			
	a						
a							
			b				
						c	
							d
					c		
		b					

4. Yes, it is stable because the algorithm scans its input right-to-left and puts equal elements into their section of the sorted array right-to-left as well.

5. **for** $i \leftarrow 0$ **to** $n - 1$ **do** $S[A[i] - 1] \leftarrow A[i]$

6. Vertex u is an ancestor of vertex v in a rooted ordered tree T if and only if the following two inequalities hold

$$preorder(u) \le preorder(v) \quad \text{and} \quad postorder(u) \ge postorder(v),$$

   where *preorder* and *postorder* are the numbers assigned to the vertices by the preorder and postorder traversals of T, respectively. Indeed, preorder traversal visits recursively the root and then the subtrees numbered from left to right. Therefore,

$$preorder(u) \le preorder(v)$$

if and only if either u is an ancestor of v (i.e., u is on the simple path from the root's tree to v) or u is to the left of v (i.e., u and v are not on the same simple path from the root to a leaf and $T(u)$ is to the left of $T(v)$ where $T(u)$ and $T(v)$ are the subtrees of the nearest common ancestor of u and v, respectively). Similarly, postorder traversal visits recursively the subtrees numbered from left to right and then the root. Therefore,

$$postorder(u) \geq postorder(v)$$

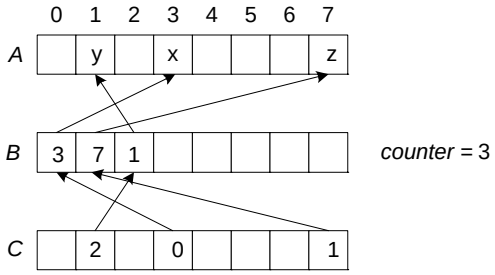
if and only if either u is an ancestor of v or v is to the left of u. Hence,

$$preorder(u) \leq preorder(v) \quad \text{and} \quad postorder(u) \geq postorder(v)$$

is necessary and sufficient for u to be an ancestor of v.

The time efficiencies of both traversals are in $O(n)$ (Section 5.3); once the preorder and postorder numbers are precomputed, checking the two inequalities takes constant time for any given pair of the vertices.

7. a. The following diagram depicts the results of these assignments (the values of the unspecified elements in the arrays are undefined):



b. $A[i]$ is initialized if and only if $0 \leq C[i] \leq counter - 1$ and $B[C[i]] = i$. (It is useful to note that the elements of array C define the inverse to the mapping defined by the elements of array B.) Hence, if these two conditions hold, $A[i]$ contains the value it has been initialized with; otherwise, it has not been initialized.

8. Count for each statue the total number of statues shorter than it, as it is done by comparison counting sort. Then use these counts to move the statues from their current positions $i = 0, 1, ..., 9$ to their target positions $Count[0], Count[1],...,Count[9]$. For example, this can be done as follows. Take the first statue that is not in its target position, i.e., $i \neq Count[i]$, and move it to its target position $Count[i]$, move the statue from the current position $Count[i]$ to that statue's target position $Count[Count[i]]$, and so on. If there remains a statue still not in its target position, repeat this step

starting with the next such stature. Since every statue is moved directly to its final position, the algorithm obviously minimizes the total distance that the statues are moved. Note that the target positions can also be found by sorting on paper pairs (statue id, statue height) in increasing order of heights by any sorting algorithm, where statue id can be any statue identifier such as its initial position or inventory number or some name.

9. n/a

10. Taking into account the board's symmetries, there are 765 essentially different positions in this game..It is easier to use the minimax algorithm with the standard position evaluation function defined as the difference between the number of lines still open for the computer win and that for the opponent win.

Solutions to Exercises 7.2

1. The shift table for the pattern BAOBAB in a text comprised of English letters, the period, and a space will be

c	A	B	C	D	.	.	.	O	.	.	.	Z	.	_
$t(c)$	1	2	6	6	6			3	6			6	6	6

The actual search will proceed as shown below:

```
B E S S _ K N E W _ A B O U T _ B A O B A B S
B A O B A B                   B A O B A B
            B A O B A B             B A O B A B
                B A O B A B
```

2. a. For the pattern TCCTATTCTT and the alphabet {A, C, G, T}, the shift table looks as follows:

c	A	C	G	T
$t(c)$	5	2	10	1

b. Below the text and the pattern, we list the characters of the text that are aligned with the last T of the pattern, along with the corresponding number of character comparisons (both successful and unsuccessful) and the shift size:

the text: TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
the pattern: TCCTATTCTT

T: 2 comparisons, shift 1
C: 1 comparison, shift 2
T: 2 comparisons, shift 1
A: 1 comparison, shift 5
T: 8 comparisons, shift 1
T: 3 comparisons, shift 1
T: 3 comparisons, shift 1
A: 1 comparison, shift 5
T: 2 comparisons, shift 1
C: 1 comparison, shift 2
C: 1 comparison, shift 2
T: 2 comparisons, shift 1
A: 1 comparison, shift 5
T: 10 comparisons to stop the successful search

3. a. For the pattern 00001, the shift table is

c	0	1
$t(c)$	1	5

The algorithm will make one unsuccessful comparison and then shift the pattern one position to the right on each of its trials:

```
0 0 0 0 0 0                                   0 0 0 0 0
0 0 0 0 1
  0 0 0 0 1
                  etc.
                                              0 0 0 0 1
```

The total number of character comparisons will be $C = 1 \cdot 996 = 996$.

b. For the pattern `10000`, the shift table is

c	`0`	`1`
$t(c)$	1	4

The algorithm will make four successful and one unsuccessful comparison and then shift the pattern one position to the right on each of its trials:

```
0 0 0 0 0 0                                   0 0 0 0 0
1 0 0 0 0
  1 0 0 0 0 0
                  etc.
                                              1 0 0 0 0
```

The total number of character comparisons will be $C = 5 \cdot 996 = 4980$.

c. For the pattern `01010`, the shift table is

c	`0`	`1`
$t(c)$	2	1

The algorithm will make one successful and one unsuccessful comparison and then shift the pattern two positions to the right on each of its trials:

```
0 0 0 0 0 0                                   0 0 0 0 0 0
0 1 0 1 0
    0 1 0 1 0
                        etc.
                                              0 1 0 1 0
```

The left end of the pattern in the trials will be aligned against the text's characters in positions 0, 2, 4, ..., 994, which is 498 trials. (We can also get this number by looking at the positions of the right end of the pattern. This leads to finding the largest integer k such that $4 + 2(k - 1) \leq 999$, which is $k = 498$.) Thus, the total number of character comparisons will be $C = 2 \cdot 498 = 996$.

4. a. The worst case: e.g., searching for the pattern $\underbrace{\texttt{10...0}}_{m-1}$ in the text of n 0's. $C_w = m(n-m+1)$.

   b. The best case: e.g., searching for the pattern $\underbrace{\texttt{0...0}}_{m}$ in the text of n 0's. $C_b = m$.

5. Yes: e.g., for the pattern $\underbrace{\texttt{10...0}}_{m-1}$ and the text $\underbrace{\texttt{0...0}}_{n}$, $C_{bf} = n-m+1$ while $C_{Horspool} = m(n-m+1)$.

6. We can shift the pattern exactly in the same manner as we would in the case of a mismatch, i.e., by the entry $t(c)$ in the shift table for the text's character c aligned against the last character of the pattern.

7. a. For the pattern 00001, the shift tables will be filled as follows:

   the bad-symbol table

c	0	1
$t_1(c)$	1	5

   .

   the good-suffix table

k	the pattern	d_2
1	0000**1**	5
2	000**01**	5
3	00**001**	5
4	0**0001**	5

   On each of its trials, the algorithm will make one unsuccessful comparison and then shift the pattern by $d_1 = \max\{t_1(0) - 0,\ 1\} = 1$ position to the right without consulting the good-suffix table:

```
0 0 0 0 0 0                                   0 0 0 0 0
0 0 0 0 1
  0 0 0 0 1
            etc.
                                              0 0 0 0 1
```

   The total number of character comparisons will be $C = 1 \cdot 996 = 996$.

   b. For the pattern 10000, the shift tables will be filled as follows:

   the bad-symbol table

$c.$	0	1
$t_1(c)$	1	4

   .

   the good-suffix table

k	the pattern	d_2
1	1000**0**	3
2	100**00**	2
3	10**000**	1
4	**10000**	5

   .

   On each of its trials, the algorithm will make four successful and one

unsuccessful comparison and then shift the pattern by the maximum of $d_1 = \max\{t_1(0) - 4,\ 1\} = 1$ and $d_2 = t_2(4) = 5$, i.e., by 5 characters to the right:

```
0 0 0 0 0 0 0 0 0 0                          0 0 0 0 0
1 0 0 0 0
          1 0 0 0 0
                          etc.
                                             1 0 0 0 0
```

The total number of character comparisons will be $C = 5 \cdot 200 = 1000$.

c. For the pattern `01010`, the shift tables will be filled as follows:

the bad-symbol table

c	0	1
$t_1(c)$	2	1

.

the good-suffix table

k	the pattern	d_2
1	0101**0**	4
2	010**10**	4
3	01**010**	2
4	0**1010**	2

.

On each trial, the algorithm will make one successful and one unsuccessful comparison. The shift's size will be computed as the maximum of $d_1 = \max\{t_1(0) - 1,\ 1\} = 1$ and $d_2 = t_2(1) = 4$, which is 4. If we count character positions starting with 0, the right end of the pattern in the trials will be aligned against the text's characters in positions 4, 8, 12, ..., with the last term in this arithmetic progression less than or equal to 999. This leads to finding the largest integer k such that $4 + 4(k - 1) \le 999$, which is $k = 249$.

```
0 0 0 0 0 0                                  0 0 0 0 0 0
0 1 0 1 0
        0 1 0 1 0
                    etc.
                                       0 1 0 1 0
```

Thus, the total number of character comparisons will be $C = 2 \cdot 249 = 498$.

8. a. Yes, the Boyer-Moore algorithm can get by with just the bad-symbol shift table.

   b. No: The bad-symbol table is necessary because it's the only one used by the algorithm if the first pair of characters does not match.

9. a. Horspool's algorithm can also compare the remaining $m - 1$ characters of the pattern from left to right because it shifts the pattern based only on the text's character aligned with the last character of the pattern.

b. The Boyer-Moore algorithm must compare the remaining $m-1$ characters of the pattern from right to left because of the good-suffix shift table.

10. n/a

11. a. The following brute-force algorithm is based on straightforward checking of the circular shift definition.

Algorithm *RightCyclicShift*($S[0..n-1]$, $T[0..n-1]$)
//Checks by brute force whether string T is a right cyclic shift of string S
//Input: Strings $S[0..n-1]$ and $T[0..n-1]$
//Output: Returns **true** if T is a right cyclic shift of S and **false** otherwise
for $i \leftarrow 0$ **to** $n-1$ **do** //try cyclic shift i positions to the right
    $k \leftarrow 0$ //number of matched characters
    while $k \leq n-1$ **and** $S[(i+k) \bmod n] = T[k]$ **do**
        $k \leftarrow k+1$
    if $k = n$ **return true**
return false

The algorithm uses no extra space, and its time efficiency is clearly $O(n^2)$. (It's $\Theta(n^2)$ in the worst case.)

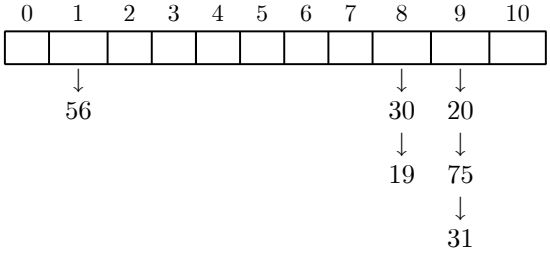
b. A more time-efficient algorithm is to append the $(n-1)$-character prefix of S to the end of S and then search for T in this expanded string by the Boyer-Moore algorithm. The time efficiencies of appending the prefix and searching for the pattern of length n in the text of length $2n-1$ add to $\Theta(n)+O(n)=\Theta(n)$. The extra space used is $\Theta(n)$ for the appended prefix and $\Theta(|\Sigma|)+\Theta(n)$ for the two shift tables, where $|\Sigma|$ is the character alphabet size. Hence the space efficiency of the algorithm is $\Theta(n)+\Theta(|\Sigma|)$.

Solutions to Exercises 7.3

1. a.
   The list of keys: 30, 20, 56, 75, 31, 19
   The hash function: $h(K) = K \bmod 11$
   The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

   The open hash table:



   b. The largest number of key comparisons in a successful search in this table is 3 (in searching for $K = 31$).

   c. The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 2 = \frac{10}{6} \approx 1.7.$$

2. a.
   The list of keys: 30, 20, 56, 75, 31, 19
   The hash function: $h(K) = K \bmod 11$
   The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

0	1	2	3	4	5	6	7	8	9	10
								30		
								30	20	
	56							30	20	
	56							30	20	75
31	56							30	20	75
31	56	19						30	20	75

   b. The largest number of key comparisons in a successful search is 6 (when searching for $K = 19$).

   c. The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 6 = \frac{14}{6} \approx 2.3.$$

3. The number of different values of such a function would be obviously limited by the size of the alphabet. Besides, it is usually not the case that the probability of a word to start with a particular letter is the same for all the letters.

4. The probability of all n keys to be hashed to a particular address is equal to $\left(\frac{1}{m}\right)^n$. Since there are m different addresses, the answer is $\left(\frac{1}{m}\right)^n m = \frac{1}{m^{n-1}}$.

5. The probability of n people having different birthdays is $\frac{364}{365}\frac{363}{365}\cdots\frac{365-(n-1)}{365}$. The smallest value of n for which this expression becomes less than 0.5 is 23. Sedgewick and Flajolet [SF96] give the following analytical solution to the problem:

$$(1-\frac{1}{M})(1-\frac{2}{M})\cdots(1-\frac{n-1}{M}) \approx \frac{1}{2} \quad \text{where } M = 365.$$

Taking the natural logarithms of both sides yields

$$\ln(1-\frac{1}{M})(1-\frac{2}{M})\cdots(1-\frac{n-1}{M}) \approx -\ln 2 \quad \text{or} \quad \sum_{k=1}^{n-1} \ln(1-\frac{k}{M}) \approx -\ln 2.$$

Using $\ln(1-x) \approx -x$, we obtain

$$\sum_{k=1}^{n-1} \frac{k}{M} \approx \ln 2 \quad \text{or} \quad \frac{(n-1)n}{2M} \approx \ln 2. \quad \text{Hence,} \quad n \approx \sqrt{2M \ln 2} \approx 22.5.$$

The implication for hashing is that we should expect collisions even if the size of a hash table is much larger (by more than a factor of 10) than the number of keys.

6. a. If all the keys are known to be distinct, a new key can always be inserted at the beginning of its linked list; this will make the insertion operation $\Theta(1)$. This will not change the efficiencies of search and deletion, however.

b. Searching in a sorted list can be stopped as soon as a key larger than the search key is encountered. Both deletion (that must follow a search) and insertion will benefit for the same reason. To sort a dictionary stored in linked lists of a hash table, we can merge the k nonempty lists to get the entire dictionary sorted. (This operation is called the *k-way merge*.) To do this efficiently, it's convenient to arrange the current first elements of the lists in a min-heap.

7. Insert successive elements of the list in a hash table until a matching element is encountered or the list is exhausted. The worst-case efficiency will be in $\Theta(n^2)$: all n distinct keys are hashed to the same address so that the number of key comparisons will be $\sum_{i=1}^{n}(i-1) \in \Theta(n^2)$. The average-case efficiency, with keys distributed about evenly so that searching for each of them takes $\Theta(1)$ time, will be in $\Theta(n)$.

8. In each cell of the table, the first and second entries are the average-case and worst-case efficiencies, respectively.

	unordered array	ordered array	binary search tree	balanced search tree	hashing
search	$\Theta(n)$ $\Theta(n)$	$\Theta(\log n)$ $\Theta(\log n)$	$\Theta(\log n)$ $\Theta(n)$	$\Theta(\log n)$ $\Theta(\log n)$	$\Theta(1)$ $\Theta(n)$
insertion	$\Theta(1)$ $\Theta(1)$	$\Theta(n)$ $\Theta(n)$	$\Theta(\log n)$ $\Theta(n)$	$\Theta(\log n)$ $\Theta(\log n)$	$\Theta(1)$ $\Theta(n)$
deletion	$\Theta(1)$ $\Theta(1)$	$\Theta(n)$ $\Theta(n)$	$\Theta(\log n)$ $\Theta(n)$	$\Theta(\log n)$ $\Theta(\log n)$	$\Theta(1)$ $\Theta(n)$

9. Representation change—one of the three varieties of transform-and-conquer.

10. n/a

Solutions to Exercises 7.4

1. Here are a few common examples of using an index: labeling drawers of a file cabinet with, say, a range of letters; an index of a book's terms indicating the page or pages on which the term is defined or mentioned; marking a range of pages in an address book, a dictionary; or an encyclopedia; marking a page of a telephone book or a dictionary with the first and last entry on the page; indexing areas of a geographic map by dividing the map into square regions.

2. a.

$$\begin{aligned}
& 1 + \sum_{i=1}^{h-1} 2\lceil m/2 \rceil^{i-1}(\lceil m/2 \rceil - 1) + 2\lceil m/2 \rceil^{h-1} \\
= \;& 1 + 2(\lceil m/2 \rceil - 1) \sum_{i=1}^{h-1} \lceil m/2 \rceil^{i-1} + 2\lceil m/2 \rceil^{h-1} \\
= \;& 1 + 2(\lceil m/2 \rceil - 1) \sum_{j=0}^{h-2} \lceil m/2 \rceil^{j} + 2\lceil m/2 \rceil^{h-1} \\
= \;& 1 + 2(\lceil m/2 \rceil - 1) \frac{\lceil m/2 \rceil^{h-1} - 1}{(\lceil m/2 \rceil - 1)} + 2\lceil m/2 \rceil^{h-1} \\
= \;& 1 + 2\lceil m/2 \rceil^{h-1} - 2 + 2\lceil m/2 \rceil^{h-1} \\
= \;& 4\lceil m/2 \rceil^{h-1} - 1.
\end{aligned}$$

   b. The inequality

$$n \geq 4\lceil m/2 \rceil^{h-1} - 1$$

   is equivalent to

$$\frac{n+1}{4} \geq \lceil m/2 \rceil^{h-1}.$$

   Taking the logarithms base $\lceil m/2 \rceil$ of both hand sides yields

$$\log_{\lceil m/2 \rceil} \frac{n+1}{4} \geq \log_{\lceil m/2 \rceil} \lceil m/2 \rceil^{h-1}$$

   or

$$\log_{\lceil m/2 \rceil} \frac{n+1}{4} \geq h - 1.$$

   Hence,

$$h \leq \log_{\lceil m/2 \rceil} \frac{n+1}{4} + 1$$

   or, since h is an integer,

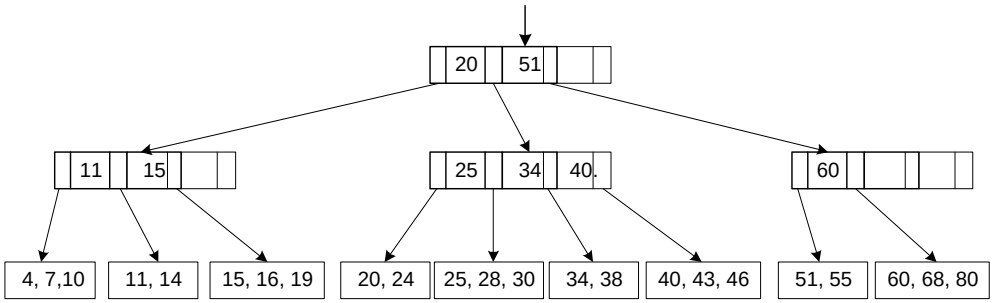
$$h \leq \lfloor \log_{\lceil m/2 \rceil} \frac{n+1}{4} \rfloor + 1.$$

3. If the tree's root is stored in main memory, the number of disk accesses will be equal to the number of the levels minus 1, which is exactly the height of the tree. So, we need to find the smallest value of the order m so that the height of the B-tree with $n = 10^8$ keys does not exceed 3. Using the upper bound of the B-tree's height, we obtain the following inequality

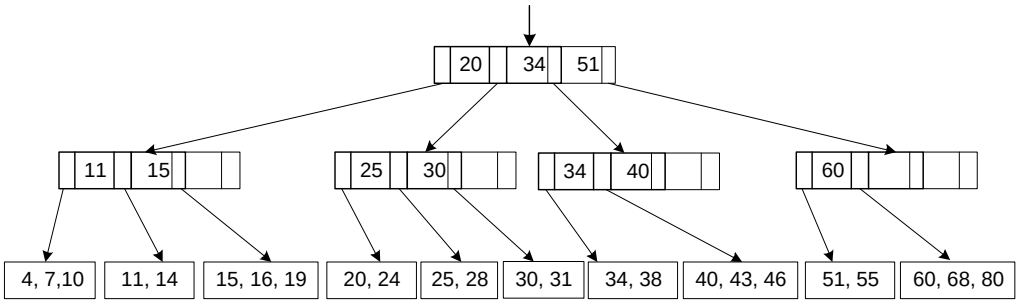
$$\lfloor \log_{\lceil m/2 \rceil} \frac{n+1}{4} \rfloor + 1 \leq 3 \quad \text{or} \quad \lfloor \log_{\lceil m/2 \rceil} \frac{n+1}{4} \rfloor \leq 2.$$

By "trial and error," we can find that the smallest value of m that satisfies this inequality is 585.

4. Since there is enough room for 30 in the leaf for it, the resulting B-tree will look as follows



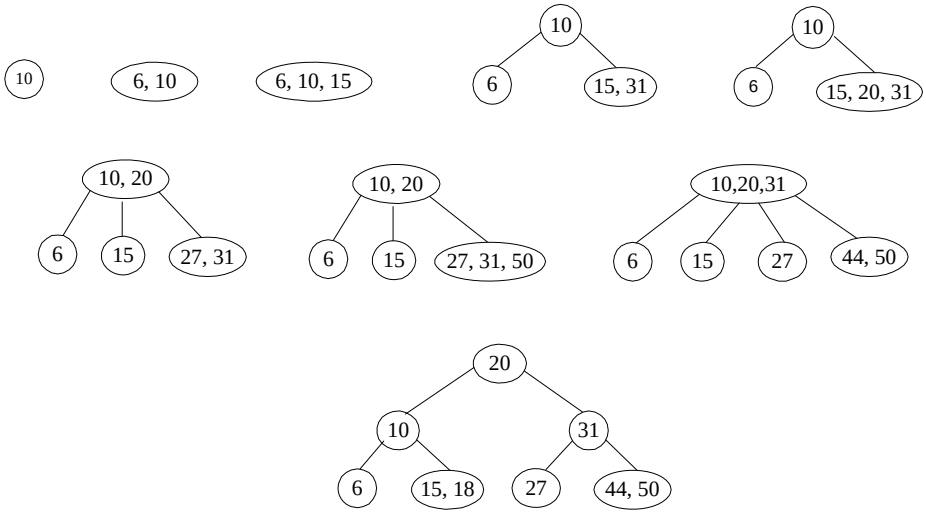
Inserting 31 will require the leaf's split and then its parent's split:



5. Starting at the root, follow the chain of the rightmost pointers to the (rightmost) leaf. The largest key is the last key in that leaf.

6. a. Constructing a top-down 2-3-4 tree by inserting the following list of keys in the initially empty tree:

10, 6, 15, 31, 20, 27, 50, 44, 18.



b. The principal advantage of splitting full nodes (4-nodes with 3 keys) on a way down during insertion of a new key lies in the fact that if the appropriate leaf turns out to be full, its split will never cause a chain reaction of splits because the leaf's parent will always have a room for an extra key. (If the parent is full before the insertion, it is split before the leaf is reached.) This is not the case for the insertion algorithm employed for 2-3 trees (see Section 6.3).

The disadvantage of splitting full nodes on the way down lies in the fact that it can lead to a taller tree than necessary. For the list of part (a), for example, the tree before the last one had a room for key 18 in the leaf containing key 15 and therefore didn't require a split executed by the top-down insertion.

7. n/a

Solutions to Exercises 8.1

1. Both techniques solve a problem by dividing it into several subproblems. But smaller subproblems in divide-and-conquer do not overlap; therefore their solutions are not stored for reuse. Smaller subproblems in dynamic programming do overlap; therefore their solutions are stored for reuse.

2. The application of the dynamic programming algorithm to the input 5, 1, 2, 10, 6, 2 in section 8.1 yielded the following table:

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	17

   Using the data in the first six columns, we conclude that the largest amount of money that can be obtained for the input 5, 1, 2, 10, 6 is $F(5) = 15$, which is obtained by taking coins $c_4 = 10$ and $c_1 = 5$.

3. a. The time efficiency analysis of the algorithm in question is identical to that of the top-down computation of the nth Fibonacci number in Section 2.5: see recurrence (2.11) for the number of additions made by both algorithms. Hence, the time efficiency class is $\Theta(\phi^n)$ where $\phi = (1+\sqrt{5})/2$.

   b. If an exhaustive search algorithm generates all the subsets of the coin row given before checking which of them don't include adjacent coins, the number of the subsets will be equal 2^n, which answers the question. But even the number of subsets with no adjacent coins is exponential as well. Indeed, let $S(n)$ be the number of such subsets including the empty one. They can be divided into the subsets that contain the first coin and the subsets that do not contain it. The number of the subsets of the first kind is equal to $S(n-2)$; the number of the subsets of the second kind is equal to $S(n-1)$. Hence we have the recurrence

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

   It's easy to see that the terms of the sequence $S(n)$, defined by these formulas, are nothing else but the Fibonacci numbers running two terms "ahead" of their canonical counterparts defined the initial conditions $F(0) = 0$ and $F(1) = 1$. Hence $S(n) = F(n+2) \in \Theta(\phi^{n+2}) = \Theta(\phi^n)$, which answers the question posed by the exercise.

4. The application of the dynamic programming algorithm to the instance

given yields the following table

$F[0] = 0$

n	0	1	2	3	4	5	6	7	8	9
F	0									

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6	7	8	9
F	0	1								

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2							

$F[3] = \min\{F[3-1],\ F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1						

$F[4] = \min\{F[4-1],\ F[4-3]\} + 1 = 2$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2					

$F[5] = \min\{F[5-1],\ F[5-3],\ F[5-5]\} + 1 = 2$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2	1				

$F[6] = \min\{F[6-1],\ F[6-3],\ F[6-5]\} + 1 = 2$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2	1	2			

$F[7] = \min\{F[7-1],\ F[7-3],\ F[7-5]\} + 1 = 3$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2	1	2	3		

$F[8] = \min\{F[8-1],\ F[8-3],\ F[8-5]\} + 1 = 2$

n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2	1	2	3	2	

$F[9] = \min\{F[9-1],\ F[9-3],\ F[9-5]\} + 1 = 3$

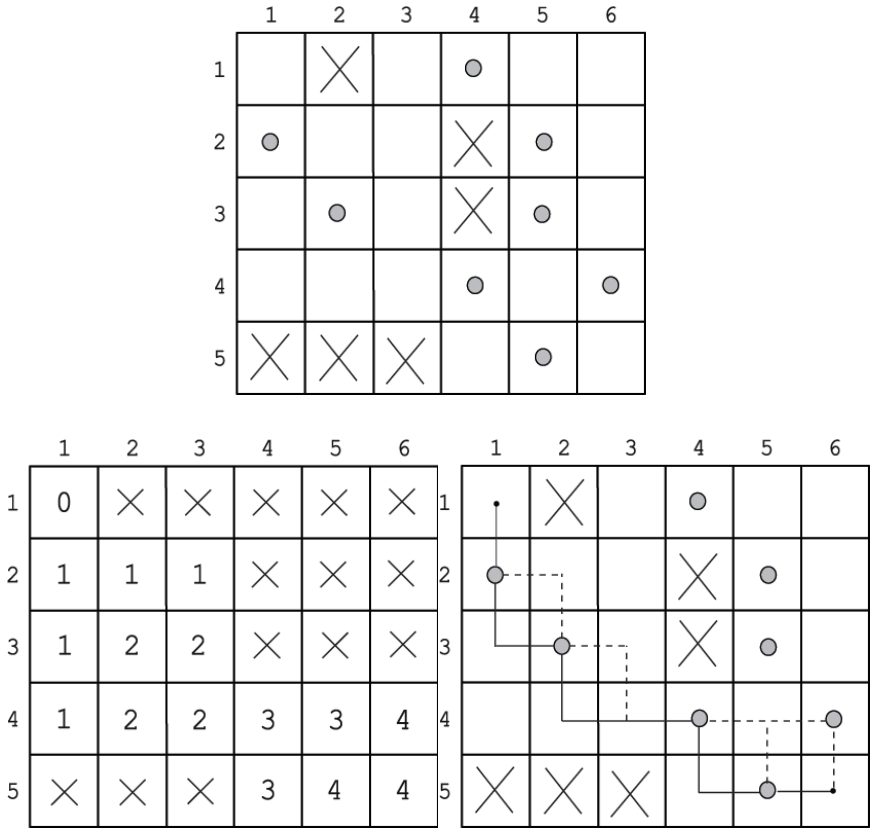
n	0	1	2	3	4	5	6	7	8	9
F	0	1	2	1	2	1	2	3	2	**3**

Application of Algorithm *MinCoinChange* to amount $n = 9$ and coin denominations 1, 3, and 5

The minimum number of coins obtained is $F(9) = 3$. There are two optimal coin sets: $\{1, 3, 5\}$ and $\{3, 3, 3\}$.

5. Formula (8.5) used for computing the largest number of coins that can be brought to a cell needs to be adjusted as follows. If a cell is inadmissible or has no admissible neighbors above or to the left of it, it is marked as inadmissible (if it hasn't been already marked as such) and no value is computed for it. If a cell has just one admissible neighbor above or to the left of it, only this value is used in formula (8.5). Otherwise, the algorithm proceeds filling the table exactly the same way as it's done in the section. The results of its application to the board given are shown below. The largest number of coins that can be brought by the robot to the lower

right corner is 4; there are 12 different paths for the robot to do this.



Application of the dynamic programming algorithm to the board shown in the top figure.

.

6. Let $F(n)$ be the maximum price for a given rod of length n. We have the following recurrence for its values:

$$\begin{aligned} F(n) &= \max_{1 \le j \le n} \{p_j + F(n-j)\} \quad \text{for } n > 0, \\ F(0) &= 0. \end{aligned}$$

Using this recurrence, we can fill a one-dimensional array with $n+1$ consecutive values of F. The last value, $F(n)$, will be the maximum possible price in question. Since computing each value of $F(i)$ requires i additions (and i integer subtractions), the total number of additions is equal to $1 + 2 + \cdots + n = n(n+1)/2$, making the algorithm's time efficiency quadratic. The space efficiency of the algorithm is obviously linear since it uses an additional array of $n+1$ elements.

Actual cuts yielding the maximum price can be obtained by backtracing (see the examples in Section 8.1 and the solution to Problem 5 in these exercises).

Note: The rod-cutting problem is discussed in Cormen et al. *Introduction to Algorithms, 3rd edition,* Section 15.1.

7. a. With no loss of generality, we can assume that the rook is initially located in the lower left corner of a chessboard, whose rows and columns are numbered from 1 to 8 bottom up and left to right, respectively. Let $P(i, j)$ be the number of the rook's shortest paths from square (1,1) to square (i, j) in the ith row and the jth column, where $1 \leq i, j \leq 8$. Any such path will be composed of vertical and horizontal moves directed toward the goal. Obviously, $P(i, 1) = P(1, j) = 1$ for any $1 \leq i, j \leq 8$. In general, any shortest path to square (i, j) is reached either from its left neighbor, i.e., square $(i, j - 1)$, or from its neighbors below, i.e., square $(i - 1, j)$. Hence we have the following recurrence

$$\begin{aligned} P(i, j) &= P(i, j - 1) + P(i - 1, j) \text{ for } 1 < i, j \leq 8, \\ P(i, 1) &= P(1, j) = 1 \text{ for } 1 \leq i, j \leq 8. \end{aligned}$$

Using this recurrence, we can compute the values of $P(i, j)$ for each square $(i.j)$ of the board. This can be done either row by row, or column by column, or diagonal by diagonal. (One can also take advantage of the board's symmetry to make the computations only for the squares either on and above or on and below the board's main diagonal.) The results are given in the diagram below:

1	8	36	120	330	792	1716	3432
1	7	28	84	210	462	924	1716
1	6	21	56	126	252	462	792
1	5	15	35	70	126	210	330
1	4	10	20	35	56	84	120
1	3	6	10	15	21	28	36
1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

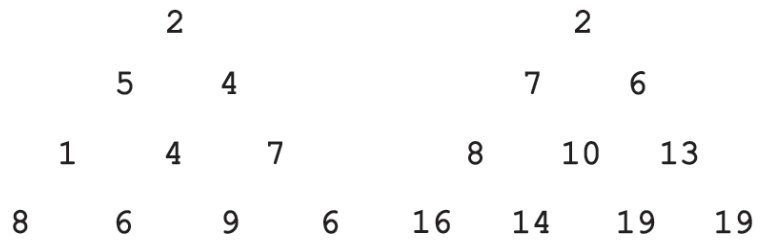
b. Any shortest path from square (1,1) to square (8,8) can be thought of as 14 consecutive moves to adjacent squares, seven of which being up while the other seven being to the right. For example, the shortest path composed of the vertical move from (1,1) to (8,1) followed by the horizontal move from (8,1) to (8,8) corresponds to the following sequence of 14 one-square moves:

$$(u, u, u, u, u, u, u, r, r, r, r, r, r, r),$$

where u and r stand for a move up and to the right, respectively. Hence, the total number of distinct shortest paths is equal to the number of different ways to choose seven u-positions among the total of 14 possible positions, which is equal to $C(14, 7)$.

Note: The problem is discussed in Martin Gardner's *aha!Insight*, Scientific American/W.H.Freeman, p. 10.

8. Using the standard dynamic programming technique, compute the minimum sum along a descending path from the apex to each number in the triangle. Start with the apex, for which this sum is obviously equal to the number itself. Then compute the sums moving top down and, say, left to right across the triangle's rows as follows. For any number that is either the first or the last in its row, add the sum previously computed for the adjacent number in the preceding row and the number itself; for any number that is neither the first nor the last in its row, add the smaller of the previously computed sums for the two adjacent numbers in the preceding row and the number itself. When all such sums are computed for the numbers at the base of the triangle, find the smallest among them.
The figure below illustrates the algorithm for the triangle given in the problem's statement.



(a) (b)

Application of the dynamic programming algorithm for the minimum-sum descent problem: (a) input triangle; (b) triangle of minimum sums along descending paths with 14 being the smallest

The time efficiency of the algorithm is obviously quadratic: it spends constant time to find a minimum sum on a path to each of $n(n+1)/2$ numbers in the triangle and then needs a linear time to find the smallest among n such sums for the base of the triangle.

Note: The problem is analogous to Problem 18 on the Project Euler website at *projecteuler.net* (accessed February 14, 2011) .

9. The recurrence underlying the algorithm in question is

$$\begin{aligned} C(n,k) &= C(n-1,k-1) + C(n-1,k) \quad \text{for } n > k > 0, \\ C(n,0) &= C(n,n) = 1. \end{aligned}$$

Here is pseudocode of the dynamic programming algorithm based on these formulas.

Algorithm *Binomial*(n,k)

```
//Computes C(n,k) by the dynamic programming algorithm
//Input: A pair of nonnegative integers n ≥ k ≥ 0
//Output: The value of C(n,k)
for i ← 0 to n do
    for j ← 0 to min(i,k) do
        if  j = 0 or j = i
            C[i,j] ← 1
        else C[i,j] ← C[i-1,j-1] + C[i-1,j]
return C[n,k]
```

The algorithm computes all the binomial coefficients $C(i,j)$ for $0 \le i \le n$ and $0 \le j \le \min(i,k)$, which fill the triangular table with $k+1$ rows followed by a rectangular table with $n-k$ rows and $k+1$ columns. One addition is made to compute each binomial coefficient, except those columns 0 and k in the triangular table and those in column 0 in the rectangular table. Therefore we can compute $A(n,k)$, the total number of additions made by this algorithm in computing $C(n,k)$, as follows:

$$\begin{aligned} A(n,k) &= \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1 = \sum_{i=1}^{k} (i-1) + \sum_{i=k+1}^{n} k \\ &= \frac{(k-1)k}{2} + k(n-k). \end{aligned}$$

The simple algebra yields

$$\frac{(k-1)k}{2} + k(n-k) = nk - \frac{1}{2}k^2 - \frac{1}{2}k.$$

So, we can obtain an upper bound by eliminating the negative terms:

$$nk - \frac{1}{2}k^2 - \frac{1}{2}k \le nk \quad \text{for all } n, k \ge 0.$$

We can get a lower bound by considering $n \ge 2$ and $0 \le k \le n$:

$$nk - \frac{1}{2}k^2 - \frac{1}{2}k \ge nk - \frac{1}{2}nk - \frac{1}{2}k\frac{1}{2}n = \frac{1}{4}nk.$$

Hence $A(n, k) \in \Theta(nk)$, which indicates the time efficiency class of the algorithm.

