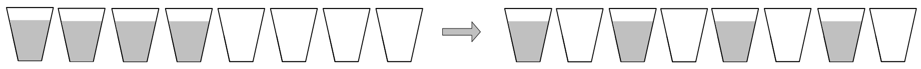
This file contains the exercises, hints, and solutions for Chapter 4 of the book "Introduction to the Design and Analysis of Algorithms," 3d edition, by A. Levitin. The problems that might be challenging for at least some students are marked by ▷; those that might be difficult for a majority of students are marked by ▶ .

Exercises 4.1

1. *Ferrying soldiers* A detachment of n soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier. How can the soldiers get across the river and leave the boys in joint possession of the boat? How many times need the boat pass from shore to shore?

2. *Alternating glasses* a. There are $2n$ glasses standing next to each other in a row, the first n of them filled with a soda drink while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of glass moves.



   b. Solve the same problem if $2n$ glasses—n with a drink and n empty—are initially in a random order.

3. *Marking cells* Design an algorithm for the following task. For any even n, mark n cells on an infinite sheet of graph paper so that each marked cell has an odd number of marked neighbors. Two cells are considered neighbors if they are next to each other either horizontally or vertically but not diagonally. The marked cells must form a contiguous region, that is a region in which there is a path between any pair of marked cell that goes through a sequence of marked neighbors. [Kor05]

4. Design a decrease-by-one algorithm for generating the power set of a set of n elements. (The power set of a set S is the set of all the subsets of S, including the empty set and S itself.)

5. Consider the following algorithm to check connectivity of a graph defined by its adjacency matrix.

```
Algorithm Connected(A[0..n − 1, 0..n − 1])

//Input: Adjacency matrix A[0..n − 1, 0..n − 1]) of an undirected graph G
//Output: 1 (true) if G is connected and 0 (false) if it is not
if n = 1 return 1      //one-vertex graph is connected by definition
else
    if not Connected(A[0..n − 2, 0..n − 2]) return 0
```

```
        else for $j \leftarrow 0$ to $n-2$ do
                if $A[n-1, j]$ return 1
            return 0
```

Does this algorithm work correctly for every undirected graph with $n > 0$ vertices? If you answer "yes," indicate the algorithm's efficiency class in the worst case; if you answer "no," explain why.

6. *Team ordering* You have results of a completed round-robin tournament in which n teams played each other once. Each game ended either with a victory of one the teams or with a tie. Design an algorithm that lists the teams in a sequence so that every team did not loose the game with the team listed immediately after it. What is the time efficeincy class of your algorithm?

7. Apply insertion sort to sort the list E, X, A, M, P, L, E in alphabetical order.

8. a. What sentinel should be put before the first element of an array being sorted in order to avoid checking the in-bound condition $j \geq 0$ on each iteration of the inner loop of insertion sort?

   b. Will the version with the sentinel be in the same efficiency class as the original version?

9. Is it possible to implement insertion sort for sorting linked lists? Will it have the same $O(n^2)$ efficiency as the array version?

10. Consider the following version of insertion sort.

```
Algorithm InsertSort2($A[0..n-1]$)
for $i \leftarrow 1$ to $n-1$ do
    $j \leftarrow i-1$
    while $j \geq 0$ and $A[j] > A[j+1]$ do
        swap($A[j]$, $A[j+1]$)
        $j \leftarrow j-1$
```

    What is its time efficiency? How is it compared to that of the version given in the text?

11. Let $A[0..n-1]$ be an array of n sortable elements. (For simplicity, you can assume that all the elements are distinct.) Recall that a pair of its elements $(A[i], A[j])$ is called an ***inversion*** if $i < j$ and $A[i] > A[j]$.

    a. What arrays of size n have the largest number of inversions and what is this number? Answer the same questions for the smallest number of inversions.

b.▶ Show that the average-case number of key comparisons in insertion sort is given by the formula

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

12. Shellsort (more accurately Shell's sort) is an important sorting algorithm which works by applying insertion sort to each of several interleaving sublists of a given list. On each pass through the list, the sublists in question are formed by stepping through the list with an increment h_i taken from some predefined decreasing sequence of step sizes, $h_1 > ... > h_i > ... > 1$, which must end with 1. (The algorithm works for any such sequence, though some sequences are known to yield a better efficiency than others. For example, the sequence 1, 4, 13, 40, 121, ... , used, of course, in reverse, is known to be among the best for this purpose.)

a. Apply shellsort to the list

$$S,\ H,\ E,\ L,\ L,\ S,\ O,\ R,\ T,\ I,\ S,\ U,\ S,\ E,\ F,\ U,\ L$$

b. Is shellsort a stable sorting algorithm?

Hints to Exercises 4.1

1. Solve the problem for $n = 1$.

2. You may consider pouring soda from a filled glass into an empty glass as one move.

3. It's easier to use the bottom-up approach.

4. Use the fact that all the subsets of an n-element set $S = \{a_1, ..