The space efficiency of the above algorithm is also $\Theta(nk)$ since the computed binomial coefficients occupy their own memory cells in the rectangular table with $n + 1$ rows and $k + 1$ columns. Considering only the memory cells actually used by the algorithm still yields the same asymptotic class:

$$S(n, k) = \sum_{i=0}^{k}(i + 1) + \sum_{i=k+1}^{n}(k + 1) = \frac{(k + 1)(k + 2)}{2} + (k + 1)(n - k) \in \Theta(nk).$$

The following algorithm uses just one-dimensional table by storing a new row of binomial coefficients over its predecessor.

Algorithm *Binomial2*(n, k)
//Computes $C(n, k)$ by the dynamic programming algorithm
//with a one-dimensional table
//Input: A pair of nonnegative integers $n \ge k \ge 0$
//Output: The value of $C(n, k)$
for $i \leftarrow 0$ **to** n **do**
    if $i \le k$ //in the triangular part
        $T[i] \leftarrow 1$ //the diagonal element
        $u \leftarrow i - 1$ //the rightmost element to be computed
    else $u \leftarrow k$ //in the rectangular part
    //overwrite the preceding row moving right to left
    for $j \leftarrow u$ **downto** 1 **do**
        $T[j] \leftarrow T[j - 1] + T[j]$
return $T[k]$

The space efficiency of *Binomial2* is obviously $\Theta(n)$.

10. After topological sorting of the digraph's vertices, the following formula for the length of the longest path to vertex u is all but obvious:

$$d_u = \max_{(v,u)\in E}\{d_v + w(v, u)\}$$

a. **Algorithm** *DagLongestPath*(G)

```
//Finds the length of a longest path in a dag
//Input: A weighted dag G = ⟨V, E⟩
//Output: The length of its longest path dmax
topologically sort the vertices of G
for every vertex v do
    d_v ← 0 //the length of the longest path to v
for every vertex v taken in topological order do
    for every vertex u such that (v, u) ∈ E do
        d_u ← max{d_v + w(v, u), d_u}
dmax ← 0
for every vertex v do
    dmax ← max{d_v, dmax}
return dmax
```

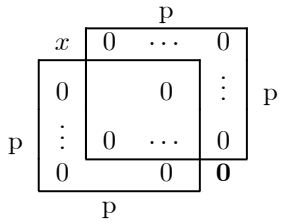
b. The dag in question will have $n + 1$ vertices, placed for convenience in a row mimicking the coin row: vertex 0 for the start and the other n vertices for the coins in the order given. The start vertex will be connected to each of the other vertices; each of the coin-representing vertices will have an outgoing edge to each of the coin-representing vertices after its immediate successor. The weight of an edge will be the value of the coin represented by the vertex the edge points to.

11. We will assume that the rows and columns of a given matrix B are numbered from 1 to m and from 1 to n, respectively. Let $F(i, j)$ be the order of the largest all-zero submatrix of a given matrix B with its low right corner at (i, j). If $b_{ij} = 1$, $F(i, j) = 0$ according to the definition of $F(i, j)$. The nontrivial case is that of $b_{ij} = 0$. In this case, one can prove that

$$\begin{aligned} F(i,j) &= \min\{F(i-1,j),\ F(i,j-1),\ F(i-1,j-1)\} + 1 \quad \text{for } 1 \le i \le m,\ 1 \le j \le n \\ F(0,j) &= 0 \quad \text{for } 0 \le j \le n \quad \text{and} \quad F(i,0) = 0 \quad \text{for } 0 \le i \le m. \end{aligned}$$

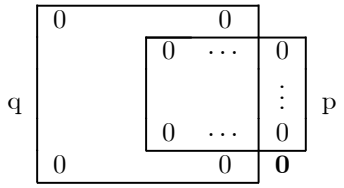
Indeed, if $b_{ij} = 0$ and at least one of $b_{i-1,j}$, $b_{i,j-1}$, and $b_{i-1,j-1}$ is 1, then $F(i, j) = 1$ and $\min\{F(i-1, j), F(i, j-1), F(i-1, j-1)\} + 1 = 1$.

Consider now the case of $b_{ij} = 0$ and $F(i-1, j) = F(i, j-1) = p > 0$, depicted below with b_{ij} shown in bold.



If $x = 0$, $F(i-1, j-1) \ge p$ and $\min\{F(i-1, j), F(i, j-1), F(i-1, j-1)\}+1 = p+1 = F(i, j)$. If $x = 1$, $F(i-1, j-1) = p-1$ and $\min\{F(i-1, j), F(i, j-1), F(i-1, j-1)\} + 1 = (p-1) + 1 = p = F(i, j)$.

Consider the case of $b_{ij} = 0$, $F(i-1, j) = p$ and $F(i, j-1) = q$, where $p \neq q$. Without loss of generality, we assume that $p < q$.



Then $F(i-1, j-1) \geq p$ and $\min\{F(i-1, j),\, F(i, j-1),\, F(i-1, j-1)\} + 1 = p + 1 = F(i, j)$.

Using the above recurrence, one can use it to fill the $m \times n$ table of $F(i, j)$ values row by row top to bottom and each row left to write and then find the largest value in this table. Here is an example.

$B =$

	1	2	3	4	5	6
1	0	0	1	0	0	0
2	1	1	0	0	0	0
3	0	0	1	0	0	0
4	1	0	0	1	0	0

$F =$

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	1	0	1	1	1
2	0	0	0	1	1	2	2
3	0	1	1	0	1	2	3
4	0	0	1	1	0	1	2

12. a. Let $P(i, j)$ be the probability of A winning the series if A needs i more games to win the series and B needs j more games to win the series. If team A wins the game, which happens with probability p, A will need $i - 1$ more wins to win the series while B will still need j wins. If team A looses the game, which happens with probability $q = 1 - p$, A will still need i wins while B will need $j - 1$ wins to win the series. This leads to the recurrence

$$P(i, j) = pP(i-1, j) + qP(i, j-1) \quad \text{for } i, j > 0.$$

The initial conditions follow immediately from the definition of $P(i, j)$:

$$P(0, j) = 1 \quad \text{for } j > 0, \quad P(i, 0) = 0 \quad \text{for } i > 0.$$

b. Here is the dynamic programming table in question, with its entries rounded-off to two decimal places. (It can be filled either row-by-row, or column-by-column, or diagonal-by-diagonal.)

$i \setminus j$	0	1	2	3	4
0		1	1	1	1
1	0	0.40	0.64	0.78	0.87
2	0	0.16	0.35	0.52	0.66
3	0	0.06	0.18	0.32	0.46
4	0	0.03	0.09	0.18	0.29

Thus, $P[4, 4] \approx 0.29$.

c. **Algorithm** *WorldSeries*(n, p)
//Computes the odds of winning a series of n games
//Input: A number of wins n needed to win the series
// and probability p of one particular team winning a game
//Output: The probability of this team winning the series
$q \leftarrow 1 - p$
for $j \leftarrow 1$ **to** n **do**
  $P[0, j] \leftarrow 1.0$
for $i \leftarrow 1$ **to** n **do**
  $P[i, 0] \leftarrow 0.0$
  for $j \leftarrow 1$ **to** n **do**
    $P[i, j] \leftarrow p * P[i - 1, j] + q * P[i, j - 1]$
return $P[n, n]$

Both the time efficiency and the space efficiency are in $\Theta(n^2)$ because each entry of the $n + 1$-by-$n + 1$ table (except $P[0, 0]$, which is not computed) is computed in $\Theta(1)$ time.

Solutions to Exercises 8.2

1. a.

		capacity j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$, $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$, $v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1$, $v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4$, $v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5$, $v_5 = 50$	5	0	15	20	35	40	55	65

The maximal value of a feasible subset is $F[5, 6] = 65$. The optimal subset is {item 3, item 5}.

b.-c. The instance has a unique optimal subset in view of the following general property: An instance of the knapsack problem has a unique optimal solution if and only if the algorithm for obtaining an optimal subset, which retraces backward the computation of $F[n, W]$, encounters no equality between $F[i-1, j]$ and $v_i + F[i-1, j-w_i]$ during its operation.

2. a. **Algorithm** *DPKnapsack*$(w[1..n], v[1..n], W)$
//Solves the knapsack problem by dynamic programming (bottom up)
//Input: Arrays $w[1..n]$ and $v[1..n]$ of weights and values of n items,
// knapsack capacity W
//Output: The table $F[0..n, 0..W]$ that contains the value of an optimal
// subset in $F[n, W]$ and from which the items of an optimal
// subset can be found
for $i \leftarrow 0$ **to** n **do** $F[i, 0] \leftarrow 0$
for $j \leftarrow 1$ **to** W **do** $F[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** W **do**
    if $j - w[i] \geq 0$
      $F[i, j] \leftarrow \max\{F[i-1, j],\ v[i] + F[i-1, j-w[i]]\}$
    else $F[i, j] \leftarrow F[i-1, j]$
return $F[0..n, 0..W]$

b. **Algorithm** *OptimalKnapsack*$(w[1..n], v[1..n], F[0..n, 0..W])$
//Finds the items composing an optimal solution to the knapsack problem
//Input: Arrays $w[1..n]$ and $v[1..n]$ of weights and values of n items,
// and table $F[0..n, 0..W]$ generated by
// the dynamic programming algorithm
//Output: List $L[1..k]$ of the items composing an optimal solution
$k \leftarrow 0$ //size of the list of items in an optimal solution
$j \leftarrow W$ //unused capacity

for $i \leftarrow n$ **downto** 1 **do**
    if $F[i, j] > F[i-1, j]$
        $k \leftarrow k + 1$; $L[k] \leftarrow i$ //include item i
        $j \leftarrow j - w[i]$
return L

Note: In fact, we can also stop the algorithm as soon as j, the unused capacity of the knapsack, becomes 0.

3. The algorithm fills a table with $n + 1$ rows and $W + 1$ columns, spending $\Theta(1)$ time to fill one cell (either by applying (8.6) or (8.7). Hence, its time efficiency and its space efficiency are in $\Theta(nW)$.

   In order to identify the composition of an optimal subset, the algorithm repeatedly compares values at no more than two cells in a previous row. Hence, its time efficiency class is in $O(n)$.

4. Both assertions are true:

   a. $F(i, j-1) \leq F(i, j)$ for $1 \leq j \leq W$ is true because it simply means that the maximal value of a subset that fits into a knapsack of capacity $j - 1$ cannot exceed the maximal value of a subset that fits into a knapsack of capacity j.

   b. $F(i-1, j) \leq F(i, j)$ for $1 \leq i \leq n$ is true because it simply means that the maximal value of a subset of the first $i - 1$ items that fits into a knapsack of capacity j cannot exceed the maximal value of a subset of the first i items that fits into a knapsack of the same capacity j.

5. Here, the recurrence underlying the dynamic-programming algorithm is

$$\begin{aligned} F(W) &= \max_{j:\, W \geq w_j} \{F(W - w_j)\} + v_j, \\ F(W) &= 0 \text{ if } W < w_j \text{ for all } 1 \leq j \leq n. \end{aligned}$$

   Using this recurrence, the algorithm can fill the one-row table table in the same way such a table is filled for the change-making problem discussed in Section 8.1.

6. In the table below, the cells marked by a minus are the ones for which no entry is computed for the instance in question; the only nontrivial entry that is retrieved without recomputation is $(2, 1)$.

		capacity j						
	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$, $v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2$, $v_2 = 20$	2	0	0	20	-	-	45	45
$w_3 = 1$, $v_3 = 15$	3	0	15	20	-	-	-	60
$w_4 = 4$, $v_4 = 40$	4	0	15	-	-	-	-	60
$w_5 = 5$, $v_5 = 50$	5	0	-	-	-	-	-	65

7. Since some of the cells of a table with $n + 1$ rows and $W + 1$ columns are filled in constant time, both the time and space efficiencies are in $O(nW)$. But all the entries of the table need to be initialized (unless virtual initialization of Problem 7 in Exercises 7.1 is used); this puts them in $\Omega(nW)$. Hence, both efficiencies are in $\Theta(nW)$.

8. For the problem of computing a binomial coefficient, we know in advance which cells of the table need to be computed. Therefore unnecessary computations can be avoided by the bottom-up dynamic programming algorithm as well. Also, using the memory function method requires $\Theta(nk)$ space, whereas the bottom-up algorithm needs only $\Theta(n)$ because the next row of the table can be written over its immediate predecessor.

9. n/a

Solutions to Exercises 8.3

1. The instance of the problem in question is defined by the data

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The table entries for the dynamic programming algorithm are computed as follows:

$$C[1,2] = \min \begin{array}{ll} k=1: & C[1,0]+C[2,2]+\sum_{s=1}^{2} p_s = 0+0.2+0.3 = 0.5 \\ k=2: & C[1,1]+C[3,2]+\sum_{s=1}^{2} p_s = 0.1+0+0.3 = \mathbf{0.4} \end{array} = 0.4$$

$$C[2,3] = \min \begin{array}{ll} k=2: & C[2,1]+C[3,3]+\sum_{s=2}^{3} p_s = 0+0.4+0.6 = 1.0 \\ k=3: & C[2,2]+C[4,3]+\sum_{s=2}^{3} p_s = 0.2+0+0.6 = \mathbf{0.8} \end{array} = 0.8$$

$$C[3,4] = \min \begin{array}{ll} k=3: & C[3,2]+C[4,4]+\sum_{s=3}^{4} p_s = 0+0.3+0.7 = \mathbf{1.0} \\ k=4: & C[3,3]+C[5,4]+\sum_{s=3}^{4} p_s = 0.4+0+0.7 = 1.1 \end{array} = 1.0$$

$$C[1,3] = \min \begin{array}{ll} k=1: & C[1,0]+C[2,3]+\sum_{s=1}^{3} p_s = 0+0.8+0.7 = 1.5 \\ k=2: & C[1,1]+C[3,3]+\sum_{s=1}^{3} p_s = 0.1+0.4+0.7 = 1.2 \\ k=3: & C[1,2]+C[4,3]+\sum_{s=1}^{3} p_s = 0.4+0+0.7 = \mathbf{1.1} \end{array} = 1.1$$

$$C[2,4] = \min \begin{array}{ll} k=2: & C[2,1]+C[3,4]+\sum_{s=2}^{4} p_s = 0+1.0+0.9 = 1.9 \\ k=3: & C[2,2]+C[4,4]+\sum_{s=2}^{4} p_s = 0.2+0.3+0.9 = \mathbf{1.4} \\ k=4: & C[2,3]+C[5,4]+\sum_{s=2}^{4} p_s = 0.8+0+0.9 = 1.7 \end{array} = 1.1$$

$$C[1,4] = \min \begin{array}{ll} k=1: & C[1,0]+C[2,4]+\sum_{s=1}^{s=4} p_s = 0+1.4+1.0 = 2.4 \\ k=2: & C[1,1]+C[3,4]+\sum_{s=1}^{4} p_s = 0.1+1.0+1.0 = 2.1 \\ k=3: & C[1,2]+C[4,4]+\sum_{s=1}^{4} p_s = 0.4+0.3+1.0 = \mathbf{1.7} \\ k=4: & C[1,3]+C[5,4]+\sum_{s=1}^{4} p_s = 1.1+0+1.0 = 2.1 \end{array} = 1.7$$

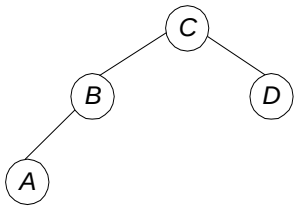
the main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

the root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

the optimal tree



2. a. The number of times the innermost loop is executed is given by the sum

$$\begin{aligned}
\sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=i}^{i+d}1 &= \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}(i+d-i+1) = \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}(d+1) \\
&= \sum_{d=1}^{n-1}(d+1)(n-d) = \sum_{d=1}^{n-1}(dn+n-d^2-d) \\
&= \sum_{d=1}^{n-1}nd + \sum_{d=1}^{n-1}n - \sum_{d=1}^{n-1}d^2 - \sum_{d=1}^{n-1}d \\
&= n\frac{(n-1)n}{2} + n(n-1) - \frac{(n-1)n(2n-1)}{6} - \frac{(n-1)n}{2} \\
&= \frac{1}{2}n^3 - \frac{2}{6}n^3 + O(n^2) \in \Theta(n^3).
\end{aligned}$$

b. The algorithm generates the $(n+1)$-by-$(n+1)$ table C and the n-by-n table R and fills about one half of each. Hence, the algorithm's space efficiency is in $\Theta(n^2)$.

3. Call *OptimalTree*$(1, n)$ below:

Algorithm *OptimalTree*(i, j)
//Input: Indices i and j of the first and last keys of a sorted list of keys
//composing the tree and table $R[1..n, 1..n]$ obtained by dynamic
//programming
//Output: Indices of nodes of an optimal binary search tree in preorder
if $i \le j$
    $k \leftarrow R[i, j]$
    print(k)
    OptimalTree$(i, k-1)$
    OptimalTree$(k+1, j)$

4. Precompute $S_k = \sum_{s=1}^{k} p_s$ for $k = 1, 2, ..., n$ and set $S_0 = 0$. Then $\sum_{s=i}^{j} p_s$ can be found as $S_j - S_{i-1}$ for any $1 \le i \le j \le n$.

5. False. Here is a simple counterexample: $A(0.3)$, $B(0.3)$, $C(0.4)$. (The numbers in the parentheses indicate the search probabilities.) The average number of comparisons in a binary search tree with C in its root is 0.3·2 + 0.3·3 + 0.4·1 = 1.9, while the average number of comparisons in the binary search tree with B in its root is 0.3·1 + 0.3·2 + 0.4·2 = 1.7.

6. The binary search tree in question should have a maximal number of nodes on each of its levels except the last one. (For simplicity, we can put all the nodes of the last level in their leftmost positions possible to make it complete.) The keys of a given sorted list can be distributed among the nodes of the binary tree by performing its in-order traversal.

   Let p/n be the probability of searching for each key, where $0 \le p \le 1$. The complete binary tree with 2^k nodes will have 2^i nodes on level i for $i = 0, ..., k-1$ and one node on level k. Hence, the average number of comparisons in a successful search will be given by the following formula:

$$\begin{aligned} C(2^k) &= \sum_{i=0}^{k-1} (p/2^k)(i+1)2^i + (p/2^k)(k+1) \\ &= (p/2^k)\frac{1}{2}\sum_{i=0}^{k-1}(i+1)2^{i+1} + (p/2^k)(k+1) \\ &= (p/2^k)\frac{1}{2}\sum_{j=1}^{k} j2^j + (p/2^k)(k+1) \\ &= (p/2^k)\frac{1}{2}[(k-1)2^{k+1}+2] + (p/2^k)(k+1) \\ &= (p/2^k)[(k-1)2^k + k + 2]. \end{aligned}$$

7. a. Let $b(n)$ be the number of distinct binary trees with n nodes. If the left subtree of a binary tree with n nodes has k nodes ($0 \le k \le n-1$), the right subtree must have $n-1-k$ nodes. The number of such trees is therefore $b(k)b(n-1-k)$. Hence,

$$b(n) = \sum_{k=0}^{n-1} b(k)b(n-1-k) \quad \text{for } n > 0, \quad b(0) = 1.$$

   b. Substituting the first five values of n into the formula above and into the formula for the nth Catalan number yields the following values:

n	0	1	2	3	4	5
$b(n) = c(n)$	1	1	2	5	14	42

c.

$$\begin{aligned} b(n) &= c(n) = \binom{2n}{n}\frac{1}{n+1} = \frac{(2n)!}{(n!)^2}\frac{1}{n+1} \\ &\approx \frac{\sqrt{2\pi 2n}(2n/e)^{2n}}{[\sqrt{2\pi n}(n/e)^n]^2}\frac{1}{n+1} = \frac{\sqrt{4\pi n}(2n/e)^{2n}}{2\pi n(n/e)^{2n}}\frac{1}{n+1} \\ &\approx \frac{1}{\sqrt{\pi n}}\left(\frac{2n/e}{n/e}\right)^{2n}\frac{1}{n+1} = \frac{1}{\sqrt{\pi n}}2^{2n}\frac{1}{n+1} \in \Theta(4^n n^{-3/2}). \end{aligned}$$

This implies that finding an optimal binary search tree by exhaustive search is feasible only for very small values of n and is, in general, vastly inferior to the dynamic programming algorithm.

8. The dynamic programming algorithm finds the root a_{kmin} of an optimal binary search tree for keys $a_i, ..., a_j$ by minimizing $\{C[i,k-1]+C[k+1,j]\}$ for $i \le k \le j$. As pointed out at the end of Section 8.3 (see also [KnuIII], p. 456, Exercise 27), $R[i,j-1] \le kmin \le R[i+1,j]$. This observation allows us to change the bounds of the algorithm's innermost loop to the lower bound of $R[i,j-1]$ and the upper bound of $R[i+1,j]$, respectively. The number of times the innermost loop of the modified algorithm will be executed can be estimated as suggested by Knuth (see [KnuIII], p.439):

$$\begin{aligned} \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=R[i,j-1]}^{R[i+1,j]} 1 &= \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=R[i,i+d-1]}^{R[i+1,i+d]} 1 \\ &= \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}(R[i+1,i+d]-R[i,i+d-1]+1) \\ &= \sum_{d=1}^{n-1}(\sum_{i=1}^{n-d}(R[i+1,i+d]-R[i,i+d-1])+\sum_{i=1}^{n-d}1). \end{aligned}$$

By "telescoping" the first sum, we can see that all its terms except the two get cancelled to yield

$\sum_{i=1}^{n-d}(R[i+1,i+d]-R[i,i+d-1]) =$
$= (R[2,1+d]-R[1,1+d-1])$
$+(R[3,2+d]-R[2,2+d-1])$
$+ \ldots$
$+(R[n-d+1,n-d+d]-R[n-d,n-d+d-1])$
$= R[n-d+1,n]-R[1,d].$

Since the second sum $\sum_{i=1}^{n-d}1$ is equal to $n-d$, we obtain

$$\sum_{i=1}^{n-d}(R[i+1,i+d]-R[i,i+d-1])+\sum_{i=1}^{n-d}1 = R[n-d+1,n]-R[1,d]+n-d < 2n.$$

Hence,

$$\sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=R[i,j-1]}^{R[i+1,j]} 1 = \sum_{d=1}^{n-1}(R[n-d+1,n]-R[1,d]+n-d) < \sum_{d=1}^{n-1} 2n < 2n^2.$$

On the other hand, since $R[i+1,i+d]-R[i,i+d-1] \geq 0$,

$$\begin{aligned}\sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=R[i,j-1]}^{R[i+1,j]} 1 &= \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}(R[i+1,i+d]-R[i,i+d-1]+1)\\ &\geq \sum_{d=1}^{n-1}\sum_{i=1}^{n-d} 1 = \sum_{d=1}^{n-1}(n-d) = \frac{(n-1)n}{2} \geq \frac{1}{4}n^2 \text{ for } n \geq 2.\end{aligned}$$

Therefore, the time efficiency of the modified algorithm is in $\Theta(n^2)$.

9. Let $a_1, ...a_n$ be a sorted list of n distinct keys, p_i be a known probability of searching for key a_i for $i = 1, 2, ..., n$, and q_i be a known probability of searching (unsuccessfully) for a key between a_i and a_{i+1} for $i = 0, 1, ..., n$ (with q_0 being a probability of searching for a key smaller than a_1 and q_n being a probability of searching for a key greater than a_n). It's convenient to associate unsuccessful searches with external nodes of a binary search tree (see Section 5.3). Repeating the derivation of equation (8.11) for such a tree yields the following recurrence for the expected number of key comparisons

$$C[i,j] = \min_{i \leq k \leq j}\{C[i,k-1]+C[k+1,j]\}+\sum_{s=i}^{j} p_s + \sum_{s=i-1}^{j} q_s \quad \text{for } 1 \leq i < j \leq n$$

with the initial condition

$$C[i,i] = p_i + q_{i-1} + q_i \quad \text{for } i = 1, ..., n.$$

In all other respects, the algorithm remains the same.

10. **Algorithm** *MFOptimalBST*(i, j)
//Returns the number of comparisons in a successful search in a *BST*
//Input: Indices i, j indicating the range of keys in a sorted list of n keys
//and an array $P[1..n]$ of search probabilities used as a global variable
//Uses a global table $C[1..n+1, 0..n]$ initialized with a negative number
//Output: The average number of comparisons in the optimal BST
if $j = i - 1$ **return** 0
if $j = i$ **return** $P[i]$
if $C[i,j] < 0$
    $minval \leftarrow \infty$

for $k \leftarrow i$ **to** j **do**
$minval \leftarrow \min\{MFOptimalBST(i, k-1) + MFOptimalBST(k+1, j), minval\}$
$sum \leftarrow 0$; **for** $s \leftarrow i$ **to** j **do** $sum \leftarrow sum + P[s]$
$C[i, j] \leftarrow minval + sum$
return $C[i, j]$

Note: The first two lines of this pseudocode can be eliminated by an appropriate initialization of table C.

11. a. Multiplying two matrices of dimensions $\alpha \times \beta$ and $\beta \times \gamma$ by the definition-based algorithm requires $\alpha\beta\gamma$ multiplications. (There are $\alpha\gamma$ elements in the product, each requiring β multiplications to be computed.) If the dimensions of A_1, A_2, and A_3 are $d_0 \times d_1$, $d_1 \times d_2$, and $d_2 \times d_3$, respectively, then $(A_1 \cdot A_2) \cdot A_3$ will require

$$d_0 d_1 d_2 + d_0 d_2 d_3 = d_0 d_2 (d_1 + d_3)$$

multiplications, while $A_1 \cdot (A_2 \cdot A_3)$ will need

$$d_1 d_2 d_3 + d_0 d_1 d_3 = d_1 d_3 (d_0 + d_2)$$

multiplications. Here is a simple choice of specific values to make, say, the first of them be 1000 times larger than the second:

$$d_0 = d_2 = 10^3, \quad d_1 = d_3 = 1.$$

b. Let $m(n)$ be the number of different ways to compute a chain product of n matrices $A_1 \cdot ... \cdot A_n$. Any parenthesization of the chain will lead to multiplying, as the last operation, some product of the first k matrices $(A_1 \cdot ... \cdot A_k)$ and the last $n-k$ matrices $(A_{k+1} \cdot ... \cdot A_n)$. There are $m(k)$ ways to do the former, and there are $m(n-k)$ ways to do the latter. Hence, we have the following recurrence for the total number of ways to parenthesize the matrix chain of n matrices:

$$m(n) = \sum_{k=1}^{n-1} m(k)m(n-k) \quad \text{for } n > 1, \quad m(1) = 1.$$

Since parenthesizing a chain of n matrices for multiplication is very similar to constructing a binary tree of n nodes, it should come as no surprise that the above recurrence is very similar to the recurrence

$$b(n) = \sum_{k=0}^{n-1} b(k)b(n-1-k) \quad \text{for } n > 1, \quad b(0) = 1,$$

for the number of binary trees mentioned in Section 8.3. Nor is it surprising that their solutions are very similar, too: namely,

$$m(n) = b(n-1) \quad \text{for } n \geq 1,$$

where $b(n)$ is the number of binary trees with n nodes. Let us prove this assertion by mathematical induction. The basis checks immediately: $m(1) = b(0) = 1$. For the general case, let us assume that $m(k) = b(k-1)$ for all positive integers not exceeding some positive integer n (we're using the strong version of mathematical induction); we'll show that the equality holds for $n+1$ as well. Indeed,

$$\begin{aligned}
m(n+1) &= \sum_{k=1}^{n} m(k)m(n+1-k) \\
&= [\text{using the induction's assumption}] \sum_{k=1}^{n} b(k-1)b(n-k) \\
&= [\text{substituting } l = k-1] \qquad \sum_{l=0}^{n-1} b(l)b(n-1-l) \\
&= [\text{see the recurrence for } b(n)] \qquad b(n).
\end{aligned}$$

c. Let $M[i,j]$ be the optimal (smallest) number of multiplications needed for computing $A_i \cdot \ldots \cdot A_j$. If k is an index of the last matrix in the first factor of the last matrix product, then

$$\begin{aligned}
M[i,j] &= \max_{1 \leq k \leq j-1} \{M[i,k] + M[k+1,j] + d_{i-1}d_k d_j\} \quad \text{for } 1 \leq i < j \leq n, \\
M[i,i] &= 0.
\end{aligned}$$

This recurrence, which is quite similar to the one for the optimal binary search tree problem, suggests filling the $n+1$-by-$n+1$ table diagonal by diagonal as in the following algorithm:

Algorithm *MatrixChainMultiplication*$(D[0..n])$
//Solves matrix chain multiplication problem by dynamic programming
//Input: An array $D[0..n]$ of dimensions of n matrices
//Output: The minimum number of multiplications needed to multiply
//a chain of n matrices of the given dimensions and table $T[1..n, 1..n]$
//for obtaining an optimal order of the multiplications
for $i \leftarrow 1$ **to** n **do** $M[i,i] \leftarrow 0$
for $d \leftarrow 1$ **to** $n-1$ **do** //diagonal count
    for $i \leftarrow 1$ **to** $n-d$ **do**
        $j \leftarrow i + d$
        minval $\leftarrow \infty$
        for $k \leftarrow i$ **to** $j-1$ **do**

$temp \leftarrow M[i,k] + M[k+1,j] + D[i-1] * D[k] * D[j]$
if $temp < minval$
$minval \leftarrow temp$
$kmin \leftarrow k$
$T[i,j] \leftarrow kmin$
return $M[1,n]$, T

To find an optimal order to multiply the matrix chain, call *OptimalMultiplicationOrder*$(1,n)$ below:
Algorithm *OptimalOrder*(i,j)
//Outputs an optimal order to multiply n matrices
//Input: Indices i and j of the first and last matrices in $A_i...A_j$ and
//table $T[1..n,1..n]$ generated by *MatrixChainMultiplication*
//Output: $A_i...A_j$ parenthesized for optimal multiplication
if $i = j$
print("A_i")
else
$k \leftarrow T[i,j]$
print("(")
OptimalOrder(i,k)
OptimalOrder$(k+1,j)$
print(")")

Solutions to Exercises 8.4

1. Applying Warshall's algorithm yields the following sequence of matrices (in which newly updated elements are shown in bold):

$$R^{(0)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad R^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^{(2)} = \begin{bmatrix} 0 & 1 & \mathbf{1} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad R^{(3)} = \begin{bmatrix} 0 & 1 & 1 & \mathbf{1} \\ 0 & 0 & 1 & \mathbf{1} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^{(4)} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = T$$

2. a. For a graph with n vertices, the algorithm computes n matrices $R^{(k)}$ $(k = 1, 2, ..., n)$, each of which has n^2 elements. Hence, the total number of elements to be computed is n^3. Since computing each element takes constant time, the time efficiency of the algorithm is in $\Theta(n^3)$.

   b. Since one *DFS* or *BFS* traversal of a graph with n vertices and m edges, which is represented by its adjacency lists, takes $\Theta(n + m)$ time, doing this n times takes $n\Theta(n+m) = \Theta(n^2+nm)$ time. For sparse graphs (i.e., if $m \in O(n)$), $\Theta(n^2 + nm) = \Theta(n^2)$, which is more efficient than the $\Theta(n^3)$ time efficiency of Warshall's algorithm.

3. The algorithm computes the new value to be put in the ith row and the jth column by the formula

$$R^{(k)}[i, j] = R^{(k-1)}[i, j] \textbf{ or } (R^{(k-1)}[i, k] \textbf{ and } R^{(k-1)}[k, j]).$$

   The formula implies that all the elements in the kth row and all the elements in the kth column never change on the kth iteration, i.e., while computing elements of $R^{(k)}$ from elements of $R^{(k-1)}$. (Substitute k for i and k for j, respectively, into the formula to verify these assertions.) Hence, the new value of the element in the ith row and the jth column, $R^{(k)}[i, j]$, can be written over its old value $R^{(k-1)}[i, j]$ for every i, j.

4. If $R^{(k-1)}[i,k] = 0$,

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \textbf{ or } (R^{(k-1)}[i,k] \textbf{ and } R^{(k-1)}[k,j]) = R^{(k-1)}[i,j],$$

and hence the innermost loop need not be executed. And, since $R^{(k-1)}[i,k]$ doesn't depend on j, its comparison with 0 can be done outside the j loop. This leads to the following implementation of Warshall's algorithm:

Algorithm *Warshall2*$(A[1..n, 1..n])$
//Implements Warshall's algorithm with a more efficient innermost loop
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph in place of A
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        if $A[i,k]$
            for $j \leftarrow 1$ **to** n **do**
                if $A[k,j]$
                    $A[i,j] \leftarrow 1$
return A

5. First, it is easy to check that

$$r_{ij} \leftarrow r_{ij} \quad \textbf{or} \quad (r_{ik} \textbf{ and } r_{kj}).$$

is equivalent to

$$\textbf{if } r_{ik} \quad r_{ij} \leftarrow r_{ij} \textbf{ or } r_{kj}$$

Indeed, if $r_{ik} = 1$ (i.e., **true**),

$$r_{ij} \textbf{ or } (r_{ik} \textbf{ and } r_{kj}) = r_{ij} \textbf{ or } r_{kj},$$

which is exactly the value assigned to r_{ij} by the **if** statement as well. If $r_{ik} = 0$ (i.e., **false**),

$$r_{ij} \textbf{ or } (r_{ik} \textbf{ and } r_{kj}) = r_{ij},$$

i.e., the new value of r_{ij} will be the same as its previous value—exactly, the result we obtain by executing the **if** statement.

Here is the algorithm that exploits this observation and the bitwise *or* operation applied to matrix rows:

Algorithm *Warshall3*$(A[1..n, 1..n])$
//Implements Warshall's algorithm for computing the transitive closure
//with the bitwise *or* operation on matrix rows
//Input: The adjacency matrix A of a digraph
//Output: The transitive closure of the digraph in place of A
for $k \leftarrow 1$ **to** n **do**

```
    for i ← 1 to n do
        if A[i, k]
            row[i] ← row[i] bitwiseor row[k]  //rows of matrix A
return A
```

6. a. With the book's definition of the transitive closure (which considers only nontrivial paths of a digraph), a digraph has a directed cycle if and only if its transitive closure has a 1 on its main diagonal. The algorithm that finds the transitive closure by applying Warshall's algorithm and then checks the elements of its main diagonal is cubic. This is inferior to the quadratic algorithms for checking whether a digraph represented by its adjacency matrix is a dag, which were discussed in Section 4.2.

   b. No. If T is the transitive closure of an undirected graph, $T[i, j] = 1$ if and only if there is a nontrivial path from the ith vertex to the jth vertex. If $i \neq j$, this is the case if and only if the ith vertex and the jth vertex belong to the same connected component of the graph. Thus, one can find the elements outside the main diagonal of the transitive closure that are equal to 1 by a depth-first search or a breadth-first search traversal, which is faster than applying Warshall's algorithm. If $i = j$, $T[i, i] = 1$ if and only if the ith vertex is not isolated, i.e., if it has an edge to at least one other vertex of the graph. Isolated vertices, if any, can be easily identified by the graph's traversal as one-node connected components of the graph.

7. Applying Floyd's algorithm to the given weight matrix generates the following sequence of matrices:

$$D^{(0)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix} \qquad D^{(1)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \mathbf{14} \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \mathbf{5} & \infty & \mathbf{4} & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 2 & \mathbf{5} & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & \mathbf{8} & 4 & 0 \end{bmatrix} \qquad D^{(3)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 2 & \mathbf{3} & 1 & \mathbf{4} \\ 6 & 0 & 3 & 2 & \mathbf{5} \\ \infty & \infty & 0 & 4 & \mathbf{7} \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & \mathbf{6} & 4 & 0 \end{bmatrix} \qquad D^{(5)} = \begin{bmatrix} 0 & 2 & 3 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ \mathbf{10} & \mathbf{12} & 0 & 4 & 7 \\ \mathbf{6} & \mathbf{8} & 2 & 0 & 3 \\ 3 & 5 & 6 & 4 & 0 \end{bmatrix} = D$$

8. The formula
$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$
implies that the value of the element in the ith row and the jth column of matrix $D^{(k)}$ is computed from its own value and the values of the elements in the ith row and the kth column and in the kth row and the jth column in the preceding matrix $D^{(k-1)}$. The latter two cannot change their values when the elements of $D^{(k)}$ are computed because of the formulas

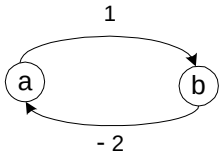
$$d_{ik}^{(k)} = \min\{d_{ik}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kk}^{(k-1)}\} = \min\{d_{ik}^{(k-1)},\ d_{ik}^{(k-1)} + 0\} = d_{ik}^{(k-1)}$$

and

$$d_{kj}^{(k)} = \min\{d_{kj}^{(k-1)},\ d_{kk}^{(k-1)} + d_{kj}^{(k-1)}\} = \min\{d_{kj}^{(k-1)},\ 0 + d_{kj}^{(k-1)}\} = d_{kj}^{(k-1)}.$$

(Note that we took advantage of the fact that the elements d_{kk} on the main diagonal remain 0's. This can be guaranteed only if the graph doesn't contain a cycle of a negative length.)

9. As a simple counterexample, one can suggest the following digraph:



Floyd's algorithm will yield:

$$D^{(0)} = \begin{bmatrix} 0 & 1 \\ -2 & 0 \end{bmatrix} \qquad D^{(1)} = \begin{bmatrix} 0 & 1 \\ -2 & -1 \end{bmatrix} \qquad D^{(2)} = \begin{bmatrix} -1 & 0 \\ -3 & -2 \end{bmatrix}$$

None of the four elements of the last matrix gives the correct value of the shortest path, which is, in fact, $-\infty$ because repeating the cycle enough times makes the length of a path arbitrarily small.

Note: Floyd's algorithm can be used for detecting negative-length cycles, but the algorithm should be stopped as soon as it generates a matrix with a negative element on its main diagonal.

10. As pointed out in the hint to this problem, Floyd's algorithm should be enhanced by recording in an n-by-n matrix index k of an intermediate vertex causing an update of the distance matrix. This is implemented in the pseudocode below:

Algorithm *FloydEnhanced*$(W[1..n, 1..n])$

```
//Input: The weight matrix W of a graph or a digraph
//Output: The distance matrix D[1..n, 1..n] and
//          the matrix of intermediate updates P[1..n, 1..n]
D ← W
for i ← 1 to n do
    for j ← 1 to n do
        P[i, j] ← 0 //initial mark
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            if D[i, k] + D[k, j] < D[i, j]
               D[i, j] ← D[i, k] + D[k, j]}
               P[i, j] ← k
```

For example, for the digraph in Fig. 8.7 whose vertices are numbered from 1 to 4, matrix P will be as follows:

$$P = \begin{bmatrix} 0 & 3 & 0 & 3 \\ 0 & 0 & 1 & 3 \\ 4 & 0 & 0 & 0 \\ 0 & 3 & 1 & 0 \end{bmatrix}.$$

The list of intermediate vertices on the shortest path from vertex i to vertex j can be then generated by the call to the following recursive algorithm, provided $D[i, j] < \infty$:

```
Algorithm ShortestPath(i, j, P[1..n, 1..n])
//The algorithm prints out the list of intermediate vertices of
//a shortest path from the ith vertex to the jth vertex
//Input: Endpoints i and j of the shortest path desired;
//         matrix P of updates generated by Floyd's algorithm
//Output: The list of intermediate vertices of the shortest path
//          from the ith vertex to the jth vertex
k ← P[i, j]
if k ≠ 0
    ShortestPath(i, k)
    print(k)
    ShortestPath(k, j)
```

11. First, for each pair of the straws, determine whether the straws intersect. (Although this can be done in $n \log n$ time by a sophisticated algorithm, the quadratic brute-force algorithm would do because of the quadratic efficiency of the subsequent step; both geometric algorithms can be found, e.g., in R. Sedgewick's "Algorithms," Addison-Wesley, 1988.) Record the obtained information in a boolean $n \times n$ matrix, which must be symmetric. Then find the transitive closure of this matrix in n^2 time by DFS or BFS (see the solution to Problem 6b in these exercises).

Solutions to Exercises 9.1

1. **Algorithm** *Change*$(n, D[1..m])$
   //Implements the greedy algorithm for the change-making problem
   //Input: A nonnegative integer amount n and
   // a decreasing array of coin denominations D
   //Output: Array $C[1..m]$ of the number of coins of each denomination
   // in the change or the "no solution" message
   for $i \leftarrow 1$ **to** m **do**
   $C[i] \leftarrow \lfloor n/D[i] \rfloor$
   $n \leftarrow n \bmod D[i]$
   if $n = 0$ **return** C
   else return "no solution"

   The algorithm's time efficiency is in $\Theta(m)$. (We assume that integer divisions take a constant time no matter how big dividends are.) Note also that if we stop the algorithm as soon as the remaining amount becomes 0, the time efficiency will be in $O(m)$.

2. a. The row-by-row version: Starting with the first row and ending with the last row of the cost matrix, select the smallest element in that row which is not in a previously marked column. After such an element is selected, mark its column to prevent selecting another element from the same column.

   The all-matrix version: Repeat the following operation n times. Select the smallest element in the unmarked rows and columns of the cost matrix and then mark its row and column.

   b. Neither of the versions always yields an optimal solution. Here is a simple counterexample:

$$C = \begin{bmatrix} 1 & 2 \\ 2 & 100 \end{bmatrix}$$

3. a. Sort the intervals in nondecreasing order of their execution time and execute them in that order.

   b. Yes, this greedy algorithm always yields an optimal solution.
   Indeed, for any ordering (i.e., permutation) of the jobs i_1, i_2, ..., i_n, the total time in the system is given by the formula

$$t_{i_1} + (t_{i_1} + t_{i_2}) + ... + (t_{i_1} + t_{i_2} + \cdots + t_{i_n}) = nt_{i_1} + (n-1)t_{i_2} + \cdots + t_{i_n}.$$

Thus, we have a sum of numbers n, $n-1$,...,1 multiplied by "weights" t_1, t_2, ...t_n assigned to the numbers in some order. To minimize such a sum, we have to assign smaller t's to larger numbers. In other words, the jobs should be executed in nondecreasing order of their execution time.

Here is a more formal proof of this fact. We will show that if jobs are executed in some order i_1, i_2, ..., i_n, in which $t_{i_k} > t_{i_{k+1}}$ for some k, then the total time in the system for such an ordering can be decreased. (Hence, no such ordering can be an optimal solution.) Let us consider the other job ordering, which is obtained by swapping the jobs k and $k+1$. Obviously, the time in the systems will remain the same for all but these two jobs. Therefore, the difference between the total time in the system for the new ordering and the one before the swap will be

$$\begin{aligned} & [(\sum_{j=1}^{k-1} t_{i_j} + t_{i_{k+1}}) + (\sum_{j=1}^{k-1} t_{i_j} + t_{i_{k+1}} + t_{i_k})] - [(\sum_{j=1}^{k-1} t_{i_j} + t_{i_k}) + (\sum_{j=1}^{k-1} t_{i_j} + t_{i_k} + t_{i_{k+1}})] \\ = \quad & t_{i_{k+1}} - t_{i_k} < 0. \end{aligned}$$

4. a. The greedy algorithm based on earliest start first doesn't always yield an optimal solution as the following counterexample shows: (0, 5), (1, 2), (3, 4). Selecting (0, 5) first makes selection of the other two intervals impossible, whereas the optimal solution comprises both of them.

   b. The greedy algorithm based on shortest duration first doesn't always yield an optimal solution as the following counterexample shows: (0, 4), (3, 6), (5, 9). Selecting (3, 6) first makes selection of the other two intervals impossible, whereas the optimal solution comprises both of them.

   c. The greedy algorithm based on earliest finish first does always yield an optimal solution. Let $I_1 = (a_{i_1}, b_{i_1}), ..., I_k = (a_{i_k}, b_{i_k})$ be k intervals composing such a solution,.listed in the order they are added by the algorithm, i.e., $b_{i_1} < \cdots < b_{i_k}$ and hence, since the intervals don't intersect, $a_{i_1} < \cdots < a_{i_k}$. Let $J_1 = (a_{j_1}, b_{j_1}), ..., J_m = (a_{j_m}, b_{j_m})$ be an optimal solution where $b_{j_1} < \cdots < b_{j_m}$ and hence $a_{j_1} < \cdots < a_{j_m}$. We need to show that $k = m$.
   First, we'll prove by induction that the earliest-finish-first algorithm does at least as well as the optimal after each of its iterations, i.e., $b_{i_r} \le b_{j_r}$ for every $r = 1, ..., k$. For $r = 1$, this assertion follows immediately from the definition of the earliest-finish-first algorithm. Now let $r > 1$; we assume that $b_{i_{r-1}} \le b_{j_{r-1}}$ to prove that $b_{i_r} \le b_{j_r}$. Since $b_{j_{r-1}} \le a_{j_r}$ and, by the inductive assumption, $b_{i_{r-1}} \le b_{j_{r-1}}$, we have $b_{i_{r-1}} \le a_{j_r}$. Thus interval J_r is available for selection by the earliest-finish-first algorithm on its rth iteration, which implies that $b_{i_r} \le b_{j_r}$.

Now we'll complete the proof by showing by contradiction that $k = m$. If $m > k$, we can use the proved above inequality for $r = k$ to get $b_{i_k} \leq b_{j_k}$ Since $m > k$, there is an interval $J_{k+1} = (a_{j_{k+1}}, b_{j_{k+1}})$ such that $a_{j_{k+1}} \geq b_{j_k} \geq b_{i_k}$. Hence J_{k+1} would have to be used by the earliest-finish-first algorithm in addition to $I_1, ..., I_k$.
Note: The proof follows the one given by Kleinberg and Tardosh in Sec. 4.1 of their book [Kle06].

5. Repeat the following step $n-2$ times: Send to the other side the pair of two fastest remaining persons and then return the flashlight with the fastest person. Finally, send the remaining two people together. Assuming that $t_1 \leq t_2 \leq ... \leq t_n$, the total crossing time will be equal to

$$(t_2+t_1)+(t_3+t_1)+\cdots+(t_{n-1}+t_1)+t_n = \sum_{i=2}^{n} t_i+(n-2)t_1 = \sum_{i=1}^{n} t_i+(n-3)t_1.$$

Note: For an algorithm that always yields a minimal crossing time, see Günter Rote, "Crossing the Bridge at Night," *EATCS Bulletin,* vol. 78 (October 2002), 241–246.

The solution to the instance of Problem 2 in Exercises 1.2 shows that the greedy algorithm doesn't always yield the minimal crossing time for $n > 3$. No smaller counterexample can be given as a simple exhaustive check for $n = 3$ demonstrates. (The obvious solution for $n = 2$ is the one generated by the greedy algorithm as well.)

6. Averaging the amount of water in the nonempty vessel successively with each of the $n - 1$ empty vessels, leaves $W/2^{n-1}$ pints in it. This is the minimum amount of water achievable for that vessel. Indeed, consider m, the minimum positive amount of water among all the vessels in their current state. (Initially, $m = W$ and our goal is to minimize it.) Since the average of two numbers is always greater than or equal to the smaller of the two, the value of m can be decreased by the averaging operation only if this operation involves a vessel containing m pints and an empty vessel. After repeating the averaging operation with each empty vessel, no empty vessel will remain making an increase in m impossible. Hence, the ultimate minimal value of m we can get here is equal to $W/2^{n-1}$ pints.

Note: The puzzle was included, in a different wording, in the exercises to the article on monovariants in the Russian magazine *Kvant* (no. 7, 1989, 63–68).

7. There are several ways to accomplish the task by sending $2n-2$ messages, which is the minimum. In particular, this can be done by the following greedy algorithm that seeks to increase as much as possible the total number of known rumors after each message sent. Number the persons from 1

to n and send the first $n-1$ messages as follows: from 1 to 2, from 2 to 3, and so on until the message combining the rumors initially known to persons 1, 2, ..., $n-1$ is sent to person n. Then send the message combining all the n rumors from person n to persons 1, 2, ..., $n-1$.

The fact that $2n-2$ messages is the smallest number needed to solve the puzzle stems from the fact that an increase of the number of persons by one requires at least two extra messages: to and from the extra person—exactly what the above algorithms provides.

8. a. Let's apply the greedy approach to the first few instances of the problem in question. For $n=1$, we have to use $w_1=1$ to balance weight 1. For $n=2$, we simply add $w_2=2$ to balance the first previously unattainable weight of 2. The weights $\{1,2\}$ can balance every integral weights up to their sum 3. For $n=3$, in the spirit of greedy thinking, we take the next previously unattainable weight: $w_3=4$. The three weights $\{1,2,4\}$ allow to weigh any integral load l between 1 and their sum 7, with l's binary expansion indicating the weights needed for load l:

load l	1	2	3	4	5	6	7
l's binary expansion	1	10	11	100	101	110	111
weights for load l	1	2	2+1	4	4+1	4+2	4+2+1

Generalizing these observations, we should hypothesize that for any positive integer n the set of consecutive powers of 2 $\{w_i = 2^{i-1}\colon\ i=1,2,...n\}$ makes it possible to balance every integral load in the largest possible range, which is up to and including $\sum_{i=1}^{n} 2^{i-1} = 2^n - 1$. The fact that every integral weight l in the range$1 \le l \le 2^n-1$ can be balanced with this set of weights follows immediately from the binary expansion of l, which yields the weights needed for weighing l. (Note that we can obtain the weights needed for a given load l by applying to it the greedy algorithm for the change-making problem with denominations $d_i = 2^{i-1}$, $i=1,2,...n$.)

In order to prove that no set of n weights can cover a larger range of consecutive integral loads, it will suffice to note that there are just 2^n-1 nonempty selections of n weights and, hence, no more than 2^n-1 sums they yield. Therefore, the largest range of consecutive integral loads they can cover cannot exceed 2^n-1.

Note: Alternatively, to prove that no set of n weights can cover a larger range of consecutive integral loads, we can prove by induction on i that if any multiset of n weights $\{w_i\colon\ i=1,...,n\}$—which we can assume without loss of generality to be sorted in nondecreasing order—can balance every integral load starting with 1, then $w_i \le 2^{i-1}$ for $i=1,2,...,n$. The basis checks out immediately: w_1 must be 1, which is equal to 2^{1-1}. For the general case, assume that $w_k \le 2^{k-1}$ for every $1 \le k < i$. The largest weight the first $i-1$ weights can balance is $\sum_{k=1}^{i-1} w_k \le \sum_{k=1}^{i-1} 2^{k-1} = 2^{i-1} - 1$.

If w_i were larger than 2^i, then this load could have been balanced neither with the first $i-1$ weights (which are too light even taken together) nor with the weights $w_i \le \cdots \le w_n$ (which are heavier than 2^i even individually). Hence, $w_i \le 2^{i-1}$, which completes the proof by induction. This immediately implies that no n weights can balance every integral load up to the upper limit larger than $\sum_{i=1}^{n} w_i \le \sum_{i=1}^{n} 2^{i-1} = 2^n - 1$, the limit attainable with the consecutive powers of 2 weights.

b. If weights can be put on both cups of the scale, then a larger range can be reached with n weights for $n > 1$. (For $n = 1$, the single weight still needs to be 1, of course.) The weights $\{1, 3\}$ enable weighing of every integral load up to 4; the weights $\{1, 3, 9\}$ enable weighing of every integral load up to 13, and, in general, the weights $\{w_i = 3^{i-1}\colon\ i = 1, 2, ..., n\}$ enable weighing of every integral load up to and including their sum of $\sum_{i=1}^{n} 3^{i-1} = (3^n - 1)/2$. A load's expansion in the ternary system indicates the weights needed. If the ternary expansion contains only 0's and 1's, the load requires putting the weights corresponding to the 1's on the opposite cup of the balance. If the ternary expansion of load l, $l \le (3^n - 1)/2$, contains one or more 2's, we can replace each 2 by (3-1) to represent it in the form

$$l = \sum_{i=1}^{n} \beta_i 3^{i-1}, \text{ where } \beta_i \in \{0, 1, -1\}, \ \ n = \lceil \log_3(l+1) \rceil.$$

In fact, every positive integer can be uniquely represented in this form, obtained from its ternary expansion as described above. For example,

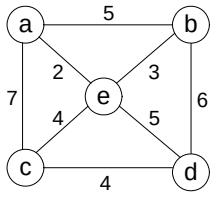
$$\begin{aligned} 5 &= 12_3 = 1 \cdot 3^1 + 2 \cdot 3^0 = 1 \cdot 3^1 + (3-1) \cdot 3^0 = 2 \cdot 3^1 - 1 \cdot 3^0 \\ &= (3-1) \cdot 3^1 - 1 \cdot 3^0 = 1 \cdot 3^2 - 1 \cdot 3^1 - 1 \cdot 3^0. \end{aligned}$$

(Note that if we start with the rightmost 2, after a simplification, the new rightmost 2, if any, will be at some position to the left of the starting one. This proves that after a finite number of such replacements, we will be able to eliminate all the 2's.) Using the representation $l = \sum_{i=1}^{n} \beta_i 3^{i-1}$, we can weigh load l by placing all the weights $w_i = 3^{i-1}$ for negative β_i's along with the load on one cup of the scale and all the weights $w_i = 3^{i-1}$ for positive β_i's on the opposite cup.

Now we'll prove that no set of n weights can cover a larger range of consecutive integral loads than $(3^n - 1)/2$. Each of the n weights can be either put on the left cup of the scale, or put on the right cup, or not to be used at all. Hence, there are $3^n - 1$ possible arrangements of the weights on the scale, with each of them having its mirror image (where all the weights are switched to the opposite pan of the scale). Eliminating this symmetry, leaves us with
just $(3^n - 1)/2$ arrangements, which can weight at most $(3^n - 1)/2$ different

integral loads. Therefore, the largest range of consecutive integral loads they can cover cannot exceed $(3^n - 1)/2$.

9. a. Applying Prim's algorithm to the graph

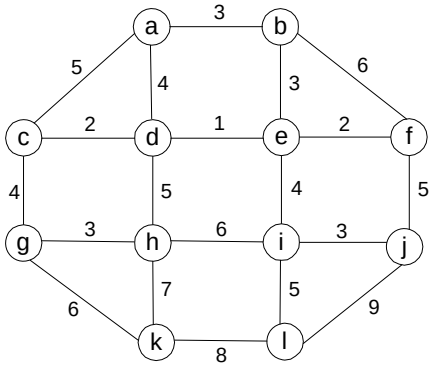


we obtain

Tree vertices	Priority queue of remaining vertices
a(-,-)	b(a,5) c(a,7) d(a,∞) **e(a,2)**
e(a,2)	**b(e,3)** c(e,4) d(e,5)
b(e,3)	**c(e,4)** d(e,5)
c(e,4)	**d(c,4)**
d(c,4)	

The minimum spanning tree found by the algorithm comprises the edges *ae*, *eb*, *ec*, and *cd*.

b. Applying Prim's algorithm to the graph given, we obtain



Tree vertices	Priority queue of fringe vertices
a(-,-)	**b(a,3)** c(a,5) d(a,4)
b(a,3)	c(a,5) d(a,4) **e(b,3)** f(b,6)
e(b,3)	*c(a,5)* **d(e,1)** f(e,2) i(e,4)
d(e,1)	**c(d,2)** f(e,2) i(e,4) h(d,5)
c(d,2)	**f(e,2)** i(e,4) h(d,5) g(c,4)
f(e,2)	**i(e,4)** h(d,5) g(c,4) j(f,5)
i(e,4)	h(d,5) g(c,4) **j(i,3)** l(i,5)
j(i,3)	h(d,5) **g(c,4)** l(i,5)
g(c,4)	**h(g,3)** l(i,5) k(g,6)
h(g,3)	**l(i,5)** k(g,6)
l(i,5)	**k(g,6)**
k(g,6)	

The minimum spanning tree found by the algorithm comprises the edges ab, be, ed, dc, ef, ei, ij, cg, gh, il, gk.

10. There is no need to check the graph's connectivity because Prim's algorithm can do it itself. If the algorithm reaches all the graph's vertices (via edges of finite lengths), the graph is connected, otherwise, it is not.

11. Prim's algorithm does work correctly on graphs with negative edge weights. One can deduce this from the fact that the proof of the algorithm's correctness doesn't preclude negative numbers. One can also reduce a graph with negative weights to one without them by adding a large enough constant C to all the weights on its edges. This operation will increase the weight of all the spanning trees of a graph given by the same amount equal $C(n-1)$, where n is the number of vertices in the graph.

12. The answer is no. In fact, the minimum spanning tree of the new graph can comprise only edges connecting the new vertex to old ones. For example,

let G be a graph of two vertices with the weight on its only edge between them being 2. If an added vertex is connected to each of the vertices in G by edges of weight 1, these two edges will comprise the minimum spanning tree of G_{new}, and hence won't include the edge of G.

13. a. The simplest and most logical solution is to assign all the edge weights to 1.

    b. Applying a depth-first search (or breadth-first search) traversal to get a depth-first search tree (or a breadth-first search tree), is conceptually simpler and for sparse graphs represented by their adjacency lists faster.

14. The number of spanning trees for any weighted connected graph is a positive finite number. (At least one spanning tree exists, e.g., the one obtained by a depth-first search traversal of the graph. And the number of spanning trees must be finite because any such tree comprises a subset of edges of the finite set of edges of the given graph.) Hence, one can always find a spanning tree with the smallest total weight among the finite number of the candidates.

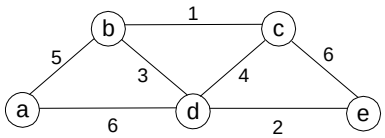
    Let's prove now that the minimum spanning tree is unique if all the weights are distinct. We'll do this by contradiction, i.e., by assuming that there exists a graph $G = (V, E)$ with all distinct weights but with more than one minimum spanning tree. Let $e_1, ..., e_{|V|-1}$ be the list of edges composing the minimum spanning tree T_P obtained by Prim's algorithm with some specific vertex as the algorithm's starting point and let T' be another minimum spanning tree. Let $e_i = (v, u)$ be the first edge in the list $e_1, ..., e_{|V|-1}$ of the edges of T_P which is not in T' (if $T_P \neq T'$, such edge must exist) and let (v, u') be an edge of T' connecting v with a vertex not in the subtree T_{i-1} formed by $\{e_1, ..., e_{i-1}\}$ (if $i = 1$, T_{i-1} consists of vertex v only). Similarly to the proof of Prim's algorithms correctness, let us replace (v, u') by $e_i = (v, u)$ in T'. It will create another spanning tree, whose weight is smaller than the weight of T' because the weight of $e_i = (v, u)$ is smaller than the weight of (v, u'). (Since e_i was chosen by Prim's algorithm, its weight is the smallest among all the weights on the edges connecting the tree vertices of the subtree T_{i-1} and the vertices adjacent to it. And since all the weights are distinct, the weight of (v, u') must be strictly greater than the weight of $e_i = (v, u)$.) This contradicts the assumption that T' was a minimum spanning tree.

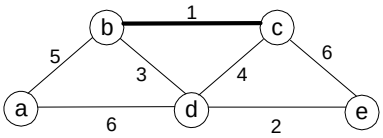
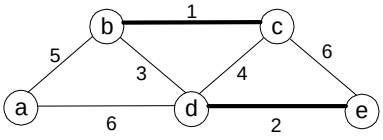
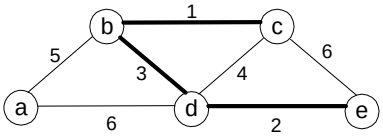
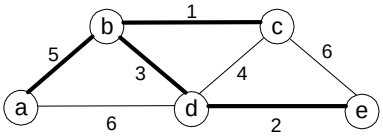
15. If a key's value in a min-heap was decreased, it may need to be pushed up (via swaps) along the chain of its ancestors until it is smaller than or equal to its parent or reaches the root. If a key's value in a min-heap was increased, it may need to be pushed down by swaps with the smaller of its current children until it is smaller than or equal to its children or reaches a

leaf. Since the height of a min-heap with n nodes is equal to $\lfloor \log_2 n \rfloor$ (by the same reason the height of a max-heap is given by this formula—see Section 6.4), the operation's efficiency is in $O(\log n)$. (Note: The old value of the key in question need not be known, of course. Comparing the new value with that of the parent and, if the min-heap condition holds, with the smaller of the two children, will suffice.)

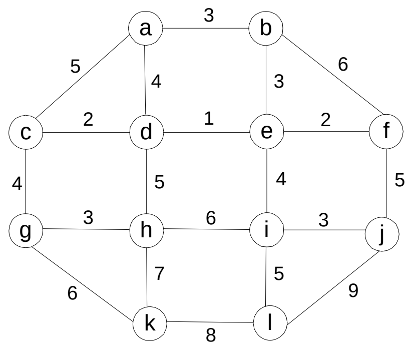
Solutions to Exercises 9.2

1. a.

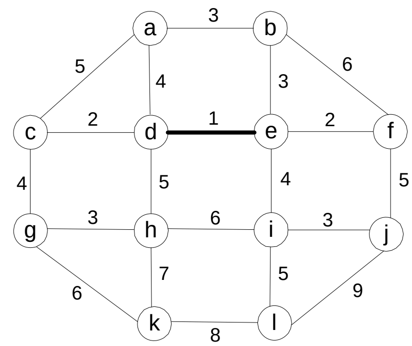


Tree edges	Sorted list of edges (selected edges are shown in bold)							Illustration
	bc 1	de 2	bd 3	cd 4	ab 5	ad 6	ce 6	
bc 1	bc 1	**de** 2	bd 3	cd 4	ab 5	ad 6	ce 6	
de 2	bc 1	de 2	**bd** 3	cd 4	ab 5	ad 6	ce 6	
bd 3	bc 1	de 2	bd 3	cd 4	**ab** 5	ad 6	ce 6	
ab 5								

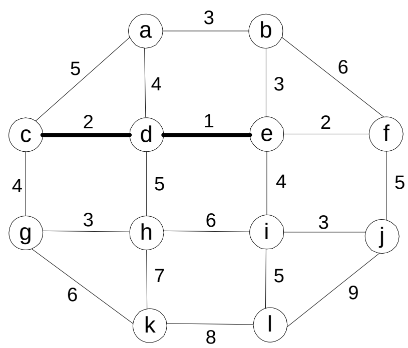
b.


a 3 b
5 4 3 6
c 2 d 1 e 2 f
4 5 4 5
g 3 h 6 i 3 j
6 7 5 9
k 8 l

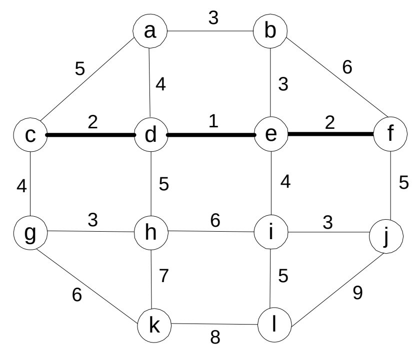
⇒


a 3 b
5 4 3 6
c 2 d 1 e 2 f
4 5 4 5
g 3 h 6 i 3 j
6 7 5 9
k 8 l

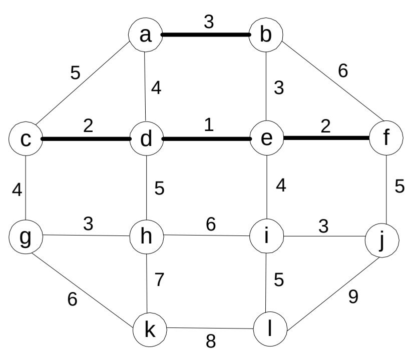
⇒


a 3 b
5 4 3 6
c 2 d 1 e 2 f
4 5 4 5
g 3 h 6 i 3 j
6 7 5 9
k 8 l

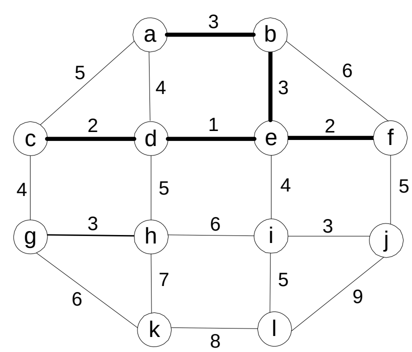
⇒


a 3 b
5 4 3 6
c 2 d 1 e 2 f
4 5 4 5
g 3 h 6 i 3 j
6 7 5 9
k 8 l

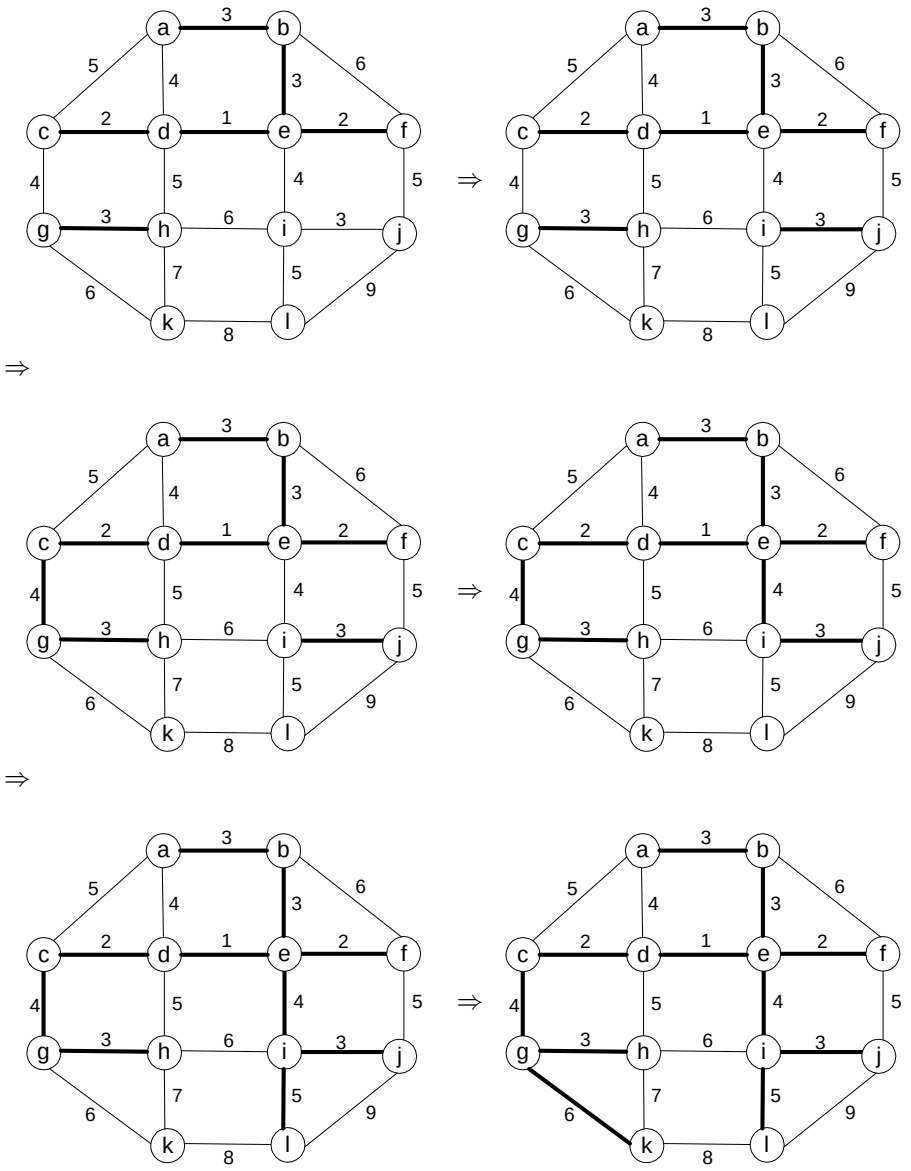
⇒


a 3 b
5 4 3 6
c 2 d 1 e 2 f
4 5 4 5
g 3 h 6 i 3 j
6 7 5 9
k 8 l

⇒


a 3 b
5 4 3 6
c 2 d 1 e 2 f
4 5 4 5
g 3 h 6 i 3 j
6 7 5 9
k 8 l

⇒



2. a. True. (Otherwise, Kruskal's algorithm would be invalid.)

   b. False. As a simple counterexample, consider a complete graph with three vertices and the same weight on its three edges

   c. True (Problem 14 in Exercises 9.1).

d. False (see, for example, the graph of Problem 1a).

3. Since the number of edges in a minimum spanning forest of a graph with $|V|$ vertices and $|C|$ connected components is equal to $|V| - |C|$ (this formula is a simple generalization of $|E| = |V| - 1$ for connected graphs), *Kruskal*(G) will never get to $|V| - 1$ tree edges unless the graph is connected. A simple remedy is to replace the loop **while** *ecounter* $< |V| - 1$ with **while** $k < |E|$ to make the algorithm stop after exhausting the sorted list of its edges.

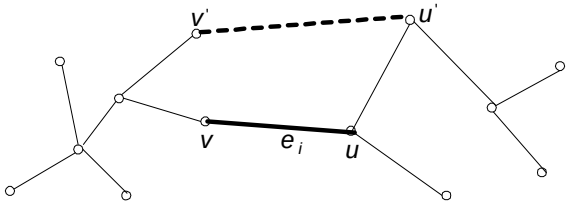
4. Both algorithms work correctly for graphs with negative edge weights. One way of showing this is to add to all the weights of a graph with negative weights some large positive number. This makes all the new weights positive, and one can "translate" the algorithms' actions on the new graph to the corresponding actions on the old one. Alternatively, you can check that the proofs justifying the algorithms' correctness do not depend on the edge weights being nonnegative.

5. Replace each weight $w(u, v)$ by $-w(u, v)$ and apply any minimum spanning tree algorithm that works on graphs with arbitrary weights (e.g., Prim's or Kruskal's algorithm) to the graph with the new weights.

6. **Algorithm** *Kruskal*(G)
   //Kruskal's algorithm with explicit disjoint-subsets operations
   //Input: A weighted connected graph $G = \langle V, E \rangle$
   //Output: E_T, the set of edges composing a minimum spanning tree of G
   sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le ... \le w(e_{i_{|E|}})$
   for each vertex $v \in V$ *make*(v)
   $E_T \leftarrow \emptyset$; *ecounter* $\leftarrow 0$ //initialize the set of tree edges and its size
   $k \leftarrow 0$ //the number of processed edges
   while *ecounter* $< |V| - 1$
   $k \leftarrow k + 1$
   if *find*$(u) \neq$ *find*(v) //u, v are the endpoints of edge e_{i_k}
   $E_T \leftarrow E_T \cup \{e_{i_k}\}$; *ecounter* $\leftarrow$ *ecounter* $+ 1$
   union(u, v)
   return E_T

7. Let us prove by induction that each of the forests F_i, $i = 0, ..., |V| - 1$, of Kruskal's algorithm is a part (i.e., a subgraph) of some minimum spanning tree. (This immediately implies, of course, that the last forest in the sequence, $F_{|V|-1}$, is a minimum spanning tree itself. Indeed, it contains all vertices of the graph, and it is connected because it is both acyclic and has $|V| - 1$ edges.) The basis of the induction is trivial, since F_0 is

made up of $|V|$ single-vertex trees and therefore must be a subgraph of any spanning tree of the graph. For the inductive step, let us assume that F_{i-1} is a subgraph of some minimum spanning tree T. We need to prove that F_i, generated from F_{i-1} by Kruskal's algorithm, is also a part of a minimum spanning tree. We prove this by contradiction by assuming that no minimum spanning tree of the graph can contain F_i. Let $e_i = (v, u)$ be the minimum weight edge added by Kruskal's algorithm to forest F_{i-1} to obtain forest F_i. (Note that vertices v and u must belong to different trees of F_{i-1}—otherwise, edge (v, u) would've created a cycle.) By our assumption, e_i cannot belong to T. Therefore, if we add e_i to T, a cycle must be formed (see the figure below). In addition to edge $e_i = (v, u)$, this cycle must contain another edge (v', u') connecting a vertex v' in the same tree of F_{i-1} as v to a vertex u' not in that tree. (It is possible that v' coincides with v or u' coincides with u but not both.) If we now delete the edge (v', u') from this cycle, we will obtain another spanning tree of the entire graph whose weight is less than or equal to the weight of T since the weight of e_i is less than or equal to the weight of (v', u'). Hence, this spanning tree is a minimum spanning tree, which contradicts the assumption that no minimum spanning tree contains F_i. This completes the correctness proof of Kruskal's algorithm.

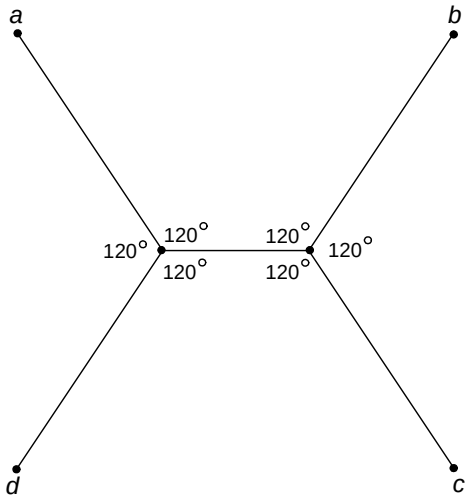


8. In the *union-by-size* version of *quick-union*, each vertex starts at depth 0 of its own tree. The depth of a vertex increases by 1 when the tree it is in is attached to a tree with at least as many nodes during a union operation. Since the total number of nodes in the new tree containing the node is at least twice as much as in the old one, the number of such increases cannot exceed $\log_2 n$. Therefore the height of any tree (which is the largest depth of the tree's nodes) generated by a legitimate sequence of unions will not exceed $\log_2 n$. Hence, the efficiency of *find*(x) is in $O(\log n)$ because *find*(x) traverses the pointer chain from the x's node to the tree's root.

9. n/a

10. n/a

11. The minimum Steiner tree that solves the problem is shown below. (The other solution can be obtained by rotating the figure 90°.)



Note: A popular discussion of Steiner trees can be found in "Last Recreations: Hydras, Eggs, and Other Mathematical Mystifications" by Martin Gardner. In general, no polynomial time algorithm is known for finding a minimum Steiner tree; moreover, the problem is known to be *NP*-hard (see Section 11.3). For the state-of-the-art information, see, e.g., The Steiner Tree Page at http://ganley.org/steiner/.

12. n/a

Solutions to Exercises 9.3

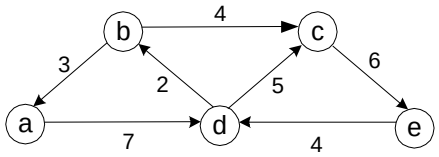
1. a. It will suffice to take into account edge directions in processing adjacent vertices.

   b. Start the algorithm at one of the given vertices and stop it as soon as the other vertex is added to the tree.

   c. If the given graph is undirected, solve the single-source problem with the destination vertex as the source and reverse all the paths obtained in the solution. If the given graph is directed, reverse all its edges first, solve the single-source problem for the new digraph with the destination vertex as the source, and reverse the direction of all the paths obtained in the solution.

   d. Create a new graph by replacing every vertex v with two vertices v' and v'' connected by an edge whose weight is equal to the given weight of vertex v. All the edges entering and leaving v in the original graph will enter v' and leave v'' in the new graph, respectively. Assign the weight of 0 to each original edge. Applying Dijkstra's algorithm to the new graph will solve the problem.

2. a.

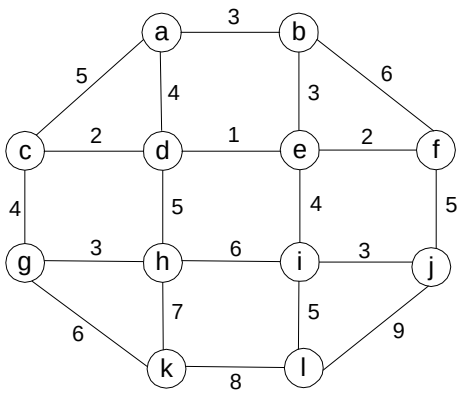


Tree vertices	Remaining vertices
a(-,0)	b(-,∞) c(–,∞) **d(a,**7) e(-,∞)
d(a,7)	**b(d,7+2)** c(d,7+5) e(-,∞)
b(d,9)	**c(d,12)** e(-,∞)
c(d,12)	**e(c,12+6)**
e(c,18)	

The shortest paths (identified by following nonnumeric labels backwards from a destination vertex to the source) and their lengths are:

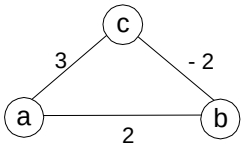
from a to d:	$a - d$	of length 7
from a to b:	$a - d - b$	of length 9
from a to c:	$a - d - c$	of length 12
from a to e:	$a - d - c - e$	of length 18

b.



Tree vertices	Fringe vertices	Shortest paths from a
a(-,0)	**b(a,3)** c(a,5) d(a,4)	to b: $a-b$ of length 3
b(a,3)	c(a,5) **d(a,4)** e(b,3+3) f(b,3+6)	to d: $a-d$ of length 4
d(a,4)	**c(a,5)** e(d,4+1) f(a,9) h(d,4+5)	to c: $a-c$ of length 5
c(a,5)	**e(d,5)** f(a,9) h(d,9) g(c,5+4))	to e: $a-d-e$ of length 5
e(d,5)	**f(e,5+2)** h(d,9) g(c,9) i(e,5+4)	to f: $a-d-e-f$ of length 7
f(e,7)	**h(d,9)** g(c,9) i(e,9) j(f,7+5)	to h: $a-d-h$ of length 9
h(d,9)	**g(c,9)** i(e,9) j(f,12) k(h,9+7))	to g: $a-c-g$ of length 9
g(c,9)	**i(e,9)** j(f,12) k(g,9+6)	to i: $a-d-e-i$ of length 9
i(e,9)	**j(f,12)** k(g,15) l(i,9+5)	to j: $a-d-e-f-j$ of length 12
j(f,12)	k(g,15) **l(i,14)**	to l: $a-d-e-i-l$ of length 14
l(i,14)	**k(g,15)**	to k: $a-c-g-k$ of length 15
k(g,15)		

3. Consider, for example, the graph

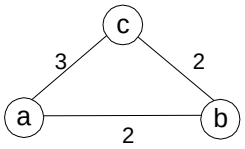


As the shortest path from a to b, Dijkstra's algorithm yields $a-b$ of length 2, which is longer than $a-c-b$ of length 1.

4. a. True: On each iteration, we add to a previously constructed tree an edge connecting a vertex in the tree to a vertex that is not in the tree. So, the resulting structure must be a tree. And, after the last operation,

it includes all the vertices of the graph. Hence, it's a spanning tree.

b. False. Here is a simple counterexample:



With vertex a as the source, Dijkstra's algorithm yields, as the shortest path tree, the tree composed of edges (a, b) and (a, c). The graph's minimum spanning tree is composed of (a, b) and (b, c).

5. **Algorithm** *SimpleDijkstra*$(W[0..n-1, 0..n-1], s)$
//Input: A matrix of nonnegative edge weights W and
// integer s between 0 and $n-1$ indicating the source
//Output: An array $D[0..n-1]$ of the shortest path lengths
// from s to all vertices
for $i \leftarrow 0$ **to** $n-1$ **do**
  $D[i] \leftarrow \infty$; $treeflag[i] \leftarrow$ **false**
$D[s] \leftarrow 0$
for $i \leftarrow 0$ **to** $n-1$ **do**
  $dmin \leftarrow \infty$
  for $j \leftarrow 0$ **to** $n-1$ **do**
    if not $treeflag[j]$ **and** $D[j] < dmin$
      $jmin \leftarrow j$; $dmin \leftarrow D[jmin]$
  $treeflag[jmin] \leftarrow$ **true**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    if not $treeflag[j]$ **and** $dmin + W[jmin, j] < \infty$
      $D[j] \leftarrow dmin + W[jmin, j]$
return D

6. We will prove by induction on the number of vertices i in tree T_i constructed by Dijkstra's algorithm that this tree contains i closest vertices to source s (including the source itself), for each of which the tree path from s to v is a shortest path of length d_v. For $i = 1$, the assertion is obviously true for the trivial path from the source to itself. For the general step, assume that it is true for the algorithm's tree T_i with i vertices. Let v_{i+1} be the vertex added next to the tree by the algorithm. All the vertices on a shortest path from s to v_{i+1} preceding v_{i+1} must be in T_i because they are closer to s than v_{i+1}. (Otherwise, the first vertex on the path from s to v_{i+1} that is not in T_i would've been added to T_i instead of v_{i+1}.) Hence, the $(i+1)$st closest vertex can be selected as the algorithm does: by minimizing the sum of d_v (the shortest distance from s to $v \in T_i$

by the assumption of the induction) and the length of the edge from v to an adjacent vertex not in the tree.

7. **Algorithm** *DagShortestPaths*(G, s)
   //Solves the single-source shortest paths problem for a dag
   //Input: A weighted dag $G = \langle V, E \rangle$ and its vertex s
   //Output: The length d_v of a shortest path from s to v and
   // its penultimate vertex p_v for every vertex v in V
   topologically sort the vertices of G
   for every vertex v **do**
   $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
   $d_s \leftarrow 0$
   for every vertex v taken in topological order **do**
   for every vertex u adjacent to v **do**
   if $d_v + w(v, u) < d_u$
   $d_u \leftarrow d_v + w(v, u)$; $p_u \leftarrow v$

   Topological sorting can be done in $\Theta(|V|+|E|)$ time (see Section 4.2). The distance initialization takes $\Theta(|V|)$ time. The innermost loop is executed for every edge of the dag. Hence, the total running time is in $\Theta(|V|+|E|)$.

8. Create a digraph by connecting numbers on adjacent levels of the triangle that can be components of a sum from the apex to the base. As a weight of an edge connecting two numbers, assign the lower of the two (i.e., the one closer to the base). Apply Dijkstra's algorithm with the source at the apex of the triangle. Stop the algorithm as soon as the shortest path to a number/vertex at the triangle's base is reached. Note that the length of the shortest path is smaller than the minimum sum from the apex to the base by the number at the apex.

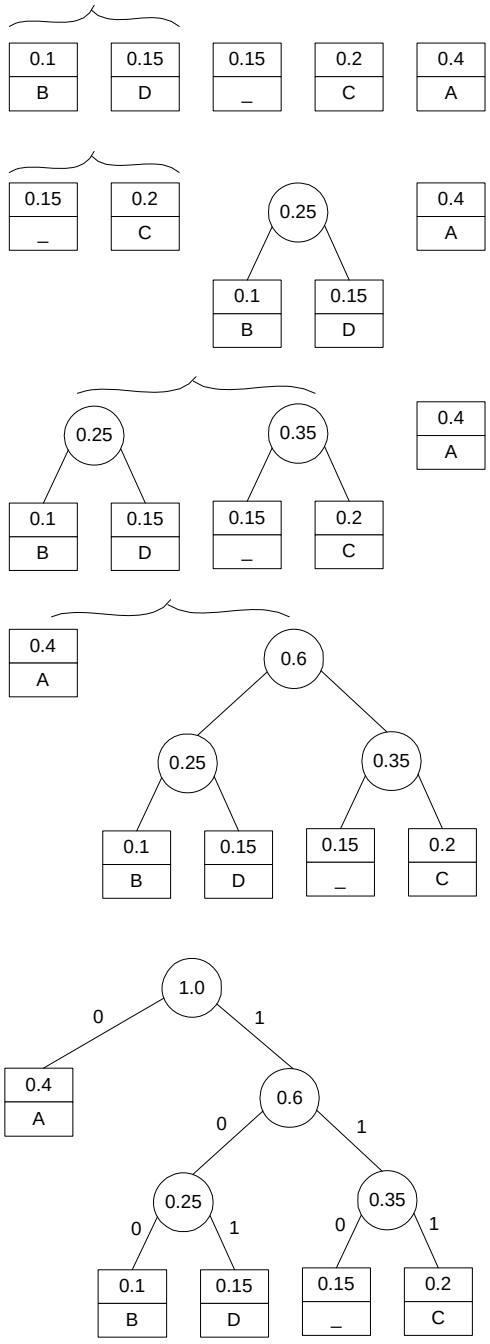
9. a. Take the two balls representing the two singled out vertices in two hands and stretch the model to get the shortest path in question as a straight line between the two ball-vertices.

   b. Hold the ball representing the source in one hand and let the rest of the model hang down: The force of gravity will make the shortest path to each of the other balls be on a straight line down.

10. n/a

Solutions to Exercises 9.4

1. a.



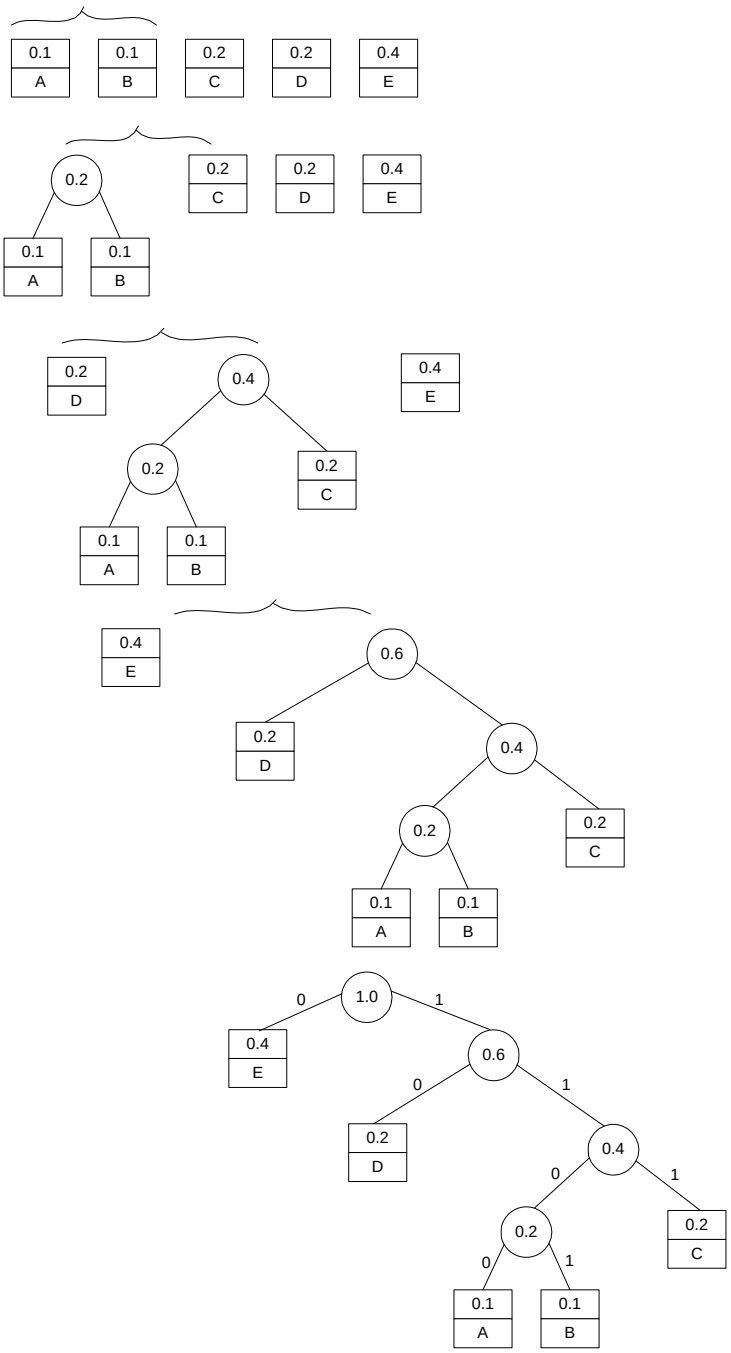
character	A	B	C	D	_
probability	0.4	0.1	0.2	0.15	0.15
codeword	0	100	111	101	110

b. The text ABACABAD will be encoded as 0100011101000101.

c. With the code of part a, 100010111001010 will be decoded as

$$\underset{B}{100}|\underset{A}{0}|\underset{D}{101}|\underset{\_}{110}|\underset{A}{0}|\underset{D}{101}|\underset{A}{0}$$

2. Here is one way:

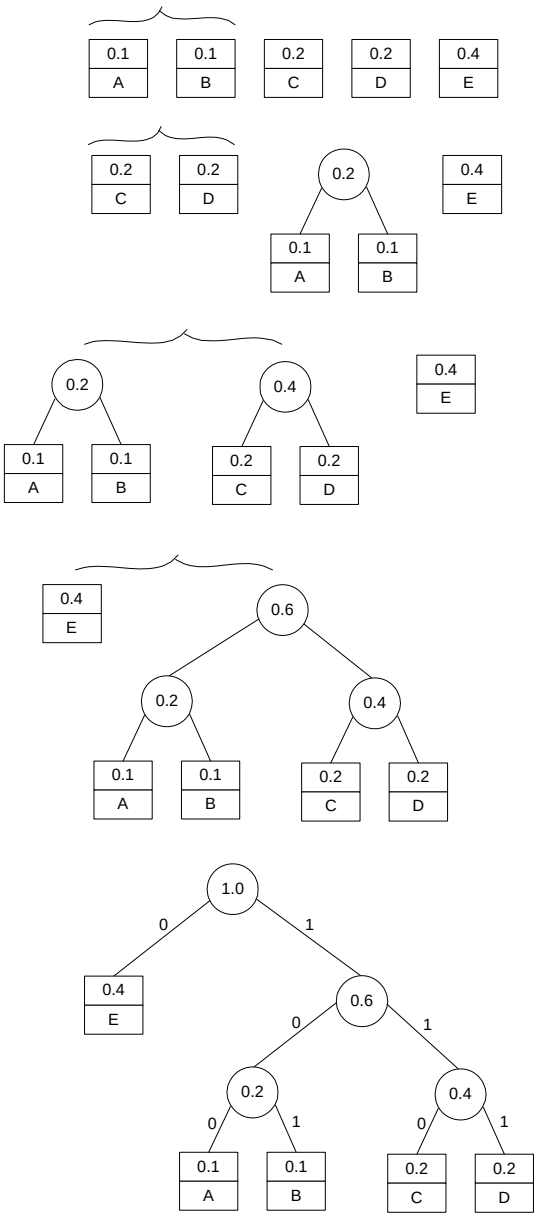


character	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4
codeword	1100	1101	111	10	0
length	4	4	3	2	1

Thus, the mean and variance of the codeword's length are, respectively,

$$\begin{aligned} \bar{l} &= \sum_{i=1}^{5} l_i p_i = 4 \cdot 0.1 + 4 \cdot 0.1 + 3 \cdot 0.2 + 2 \cdot 0.2 + 1 \cdot 0.4 = 2.2 \quad \text{and} \\ Var &= \sum_{i=1}^{5} (l_i - \bar{l})^2 p_i = (4\text{-}2.2)^2 0.1 + (4\text{-}2.2)^2 0.1 + (3\text{-}2.2)^2 0.2 + (2\text{-}2.2)^2 0.2 + (1\text{-}2.2)^2 0.4 = 1.36. \end{aligned}$$

Here is another way:



character	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4
codeword	100	101	110	111	0
length	3	3	3	3	1

Thus, the mean and variance of the codeword's length are, respectively,

$$\bar{l} = \sum_{i=1}^{5} l_i p_i = 2.2 \quad \text{and} \quad Var = \sum_{i=1}^{5} (l_i - \bar{l})^2 p_i = 0.96.$$

3. a. Yes. This follows immediately from the way Huffman's algorithm operates: after each of its iterations, the two least frequent characters that are combined on the first iteration are always on the same level of their tree in the algorithm's forest. An easy formal proof of this obvious observation is by induction.
(Note that if there are more than two least frequent characters, the assertion may be false for some pair of them, e.g., $A(\frac{1}{3})$, $B(\frac{1}{3})$, $C(\frac{1}{3})$.)

   b. Yes. Let's use the optimality of Huffman codes to prove this property by contradiction. Assume that there exists a Huffman code containing two characters c_i and c_j such that $p(c_i) > p(c_j)$ and $l(w(c_i)) > l(w(c_j))$, where $p(c_i)$ and $l(w(c_i))$ are the probability and codeword's length of c_i, respectively, and $p(c_j)$ and $l(w(c_j))$ are the probability and codeword's length of c_j, respectively. Let's create a new code by simply swapping the codewords of c_1 and c_2 and leaving the codewords for all the other characters the same. The new code will obviously remain prefix-free and its expected length $\sum_{k=1}^{n} l(w(c_k))p(c_k))$ will be smaller than that of the initial code. This contradicts the optimality of the initial Huffman code and, hence, proves the property in question.

4. The answer is $n - 1$. Since two leaves corresponding to the two least frequent characters must be on the same level of the tree, the tallest Huffman coding tree has to have the remaining leaves each on its own level. The height of such a tree is $n - 1$. An easy and natural way to get a Huffman tree of this shape is by assuming that $p_1 \leq p_2 < ... < p_n$ and having the weight W_i of a tree created on the ith iteration of Huffman's algorithm, $i = 1, 2, ..., n-2$, be less than or equal to p_{i+2}. (Note that for such inputs, $W_i = \sum_{k=1}^{i+1} p_k$ for every $i = 1, 2, ..., n - 1$.)

   As a specific example, it's convenient to consider consecutive powers of 2:

$$p_1 = p_2 \quad \text{and} \quad p_i = 2^{i-n-1} \text{ for } i = 2, ..., n.$$

   (For, say, $n = 4$, we have $p_1 = p_2 = 1/8$, $p_3 = 1/4$ and $p_4 = 1/2$.) Indeed, $p_i = 2^i/2^{n+1}$ is an increasing sequence as a function of i. Further,

$W_i = p_{i+2}$ for every $i = 1, 2, ..., n-2$, since

$$\begin{aligned} W_i &= \sum_{k=1}^{i+1} p_k = p_1 + \sum_{k=2}^{i+1} p_k = 2^2/2^{n+1} + \sum_{k=2}^{i+1} 2^k/2^{n+1} = \frac{1}{2^{n+1}}(2^2 + \sum_{k=2}^{i+1} 2^k) \\ &= \frac{1}{2^{n+1}}(2^2 + (2^{i+2} - 4)) = \frac{2^{i+2}}{2^{n+1}} = p_{i+2}. \end{aligned}$$

5. a. The following pseudocode is based on maintaining a priority queue of trees, with the priorities equal the trees' weights.

Algorithm *Huffman*$(W[0..n-1])$
//Constructs Huffman's tree
//Input: An array $W[0..n-1]$ of weights
//Output: A Huffman tree with the given weights assigned to its leaves
initialize priority queue Q of size n with one-node trees and priorities equal to the elements of $W[0..n-1]$
while Q has more than one element **do**
    $T_l \leftarrow$ the minimum-weight tree in Q
    delete the minimum-weight tree in Q
    $T_r \leftarrow$ the minimum-weight tree in Q
    delete the minimum-weight tree in Q
    create a new tree T with T_l and T_r as its left and right subtrees
                and the weight equal to the sum of T_l and T_r weights
    insert T into Q
return T

Note: See also Problem 6 for an alternative algorithm.

b. The algorithm requires the following operations: initializing a priority queue, deleting its smallest element $2(n-1)$ times, computing the weight of a combined tree and inserting it into the priority queue $n-1$ times. The overall running time will be dominated by the time spent on deletions, even taking into account that the size of the priority queue will be decreasing from n to 2. For the min-heap implementation, the time efficiency will be in $O(n \log n)$; for the array or linked list representations, it will be in $O(n^2)$. (Note: For the coding application of Huffman trees, the size of the underlying alphabet is typically not large; hence, a simpler data structure for the priority queue might well suffice.)

6. The critical insight here is that the weights of the trees generated by Huffman's algorithm for nonnegative weights (frequencies) form a nondecreasing sequence. As the hint to this problem suggests, we can then maintain two queues: one for given frequencies in nondecreasing order, the other

for weights of new trees. On each iteration, we do the following: find the two smallest elements among the first two (ordered) elements in the queues (the second queue is empty on the first iteration and can contain just one element thereafter); add their sum to the second queue; and then delete these two elements from their queues. The algorithm stops after $n-1$ iterations (where n is the alphabet's size), each of which requiring a constant time.

7. Use one of the standard traversals of the binary tree and generate a bit string for each node of the tree as follows:. Starting with the empty bit string for the root, append 0 to the node's string when visiting the node's left subtree begins and append 1 to the node's string when visiting the node's right subtree begins. At a leaf, print out the current bit string as the leaf's codeword. Since Huffman's tree with n leaves has a total of $2n-1$ nodes (see Sec. 4.4), the efficiency will be in $\Theta(n)$.

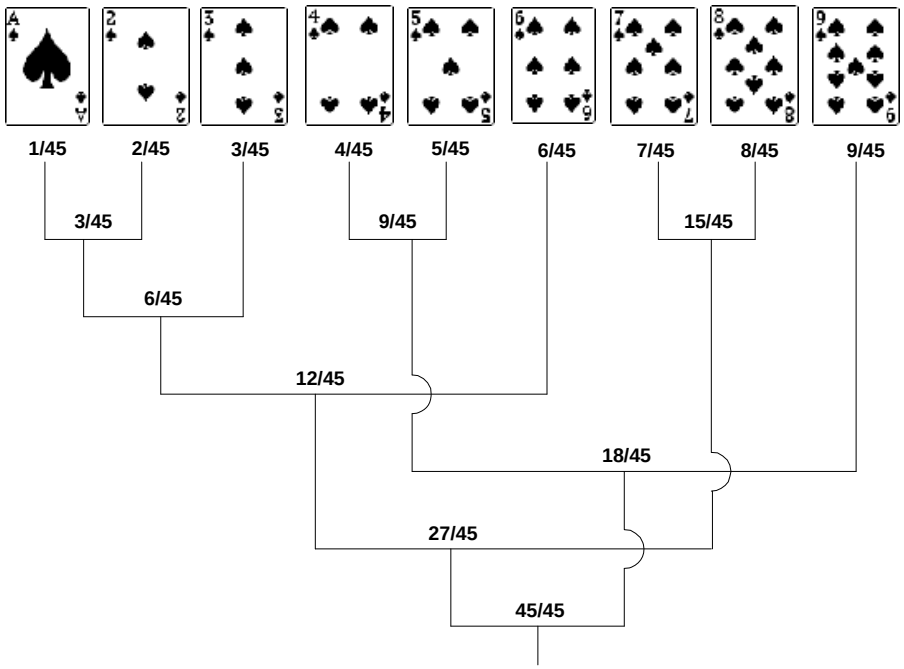
8. We can generate the codewords right to left by the following method that stems immediately from Huffman's algorithm: when two trees are combined, append 0 in front of the current bit strings for each leaf in the left subtree and append 1 in front of the current bit strings for each leaf in the right subtree. (The substrings associated with the initial one-node trees are assumed to be empty.)

9. n/a

10. The probabilities of a selected card be of a particular type is given in the following table:

card	ace	deuce	three	four	five	six	seven	eight	nine
probability	1/45	2/45	3/45	4/45	5/45	6/45	7/45	8/45	9/45

Huffman's tree for this data looks as follows:



The first question this tree implies can be phrased as follows: "Is the selected card a four, a five, or a nine?" . (The other questions can be phrased in a similar fashion.)

The expected number of questions needed to identify a card is equal to the weighted path length from the root to the leaves in the tree:

$$\bar{l} = \sum_{i=1}^{9} l_i p_i = \frac{5 \cdot 1}{45} + \frac{5 \cdot 2}{45} + \frac{4 \cdot 3}{45} + \frac{3 \cdot 5}{45} + \frac{3 \cdot 6}{45} + \frac{3 \cdot 7}{45} + \frac{3 \cdot 8}{45} + \frac{2 \cdot 9}{45} = \frac{135}{45} = 3.$$

Solutions to Exercises 10.1

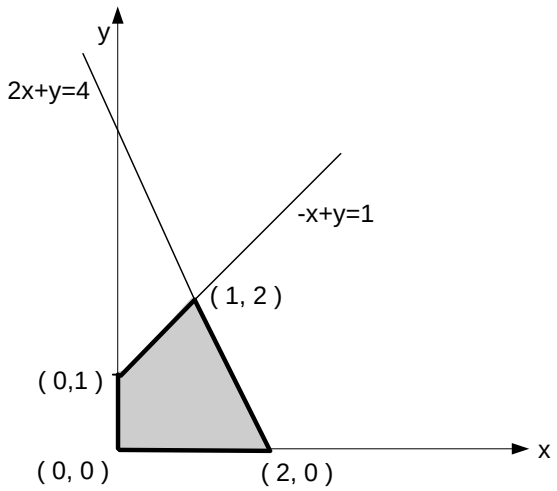
1. The problem is to find a value of x that minimizes $f(x) = \frac{1}{n}\sum_{i=1}^{n} |x - x_i|$, which can be simplified to minimizing $g(x) = \sum_{i=1}^{n} |x - x_i|$ since $\frac{1}{n}$ is a constant. Here is an iterative improvement algorithm that does this. Select an arbitrary integer location x and compute $g(x)$, then compute $g(x-1)$ and $g(x+1)$. If $g(x-1) < g(x)$ or $g(x+1) < g(x)$, replace x by $x-1$or $x+1$, respectively, and repeat this step with the new value of x; otherwise, return the current value of x as the post office location. Note that since the functions $f(x)$ and $g(x)$ are piecewise linear and convex, the algorithm will stop after a finite number of iterations for any initial integer value of x.

   The solution to the problem is the median of x_1, x_2, ..., x_n, which can be found either by sorting the points given or by a partition-based algorithm (see Section 4.5). Although the iterative improvement algorithm can find the solution faster if the initial value of x happens to be close to the median, it can hardly be considered competitive with either the presorting-based algorithm or quickselect. Also note that the latter work for noninteger input, whereas the iterative improvement algorithm does not.

2. a. The feasible region of the linear programming problem

$$\begin{array}{rl} \text{maximize} & 3x + y \\ \text{subject to} & -x + y \le 1 \\ & 2x + y \le 4 \\ & x \ge 0, \;\; y \ge 0 \end{array}$$

   is given in the following figure:



   Since the feasible region of the problem is nonempty and bounded, an

optimal solution exists and can be found at one of the extreme points of the feasible region. The extreme points and the corresponding values of the objective function are given in the following table:

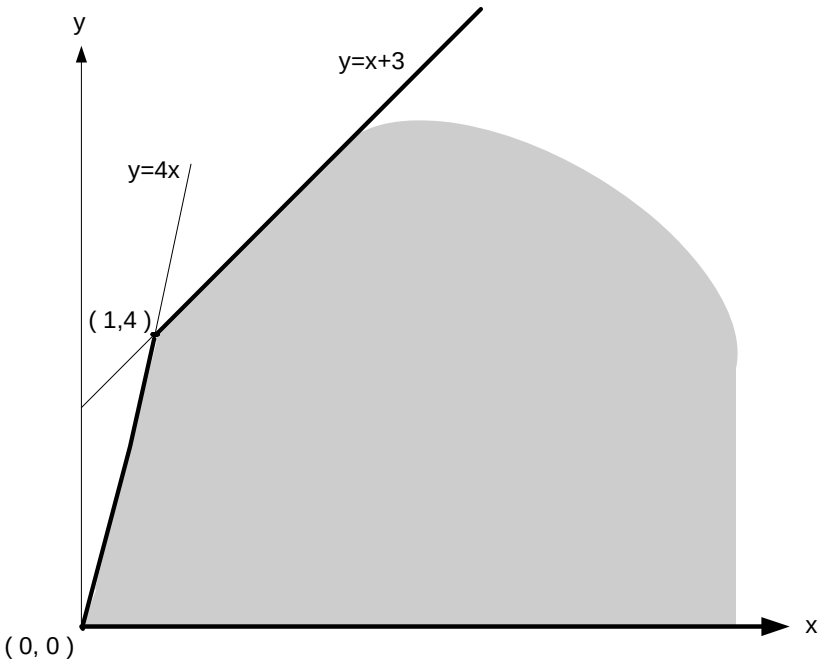
Extreme point	Value of $3x + y$
$(0, 0)$	0
$(2, 0)$	6
$(1, 2)$	5
$(0, 1)$	1

Hence, the optimal solution is $x = 2$, $y = 0$, with the corresponding value of the objective function equal to 6.

b. The feasible region of the linear programming problem

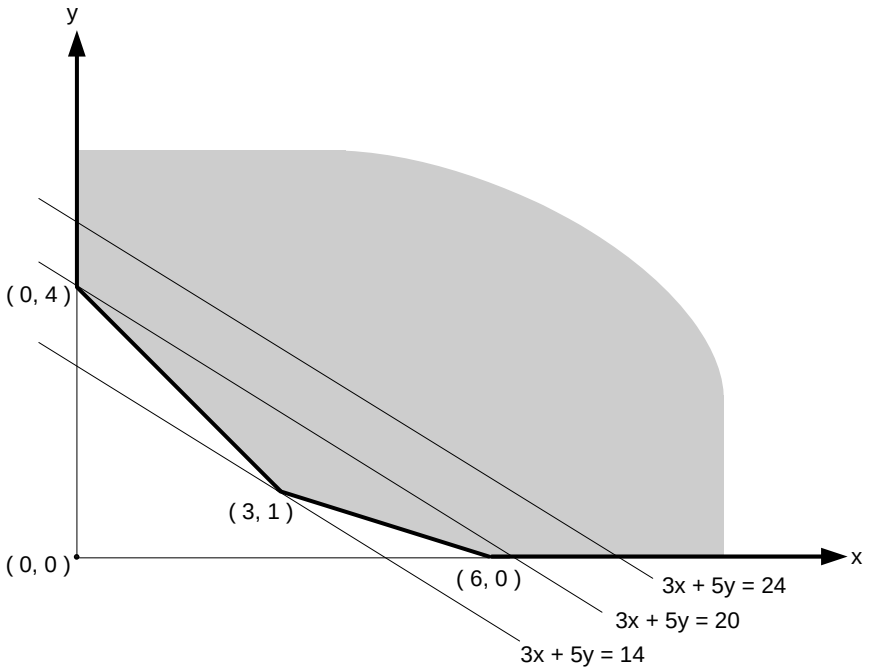
$$\begin{array}{rl} \text{maximize} & x + 2y \\ \text{subject to} & 4x \geq y \\ & x + 3 \geq y \\ & x \geq 0, \;\; y \geq 0 \end{array}$$

is given in the following figure:



The feasible region is unbounded. On considering a few level lines $x+2y = z$, it is easy to see that level lines can be moved in the north-east direction, which corresponds to increasing values of z, as far as we wish without losing common points with the feasible region. (Alternatively, we can consider a simple sequence of points, such as $(n, 0)$, which are feasible for any nonnegative integer value of n and make the objective function $z = x + 2y$ as large as we wish as n goes to infinity.) Hence, the problem is unbounded and therefore does not have a finite optimal solution.

3. The feasible region of the problem with the constraints $x + y \geq 4$, $x + 3y \geq 6$, $x \geq 0$, $y \geq 0$ is given in Figure 10.3 of Section 10.1:



a. Minimization of $z = c_1x + c_2y$ on this feasible region, with $c_1, c_2 > 0$, pushes level lines $c_1x + c_2y = z$ in the south-west direction. Any family of such lines with a slope strictly between the slopes of the boundary lines $x + y = 4$ and $x + 3y = 6$ will hit the extreme point $(3, 1)$ as the only minimum solution to the problem. For example, as mentioned in Section 10.1, we can minimize $3x + 5y$ among infinitely many other possible answers.

b. For a linear programming problem to have infinitely many solutions, the optimal level line of its objective function must contain a line segment that is a part of the feasible region's boundary. Hence, there are four qualitatively distinct answers:

i. minimize $1 \cdot x + 0 \cdot y$ (or, more generally, any objective function of the form $c \cdot x + 0 \cdot y$, where $c > 0$);

ii. minimize $x + y$ (or, more generally, any objective function of the form $cx + cy$, where $c > 0$);

iii. minimize $x + 3y$ (or, more generally, any objective function of the form $cx + 3cy$, where $c > 0$);

iv. minimize $0 \cdot x + 1 \cdot y$ (or, more generally, any objective function of the form $0 \cdot x + c \cdot y$, where $c > 0$).

c. Among infinitely many examples, there are the following: $-1 \cdot x + -1 \cdot y$, $-1 \cdot x + 0 \cdot y$, $0 \cdot x - 1 \cdot y$.

4. The problem with the strict inequalities does not have an optimal solution: The objective function values of any sequence of feasible points approaching $x_1 = 3$, $x_2 = 1$ (the optimal solution to Problem (10.2) in the text) will approach $z = 14$ as its limit, but this value will not be attained at any feasible point of the problem with the strict inequalities.

5. a. The standard form of the problem given is

$$\begin{array}{ll} \text{maximize} & 3x + y \\ \text{subject to} & -x + y + u \qquad = 1 \\ & 2x + y + \quad +v = 4 \\ & x, y, u, v \geq 0. \end{array}$$

Here are the tableaux generated by the simplex method in solving this problem:

	x	y	u	v	
u	−1	1	1	0	1
← v	2	1	0	1	4
	−3	−1	0	0	0

$\theta_v = \frac{4}{2}$

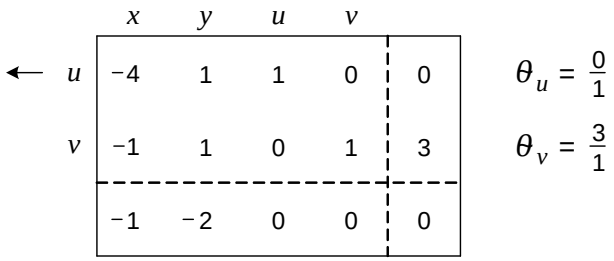
	x	y	u	v	
u	0	$\frac{3}{2}$	1	$\frac{1}{2}$	3
x	1	$\frac{1}{2}$	0	$\frac{1}{2}$	2
	0	$\frac{1}{2}$	0	$\frac{3}{2}$	6

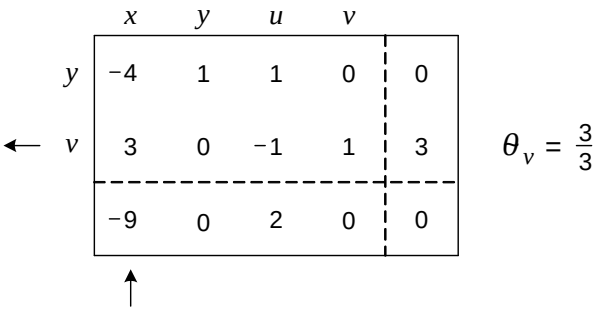
The optimal solution found is $x = 2$, $y = 0$, with the maximal value of the objective function equal to 6.

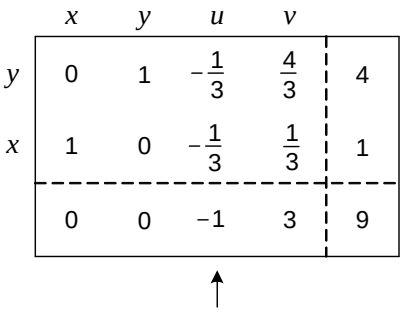
b. The standard form of the problem given is

$$\begin{array}{ll} \text{maximize} & x + 2y \\ \text{subject to} & -4x + y + u \qquad = 0 \\ & -x + y + \quad +v = 3 \\ & x, y, u, v \geq 0. \end{array}$$

Here are the tableaux generated by the simplex method in solving this problem:







Since there are no positive elements in the pivot column of the last tableau, the problem is unbounded.

6. a. To simplify the task of getting an initial basic feasible solution here, we can replace the equality constraint $x + y + z = 100$ by the inequality $x + y + z \le 100$, because an optimal solution (x^*, y^*, z^*) to the problem with the latter constraint must satisfy the former one.. (Otherwise, we would've been able to increase the maximal value of the objective function by increasing the value of z^*.) then, after an optional replacement of the objective function and the constraints by the equivalent ones without fractional coefficients, the problem can be presented in the standard form as follows

$$\begin{aligned} \text{maximize} \quad & 10x + 7y + 3z \\ \text{subject to} \quad & x + y + z + u = 100 \\ & 3x - y + \quad + v \quad = 0 \\ & x + y - 4z + \quad + w = 0 \\ & x, y, z, u, v, w \ge 0. \end{aligned}$$

Here are the tableaux generated by the simplex method in solving this problem:

		x	y	z	u	v	w		
	u	1	1	1	1	0	0	100	$\theta_u = \frac{100}{1}$
←	v	3	−1	0	0	1	0	0	$\theta_v = \frac{0}{3}$
	w	1	1	−4	0	0	1	0	$\theta_w = \frac{0}{1}$
		−10	−7	−3	0	0	0	0	

↑

		x	y	z	u	v	w		
	u	0	$\frac{4}{3}$	1	1	$-\frac{1}{3}$	0	100	$\theta_u = \frac{100}{4/3}$
	x	1	$-\frac{1}{3}$	0	0	$\frac{1}{3}$	0	0	
←	w	0	$\frac{4}{3}$	−4	0	$-\frac{1}{3}$	1	0	$\theta_w = \frac{0}{4/3}$
		0	$-\frac{31}{3}$	−3	0	$\frac{10}{3}$	0	0	

↑

		x	y	z	u	v	w		
←	u	0	0	5	1	0	−1	100	$\theta_u = \frac{100}{5}$
	x	1	0	−1	0	$\frac{1}{4}$	$\frac{1}{4}$	0	
	y	0	1	−3	0	$-\frac{1}{4}$	$\frac{3}{4}$	0	
		0	0	−34	0	$\frac{3}{4}$	$\frac{31}{4}$	0	

↑

	x	y	z	u	v	w	
z	0	0	1	$\frac{1}{5}$	0	$-\frac{1}{5}$	20
x	1	0	0	$\frac{1}{5}$	$\frac{1}{4}$	$\frac{1}{20}$	20
y	0	1	0	$\frac{3}{5}$	$-\frac{1}{4}$	$\frac{3}{20}$	60
	0	0	0	$\frac{34}{5}$	$\frac{3}{4}$	$\frac{19}{20}$	680

The optimal solution found is $x = 20$, $y = 60$, $z = 20$, with the maximal value of the objective function equal to 680.

7. The optimal solution to the problem is $x_1 = b_1, ..., x_n = b_n$. After introducing a slack variable s_i in the ith inequality $x_i = b_i$ to get to the standard form and starting with $x_1 = 0, ..., x_n = 0$, $s_1 = b_1, ..., s_n = b_n$, the simplex method will need n iterations to get to an optimal solution $x_1 = b_1, ..., x_n = b_n, s_1 = 0, ..., s_n = 0$. It follows from the fact that on each iteration the simplex method replaces only one basic variable. Here, on each of its n iterations, it will replace some slack variable s_i by the corresponding x_i.

8. The continuous version of the knapsack problem can be solved by the simplex method, because it is a special case of the general linear programming problem (see Example 2 in Section 6.6). However, it is hardly a good method for solving this problem because it can be solved more efficiently by a much simpler algorithm based on the greedy approach. You may want to design such an algorithm by yourself before looking it up in the book.
   The 0-1 version of the knapsack problem cannot be solved by the simplex method because of the integrality (0-1) constraints imposed on the problem's variables.

9. The assertion follows immediately from the fact that if $x' = (x'_1, ..., x'_n)$ and $x'' = (x''_1, ..., x''_n)$ are two distinct optimal solutions to the same linear programming problem, then any of the infinite number of points of the line segment with endpoints at x' and x'' will be an optimal solution to this problem as well. Indeed, let x^t be such a point:

$$x^t = tx' + (1-t)x'' = (tx'_1 + (1-t)x''_1, ..., tx'_n + (1-t)x''_n), \text{ where } 0 \le t \le 1.$$

First, x^t will satisfy all the constraints of the problem, whether they are linear inequalities or linear equations, because both x' and x'' do. Indeed, let the ith constraint be inequality $\sum_{j=1}^{n} a_{ij}x_j \le b_i$. Then $\sum_{j=1}^{n} a_{ij}x'_j \le b_i$ and $\sum_{j=1}^{n} a_{ij}x''_j \le b_i$. Multiplying these inequalities by t and $1-t$, respectively, and adding the results, we obtain

$$t\sum_{j=1}^{n} a_{ij}x'_j + (1-t)\sum_{j=1}^{n} a_{ij}x''_j \le tb_i + (1-t)b_i$$

or

$$\sum_{j=1}^{n}(ta_{ij}x'_j + (1-t)a_{ij}x''_j) = \sum_{j=1}^{n} a_{ij}(tx'_j + (1-t)x''_j) = \sum_{j=1}^{n} a_{ij}x^t_j \le b_i,$$

i.e., x satisfies the inequality. The same argument holds for inequalities $\sum_{j=1}^{n} a_{ij}x_j \ge b_i$ and equations $\sum_{j=1}^{n} a_{ij}x_j = b_i$.
Second, $x^t = tx' + (1-t)x''$ will maximize the value of the objective function. Indeed, if the maximal value of the objective function is z^*, then

$$\sum_{j=1}^{n} c_j x'_j = z^* \quad \text{and} \quad \sum_{j=1}^{n} c_j x''_j = z^*.$$

Multiplying these equalities by t and $1-t$, respectively, and adding the results, we will obtain

$$t\sum_{j=1}^{n} c_j x'_j + (1-t)\sum_{j=1}^{n} c_j x''_j = tz^* + (1-t)z^*$$

or

$$\sum_{j=1}^{n}(tc_j x'_j + (1-t)c_j x''_j) = \sum_{j=1}^{n} c_j(tx'_j + (1-t)x''_j) = \sum_{j=1}^{n} c_j x^t_j = z^*.$$

I.e., we proved that x^t does maximize the objective function and hence will be an optimal solution to the problem in question for any $0 \le t \le 1$.

Note: What we actually proved is the fact that the set of optimal solutions to a linear programming problem is convex. And any nonempty convex set can contain either a single point or infinitely many points.

10. a. A linear programming problem

$$\begin{array}{rl} \text{maximize} & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \text{for } i = 1, 2, ..., m \\ & x_1, x_2, ..., x_n \ge 0 \end{array}$$

can be compactly written using the matrix notations as follows:

$$\begin{array}{rl} \text{maximize} & cx \\ \text{subject to} & Ax \le b \\ & x \ge 0, \end{array}$$

where

$$c = \begin{bmatrix} c_1 & ... & c_n \end{bmatrix}, \qquad x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \qquad A = \begin{bmatrix} a_{11} & ... & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & ... & a_{mn} \end{bmatrix}, \qquad b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix},$$

$Ax \le b$ holds if and only if each coordinate of the product Ax is less than or equal to the corresponding coordinate of vector b, and $x \ge 0$ is shorthand for the nonnegativity requirement for all the variables. The ***dual*** can be written as follows:

$$\begin{array}{rl} \text{minimize} & b^T y \\ \text{subject to} & A^T y \ge c^T \\ & y \ge 0, \end{array}$$

where b^T is the transpose of b (i.e., $b^T = [b_1, ..., b_m]$), c^T is the transpose of c (i.e., c^T is the column-vector made up of the coordinates of the row-vector c), A^T is the transpose of A (i.e., the $n \times m$ matrix whose jth row is the jth column of matrix A, $j = 1, 2, ..., n$), and y is the vector-column of m new unknowns $y_1, ..., y_m$.

b. The dual of the linear programming problem

$$\begin{array}{rl} \text{maximize} & x_1 + 4x_2 - x_3 \\ \text{subject to} & x_1 + x_2 + x_3 \le 6 \\ & x_1 - x_2 - 2x_3 \le 2 \\ & x_1, x_2, x_3 \ge 0 \end{array}$$

is

$$\begin{array}{rl} \text{minimize} & 6y_1 + 2y_2 \\ \text{subject to} & y_1 + y_2 \ge 1 \\ & y_1 - y_2 \ge 4 \\ & y_1 - 2y_2 \ge -1 \\ & y_1, y_2 \ge 0. \end{array}$$

c. The standard form of the primal problem is

$$\begin{array}{ll} \text{maximize} & x_1 + 4x_2 - x_3 \\ \text{subject to} & x_1 + x_2 + x_3 + x_4 \qquad = 6 \\ & x_1 - x_2 - 2x_3 + \qquad x_5 = 2 \\ & x_1, x_2, x_3, x_4, x_5 \geq 0. \end{array}$$

The simplex method yields the following tableaux:

	x_1	x_2	x_3	x_4	x_5	
← x_4	1	1	1	1	0	6
x_5	1	−1	−2	0	1	2
	−1	−4	1	0	0	0

↑

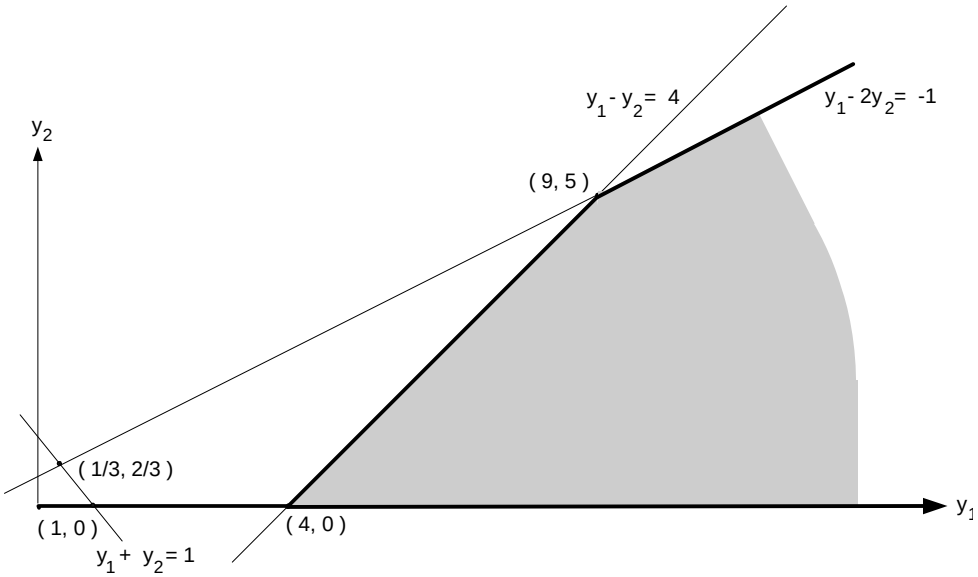
$\theta_{x_4} = \frac{6}{1}$

	x_1	x_2	x_3	x_4	x_5	
x_2	1	1	1	1	0	6
x_5	2	0	−1	1	1	8
	3	0	5	4	0	24

The found optimal solution is $x_1 = 0$, $x_2 = 6$, $x_3 = 0$.

Since the dual problem has just two variables, it is easier to solve it

geometrically. Its feasible region is presented in the following figure:



Although it is unbounded, the minimization problem in question does have a finite optimal solution $y_1 = 4$, $y_2 = 0$. Note that the optimal values of the objective functions in the primal and dual problems are equal to each other:

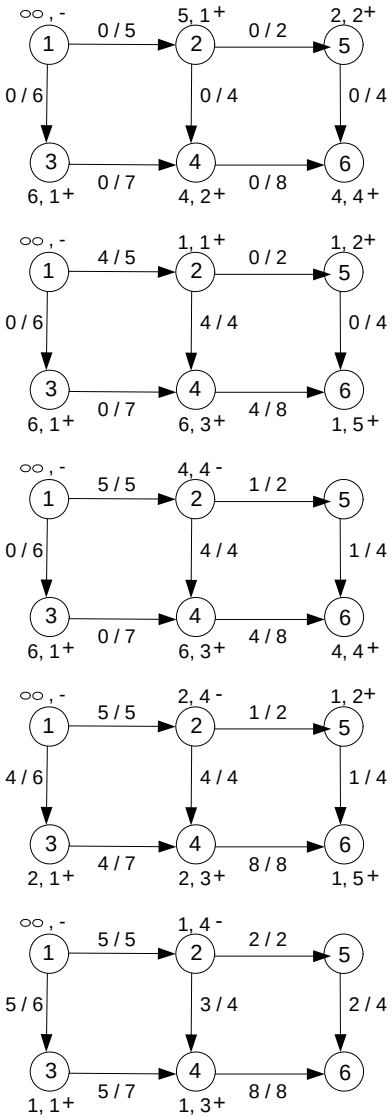
$$0 + 4 \cdot 6 - 0 = 6 \cdot 4 + 2 \cdot 0.$$

This is the principal assertion of the Duality Theorem, one of the most important facts in linear programming theory.

Solutions to Exercises 10.2

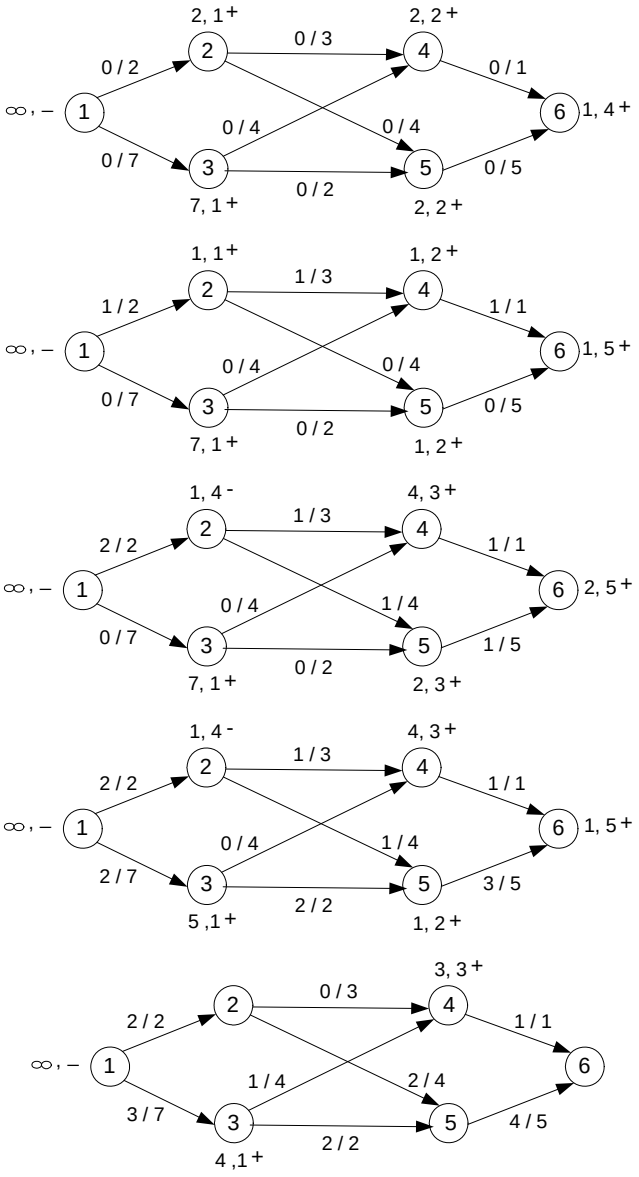
1. The definition of a source implies that a vertex is a source if and only if there are no negative elements in its row in the modified adjacency matrix. Similarly, the definition of a sink implies that a vertex is a sink if and only if there are no positive elements in its row of the modified adjacency matrix. Thus a simple scan of the adjacency matrix rows solves the problem in $O(n^2)$ time, where n is the number of vertices in the network.

2. a. Here is an application of the shortest-augmenting path algorithm to the network of Problem 2a:



The maximum flow is shown on the last diagram above. The minimum cut found is $\{(2,5\},(4,6)\}$.

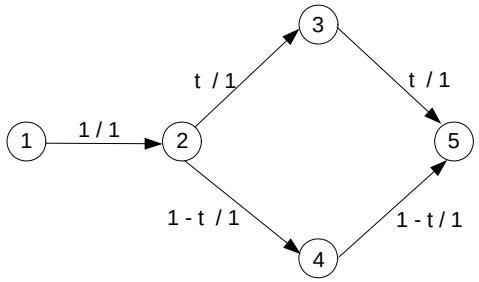
b. Here is an application of the shortest-augmenting path algorithm to the network of Problem 10.2b:



The maximum flow of value 5 is shown on the last diagram above. The

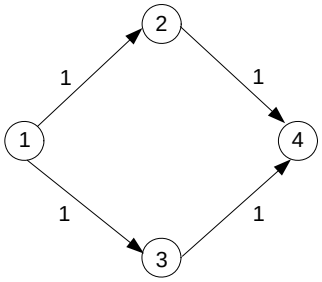
minimum cut found is $\{(1,2),(3,5),(4,6)\}$.

3. a. The maximum-flow problem may have more than one optimal solution. In fact, there may be infinitely many of them if we allow (as the definition does) non-integer edge flows. For example, for any $0 \le t \le 1$, the flow depicted in the diagram below is a maximum flow of value 1. Exactly two of them—for $t = 0$ and $t = 1$—are integer flows.



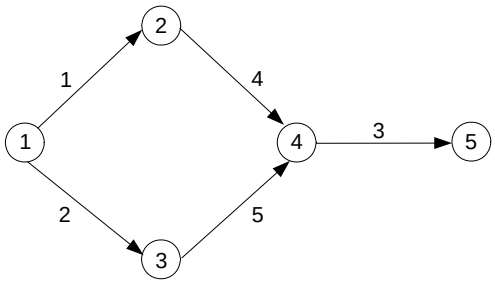
The answer does not change for networks with distinct capacities: e.g., consider the previous example with the capacities of edges $(2,3)$, $(3,5)$, $(2,4)$, and $(4,5)$ changed to, say, $2, 3, 4$, and 5, respectively.

b. The answer for the number of distinct minimum cuts is analogous to that for maximum flows: there can be more than one of them in the same network (though, of course, their number must always be finite because the number of all edge subsets is finite to begin with). For example, the network



has four minimum cuts: $\{(1,2),(1,3)\}$, $\{(1,2),(3,4)\}$, $\{(1,3),(2,4)\}$, and $\{(2,4),(3,4)\}$. The answer does not change for networks with distinct edge

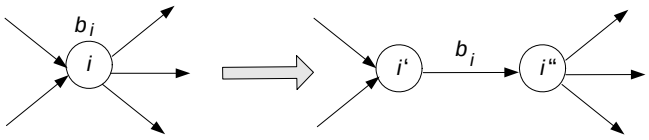
capacities. For example, the network



has two minimum cuts: $\{(1,2),(1,3)\}$ and $\{(4,5)\}$.

4. a. Add two vertices to the network given to serve as the source and sink of the new network, respectively. Connect the new source to each of the original sources and each of the original sinks to the new sink with edges of some large capacity M. (It suffices to take M greater than or equal to the sum of the capacities of the edges leaving each source of the original network.)

   b. Replace each intermediate vertex i with an upper bound b_i on a flow amount that can flow through it with two vertices i' and i'' connected by an edge of capacity b_i as shown below:



   Note that all the edges entering and leaving i in the original network should enter i' and leave i'', respectively, in the new one.

5. The problem can be solved by calling *TreeFlow*$(T(r), \infty)$, where
   Algorithm *TreeFlow*$(T(r), v)$
   //Finds a maximum flow for tree $T(r)$ rooted at r,
   //whose value doesn't exceed v (available at the root),
   //and returns its value
   if r is a leaf *maxflowval* $\leftarrow v$
   else
   maxflowval $\leftarrow 0$
   for every child c of r **do**
   $x_{rc} \leftarrow$ *TreeFlow*$(T(c), \min\{u_{rc}, v\})$
   $v \leftarrow v - x_{rc}$
   maxflowval $\leftarrow$ *maxflowval* $+ x_{rc}$
   return *maxflowval*

The efficiency of the algorithm is clearly linear because a $\Theta(1)$ call is made for each of the nodes of the tree.

6. a. Adding the $n-2$ equalities expressing the flow conservation requirements yields

$$\sum_{i=2}^{n-1} \sum_{j} x_{ji} = \sum_{i=2}^{n-1} \sum_{j} x_{ij}.$$

For any edge from an intermediate vertex p to an intermediate vertex q $(2 \le p, q \le n-1)$, the edge flow x_{pq} occurs once in both the left- and right-hand sides of the last equality and hence will cancel out. The remaining terms yield:

$$\sum_{i=2}^{n-1} x_{1i} = \sum_{i=2}^{n-1} x_{in}.$$

Adding x_{1n} to both sides of the last equation, if there is an edge from source 1 to sink n, results in the desired equality:

$$\sum_{i} x_{1i} = \sum_{i} x_{in}.$$

b. Summing up the flow-value definition $v = \sum_{j} x_{1j}$ and the flow-conservation requirement $\sum_{j} x_{ji} = \sum_{j} x_{ij}$ for every $i \in X$ $(i > 1)$, we obtain

$$v + \sum_{i:\ i \in X,\ i>1} \sum_{j} x_{ji} = \sum_{j} x_{1j} + \sum_{i:\ i \in X,\ i>1} \sum_{j} x_{ij},$$

or, since there are no edges entering the source,

$$v + \sum_{i \in X} \sum_{j} x_{ji} = \sum_{i \in X} \sum_{j} x_{ij}.$$

Moving the summation from the left-hand side to the right-hand side and splitting the sum into the sum over the vertices in X and the sum over the vertices in $\bar{X}$, we obtain:

$$\begin{aligned} v &= \sum_{i \in X} \sum_{j} x_{ij} - \sum_{i \in X} \sum_{j} x_{ji} \\ &= \sum_{i \in X} \sum_{j \in X} x_{ij} + \sum_{i \in X} \sum_{j \in \bar{X}} x_{ij} - \sum_{i \in X} \sum_{j \in X} x_{ji} - \sum_{i \in X} \sum_{j \in \bar{X}} x_{ji} \\ &= \sum_{i \in X} \sum_{j \in \bar{X}} x_{ij} - \sum_{i \in X} \sum_{j \in \bar{X}} x_{ji}, \qquad \text{Q.E.D.} \end{aligned}$$

Note that equation (10.9) expresses this general property for two special cuts: $C_1(X_1, \bar{X}_1)$ induced by $X_1 = \{1\}$ and $C_2(X_2, \bar{X}_2)$ induced by $X_2 = V - \{n\}$.

7. a.

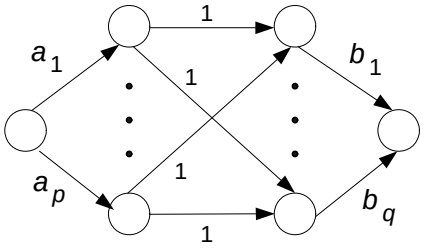
$$\begin{aligned} \text{maximize} \quad & v = x_{12} + x_{14} \\ \\ \text{subject to} \quad & x_{12} - x_{23} - x_{25} = 0 \\ & x_{23} + x_{43} - x_{36} = 0 \\ & x_{14} - x_{43} = 0 \\ & x_{25} - x_{56} = 0 \\ & 0 \le x_{12} \le 2 \\ & 0 \le x_{14} \le 3 \\ & 0 \le x_{23} \le 5 \\ & 0 \le x_{25} \le 3 \\ & 0 \le x_{36} \le 2 \\ & 0 \le x_{43} \le 1 \\ & 0 \le x_{56} \le 4. \end{aligned}$$

b. The optimal solution is $x_{12} = 2$, $x_{14} = 1$, $x_{23} = 1$, $x_{25} = 1$, $x_{36} = 2$, $x_{43} = 1$, $x_{56} = 1$.

8. n/a

9. n/a

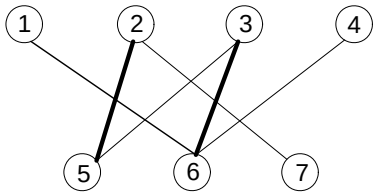
10. Solve the maximum flow problem for the following network:



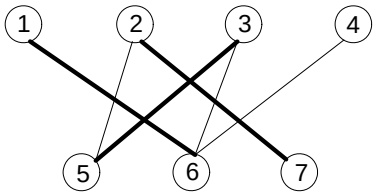
If the maximum flow value is equal to $\sum_{i=1}^{p} a_i$, then the problem has a solution indicated by the full edges of capacity 1 in the maximum flow; otherwise, the problem does not have a solution.

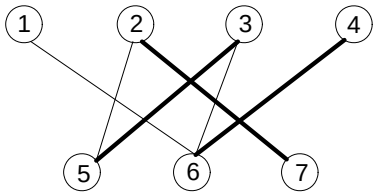
Solutions to Exercises 10.3

1. a. The matching given in the exercise is reproduced below:

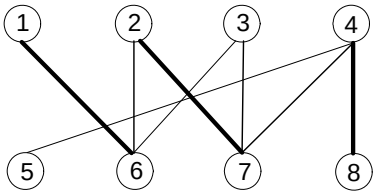


Its possible augmentations are:



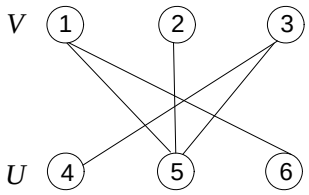


b. No augmentation of the matching given in part b (reproduced below) is possible.

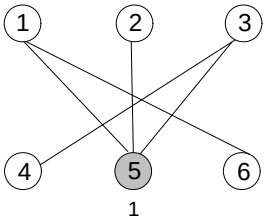


This conclusion can be arrived at either by applying the maximum-matching algorithm or by simply noting that only one of vertices 5 and 8 can be matched (and, hence, no more than three of vertices 5, 6, 7, and 8 can be matched in any matching).

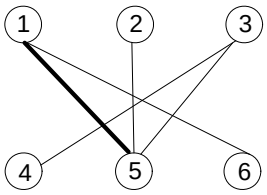
2. Here is a trace of the maximum-matching algorithm applied to the bipartite graph in question:



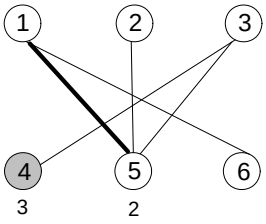
Queue: 1 2 3



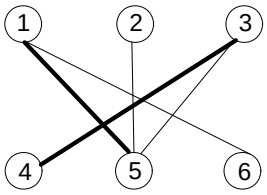
Queue: 1 2 3
↑
Augment from 5



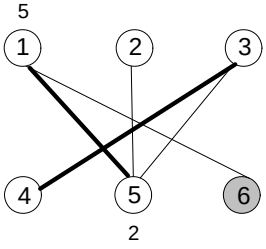
Queue: 2 3



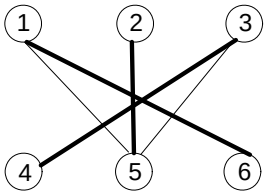
Queue: 2 3 5
↑ ↑
Augment from 4



Queue: 2



Queue: 2 5 1
↑ ↑ ↑
Augment from 6



No free vertices ⇒ maximum (here perfect) matching

3. a. The largest cardinalilty of a matching is n when all the vertices of a graph are matched (perfect matching). For example, if $V = \{v_1, v_2, ..., v_n\}$,

$U = \{u_1, u_2, ..., u_n\}$ and $E = \{(v_1, u_1), (v_2, u_2), ..., (v_n, u_n)\}$, then $M = E$ is a perfect matching of size n.
The smallest cardinality of a matching is 1. (It cannot be zero because the number of edges is assumed to be at least $n \geq 1$.) For example, in the bipartite graph with $V = \{v_1, v_2, ..., v_n\}$, $U = \{u_1, u_2, ..., u_n\}$, and $E = \{(v_1, u_1), (v_1, u_2), ..., (v_1, u_n)\}$, the size of any matching is 1.

b. Consider $K_{n,n}$, the bipartite graph in which each of the n vertices in V is connected to each of the n vertices in U. To obtain a perfect matching, there are n possible mates for vertex v_1, $n-1$ possible remaining mates for v_2, and so on until there is just one possible remaining mate for v_n. Therefore the total number of distinct perfect matchings for $K_{n,n}$ is $n(n-1)...1 = n!$.

The smallest number of distinct maximum matchings is 1. For example, if $V = \{v_1, v_2, ..., v_n\}$, $U = \{u_1, u_2, ..., u_n\}$, and $E = \{(v_1, u_1), (v_2, u_2), ..., (v_n, u_n)\}$, $M = E$ is the only perfect (and hence maximum) matching.

4. a. (i) For $V = \{1, 2, 3, 4\}$, the inequality obviously fails for $S = V$ since $|R(S)| = |U| = 3$ while $|S| = 4$.
Hence, according to Hall's Marriage Theorem, there is no matching that matches all the vertices of the set $\{1, 2, 3, 4\}$.

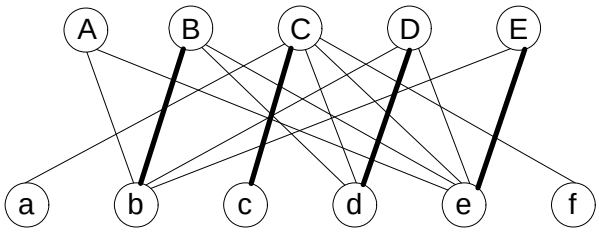
(ii) For subsets S of $V = \{5, 6, 7\}$, we have the following table:

S	$R(S)$	$\lvert R(S)\rvert \geq \lvert S\rvert$
{5}	{1,2,3}	$3 \geq 1$
{6}	{1}	$1 \geq 1$
{7}	{2,3,4}	$3 \geq 1$
{5,6}	{1,2,3}	$3 \geq 2$
{5,7}	{1,2,3,4}	$4 \geq 2$
{6,7}	{1,2,3,4}	$4 \geq 2$
{5,6,7}	{1,2,3,4}	$4 \geq 3$

Hence, according to Hall's Marriage Theorem, there is a matching that matches all the vertices of the set $\{5, 6, 7\}$. (Obviously, such a matching must be maximal.)

b. Since Hall's theorem requires checking an inequality for each subset $S \subseteq V$, the worst-case time efficiency of an algorithm based on it would be in $\Omega(2^{|V|})$. A much better solution is to find a maximum matching M^* (e.g., by the section's algorithm) and return "yes" if $|M^*| = |V|$ and "no" otherwise.

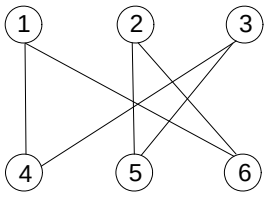
5. It is convenient to model the situation by a bipartite graph $G = \langle V, U, E \rangle$ where V represents the committees, U represents the committee members, and $(v, u) \in E$ if and only if u belongs to committee v:



There exists no matching that would match all the vertices of the set V. One way to prove it is based on Hall's Marriage Theorem (see Problem 4) whose necessary condition is violated for set $S = \{A, B, D, E\}$ with $R(S) = \{b, d, e\}$. Another way is to select a matching such as $M = \{(B, b), (C, c), (D, d), (E, e)\}$ (shown in bold) and check that the maximum-matching algorithm fails to find an augmenting path for it.

6. Add one source vertex s and connect it to each of the vertices in the set V by directed edges leaving s. Add one sink vertex t and connect each of the vertices in the set U to t by a directed edge entering t. Direct all the edges of the original graph to point from V to U. Assign 1 as the capacity of every edge in the network. A solution to the maximum-flow problem for the network yields a maximum matching for the original bipartite graph: it consists of the full edges (edges with the unit flow on them) between vertices of the original graph.

7. The greedy algorithm does not always find a maximum matching. As a counterexample, consider the bipartite graph shown below:



Since all its vertices have the same degree of 2, we can order them in numerical order of their labels: 1, 2, 3, 4, 5, 6. Using the same rule to break ties for selecting mates for vertices on the list, the greedy algorithm yields the matching $M = \{(1, 4), (2, 5)\}$, which is smaller than, say, $M^* = \{(1, 4), (2, 6), (3, 5)\}$.

8. A crucial observation is that for any edge between a leaf and its parent there is a maximum matching containing this edge. Indeed, consider a leaf v and its parent p. Let M be a maximum matching in the tree. If (v, p) is in M, we have a maximum matching desired. If M does not include (v, p), it must contain an edge (u, p) from some vertex $u \neq v$ to p because otherwise we could have added (v, p) to M to get a larger matching. But then simply replacing (u, p) by (v, p) in the matching M yields a maximum marching containing (v, p). This operation is to be repeated recursively for the smaller forest obtained.

   Based on this observation, Manber ([Man89], p. 431) suggested the following recursive algorithm. Take an arbitrary leaf v and match it to its parent.p. Remove from the tree both v and p along with all the edges incident to p. Also remove all the former sibling leaves of v, if any. This operation is to be repeated recursively for the smaller forest obtained.

   Thieling Chen [Thieling Chen, "Maximum Matching and Minimum Vertex Covers of Trees," *The Western Journal of Graduate Research*, vol. 10, no. 1, 2001, pp. 10-14] suggested to implement the same idea by processing vertices of the tree in the reverse BFS order (i.e., bottom up and right to left across each level) as in the following pseudocode:

   Algorithm *Tree-Max-Matching*
   //Constructs a maximum matching in a free tree
   //Input: A tree T
   //Output: A maximum cardinality matching M in T
   initialize matching M to the empty set
   starting at an arbitrary vertex, traverse T by BFS, numbering the visited vertices sequentially from 0 to $n-1$ and saving pointers to a visited vertex and its parent
   for $i \leftarrow n-1$ **downto** 0 **do**
       if vertex numbered i and its parent are both not marked
           add the edge between them to M
           mark the parent
   return M

   The time efficiency of this algorithm is obviously in $\Theta(n)$, where n is thenumber of vertices in an input tree.

10. No domino tiling of such a board is possible. Think of the board as being an 8×8 chessboard (with two missing squares at the diagonally opposite corners) whose squares of the board colored alternatingly black and white. Since a single domino would cover (match) exactly one black and one white square, no tiling of the board is possible because it has 32 squares of one color and 30 squares of the other color.

Solutions to Exercises 10.4

1. There are the total of $3! = 6$ one-one matchings of two disjoint 3-element sets:

	A	B	C
α	$\boxed{1,3}$	$2,2$	$3,1$
β	$3,1$	$\boxed{1,3}$	$2,2$
γ	$2,2$	$3,1$	$\boxed{1,3}$

$\{(\alpha, A), (\beta, B), (\gamma, C)\}$ is stable: no other cell can be blocking since each man has his best choice. This is obviously the man-optimal matching.

	A	B	C
α	$\boxed{1,3}$	$2,2$	$3,1$
β	$3,1$	$1,3$	$\boxed{2,2}$
γ	$2,2$	$\boxed{3,1}$	$1,3$

$\{(\alpha, A), (\beta, C), (\gamma, B)\}$ is unstable: (γ, A) is a blocking pair.

	A	B	C
α	$1,3$	$\boxed{2,2}$	$3,1$
β	$\boxed{3,1}$	$1,3$	$2,2$
γ	$2,2$	$3,1$	$\boxed{1,3}$

$\{(\alpha, B), (\beta, A), (\gamma, C)\}$ is unstable: (β, C) is a blocking pair.

	A	B	C
α	$1,3$	$\boxed{2,2}$	$3,1$
β	$3,1$	$1,3$	$\boxed{2,2}$
γ	$\boxed{2,2}$	$3,1$	$1,3$

$\{(\alpha, B), (\beta, C), (\gamma, A)\}$ is stable: all the other cells contain a 3 (the lowest rank) and hence cannot be a blocking pair. This is neither a man-optimal nor a woman-optimal matching since it's inferior to $\{(\alpha, A), (\beta, B), (\gamma, C)\}$ for the men and inferior to $\{(\alpha, C), (\beta, A), (\gamma, B)\}$ for the women.

	A	B	C
α	$1,3$	$2,2$	$\boxed{3,1}$
β	$\boxed{3,1}$	$1,3$	$2,2$
γ	$2,2$	$\boxed{3,1}$	$1,3$

$\{(\alpha, C), (\beta, A), (\gamma, B)\}$ is stable: no other cell can be blocking since each woman has her best choice. This is obviously the woman-optimal matching.

	A	B	C
α	$1, 3$	$2, 2$	$\boxed{3,1}$
β	$3, 1$	$\boxed{1,3}$	$2, 2$
γ	$\boxed{2,2}$	$3, 1$	$1, 3$

$\{(\alpha, C), (\beta, B), (\gamma, A)\}$ is unstable: (α, B) is a blocking pair.

2. **Stability-checking algorithm**
Input: A marriage matching M of n (m, w) pairs along with rankings of the
women by each man and rankings of the men by each woman
Output: "yes" if the input is stable and a blocking pair otherwise
for $m \leftarrow 1$ **to** n **do**
for each w such that m prefers w to his mate in M **do**
if w prefers m to her mate in M
return (m, w)
return "yes"

With appropriate data structures, it is not difficult to implement this algorithm to run in $O(n^2)$ time. For example, the mates of the men and the mates of the women in a current matching can be stored in two arrays of size n and all the preferences can be stored in the $n \times n$ ranking matrix containing two rankings in each cell.

3. a.

		A	B	C	
Free men:	α	1,3	$2,2$	$3,1$	α proposed to A
α, β, γ	β	$3,1$	$1,3$	$2,2$	A accepted
	γ	$2,2$	$3,1$	$1,3$	

		A	B	C	
Free men:	α	1,3	$2,2$	$3,1$	β proposed to B
β, γ	β	$3,1$	1,3	$2,2$	B accepted
	γ	$2,2$	$3,1$	$1,3$	

		A	B	C	
Free men:	α	1,3	$2,2$	$3,1$	γ proposed to C
γ	β	$3,1$	1,3	$2,2$	C accepted
	γ	$2,2$	$3,1$	1,3	

The (man-optimal) stable marriage matching is $M = \{(\alpha, A), (\beta, B), (\gamma, C)\}$.

b.

		A	B	C	
Free women:	α	$1,3$	$2,2$	$3,1$	A proposed to β
A, B, C	β	3,1	$1,3$	$2,2$	β accepted
	γ	$2,2$	$3,1$	$1,3$	

		A	B	C	
Free women:	α	$1,3$	$2,2$	$3,1$	B proposed to γ
B, C	β	3,1	$1,3$	$2,2$	γ accepted
	γ	$2,2$	3,1	$1,3$	

		A	B	C	
Free women:	α	$1,3$	$2,2$	3,1	C proposed to α
C	β	3,1	$1,3$	$2,2$	α accepted
	γ	$2,2$	3,1	$1,3$	

The (woman-optimal) stable marriage matching is $M = \{(\beta, A), (\gamma, B), (\alpha, C)\}$.

4.

iteration 1

Free men: $\alpha, \beta, \gamma, \delta$

	A	B	C	D
α	1,3	$2,3$	$3,2$	$4,3$
β	$1,4$	$4,1$	$3,4$	$2,2$
γ	$2,2$	$1,4$	$3,3$	$4,1$
δ	$4,1$	$2,2$	$3,1$	$1,4$

α proposed to A; A accepted

iteration 2

Free men: β, γ, δ

	A	B	C	D
α	1,3	$2,3$	$3,2$	$4,3$
β	$1,4$	$4,1$	$3,4$	$2,2$
γ	$2,2$	$1,4$	$3,3$	$4,1$
δ	$4,1$	$2,2$	$3,1$	$1,4$

β proposed to A; A rejected

iteration 3

Free men: β, γ, δ

	A	B	C	D
α	1,3	$2,3$	$3,2$	$4,3$
β	$1,4$	$4,1$	$3,4$	2,2
γ	$2,2$	$1,4$	$3,3$	$4,1$
δ	$4,1$	$2,2$	$3,1$	$1,4$

β proposed to D; D accepted

iteration 4

Free men: γ, δ

	A	B	C	D
α	1,3	$2,3$	$3,2$	$4,3$
β	$1,4$	$4,1$	$3,4$	2,2
γ	$2,2$	1,4	$3,3$	$4,1$
δ	$4,1$	$2,2$	$3,1$	$1,4$

γ proposed to B; B accepted

iteration 5

Free men: δ

	A	B	C	D
α	1,3	$2,3$	$3,2$	$4,3$
β	$1,4$	$4,1$	$3,4$	2,2
γ	$2,2$	1,4	$3,3$	$4,1$
δ	$4,1$	$2,2$	$3,1$	$\underline{1,4}$

δ proposed to D; D rejected

iteration 6

Free men: δ

	A	B	C	D
α	1,3	$2,3$	$3,2$	$4,3$
β	$1,4$	$4,1$	$3,4$	2,2
γ	$2,2$	$1,4$	$3,3$	$4,1$
δ	$4,1$	2,2	$3,1$	$1,4$

δ proposed to B; B replaced γ with δ

iteration 7

Free men: γ

	A	B	C	D
α	$1,3$	$2,3$	$3,2$	$4,3$
β	$1,4$	$4,1$	$3,4$	2,2
γ	2,2	$1,4$	$3,3$	$4,1$
δ	$4,1$	2,2	$3,1$	$1,4$

γ proposed to A; A replaced α with γ

iteration 8

Free men: α

	A	B	C	D
α	$1,3$	$\underline{2,3}$	$3,2$	$4,3$
β	$1,4$	$4,1$	$3,4$	2,2
γ	2,2	$1,4$	$3,3$	$4,1$
δ	$4,1$	2,2	$3,1$	$1,4$

α proposed to B; B rejected

iteration 9

Free men: α

	A	B	C	D
α	$1,3$	$2,3$	3,2	$4,3$
β	$1,4$	$4,1$	$3,4$	2,2
γ	2,2	$1,4$	$3,3$	$4,1$
δ	$4,1$	2,2	$3,1$	$1,4$

α proposed to C; C accepted

Free men: none

$M = \{(\alpha, C), (\beta, D), (\gamma, A), (\delta, B)\}$

5. a. The worst-case time efficiency of the algorithm is $\Theta(n^2)$. On the one hand, the total number of the proposals, $P(n)$, cannot exceed n^2, the total number of possible partners for n men, because a man does not propose to the same woman more than once. On the other hand, for the instance of size n where all the men and women have the identical preference list 1, 2, ..., n, $P(n) = \sum_{i=1}^{n} i = n(n+1)/2$. Thus, if $P_w(n)$ is the number of proposals made by the algorithm in the worst case,

$$n(n+1)/2 \le P_w(n) \le n^2,$$

i.e., $P_w(n) \in \Theta(n^2)$.

b. The best-case time efficiency of the algorithm is $\Theta(n)$: the algorithm makes the minimum of n proposals, one by each man, on the input that ranks first a different woman for each of the n men.

6. Assume that there are two distinct man-optimal solutions to an instance of the stable marriage problem. Then there must exist at least one man m matched to two different women w_1 and w_2 in these solutions. Since no ties are allowed in the rankings, m must prefer one of these two women to the other, say, w_1 to w_2. But then the marriage matching in which m is matched with w_2 is not man-optimal in contradiction to the assumption.

Of course, a woman-optimal solution is unique too due to the complete symmetry of these notions.

7. Assume that on the contrary there exists a man-optimal stable matching M in which some woman w doesn't have the worst possible partner in a stable matching, i.e., w prefers her partner m in M to her partner $\bar{m}$ in another stable matching $\bar{M}$. Since $(m, w) \notin \bar{M}$, m must prefer his partner $\bar{w}$ in $\bar{M}$ to w because otherwise (m, w) would be a blocking pair for $\bar{M}$. But this contradicts the assumption that M is a man-optimal stable matching in which every man, including m, has the best possible partner in a stable matching.

8. n/a

9. n/a

10. Consider an instance of the problem of the roommates with four boys α, β, γ, and δ and the following preference lists ($*$ stands for any legitimate

rating):

boy	rank 1	rank 2	rank 3
α	β	γ	δ
β	γ	α	δ
γ	α	β	δ
δ	$*$	$*$	$*$

Any pairing would have to pair one of the boys α, β, γ with δ. But any such pairing would be unstable since whoever is paired with δ will want to move out and one of the other two boys, having him rated first, will prefer him to his current roommate. For example, if α is paired with δ then β and γ are paired too while γ prefers α to β (in addition to α preferring γ to δ).

Note: This example is from the seminal paper by D. Gale and L. S. Shapley "College Admissions and the Stability of Marriage", *American Mathematical Monthly*, vol. 69 (Jan. 1962), 9-15. For an in-depth discussion of the problem of the roommates see the monograph by D. Gusfield and R.W. Irwing *The Stable Marriage Problem*: *Structure and Algorithms,*. MIT Press, 1989.

Solutions to Exercises 11.1

1. In the initial position of the puzzle, the ith light disk has exactly i dark disks to the left of it ($i = 1, 2, ..., n$). Hence the total number of the dark disks that are to the left of the light disks is initially equal to $\sum_{i=1}^{n} i = n(n+1)/2$. There are no dark disks to the left of a light disk in the final state of the puzzle. Since one move can only swap two neighboring disks—and hence decrease the total number of dark disks to the left of a light disk by one—the puzzle requires at least $n(n+1)/2$ moves to be solved. This bound is tight because the brute-force algorithm (see the solution to Problem 14 in Exercises 3.1) makes exactly this number of moves.

2. Let $M(n)$ be the number of disk moves made by some algorithm solving the Tower of Hanoi problem. We'll prove by induction that

$$M(n) \geq 2^n - 1 \text{ for } n \geq 1.$$

   For the basis case of $n = 1$, $M(1) \geq 2^1 - 1$ holds. Assume now that the inequality holds for $n \geq 1$ disks and consider the case of $n+1$ disks. Before the largest disk can be moved, all n smaller disks must be in a tower on another peg. By the inductive assumption, it will require at least $2^n - 1$ disk moves. Moving the largest disk to the destination peg will take at least 1 move. After the largest disk is moved on the destination peg for the last time, the n smaller disks will have to be moved from its tower formation on top of the largest disk, which will require at least $2^n - 1$ moves by the inductive assumption. Therefore:

$$M(n+1) \geq (2^n - 1) + 1 + (2^n - 1) = 2^{n+1} - 1.$$

   The alternative proof via setting a recurrence for the minimum number of moves $M^*(n)$ follows essentially the same logic as the proof above.

3. a. All n elements of a given array need to be processed to find its largest element (otherwise, if an unprocessed element is larger than all the others, the output cannot be correct) and just one item needs to be produced (if just the value of the largest element or a position of the largest element needs to be returned). Hence the trivial lower bound is linear. It is tight because the standard one-pass algorithm for this problem is in $\Theta(n)$.

   b. Since the existence of an edge between all $n(n-1)/2$ pairs of vertices needs to be verified in the worst case before establishing completeness of a graph with n vertices, the trivial lower bound is quadratic. It is tight because this is the amount of work done by the brute-force algorithm that simply checks all the elements in the upper-triangular part of the matrix until either a zero is encountered or no unchecked elements are left.

c. The size of the problem's output is 2^n. Hence, the lower bound is exponential. The bound is tight because efficient algorithms for subset generation (Section 4.3) spend a constant time on each of them (except, possibly, for the first one).

d. The size of the problem's input is n while the output is just one bit. Hence, the trivial lower bound is linear. It is not tight: according to the known result quoted in the section, the tight lower bound for this problem is $n \log n$.

4. The answer is no. The problem can be solved with fewer weighings by dividing the coins into three rather than two subsets with about the same number of coins each. The information-theoretic argument, if used, has to take into account the fact that one weighing reduces uncertainty more than for the number-guessing problem because it can have three rather than two outcomes.

5. Every comparison of two (distinct) elements produces one "winner" and one "loser". If less than $n-1$ comparisons are made, at least two elements will be with no losses. Hence, it would be impossible to say which of them is the largest element.

6. Recall that an inversion in an array is any pair of its elements that are out of order, i.e., $A[i] > A[j]$ while $i < j$. The maximum number of inversions in an n-element array is attained when its elements are strictly decreasing; this maximum is equal to $\sum_{i=2}^{n}(i-1) = (n-1)n/2$. Since a swap of two adjacent elements can decrease the total number of inversions only by one, the worst-case number of such swaps required for sorting an n-element array will be equal to $(n-1)n/2$ as well. This bound is tight because it is attained by both bubble sort and insertion sort.

7. Consider the following rule for the adversary to follow: Divide the set of vertices of an input graph into two disjoints subsets U and W having $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ vertices respectively (e.g., by putting the first $\lfloor n/2 \rfloor$ vertices into U and the remaining $\lceil n/2 \rceil$ vertices into W). Whenever an algorithm inquires about an edge between two vertices, reply yes if and only if both vertices belong to either U or to W. If the algorithm stops before inquiring about a pair of vertices (u, v), where $u \in U$ and $w \in W$, with the positive answer about the graph's connectivity, present the disconnected graph with all possible edges between vertices in U, all possible edges between vertices in W, and no edges between vertices of U and W, including u and w. If the algorithm stops before inquiring about a pair of vertices (u, w), where $u \in U$ and $w \in W$, with the negative answer

about the graph's connectivity, present the connected graph with all possible edges between vertices in U, all possible edges between vertices in W, and the edge between u and w. Hence, any correct algorithm must inquire about at least $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \in \Omega(n^2)$ possible edges in the worst case.

The quadratic lower bound is tight because a depth-first search traversal solves the problem in $O(n^2)$ time.

Note: In fact, all $n(n-1)/2$ potential edges need to be checked in the worst case (see [Bra96, Sec. 12.3.2]).

8. Any comparison-based algorithm will need at least $2n$ comparisons to merge two arbitrary sorted lists of sizes n and $n+1$, respectively. The proof is obtained via the adversary argument similar to the one given in the section for the case of two n-element lists.

9. For $A = \begin{bmatrix} 1 & -1 \\ 2 & 3 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & 1 \\ -1 & 2 \end{bmatrix}$, the respective transposes are

$$A^T = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix} \text{ and } B^T = \begin{bmatrix} 0 & -1 \\ 1 & 2 \end{bmatrix}.$$

Hence, $X = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$ and $Y = \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix}$ are, respectively,

$$X = \begin{bmatrix} 0 & 0 & 1 & -1 \\ 0 & 0 & 2 & 3 \\ 1 & 2 & 0 & 0 \\ -1 & 3 & 0 & 0 \end{bmatrix} \text{ and } Y = \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 2 \\ 0 & 1 & 0 & 0 \\ -1 & 2 & 0 & 0 \end{bmatrix}.$$

Thus,

$$XY = \begin{bmatrix} 0 & 0 & 1 & -1 \\ 0 & 0 & 2 & 3 \\ 1 & 2 & 0 & 0 \\ -1 & 3 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 2 \\ 0 & 1 & 0 & 0 \\ -1 & 2 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 0 \\ -3 & 8 & 0 & 0 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 3 & 7 \end{bmatrix},$$

with the product AB produced in the upper left quadrant of XY.

10. a. The formula $XY = \frac{1}{4}[(X+Y)^2 - (X-Y)^2]$ does not hold for arbitrary square matrices because it relies on commutativity of multiplication (i.e., $XY = YX$):

$$\begin{aligned} \frac{1}{4}[(X+Y)^2 - (X-Y)^2] &= \frac{1}{4}[(X+Y)(X+Y) - (X-Y)(X-Y)] \\ &= \frac{1}{4}[2XY + 2YX] = \frac{1}{2}[XY + YX] \neq XY. \end{aligned}$$

b. The problem is solved by the following reduction formula

$$\begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}^2 = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix},$$

which allows us to obtain products AB and BA as a result of matrix squaring.

11. The element uniqueness problem can be reduced to the closest numbers problem. (After solving the latter, it suffices to check whether the distance between the closest numbers is zero to solve the former.) This implies that $\Omega(n \log n)$, the lower bound for the element uniqueness problem, is also a lower bound for the closest numbers problem. The presorting-based algorithm (see Example 1 in Section 6.1)—with an $n \log n$ sorting method such as mergesort—makes this lower bound tight for the closest numbers problem as well.

12. Sorting a list of numbers is a special case of the number placement problem. This implies that $\Omega(n \log n)$, the lower bound for sorting, is also a lower bound for the number placement problem. The presorting-based algorithm for the latter (see the solution to Problem 9 in Exercises 6.1)—with an $n \log n$ sorting method such as mergesort or heapsort— makes this lower bound tight for the number placement problem as well.

Solutions to Exercises 11.2

1. a. We'll prove by induction on h that that $2^h \geq l$ for any nonempty binary tree with height $h \geq 0$ and the number of leaves l. For the basis case of $h = 0$, we have $2^0 \geq 1$. For the general case, assume that $2^h \geq l$ holds for any binary tree whose height doesn't exceed h. Consider an arbitrary binary tree T of height $h+1$; let T_L and T_R be its left and right subtrees, respectively. (One of T_L and T_R, but not both of them, can be empty.) The heights of T_L and T_R cannot exceed h, and the number of leaves in T is equal to the number of leaves in T_L and T_R. Whether or not T_L is empty, $l(T_L) \leq 2^h$. Indeed, if it's not empty, it is true by the inductive assumption; if it is empty, $l(T_L) = 0$ and therefore smaller than 2^h. By the same reasons, $l(T_R) \leq 2^h$. Hence, we obtain the following inequality for the number of leaves $l(T)$ in T:

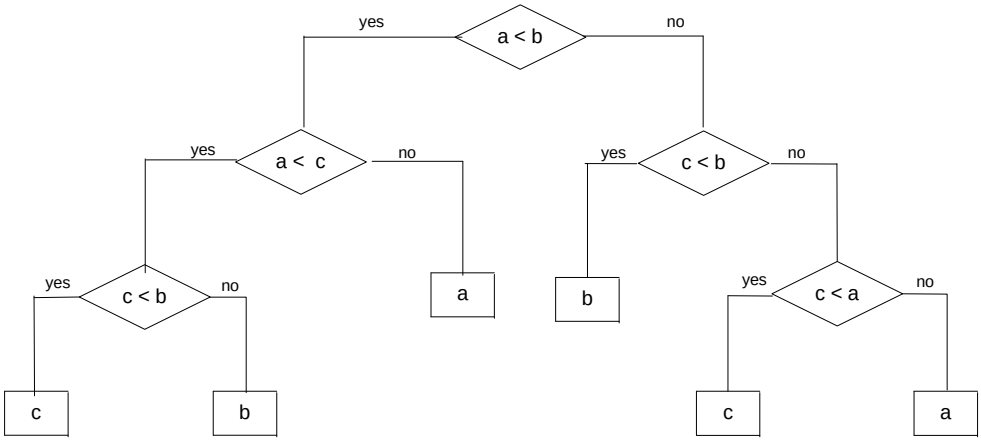
$$l(T) = l(T_L) + l(T_R) \leq 2^h + 2^h = 2^{h+1}.$$

   Taking binary logarithms of both hand sides of the proved inequality $l \leq 2^h$ yields $\log_2 l \leq h$ and, since h is an integer, $h \geq \lceil \log_2 l \rceil$.

   b. The proof is analogous to the one given for part a.

2. a. Since the problem has three possible outcomes, the information-theoretic lower bound is $\lceil \log_2 3 \rceil = 2$.

   b. Here is a decision tree for an algorithm for the problem of finding the median of a three-element set $\{a, b, c\}$:



   Note: The problem can also be solved by sorting a, b, c by one of the sorting algorithms.

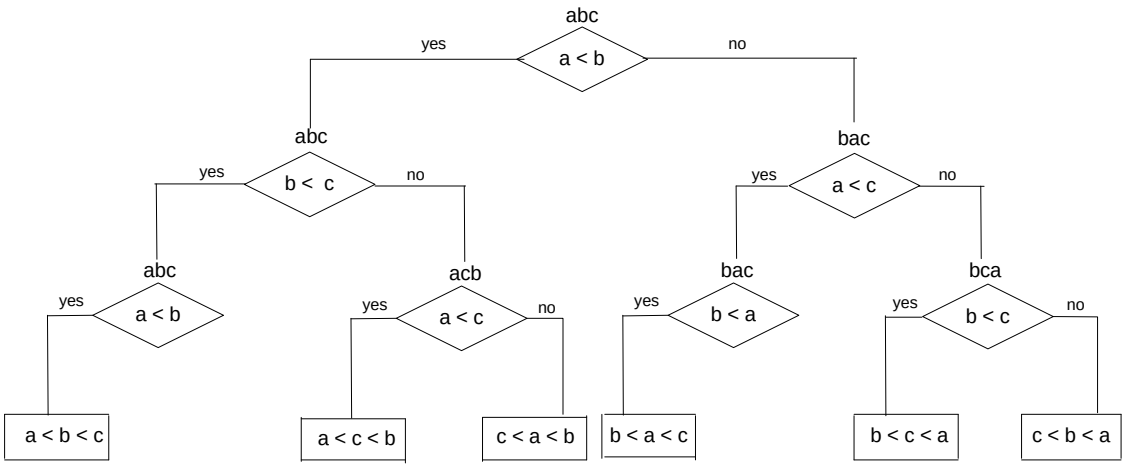
   c. It is impossible to find the median of three arbitrary real numbers in less than three comparisons (with a comparison-based algorithm). The first comparison will establish endpoints of some interval. The second

comparison (unless it's the same as the first one) will compare the third number with one of these endpoints. If that third number is compared to the left end of the interval and it is larger than it, it would be impossible to say without one more comparison whether this number or the right endpoint is the median. Similarly, if the third number is compared with the right end of the interval and it is smaller than it, it would be impossible to say without one more comparison whether this number or the left endpoint is the median. (This proof can be rephrased as an adversary argument.)

Note: As mentioned in Section 11.1, the following lower bound is known for comparison-based algorithms for finding the median of an n-element set (see, e.g., [Baa00, p.240]):

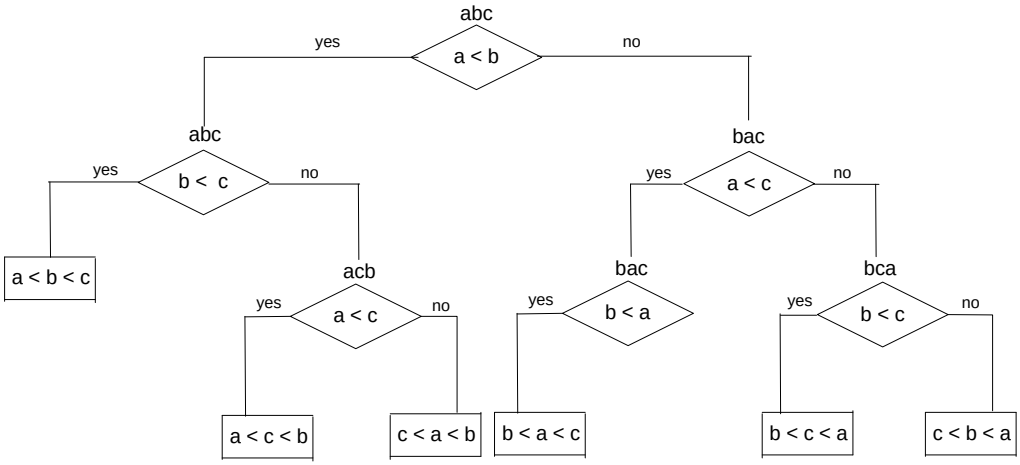
$$C_w(n) \geq \frac{3}{2}(n-1) \quad \text{for odd } n.$$

3. a. Here is a decision tree for sorting an array of three distinct elements a, b, and c by basic bubble sort:



The algorithm makes exactly three comparisons on any of its inputs.

b. Here is a decision tree for sorting an array of three distinct elements by enhanced bubble sort:



The algorithm makes three comparisons in the worst case and $(2+3+3+3+3+3)/6 = 2\frac{5}{6}$ comparisons in the average case.

4. $\lceil \log_2 4! \rceil = 5$. Mergesort (as described in Section 5.1) sorts any four-element array with no more than five comparisons: two comparisons are made to sort the two halves and no more than three comparisons are needed to merge them.

5. The following solution is presented by Knuth [KnuIII, pp. 183–184]. Let a, b, c, d, e be five distinct elements to be sorted. First, we compare a with b and c with d. Without loss of generality we can assume that $a < b$ and $c < d$. This can be depicted by the following digraph of five vertices, in which a path indicates an ordered subsequence:

$$\begin{array}{ccc} b & & d \\ \uparrow & & \uparrow \\ a & & c \end{array} \qquad e$$

Then we compare the larger elements of the two pairs; without loss of generality, we can assume that $b < d$:

$$\begin{array}{ccc} b & \rightarrow & d \\ \uparrow & & \uparrow \\ a & & c \end{array} \qquad e$$

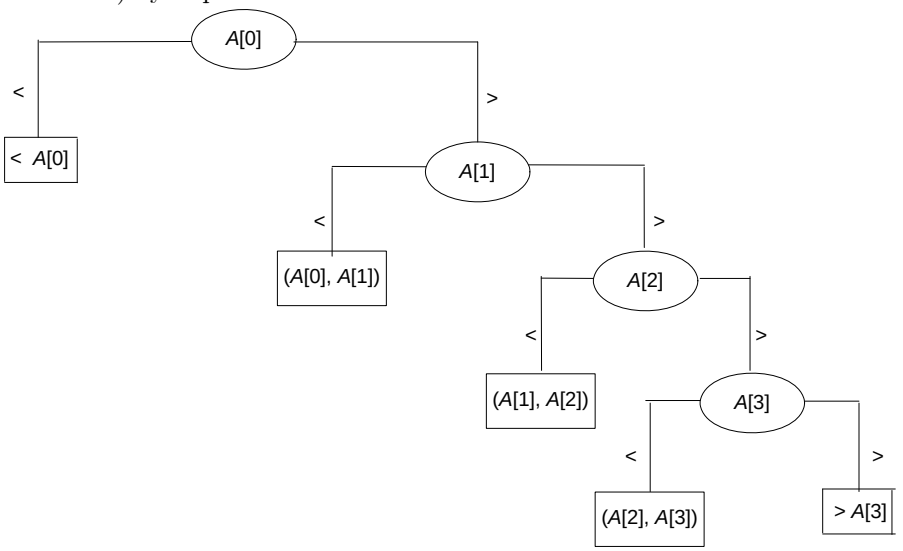
So far, we performed 3 comparisons. Now, we insert e in its appropriate position in the ordered subsequence $a < b < d$; it can be done in 2 comparisons if we start by comparing e with b. This results in one of the four

following situations:

$$\begin{array}{llll} e \to a \to b \to d & a \to e \to b \to d & a \to b \to e \to d & a \to b \to d \to e \\ \uparrow & \uparrow & \uparrow & \uparrow \\ c & c & c & c \end{array}$$

What remains to be done is to insert c in its appropriate position. Taking into account the information that $c < d$, it can be done in 2 comparisons in each of the four cases depicted above. (For example, for the first case, compare c with a and then, depending on the result, with either e or b.)

6. Here is a decision tree for searching a four-element ordered list (indexed from 0 to 3) by sequential search:



7. a. Since $a \le b$ implies $\lceil a \rceil \le \lceil b \rceil$ (because $a \le b \le \lceil b \rceil$, and therefore, since $\lceil b \rceil$ is an integer, $\lceil a \rceil \le \lceil b \rceil$ by the definition of $\lceil a \rceil$), it will suffice to show that

$$\log_3(2n+1) \le \log_2(n+1) \text{ for every } n \ge 1.$$

Using elementary properties of logarithms, we obtain the following chain of equivalent inequalities:

$$\begin{array}{rcl} \dfrac{\log_2(2n+1)}{\log_2 3} & \le & \log_2(n+1) \\ \log_2(2n+1) & \le & \log_2 3 \log_2(n+1) \\ \log_2(2n+1) & \le & \log_2(n+1)^{\log_2 3} \end{array}$$

$$(2n+1) \le (n+1)^{\log_2 3}.$$

One way to prove that the last inequality holds for every $n \geq 1$ is to consider the function

$$f(x) = (x+1)^{\log_2 3} - (2x+1)$$

and verify that $f(1) = 2^{\log_2 3} - 3 = 0$ and $f'(1) = \log_2 3 (1+1)^{\log_2 3 - 1} - 2 = \log_2 3 \cdot 2^{\log_2 (3/2)} - 2 = \log_2 3 \cdot (3/2) - 2 > 0$.

b. The easiest way is to prove a stronger assertion:

$$\lim_{n\to\infty} \frac{\lceil \log_2 (n+1) \rceil}{\lceil \log_3 (2n+1) \rceil} > 1,$$

which implies that

$$\frac{\lceil \log_2 (n+1) \rceil}{\lceil \log_3 (2n+1) \rceil} > 1 \quad \text{for every } n \geq n_0.$$

To get rid of the ceiling functions, we can use

$$\frac{f(n)-1}{g(n)+1} < \frac{\lceil f(n) \rceil}{\lceil g(n) \rceil} < \frac{f(n)+1}{g(n)-1}$$

where $f(n) = \log_2 (n+1)$ and $g(n) = \log_3 (2n+1)$ for $n > 1$ and show that

$$\lim_{n\to\infty} \frac{f(n)-1}{g(n)+1} = \lim_{n\to\infty} \frac{f(n)+1}{g(n)-1} > 1.$$

Indeed, using l'Hôpital's Rule, we obtain the following:

$$\begin{aligned} \lim_{n\to\infty} \frac{f(n)-1}{g(n)+1} &= \lim_{n\to\infty} \frac{\log_2 (n+1) - 1}{\log_3 (2n+1) + 1} \\ &= \lim_{n\to\infty} \frac{\frac{1}{n+1} \cdot \frac{1}{\ln 2}}{\frac{1}{2n+1} \cdot \frac{2}{\ln 3}} = \frac{\ln 3}{\ln 2} = \log_2 3. \end{aligned}$$

Computing the limit of $\frac{f(n)+1}{g(n)-1}$ in the exactly same way yields the same result. Therefore, according to a well-known theorem of calculus,

$$\lim_{n\to\infty} \frac{\lceil f(n) \rceil}{\lceil g(n) \rceil} = \lim_{n\to\infty} \frac{\lceil \log_2 (n+1) \rceil}{\lceil \log_3 (2n+1) \rceil} = \log_2 3 > 1,$$

and hence there exists n_0 such that

$$\frac{\lceil \log_2 (n+1) \rceil}{\lceil \log_3 (2n+1) \rceil} > 1 \text{ for every } n \geq n_0.$$

8. Since any of the n numbers can be the maximum, the number of leaves in a decision tree of any algorithm solving the problem will be at least n. Hence, according to inequality (11.1), the lower bound for the number of comparisons made by any comparison-based algorithm in the worst case is $\lceil \log_2 n \rceil$. This bound is not tight. At least $n-1$ comparisons are needed because at least $n-1$ numbers should "lose" their comparions before the maximum is found.

9. a. Before a winner can be determined, $n-1$ players must lose a game. Since each game results in one loser, $n-1$ games are played in a single-elimination tournament with n players.

   b. The tournament tree has n leaves (the number of players) and $n-1$ internal nodes (the number of games—see part a). Hence the total number of nodes in this binary tree is $2n-1$, and, since it is complete, its height h is equal to

   $$h = \lfloor \log_2(2n-1) \rfloor = \lceil \log_2(2n-1+1) \rceil - 1 = \lceil \log_2 2 + \log_2 n \rceil - 1 = \lceil \log_2 n \rceil.$$

   Alternatively, one can solve the recurrence relation for the number of rounds

   $$R(n) = R(\lceil n/2 \rceil) + 1 \quad \text{for } n > 1, \quad R(1) = 0,$$

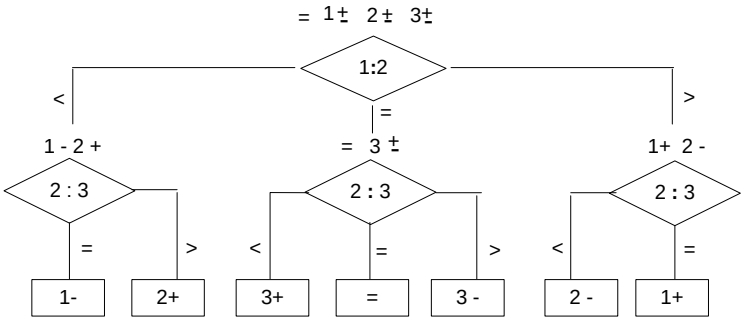
   to obtain $R(n) = \lceil \log_2 n \rceil$.

   c. The second best player can be any player who lost to the winner and nobody else. These players can be made play their own knock-out tournament by following the path from the leaf representing the winner to the root and assuming that the winner lost the first match. This will require no more than $\lceil \log_2 n \rceil - 1 = \lfloor \log_2(n-1) \rfloor$ games.

10. a. Since each of the n coins can be either lighter or heavier than all the others and all the coins can be genuine, the total number of possible outcomes is $2n+1$. Since each weighing can have three results, the smallest number of weighings must be at least $\lceil \log_3(2n+1) \rceil$.

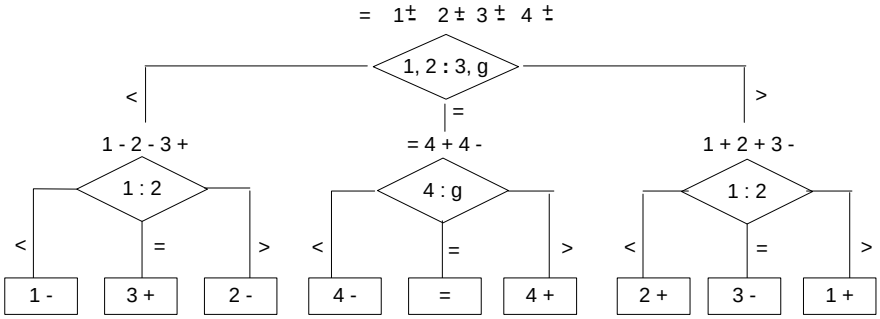
    b. In the decision tree below, the coins are numbered from 1 to 3. Internal nodes indicate weighings, with the coins being weighted listed inside the node. (E.g., the root corresponds to the first weighing in which coin 1 and 2 are put on the left and right cup of the scale, respectively). Edges to a node's children are marked according to the node's weighing outcome: $<$ means that the left cup's weight is smaller than that of the right's cup; $=$ means that the weights are equal; $>$ means that the left cup's weight is larger than that of the right cup. Leaves indicate final outcomes: $=$ means that all the coins are genuine, a particular coin followed by $+$ or $-$

means this coins is heavier or lighter, respectively. A list above an internal node indicates the outcomes that are still possible before the weighing indicated inside the node. For example, before coins 1 and 2 are put on the left and right cups, respectively, in the first weighing of the algorithm, the list = 1± 2± 3± indicates that either all the coins are genuine (=), or coin 1 is either heavier or lighter (1±) or coin 2 is either heavier or lighter (2±) or coin 3 is either heavier or lighter (3±).



c. There are two reasonable possibilities for the first weighing: to weigh two coins (i.e., one on each cup of the scale) and to weigh four coins (i.e., two on each cup). If an algorithm weighs two coins and they weigh the same, five outcomes remain possible for the two other coins. If an algorithm weighs four coins and they don't weigh the same, four outcomes remain possible. Since more than three possible outcomes cannot be resolved by one weighing, no algorithm will be able to solve the puzzle with the total of two weighings.

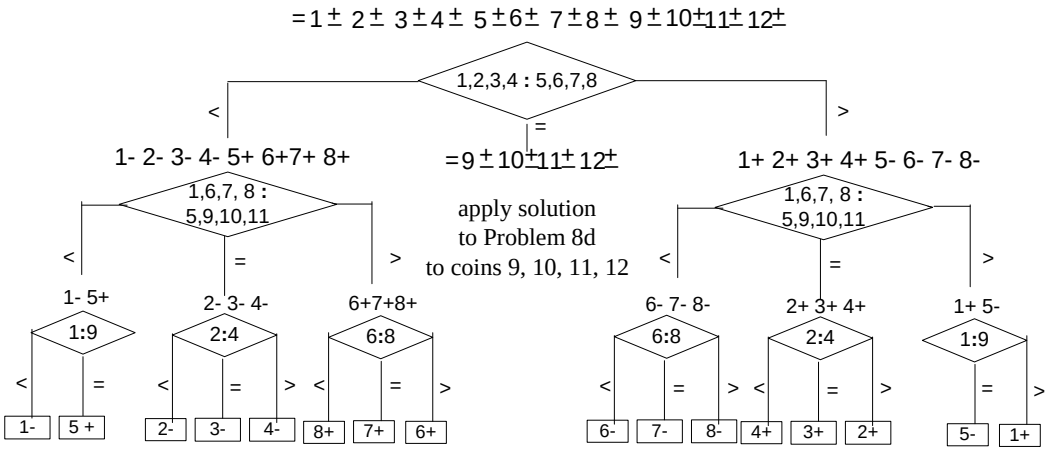
d. Here is a decision three of an algorithm that solves the fake-coin puzzle for $n = 4$ in two weighings by using an extra coin (denoted g) known to be genuine:



(The coins are assumed to be numbered form 1 to 4. All possible outcomes before a weighing are listed above the weighing's diamond box. A

plus or minus next to a coin's number means that the coin is heavier or lighter than the genuine one, respectively; the equality sign in the leaf means that all the coins are genuine.)

e. Here is a decision tree of an algorithm that solves the fake-coin puzzle for 12 coins in three weighings



Note: This problem is discussed on many puzzle-related sites on the Internet (see, e.g., http://www.cut-the-knot.com/blue/weight1.shtml). The attractiveness of the solution given above lies in its symmetry: the second weighings involve the same coins if the first one tips the scale either way, and the subsequent round of weighings involves the same pairs of coins. (In fact, the problem has a completely non-adaptive solution, i.e., a choice of what to put on the balance for the second weighing doesn't depend on the outcome of the first one, and a choice of what to weigh in the third round doesn't depend on what happened on either the first or second weighing.)

11. Any algorithm to assemble the puzzle can be represented by a binary tree whose leaves represents the single pieces and internal nodes represent connecting two sections (its children). Since each internal node in such a tree has two children, the leaves can be interpreted as extended nodes of a tree formed by its internal nodes. According to equality (4.5), the number of internal nodes (moves) is equal to $n - 1$, one less than the number of leaves (single pieces), for any such tree (assembling algorithm).

Note 1: Alternatively, one can reason that the task starts with n one-piece sections and ends with a single section. Since each move connects two sections and hence decreases the total number of sections by 1, the total

number of moves made by any algorithm that doesn't disconnect already connected pieces is equal to $n - 1$.
Note 2: This puzzle was published in *Mathematics Magazine*, vol. 26, p. 169. See also Problem 11 in Exercises 5.3 for a better known version of this puzzle (chocolate bar puzzle).

Solutions to Exercises 11.3

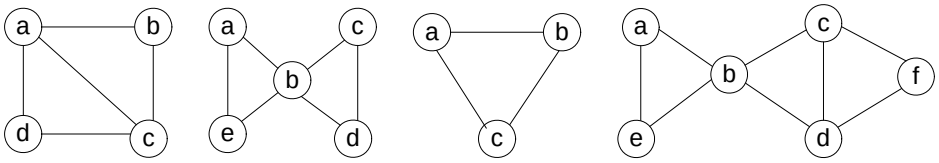
1. Yes, it's decidable. Theoretically, we can simply generate all the games (i.e., all the sequences of all legal moves of both sides) from the position given and check whether one of them is a win for the side that moves next.

2. First, $n^{\log_2 n}$ grows to infinity faster than any polynomial function n^k. This follows immediately from the fact that while the exponent k of n^k is fixed, the exponent $\log_2 n$ of $n^{\log_2 n}$ grows to infinity. (More formally:

$$\lim_{n\to\infty} \frac{n^k}{n^{\log_2 n}} = \lim_{n\to\infty} n^{k-\log_2 n} = 0.)$$

   We cannot say whether or not this algorithm solves the problem in polynomial time because the O-notation doesn't preclude the possibility that this algorithm is, in fact, polynomial-time. Even if this algorithm is not polynomial, there might be another which is. Hence, the correct answer is (c).

3. Here are examples of graphs required:



(a) (b) (c) (d)

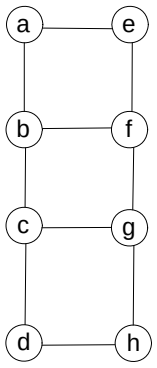
   a. Graph (a) has a Hamiltonian circuit ($a-b-c-d-a$) but no Eulerian circuit because it has vertices of odd degrees (a and c).

   b. Graph (b) has an Eulerian circuit ($a-b-c-d-b-e-a$) but no Hamiltonian circuit. If a Hamiltonian circuit existed, we could consider a as its starting vertex without loss of generality. But then it would have to visit b at least twice: once to reach c and the other to return to a.

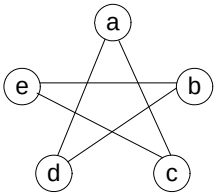
   c. Graph (c) has both a Hamiltonian circuit and an Eulerian circuit ($a-b-c-a$).

   d. Graph (d) has a cycle that includes all the vertices ($a-b-c-f-d-b-e-a$). It has neither a Hamiltonian circuit (by the same reason graph (b) doesn't) nor an Eulerian circuit (because vertices c and d have odd degrees).

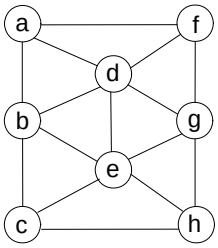
4. The chromatic numbers for the graphs below are 2, 3, and 4, respectively.



(a)



(b)



(c)

a. The chromatic number of graph (a) is 2 because we can assign color 1 to vertices a, f, c, and h and color 2 to vertices e, b, g, and d. (Note: This is an example of a bipartite graph. A *bipartite graph* is a graph whose vertices can be partitioned into two disjoint sets V_1 and V_2 so that every edge of the graph connects a vertex from V_1 and a vertex from V_2 . A graph is bipartite if and only if it doesn't have a cycle of an odd length.)

b. The chromatic number of graph (b) is 3. It needs at least 3 colors because it has an odd-length cycle: $a - c - e - b - d - a$ and 3 colors will suffice if we assign, for example, color 1 to vertices a and b, color 2 to vertices c and d, and color 3 to vertex e.

c. The chromatic number of graph (c) is 4. It needs at least 3 colors because it has cycles of length 3. But 3 colors will not suffice. If we try to use just 3 colors and assign, without loss of generality, colors 1, 2, and 3 to vertices a, f, and d, then we'll have to assign color 2 to vertex b and color 1 to vertex g. But then vertex e, being adjacent to b, d, and g, will require a different color, i.e., color 4. Coloring e with color 4 does yield a legitimate coloring with four colors, with c and h colored with, say, colors 1 and 3, respectively. Hence the chromatic number of this graph is equal to 4.

5. Consider a depth-first search forest obtained by a DFS traversal of a given graph. If the DFS forest has no back edges, the graph's vertices can be colored in two colors by alternating the colors on odd and even levels of the forest. (Hence, an acyclic graph is always 2-colorable.) If the DFS forest has back edges, it is clear that it is 2-colorable if and only if every back edge connects vertices on the levels of different parity: the one on an even level and the other on an odd level. This property can be checked

by a minor modification of depth-first search, which, of course, is a polynomial time algorithm.

Note: A breadth-first search forest can be used in the same manner. Namely, a graph is two colorable if and only if its BFS forest has no cross edges connecting vertices on the same level.

6. The brute-force algorithm is in $O(n)$. However, a proper measure of size for this problem is b, the number of bits in n's binary expansion. Since $b = \lfloor \log_2 n \rfloor + 1$, $O(n) = O(2^b)$.

7. a. Determine whether there is a subset of a given set of n items that fits into the knapsack and has a total value not less than a given positive integer m. To verify a proposed solution to this problem, sum up the weights and (separately) the values of the subset proposed: the weight sum should not exceed the knapsack's capacity, and the value sum should be not less than m. The time efficiency of this algorithm is obviously in $O(n)$.

   b. Determine whether one can place a given set of items into no more than m bins. A proposed solution can be verified by checking that the number of bins in this solution doesn't exceed m and that the sum of the sizes of the items assigned to the same bin doesn't exceed the bin capacity for each of the bins. The time efficiency of this algorithm is obviously in $O(n)$ where n is the number of items in the given instance.

8. The partition problem for n given integers s_i, $i = 1, ..., n$, can be expressed as a selection of 0–1 variables x_i, $i = 1, ..., n$ such that

$$\sum_{i=1}^{n} x_i s_i = \frac{1}{2} \sum_{i=1}^{n} s_i.$$

   This is equivalent to

$$2 \sum_{i=1}^{n} x_i s_i = \sum_{i=1}^{n} s_i \quad \text{or} \quad \sum_{i=1}^{n} x_i 2 s_i = \sum_{i=1}^{n} s_i.$$

   Denoting $w_i = 2s_i$, $W = \sum_{i=1}^{n} s_i$, and selecting $m = W$ as the value's lower bound in the decision knapsack problem, we obtain the following (equivalent) instance of the latter:

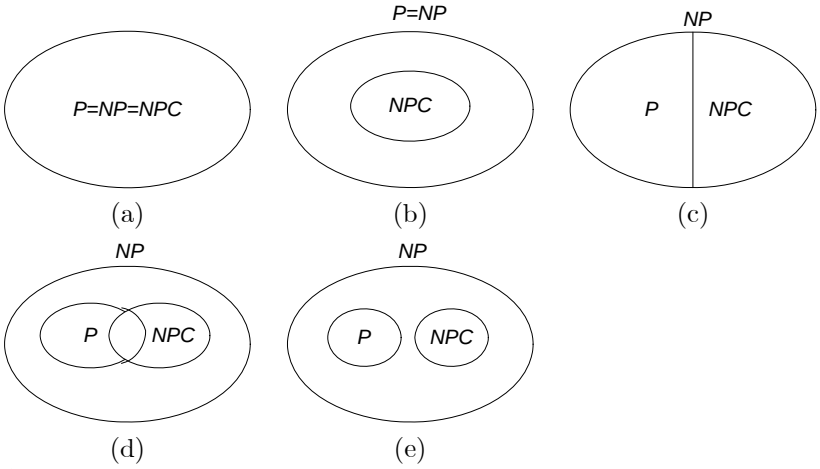
$$\sum_{i=1}^{n} x_i w_i \geq m \ (m = W)$$
$$\sum_{i=1}^{n} x_i w_i \leq W$$
$$x_i \in \{0, 1\} \text{ for } i = 1, ..., n.$$

9. The reductions follow from the following result (see, e.g., [Gar79]), which immediately follows from the definitions of the clique, independent set, vertex cover, and that of the complement graph: For any graph $G = \langle V, E \rangle$ and subset $V' \subseteq V$, the following statements are equivalent:

   i. V' is a vertex cover for G.
   ii. $V - V'$ is an independent set for G.
   iii. $V - V'$ is a clique in the complement of G.

10. Since the problem is a different wording of the CNF-satisfiability problem, it is *NP*-complete.

11. The given diagrams are:



   (a) is impossible because the trivial decision problem whose solution is "yes" for all inputs is clearly not *NP*-complete.

   (b) is possible (depicts the case of $P = NP$)

   (c) is impossible because, as mentioned in Section 11.3, there must exist problems in *NP* that are neither in P nor *NP*-complete, provided $P \neq NP$.

   (d) is impossible because it implies the existence of an *NP*-complete problem that is in P, which requires $P = NP$.

   (e) is possible (depicts the case of $P \neq NP$).

12. a. Create a graph in which vertices represent the knights and an edge connects two vertices if and only if the knights represented by the vertices can sit next to each other. Then a solution to King Arthur's problem exits if and only if the graph has a Hamiltonian circuit.

b. According to Dirac's theorem, a graph with $n \geq 3$ vertices has a Hamiltonian circuit if the degree of each of its vertices is greater than or equal to $n/2$. A more general sufficient condition—the sum of degrees for each pair of nonadjacent vertices is greater than or equal to n—is due to Ore (see, e.g., *Graph Theory and Its Applications* by J. Gross and J. Yellen, CRC Press, 1999).

Solutions to Exercises 11.4

1. a. For $\alpha = 3.1415$, the relative error of the approximation is

$$\frac{|\alpha - \alpha^*|}{|\alpha^*|} = \frac{|3.1415 - \pi|}{\pi} \approx 2.9 \cdot 10^{-5}.$$

Since $2.9 \cdot 10^{-5} < 5 \cdot 10^{-5}$ but $2.9 \cdot 10^{-5} > 5 \cdot 10^{-6}$, the number of significant digits is 5.

b. For $\alpha = 3.1417$, the relative error of the approximation is

$$\frac{|\alpha - \alpha^*|}{|\alpha^*|} = \frac{|3.1417 - \pi|}{\pi} \approx 3.4 \cdot 10^{-5}.$$

Since $3.4 \cdot 10^{-5} < 5 \cdot 10^{-5}$ but $3.4 \cdot 10^{-5} > 5 \cdot 10^{-6}$, the number of significant digits is also 5. (Note that the correct value of π is 3.14159265....)

2. Since $|\alpha - \alpha^*| \leq 10^{-2}$ is equivalent to $\alpha - 10^{-2} \leq \alpha^* \leq \alpha + 10^{-2}$,

$$1.49 \leq \alpha^* \leq 1.51.$$

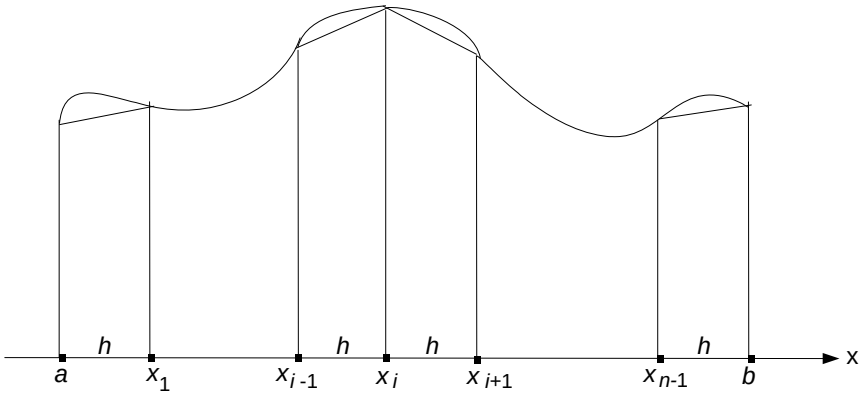
These bounds imply the following for the possible values of the relative error

$$0 \leq \frac{|\alpha - \alpha^*|}{|\alpha^*|} \leq \frac{10^{-2}}{1.49}.$$

3. $\sum_{i=0}^{5} \frac{0.5^i}{i!} = 1.648697916\bar{} \approx 1.6487$. The absolute error of approximating $\sqrt{e} = 1.648721....$ by this sum is 0.00002..., which is smaller than the required 10^{-4} (see the text immediately following formula (11.8)).

4. The composite trapezoidal rule is based on dividing the interval $a \leq x \leq b$ into n equal subintervals of length $h = (b-a)/n$ by the points $x_i = a + ih$ for $i = 0, 1, ..., n$ and approximating the area between the graph of $f(x)$ and the x axis (equal to $\int_a^b f(x)dx$) by the sum of the areas of the trapezoid

strips:



The area of the ith such trapezoid (with the lengths of two parallel sides $f(x_i)$ and $f(x_{i+1})$ and height h) is obtained by the standard formula

$$A_i = \frac{h}{2}[f(x_i) + f(x_{i+1})].$$

The entire area A is approximated by the sum of these areas:

$$\begin{aligned}\int_a^b f(x)dx &= \sum_{i=0}^{n-1} A_i \approx \sum_{i=0}^{n-1} \frac{h}{2}[f(x_i) + f(x_{i+1})] \\ &= \frac{h}{2}[f(x_0) + f(x_1) + f(x_1) + f(x_2) + ... + f(x_{n-1}) + f(x_n)] \\ &= \frac{h}{2}[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)].\end{aligned}$$

5. a. Applying formula (11.7) to function $f(x) = x^2$ on the interval $0 \le x \le 1$ divided into four equal subintervals of length $h = (1-0)/4 = 0.25$ yields

$$\int_0^1 x^2 dx \approx \frac{0.25}{2}[0^2 + 2(0.25^2 + 0.5^2 + 0.75^2) + 1^2] = 0.34375.$$

The exact value is

$$\int_0^1 x^2 dx = \frac{1}{3} = 0.\bar{3}.$$

Hence, the (magnitude of) truncation error of this approximation is $0.34375 - 0.\bar{3} = 0.010416\bar{6}$.
Formula (11.9) yields the following upper bound for the magnitude of the truncation error:

$$\frac{(b-a)h^2}{12} M_2 = \frac{(1-0)0.25^2}{12} 2 = 0.010416\bar{6},$$

which is exactly equal to the actual magnitude of the truncation error.

b. Applying formula (11.7) to function $f(x) = 1/x$ on the interval $1 \le x \le 3$ divided into four equal subintervals of length $h = (3-1)/4 = 0.5$ yields

$$\int_1^3 \frac{1}{x}dx \approx \frac{0.5}{2}[\frac{1}{1} + 2(\frac{1}{1.5} + \frac{1}{2} + \frac{1}{2.5}) + \frac{1}{3}] = 1.11\bar{6}.$$

The exact value is

$$\int_1^3 \frac{1}{x}dx = \ln 3 - \ln 1 = 1.09861....$$

Hence, the (magnitude of) truncation error of this approximation is $1.11\bar{6} - 1.09861... = 0.01805...$

Before we use formula (11.9), let us find

$$M_2 = \max_{1 \le x \le 3} |f''(x)| \text{ where } f(x) = \frac{1}{x}.$$

Since $f(x) = 1/x$, $f'(x) = -1/x^2$ and $f''(x) = 2/x^3$, $M_2 = 2$. Formula (11.9) yields the following upper bound for the magnitude of the truncation error:

$$\frac{(b-a)h^2}{12}M_2 = \frac{(3-1)0.5^2}{12}2 = \frac{1}{12} = 0.08\bar{3},$$

which is (appropriately) larger than the actual truncation error of 0.01805....

6. Since $f(x) = e^{\sin x}$, $f'(x) = e^{\sin x}\cos x$ and

$$\begin{aligned} f''(x) &= [e^{\sin x}\cos x]' = [e^{\sin x}]'\cos x + e^{\sin x}[\cos x]' = e^{\sin x}\cos^2 x - e^{\sin x}\sin x \\ &= e^{\sin x}(\cos^2 x - \sin x) = e^{\sin x}(1 - \sin^2 x - \sin x). \end{aligned}$$

Since $\sin x$ is increasing from 0 to $\sin 1$ and $1 - \sin^2 x - \sin x$ is decreasing from 1 to $1 - \sin^2 1 - \sin 1 \approx -0.55$ on the interval $0 \le x \le 1$,

$$|f''(x)| = |e^{\sin x}(1 - \sin^2 x - \sin x)| = e^{\sin x}|1 - \sin^2 x - \sin x| < e^{\sin 1} < e^1 < 3.$$

Using formula (11.9), we get

$$\frac{b-a}{12}\left(\frac{b-a}{n}\right)^2 M_2 < \frac{1}{12}\frac{1}{n^2}3 < \varepsilon,$$

where ε is a given upper error limit. Solving the last inequality for n yields

$$n^2 > \frac{1}{4\varepsilon} \quad \text{or} \quad n > \frac{1}{2\sqrt{\varepsilon}}.$$

Hence, the answers for $\varepsilon = 10^{-4}$ and $\varepsilon = 10^{-6}$ are $n > 50$ and $n > 500$, respectively.

7. The solution to the first system is $x = 1$, $y = 1$; the solution to the second one is $x = 8.5$, $y = -2$. Both systems are ill-conditioned.

8. In addition to the points made in Section 11.4 about solving quadratic equations, the program should handle correctly the case of $a = 0$ as well: If $b \neq 0$, the equation $bx = c$ has a single root $x = -b/c$. If $b = 0$ and $c \neq 0$, the equation $0x = c$ has no roots. If $b = 0$ and $c = 0$, any number is a root of $0x = 0$.

9. a. Let us prove by induction that $x_n > \sqrt{D}$ for $n = 0, 1, ...$, if $x_0 > \sqrt{D}$. Indeed, $x_0 > \sqrt{D}$ by assumption. Further, assuming that $x_n > \sqrt{D}$, we can prove that this implies that $x_{n+1} > \sqrt{D}$ as follows:

$$x_{n+1} - \sqrt{D} = \frac{1}{2}(x_n + \frac{D}{x_n}) - \sqrt{D} = \frac{x_n^2 - 2x_n\sqrt{D} + D}{2x_n} = \frac{(x_n - \sqrt{D})^2}{2x_n} > 0.$$

Now, consider

$$x_{n+1} - x_n = \frac{1}{2}(x_n + \frac{D}{x_n}) - x_n = \frac{1}{2}(\frac{D}{x_n} - x_n) = \frac{D - x_n^2}{2x_n} < 0,$$

i.e., the sequence $\{x_n\}$ is strictly decreasing. Since it is also bound below (e.g., by $\sqrt{D}$), it must have a limit (to be denoted by l) according to a well-known theorem of calculus. Taking the limit of both sides of

$$x_{n+1} = \frac{1}{2}(x_n + \frac{D}{x_n})$$

as n goes to infinity yields

$$l = \frac{1}{2}(l + \frac{D}{l}),$$

whose nonnegative solution is $l = \sqrt{D}$.

(Note that $x_0 = (1+D)/2 > \sqrt{D}$ for any $D \neq 1$ because $(1+D)/2 - \sqrt{D} = (1-\sqrt{D})^2/2 > 0$. If $D = 1$, $x_0 = (1+D)/2 = 1$ and all the other elements of the approximation sequence are also equal to 1.)

b. We will take advantage of the equality

$$x_{n+1} - \sqrt{D} = \frac{(x_n - \sqrt{D})^2}{2x_n},$$

which was derived in the solution to part (a) above. Since $x_0 = (1 + D)/2 > \sqrt{D}$ for any $D \neq 1$, according to the fact proved by induction in part (a) $x_n > \sqrt{D} \geq 0.5$. Therefore,

$$x_{n+1} - \sqrt{D} = \frac{(x_n - \sqrt{D})^2}{2x_n} \leq (x_n - \sqrt{D})^2.$$

Applying the last inequality n times yields

$$|x_n - \sqrt{D}| \leq (x_{n-1} - \sqrt{D})^2 \leq (x_{n-2} - \sqrt{D})^4 \leq .. \leq (x_0 - \sqrt{D})^{2^n}.$$

For $0.25 \leq D < 1$, $0.5 \leq \sqrt{D} < 1$, and hence

$$x_0 - \sqrt{D} = \frac{1+D}{2} - \sqrt{D} = \frac{(1 - 2\sqrt{D} + D)}{2} = \frac{(1-\sqrt{D})^2}{2} \leq 2^{-3}.$$

By combining the last two inequalities, we obtain

$$|x_n - \sqrt{D}| \leq (x_0 - \sqrt{D})^{2^n} \leq 2^{-3 \cdot 2^n}.$$

In particular, for $n = 4$,

$$|x_4 - \sqrt{D}| \leq 2^{-48} < 4 \cdot 10^{-15}.$$

10. We will roundoff the numbers to 6 decimal places.

$$\begin{aligned} x_0 &= \frac{1}{2}(1+3) = 2.000000 \\ x_1 &= \frac{1}{2}(x_0 + \frac{3}{x_0}) = 1.750000 \\ x_2 &= \frac{1}{2}(x_1 + \frac{3}{x_1}) \doteq 1.732143 \\ x_3 &= \frac{1}{2}(x_2 + \frac{3}{x_2}) \doteq 1.732051 \\ x_4 &= \frac{1}{2}(x_3 + \frac{3}{x_3}) \doteq 1.732051 \end{aligned}$$

Thus, $x_4 \doteq 1.732051$ and, had we continued, all other approximations would've been the same. The exact value of $\sqrt{3}$ is 1.73205080.... Hence, we can bound the absolute error by

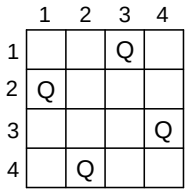
$$1.732051 - 1.73205080... < 2 \cdot 10^{-7}$$

and the relative error by

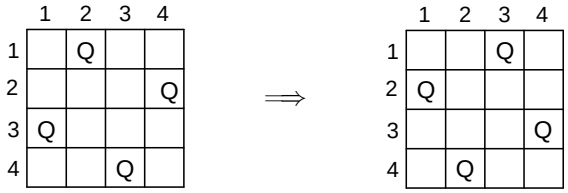
$$\frac{1.732051 - 1.73205080...}{1.73205080...} < \frac{2 \cdot 10^{-7}}{1.73205080} < 1.2 \cdot 10^{-7}.$$

Solutions to Exercises 12.1

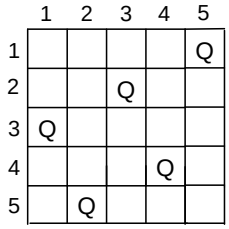
1. a. The second solution reached by the backtracking algorithm is



b. The second solution can be found by reflection of the first solution with respect to the vertical line through the middle of the board:

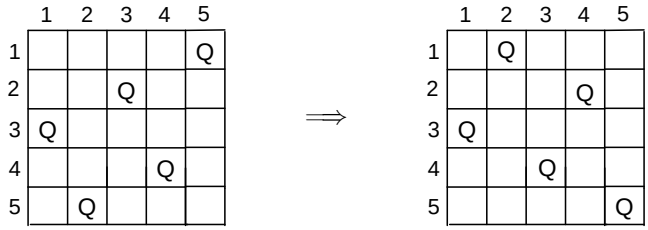


2. a. The last solution to the 5-queens puzzle found by backtracking last is

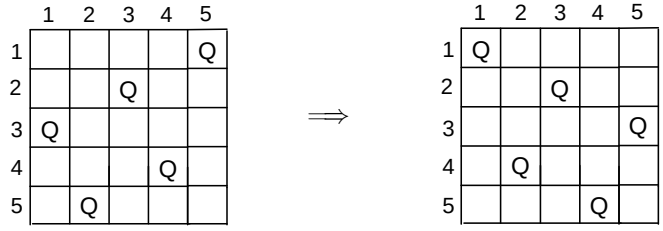


It is obtained by placing each queen in the last possible column of the queen's row, staring with the first queen and ending with the fifth one. (Any placement with the first queen in column $j < 5$ is found by the backtracking algorithm before a placement with the first queen in column 5, and so on.)

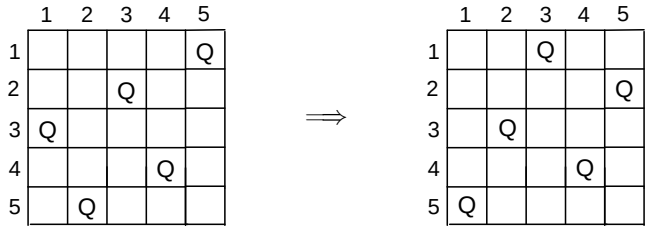
b. The solution obtained using the board's symmetry with respect to its middle row is



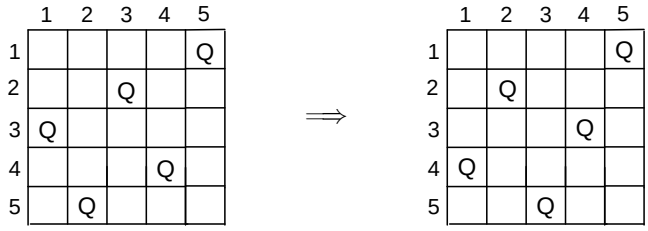
The solution obtained using the board's symmetry with respect to its middle column is



The solution obtained using the board's symmetry with respect to its main north-west–south-east diagonal is

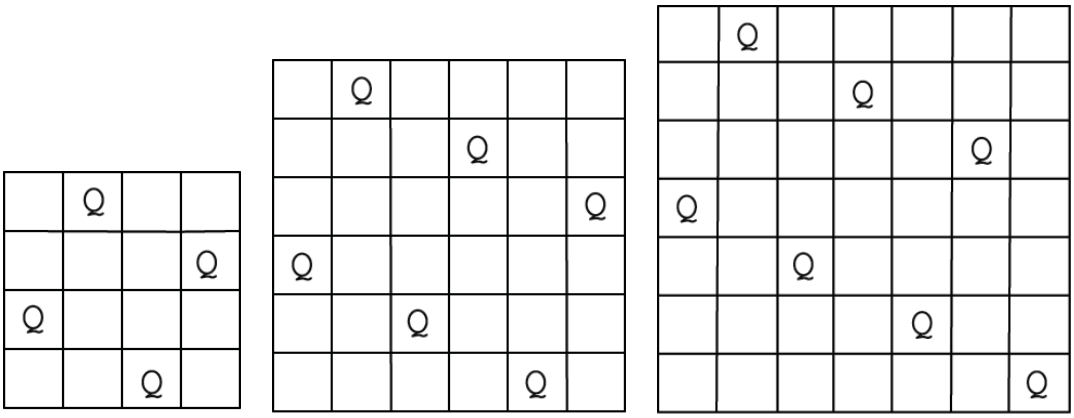


The solution obtained using the board's symmetry with respect to its main south-west–north-east diagonal is



3. For a discussion of efficient implementations of the backtracking algorithm for the n-queens problem, see Timothy Rolfe, "Optimal Queens," *Dr. Dobb's Journal,* May 2005, pp. 32–37.

4. The solution for $n = 4$ obtained by backtracking in the text (see Figure 12.2) suggests the following structure of solutions for other even n's. For the first $n/2$ rows, place the queens in columns 2, 4, ..., $n/2$; for the last $n/2$ rows, place the queens in columns 1, 3, ..., $n - 1$. Indeed, this works not only for any $n = 4+6k$, but also for any $n = 6k$. Moreover, since none of these solutions places a queen on the main diagonal, the solution can be extended to yield a solution for the next value of n by just adding a queen in the last column of the last row (see the figures below for examples).

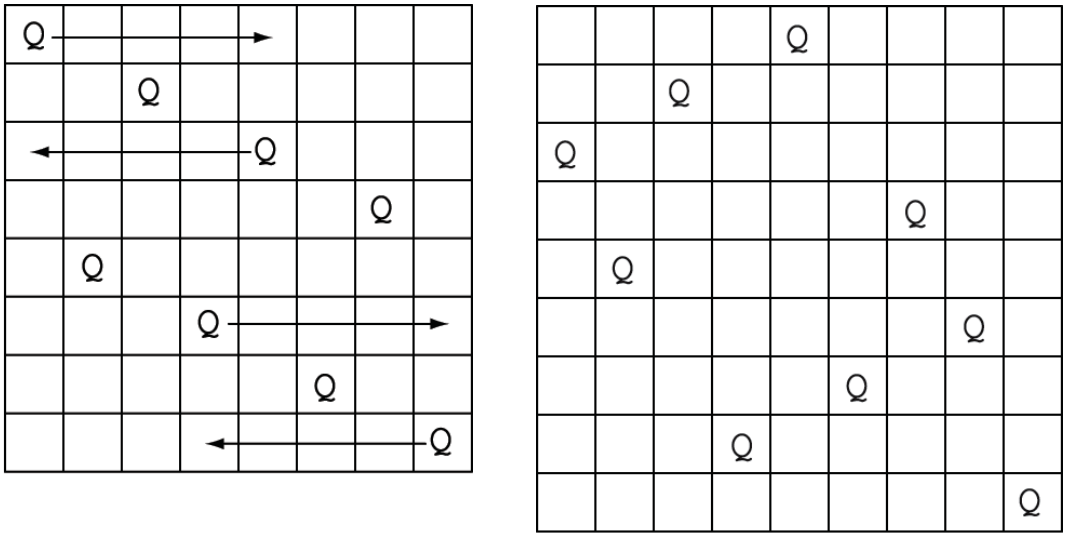


(a) (b) (c)

Solutions to the n-queens problems for (a) $n = 4$; (b) $n = 6$; (c) $n = 7$

Unfortunately, this does not work for $n = 8 + 6k$. Although it is possible to place queens on such boards as it was done above and then rearrange them, it is easier to start with queens in odd-numbered columns of the first $n/2$ rows followed by queens in even-number columns. Then we can get a nonattacking position by exchanging the columns of the queens in rows 1 and $n/2 - 1$ and the columns of the queens in rows $n/2 + 2$ and n (see the case of $n = 8$ below). Since the solution obtained in this way has no queen on the main diagonal, it can be expanded to a solution for $n = 9 + 6k$ by placing the extra queen at the

last column of the last row of the larger board.



(d) (e)

Solutions to the n-queens problems for (d) $n = 8$ and (e) $n = 9$

Thus we have the following algorithm that generates column numbers for placing $n > 3$ queens in consecutive rows of an $n \times n$ board.

Compute the remainder r of division n by 6.

Case 1 (r is neither 2 nor 3): Write a list of the consecutive even numbers from 2 to n inclusive and append to it the consecutive odd numbers from 1 to n inclusive.

Case 2 (r is 2): Write a list of the consecutive odd numbers from 1 to n inclusive and then swap the first and penultimate numbers. Append to the list the consecutive even numbers from 2 to n inclusive and then swap number 4 and the last number in the expanded list.

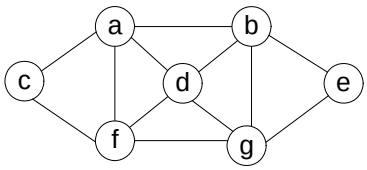
Case 3 (r is 3) Apply the directives of Case 2 to $n - 1$ instead of n and append n to the created list.

Note: The n-queens problem is one of the most well-known problems in recreational mathematics, which has attracted the attention of mathematicians since the middle of the 19th century. The search for efficient algorithms for solving it started much later, of course. Among a variety of approaches, solving the problem by backtracking—as it's done in the text—has become a standard way to introduce this general technique in algorithm textbooks. In addition to the educational merits, backtracking has the advantage of finding, at least in principle, all the solutions to the problem. If just one solution is the goal, much simpler methods can be employed. The survey paper by J. Bell and B. Stevens [Bel09] contains more than half a dozen sets of formulas for direct computation of queens' positions, including the earliest one by E. Pauls published in 1874.

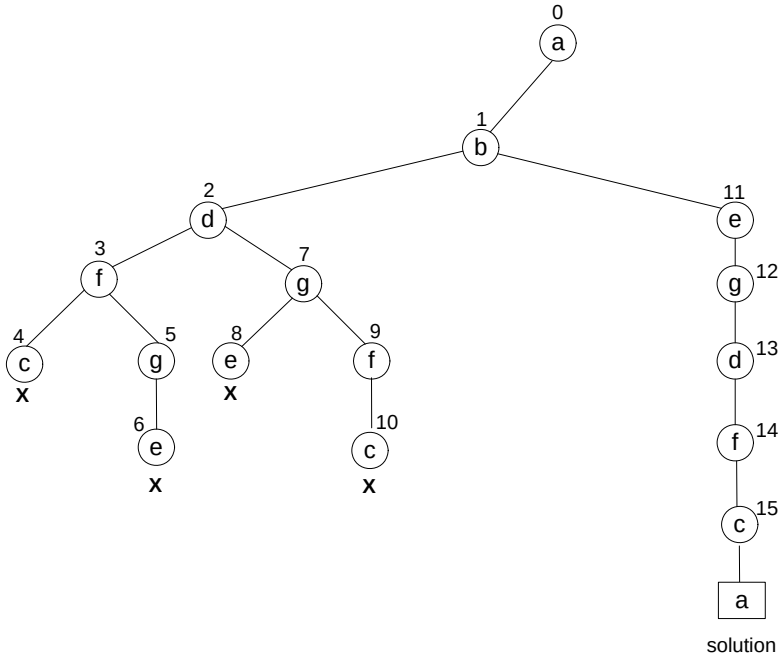
Surprisingly, this alternative has been all but ignored by computer scientists until B. Bernhardsson alerted the community of its existence in "Explicit solutions to the n-queens problem for all n," *SIGART Bulletin*, vol. 2, issue 2, (April 1991), p. 7.

The above algorithm follows an outline by D. Ginat in his Colorful Challenges column.(*SIGCSE Bulletin*, vol. 38, no. 2, June 2006, 21–22). It is rather straightforward in that it places queens in the first available square of each row of the board. The nontrivial part is that for some n's we have to start with the second square in the first row, and for some n's we have to swap two pairs of queens to satisfy the nonattacking requirement. It is also worth noting that the solutions for $n \times n$ boards where n is odd are obtained, in fact, as extensions of the solutions for $(n\text{-}1)\times(n\text{-}1)$ boards.

5. Finding a Hamiltonian circuit in the graph

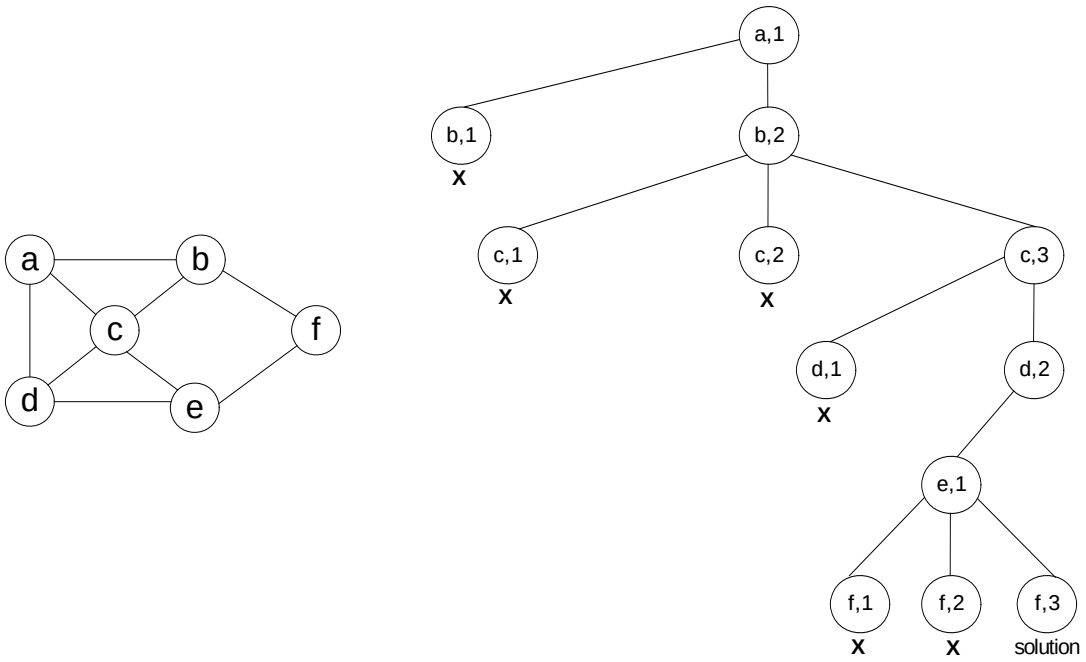


by backtracking yields the following state-space tree:

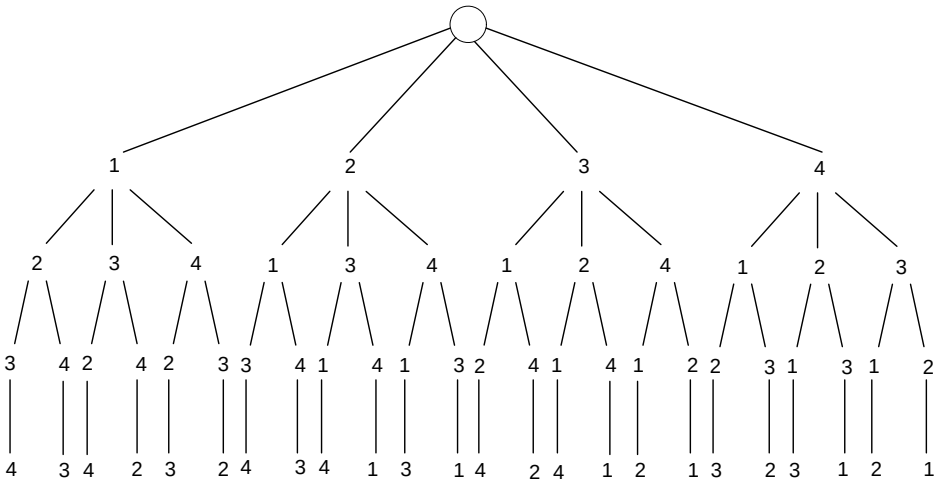


6. Here are the graph and a state-space tree for solving the 3-coloring problem

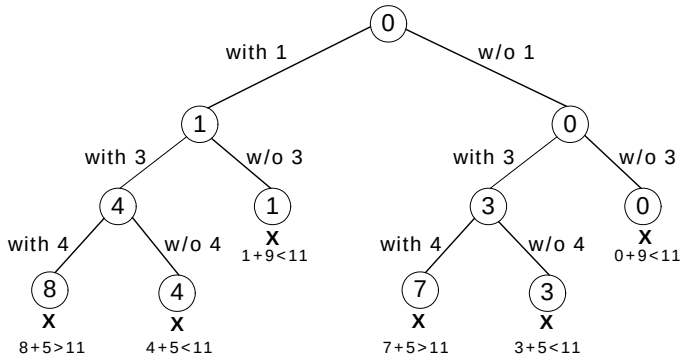
for it using backtracking



7. Here is a state-space tree for generating all permutations of {1, 2, 3, 4} by backtracking:



8. a. Here is a state-space tree for the given instance of the subset-sum problem: $A = \{1, 3, 4, 5\}$ and $d = 11$:



There is no solution to this instance of the problem.

b. The algorithm will still work correctly but the state-space tree will contain more nodes than necessary.

9. Eliminate **else** from the template's pseudocode.

10. n/a

11. n/a

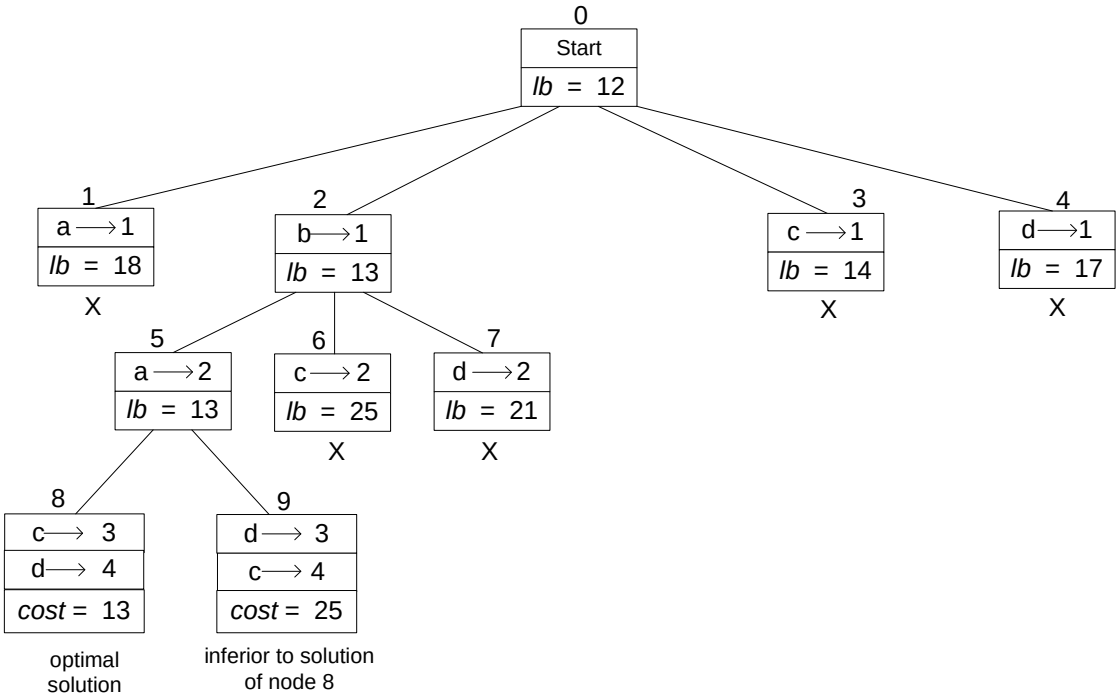
Solutions to Exercises 12.2

1. The heap and min-heap for maximization and minimization problems, respectively.

2. The instance discussed in the section is specified by the matrix

$$C = \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} & \\ 9 & 2 & 7 & 8 & \text{person } a \\ 6 & 4 & 3 & 7 & \text{person } b \\ 5 & 8 & 1 & 8 & \text{person } c \\ 7 & 6 & 9 & 4 & \text{person } d \end{array}$$

Here is the state-space tree in question:



The optimal assignment is $b \to 1$, $a \to 2$, $c \to 3$, $d \to 4$.

3. a. An $n \times n$ matrix whose elements are all the same is one such example.

   b. In the best case, the state-space tree will have just one node developed on each of its levels. Accordingly, the total number of nodes will be

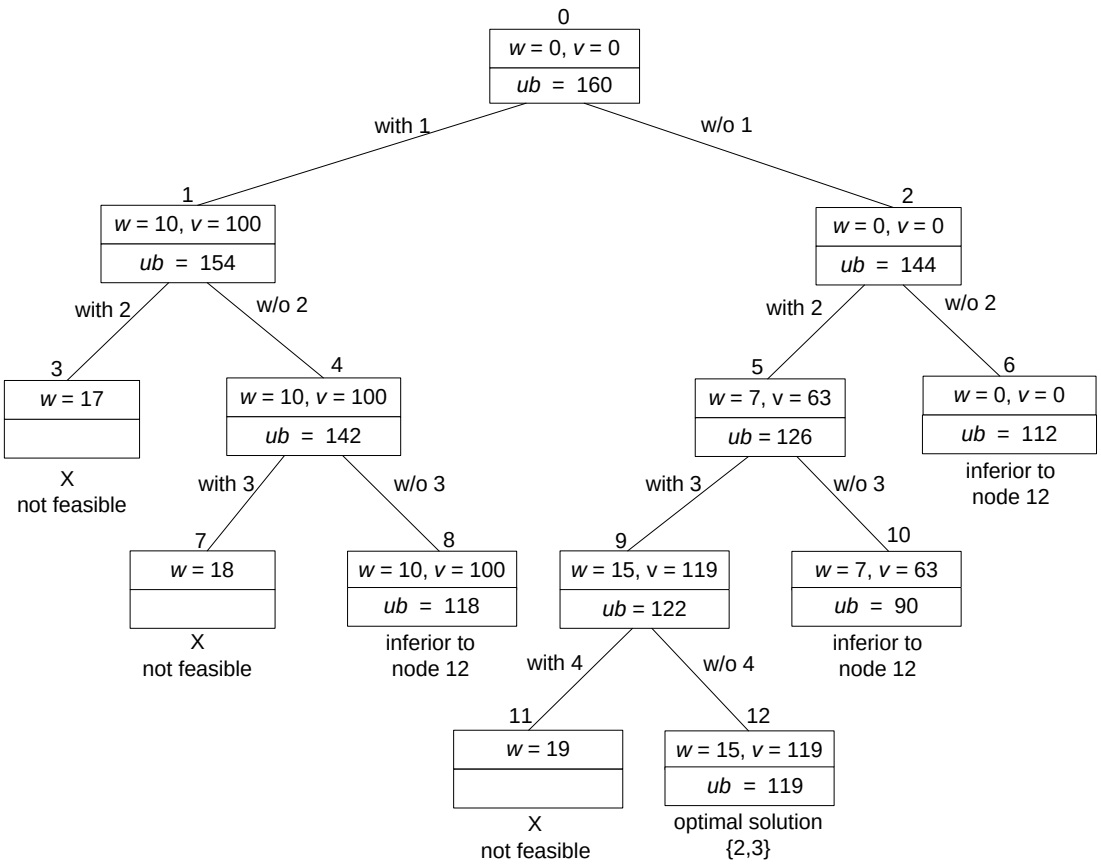
$$1 + n + (n-1) + \cdots + 2 = \frac{n(n+1)}{2}.$$

4. n/a

5. The instance in question is specified by the following table, in which the items are listed in nonincreasing order of their value-to-weight ratios:

item	weight	value	
1	10	\$100	
2	7	\$63	$W = 16$
3	8	\$56	
4	4	\$12	

Here is the state-space tree of the best-first branch-and-bound algorithm with the simple bounding function indicated in the section:

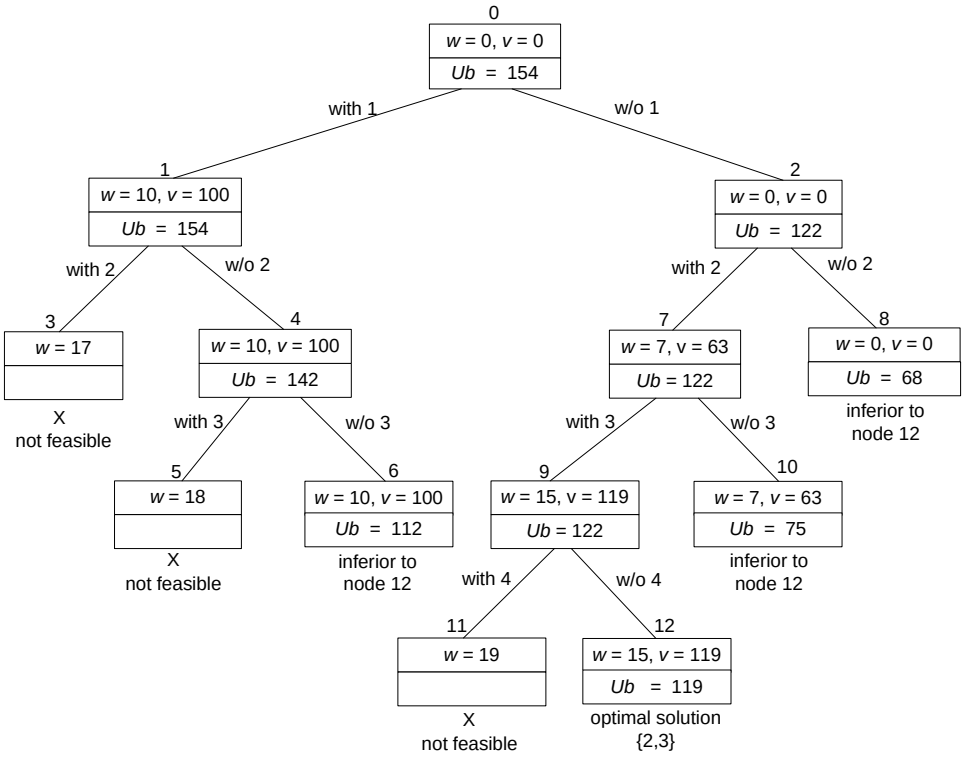


The found optimal solution is {item 2, item 3} of value \$119.

6. a. We assume that the items are sorted in nonincreasing order of their efficiencies: $v_1/w_1 \geq \cdots \geq v_n/w_n$ and, hence, new items are added in this order. Consider a subset $S = \{\text{item } i_1, \ldots, \text{item } i_k\}$ represented by a node in a branch-and-bound tree; the total weight and value of the items in S are $w(S) = w_{i_1} + \cdots + w_{i_k}$ and $v(S) = v_{i_1} + \cdots + v_{i_k}$, respectively. We can compute the upper bound $Ub(S)$ on the value of any subset that can be obtained by adding items to S as follows. We'll add to $v(S)$ the consecutive values of the items not included in S as long as the total weight doesn't exceed the knapsack's capacity. If we do encounter an item that violates this requirement, we'll determine its fraction needed to fill the knapsack to full capacity and use the product of this fraction and that item's efficiency as the last addend in computing $Ub(S)$. For example, for the instance of Problem 5, the upper bound for the root of the tree will be computed as follows: $Ub = 100 + (6/7)63 = 154$.

b. The instance is specified by the following data

item	weight	value	
1	10	\$100	
2	7	\$63	$W = 16$
3	8	\$56	
4	4	\$12	



The found optimal solution is {item 2, item 3} of value \$119.

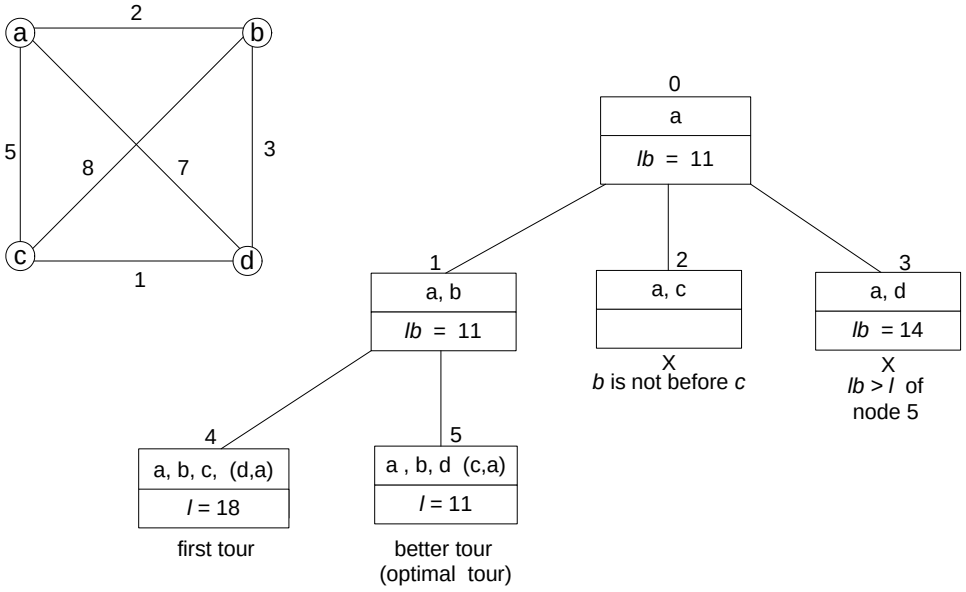
7. n/a

8. a. Any Hamiltonian circuit must have exactly two edges incident with each vertex of the graph: one for leaving the vertex and the other for entering it. The length of the edge leaving the ith vertex, $i = 1, 2, ...n$, is given by some nondiagonal element $D[i, j]$ in the ith row of the distance matrix. The length of the entering edge is given by some nondiagonal element $D[j', i]$ in the ith column of the distance matrix, which is not symmetric to the element in the ith row ($j' \neq j$) because the two edges must be distinct; this element is equal to some other element $D[i, j']$ in the ith row of the symmetric distance matrix. Hence, for any Hamiltonian circuit, the sum of the lengths of two edges incident with the ith vertex must be greater than or equal to s_i, the sum of the two smallest elements (not on the main diagonal of the matrix) in the ith row . Summing up these inequalities for all n vertices yields $2l \geq s$, where l is the length of any tour. Since l must be an integer if all the distances are integers, $l \geq \lceil s/2 \rceil$, which proves the assertion.

   b. Redefine s_i as the sum of the smallest element in the ith row and the smallest element in the ith column (neither on the main diagonal of the matrix).

9. Without loss of generality, we'll consider a as the starting vertex and

ignore the tours in which c is visited before b.



The found optimal tour is a, b, d, c, a of length 11.

10. n/a

Solutions to Exercises 12.3

1. a. The nearest-neighbor algorithm yields the tour $1-3-2-5-4-1$ of length 58.

   b. To compute the accuracy ratio, we'll have to find the length of the optimal tour for the instance given by the distance matrix

$$\begin{bmatrix} 0 & 14 & 4 & 10 & \infty \\ 14 & 0 & 5 & 8 & 7 \\ 4 & 5 & 0 & 9 & 16 \\ 10 & 8 & 9 & 0 & 32 \\ \infty & 7 & 16 & 32 & 0 \end{bmatrix}$$

   Generating all the finite-length tours that start and end at city 1 and visit city 2 before city 3 (see Section 3.4) yields the following:

   $1-2-3-5-4-1$ of length 77
   $1-2-4-5-3-1$ of length 74
   $1-2-5-3-4-1$ of length 56
   $1-2-5-4-3-1$ of length 66
   $1-4-2-5-3-1$ of length 45
   $1-4-5-2-3-1$ of length 58,

   with the tour $1-4-2-5-3-1$ of length 45 being optimal. Hence, the accuracy ratio of the approximate solution obtained by the nearest-neighbor algorithm is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{58}{45} \approx 1.29.$$

2. a. This is a pseudocode of a straightforward implementation.

   Algorithm *NearestNeighbor*($D[1..n, 1..n]$, s)
   //Implements the nearest-neighbor heuristic for the TSP
   //Input: A matrix $D[1..n, 1..n]$ of intercity distances and
   // an index s of the starting city
   //Output: A list $Tour$ of the vertices composing the tour obtained
   for $i \leftarrow 1$ **to** n **do** $Visited[i] \leftarrow$ **false**
   initialize a list $Tour$ with s
   $Visited[s] \leftarrow$ **true**
   $current \leftarrow s$
   for $i \leftarrow 2$ **to** n **do**
   find column j with smallest element in row *current* & unmarked col.
   $current \leftarrow j$
   $Visited[j] \leftarrow$ **true**
   add j to the end of list $Tour$

add s to the end of list $Tour$
return $Tour$

b. With either implementation (see the hint), the algorithm's time efficiency is in $\Theta(n^2)$.

3. The tour in question is *a, b, d, e, c, a* of length 38, which is not the same as the length of the tour based on the counter-clockwise walk around the minimum spanning tree.

4. The assertion in question follows immediately from the following generalization of the triangle inequality. If distances satisfy the triangle inequality, then they also satisfy its extension to an arbitrary positive number k of intermediate cities:

$$d[i, j] \leq d[i, m_1] + d[m_1, m_2] + ... + d[m_{k-1}, m_k] + d[m_k, j].$$

We will prove this by mathematical induction. The basis case of $k = 1$ is the triangle inequality itself:

$$d[i, j] \leq d[i, m_1] + d[m_1, j].$$

For the general case, we assume that for any two cities i and j and any set of $k \geq 1$ cities $m_1, m_2, ..., m_k$

$$d[i, j] \leq d[i, m_1] + d[m_1, m_2] + ... + d[m_{k-1}, m_k] + d[m_k, j]$$

to prove that for any two cities i and j and any set of $k + 1$ cities $m_1, m_2, ..., m_{k+1}$

$$d[i, j] \leq d[i, m_1] + d[m_1, m_2] + ... + d[m_k, m_{k+1}] + d[m_{k+1}, j].$$

Indeed, by the triangle inequality applied to m_k, j, and one intermediate city m_{k+1}, we obtain

$$d[m_k, j] \leq d[m_k, m_{k+1}] + d[m_{k+1}, j].$$

Replacing $d[m_k, j]$ in the inductive assumption by $d[m_k, m_{k+1}]+d[m_{k+1}, j]$ yields the inequality we wanted to prove.

5. Computing n value-to-weight ratios is in $\Theta(n)$. The time efficiency of sorting depends on the sorting algorithm used: it's in $O(n^2)$ for elementary sorting algorithms and in $O(n \log n)$ for advanced sorting algorithms such as mergesort. The third step is in $O(n)$. Hence, the running time of the entire algorithm will be dominated by the sorting step and will be in $\Theta(n) + O(n \log n) + O(n) = O(n \log n)$, provided an efficient sorting algorithm is used.

6. i. Let us prove that

$$f(s^*) \le 2f(s_a)$$

for any instance of the knapsack problem, where $f(s_a)$ is the value of the approximate solution obtained by the enhanced greedy algorithm and $f(s^*)$ is the optimal value of the exact solution for the same instance. In the trivial case when all the items fit into the knapsack, $f(s_a) = f(s^*)$ and the inequality obviously holds. Let I be a nontrivial instance. We can assume without loss of generality that the items are numbered in nonincreasing order of their efficiency (i.e., value to weight) ratios, that each of them fits into the knapsack, and that the item of the largest value has index m. Let s^* be the exact optimal solution for this discrete instance I and let k be the index of the first item that does not fit into the knapsack. Let c^* be the (exact) solution to the continuous counterpart of instance I. We have the following upper estimate for $f(s^*)$:

$$f(s^*) \le f(c^*) < \sum_{i=1}^{k-1} v_i + v_k \le f(s_a) + v_m \le f(s_a) + f(s_a) = 2f(s_a).$$

ii. Consider, for example, the family of instances defined as follows:

item	*weight*	*value*	$\frac{value}{weight}$
1	$(W+1)/2$	$(W+2)/2$	>1
2	$W/2$	$W/2$	1
3	$W/2$	$W/2$	1

with the knapsack's capacity $W \ge 2$, which will serve as the family's parameter.

The optimal solution s^* to any instance of this family is {item 2, item 3} of value W. The approximate solution s_a obtained by the enhanced greedy algorithm is {item 1} of value $(W+2)/2$. Hence,

$$\frac{f(s^*)}{f(s_a)} = \frac{W}{(W+2)/2} = \frac{2}{1+2/W},$$

which can be made as close to 2 as we wish by making W sufficiently large.

7. a. The first-fit algorithm places items 1, 3, and 4 into the first bin, item 2 into the second bin, and item 5 into the third bin. This solution is not optimal because it's inferior to the solution that places items 1 and 5 into one bin and items 2, 3, and 4 into the other. The latter solution is optimal because the two bins is the smallest number possible since $\sum_{i=1}^{5} s_i > 1$ and hence all the items cannot fit into a single bin.

b. The basis operation is to check whether the current item fits into a particular bin. (It can be done in constant time if the amount of remaining space is maintained for each started bin.) The worst-case input will force the algorithm to check all $i-1$ started bins when placing the ith item for $i = 2, ..., n$. For example, this will happen for any input with items of the same size that is greater than 0.5. Hence, the worst-case efficiency of the first-fit is quadratic because

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

(Note: If we don't assume that the size of each input item doesn't exceed the bin's capacity, the algorithm will have to make i comparisons on its ith iteration in the worst case. This will not change the worst-case efficiency class of the algorithm since $\sum_{i=1}^{n} i = n(n+1)/2 \in \Theta(n^2)$.)

c. Obviously, *FF* is a polynomial time algorithm. We'll prove first that for any instance of the problem that requires more than one bin (i.e., $\sum_{i=1}^{n} s_i > 1$),

$$B_{FF} < 2\sum_{i=1}^{n} s_i,$$

where B_{FF} is the number of bins in the approximate solution obtained by the first-fit (*FF*) algorithm. In any solution obtained by this algorithm, there are no more than one bin that is half full or less, i.e., $S_k \le 0.5$, where S_k is the sum of the sizes of the items that go into bin k, $k = 1, 2, ..., B_{FF}$. (Indeed, if there were more that one such bin, the algorithm would've placed all the items in the later-filled bin of the two into the earlier-filled one at the latest.) Let $\tilde{k}$ be the index of the bin in the solution with the smallest sum of the item sizes in it and let $\bar{k}$ be the index of any other bin in the solution. (Since $\sum_{i=1}^{n} s_i > 1$, such other bin must exist.) The sum S of the sizes of the items in these two bins must exceed 1. Therefore, we have the following:

$$\sum_{i=1}^{n} s_i = \sum_{k=1}^{B_{FF}} S_k = \sum_{k=1,\ k\neq\tilde{k},\ k\neq\bar{k}}^{B_{FF}} S_k + S > 0.5(B_{FF}-2) + 1 = 0.5B_{FF},$$

which proves the inequality $B_{FF} < 2\sum_{i=1}^{n} s_i$. Combining this with the obvious observation that $\sum_{i=1}^{n} s_i \le B^*$, we obtain

$$B_{FF} < 2\sum_{i=1}^{n} s_i \le 2B^*.$$

For the trivial case of $\sum_{i=1}^{n} s_i \le 1$, *FF* puts all the items in the first bin, and hence $B_{FF} = B^* < 2B^*$ as well.

Note: The best (and tight) bound for the first fit is

$$B_{FF} \leq \lceil 1.7B^* \rceil \text{ for all inputs.}$$

(See the survey by Coffman et al. in "Approximation Algorithms for *NP*-hard Problems," edited by D.S. Hochbaum, PWS Publishing, 1995.)

8. a. The first-fit decreasing algorithm (*FFD*) places items of sizes 0.7, 0.2, and 0.1 in the first bin and items of sizes 0.5 and 0.4 in the second one. Since $B^* \geq \lceil \sum_{i=1}^{5} s_i \rceil = 2$, at least two bins are necessary, making the solution obtained by *FFD* optimal.

   b. The answer is no: if it did, *FFD* would be a polynomial time algorithm that solves this *NP*-hard problem. Here is one counterexample:

   $$s_1 = 0.5, \quad s_2 = 0.4, \quad s_3 = 0.3, \quad s_4 = 0.3, \quad s_5 = 0.25, \quad s_6 = 0.25.$$

   (*FFD* yields a solution with 3 bins while the optimal number of bins is 2.)

   c. Obviously, *FFD* is a polynomial time algorithm. If *FFD* yields B_{FFD} bins while the optimal number of bins is B^*, we know from the properties quoted in the hint that the number of items in the extra bins is at most $B^* - 1$, with each of the items be of size at most 1/3. Therefore the total number of extra bins is at most $\lceil (B^* - 1)/3 \rceil$, and we have the following upper bound on the approximation's accuracy ratio:

   $$B_{FFD} \leq B^* + \lceil (B^* - 1)/3 \rceil \leq B^* + \frac{B^* + 1}{3}.$$

   (You can check the validity of the last replacement of $\lceil (B^* - 1)/3 \rceil$ by $(B^* + 1)/3$ by considering separately three cases: $B^* = 3i$, $B^* = 3i + 1$, and $B^* = 3i + 2$.). Finally,

   $$B^* + \frac{B^* + 1}{3} = (\frac{4}{3} + \frac{1}{3B^*})B^* \leq (\frac{4}{3} + \frac{1}{3 \cdot 2})B^* = 1.5B^*.$$

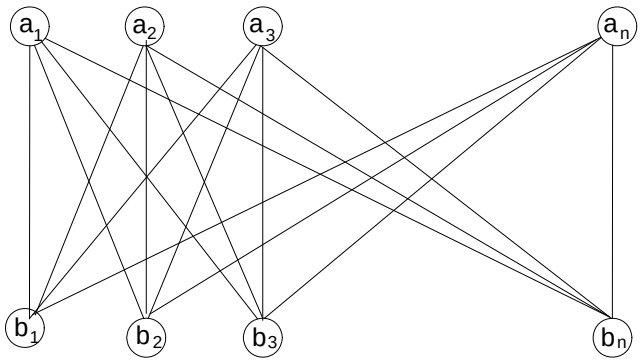
   (We used the observation that when $B^* \geq 2$, $\frac{1}{3B^*}$ is the largest when $B^* = 2$. When $B^* = 1$, *FFD* yields an exact solution, and the inequality $B_{FFD} \leq 1.5B^*$ checks out directly.)

   d. Note the two versions of this task. The easy one would simply compare which of the two greedy algorithms yields a more accurate solution more often. It is easy because, in this form, one doesn't need to know the optimal number of bins in a generated instance. The much more difficult version is to compare the average accuracy of the two approximation algorithms, which would require information about the optimal number of bins in each of the generated instances.

9. a. Initialize the vertex cover to the empty set. Repeat the following until no edges are left: select an arbitrary edge, add both its endpoints to the vertex cover, and remove from the graph all the edges incident with either of these two endpoint vertices.

   Let $L = e_1, e_2, ..., e_k$ be the list of edges chosen by the algorithm. The number of vertices in the vertex cover the algorithm returns, $|VC_a|$, is $2k$. Since no two edges in L have a common vertex, a minimum vertex cover of the graph must include at least one endpoint of each edge in L; therefore the number of vertices in it, $|VC^*|$, is at least k. Hence $|VC_a| \leq 2|VC^*|$.

   b. No. Consider, for example, the complete bipartite graph $K_{n,n}$ with vertices a_1, b_1, ..., a_n, b_n:



   Selecting edges (a_i, b_i), $i = 1, 2, ..., n$, in the 2-approximation vertex-cover algorithm of part a, yields the vertex cover with $2n$ vertices and hence 0 independent vertices. The maximum independent set has, in fact, n vertices (all a vertices or all b vertices).

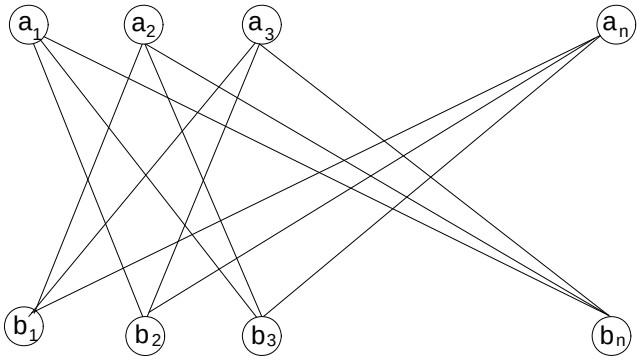
10. a. The simplest greedy heuristic, called *sequential coloring*, is to color a vertex in the first available color, i.e., the first color not used for coloring any vertex adjacent to it. (Vertices are colored in the order given by the graph's data structure.)

    Algorithm $SC(G)$
    //Implements sequential coloring of a given graph
    //Input: A graph $G = \langle V, E \rangle$
    //Output: An array $Color$ of numeric colors assigned to the vertices
    for $i \leftarrow 1$ **to** $|V|$ **do**
        $Color[i] = 0$ //0 signifies no color
    for $i \leftarrow 1$ **to** $|V|$ **do**
        $c \leftarrow 1$
        while $Color[i] = 0$ **do**
            if no vertex adjacent to v_i has color c

$Color[i] \leftarrow c$
else $c \leftarrow c + 1$
return $Color$

The algorithm's time efficiency is clearly in $O(|V|^3)$ because for each of the $|V|$ vertices, the algorithm checks no more than $|V|$ colors for up to $O(|V|)$ vertices adjacent to it.

b. Consider the following sequence of graphs G_n with $2n$ vertices specified in the order a_1, b_1, ..., a_n, b_n:



The smallest number of colors is 2 (color all the a_i's with color 1 and all the b_i's with color 2). The sequential coloring yields n colors: one for each pair of vertices a_i and b_i. Hence, for this sequence of graphs,

$$\frac{\varkappa_a(G_n)}{\varkappa^*(G_n)} = \frac{n}{2},$$

which is not bounded above.

Solutions to Exercises 12.4

1. a. Here is a solution as described at http://www.sosmath.com/algebra/factor/fac11/fac11.html:

   First, substitute $x = y - b/3a$ to reduce the general cubic equation

   $$ax^3 + bx^2 + cx + d = 0$$

   to the "depressed" cubic equation

   $$y^3 + Ay = B.$$

   Then solve the system

   $$\begin{aligned} 3st &= A \\ s^3 - t^3 &= B \end{aligned}$$

   by substituting $s = A/3t$ into the second equation to get the "tri-quadratic" equation for t

   $$t^6 + Bt^3 - \frac{A^3}{27} = 0.$$

   (The last equation can be solved as a quadratic equation after substitution $u = t^3$.) Finally,

   $$y = s - t$$

   yields the y's value, from which we get the root as

   $$x = y - b/3a.$$

   b. Transform-and-conquer.

2. a. Equation $xe^x - 1 = 0$ is equivalent to $e^x = 1/x$. The graphs of $f_1(x) = e^x$ and $f_2(x) = 1/x$ clearly have a single common point between 0 and 1.

   b. Equation $x - \ln x = 0$ is equivalent to $x = \ln x$, and the graphs of $f_1(x) = x$ and $f_2(x) = \ln x$ clearly don't intersect.

   c. Equation $x \sin x - 1 = 0$ is equivalent to $\sin x = 1/x$, and the graphs of $f_1(x) = \sin x$ and $f_2(x) = 1/x$ clearly intersect at infinitely many points.

3. a. For a polynomial $p(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$ of an odd degree, where $a_n > 0$,

   $$\lim_{x \to -\infty} p(x) = -\infty \quad \text{and} \quad \lim_{x \to +\infty} p(x) = +\infty.$$

Therefore there exist real numbers a and b, $a < b$, such that $p(a) < 0$ and $p(b) > 0$. In addition, any polynomial is a continuous function everywhere. Hence, by the theorem mentioned in conjunction with the bisection method, $p(x)$ must have a root between a and b.
The case of $a_n < 0$ is reduced to the one with the positive coefficient by considering $-p(x)$.

b. By definition of division of $p(x)$ by $x - x_0$, where x_0 is a root of $p(x)$,

$$p(x) = q(x)(x - x_0) + r,$$

where $q(x)$ and r are the quotient and remainder of this division, respectively. Substituting x_0 for x into the above equation and taking into account that $p(x_0) = 0$ proves that remainder r is equal to 0:

$$p(x_0) = r = 0.$$

Hence,

$$p(x) = q(x)(x - x_0).$$

This implies that if one root of an n-degree polynomial $p(x)$ is known, the other roots can be found by solving

$$q(x) = 0,$$

where $q(x)$—the quotient of the division of $p(x)$ by $x - x_0$ (x_0 is a known root)—is a polynomial of degree $n - 1$.

c. Differentiating both hand sides of equality

$$p(x) = q(x)(x - x_0)$$

yields

$$p'(x) = q'(x)(x - x_0) + q(x).$$

Substituting x_0 for x results in

$$p'(x_0) = q'(x_0)(x_0 - x_0) + q(x_0) = q(x_0).$$

4. Since x_n is the middle point of interval $[a_n, b_n]$, its distance to any point within that interval, including root x^*, cannot exceed the interval's half length, which is $(b_n - a_n)/2$. That is

$$|x_n - x^*| \leq \frac{b_n - a_n}{2} \text{ for } n = 1, 2, ...$$

But the length of the intervals $[a_n, b_n]$ is halved on each iteration. Hence,

$$b_n - a_n = \frac{b_{n-1} - a_{n-1}}{2} = \frac{b_{n-2} - a_{n-2}}{2^2} = ... = \frac{b_1 - a_1}{2^{n-1}}.$$

(Use mathematical induction, if you prefer a more formal proof.) Thus,

$$\frac{b_n - a_n}{2} = \frac{b_1 - a_1}{2^n}.$$

Substituting this in the inequality above yields

$$|x_n - x^*| \leq \frac{b_1 - a_1}{2^n} \quad \text{for } n = 1, 2, ...$$

5. The graph of $f(x) = x^3 + x - 1$ makes it obvious that this polynomial has a single real root that lies in the interval $0 < x < 1$. (It also follows from the fact that this polynomial has an odd degree and its derivative is positive for every x.) Solving inequality (12.8) with $a = 0$, $b = 1$, and $\epsilon = 10^{-2}$, i.e.,

$$n > \log_2 \frac{1 - 0}{10^{-2}},$$

yields $n \geq 7$. The following table contains the results of the first seven iterations of the bisection method:

n	a_n	b_n	x_n	$f(x_n)$
1	0.0-	1.0+	0.5	-0.375
2	0.5-	1.0+	0.75	0.171875
3	0.5-	0.75+	0.625	-0.130859
4	0.625-	0.75+	0.6875	0.012451
5	0.625-	0.6875+	0.65625	-0.061127
6	0.65625-	0.6875+	0.671875	-0.024830
7	0.671875-	0.6875+	0.6796875	-0.006314

Thus, the obtained approximation is $x_7 = 0.6796875$.

6. Substituting $(a_n, f(a_n))$ and $(b_n, f(b_n))$, the two given points, into the standard straight-line equation

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1),$$

we obtain

$$y - f(a_n) = \frac{f(b_n) - f(a_n)}{b_n - a_n}(x - a_n).$$

Setting y to 0 to find the line's x-intercept, we obtain the following equation for x_n

$$-f(a_n) = \frac{f(b_n) - f(a_n)}{b_n - a_n}(x_n - a_n).$$

Solving for x_n yields

$$x_n = a_n - \frac{f(a_n)(b_n - a_n)}{f(b_n) - f(a_n)},$$

or, after standard algebraic simplifications,

$$x_n = \frac{a_n f(b_n) - f(a_n) b_n}{f(b_n) - f(a_n)},$$

which is the formula for the approximation sequence of the method of false position.

7. For $f(x) = x^3 + x - 1$,

$$f'(x) = 3x^2 + 1 \geq 1 \text{ for every } x.$$

Hence, according to inequality (12.12), we can stop the iterations as soon as

$$|x_n - x^*| \leq |f(x_n)| < 10^{-2}.$$

The following table contains the results of the first four iterations of the method of false position:

n	a_n	b_n	x_n	$f(x_n)$
1	0.0-	1.0+	0.5	-0.375
2	0.5-	1.0+	0.636364	-0.105935
3	0.636364-	1.0+	0.671196	-0.026428
4	0.671196-	1.0+	0.679662	-0.006375

Thus, the obtained approximation is $x_4 = 0.679662$.

8. Using the standard equation for the tangent line to the graph of the function $f(x)$ at $(x_n, f(x_n))$, we obtain

$$y - f(x_n) = f'(x_n)(x - x_n).$$

Setting y to 0 to find its x-intercept, which is x_{n+1} of Newton's method, yields

$$-f(x_n) = f'(x_n)(x_{n+1} - x_n).$$

Solving for x_{n+1} yields, if $f'(x_n) \neq 0$,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

which is the formula for the approximation sequence of Newton's method.

9. For $f(x) = x^3 + x - 1$,

$$f'(x) = 3x^2 + 1 \geq 1 \text{ for every } x.$$

Hence, according to inequality (12.12), we can stop the iterations as soon as

$$|x_n - x^*| \leq |f(x_n)| < 10^{-2}.$$

The following table contains the results of the first two iterations of Newton's method, with $x_0 = 1$:

n	x_n	x_{n+1}	$f(x_n)$
0	1.0	0.75	0.171875
1	0.75	0.686047	0.008941

Thus, the obtained approximation is $x_2 = 0.686047$.

10. Equation $\sqrt[3]{x} = 0$ has $x = 0$ as its only root. Using the geometric interpretation of Newton's method, it is easy to see that the approximation sequence (the x-intercepts of the tangent lines) diverges for any initial approximation $x_0 \neq 0$. Here is a formal proof of this fact. The approximation sequence of Newton's method is given by the formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^{1/3}}{\frac{1}{3}x_n^{-2/3}} = x_n - 3x_n = -2x_n.$$

This equality means that if $x_0 \neq 0$, each next approximation x_{n+1} is twice as far from 0, the equation's root, as its predecessor x_n. Hence, sequence $\{x_n\}$ diverges for any initial approximation $x_0 \neq 0$.

11. You can find a solution to this classic puzzle at http://plus.maths.org/issue9/puzzle/solution.html (retrieved Nov. 24, 2011).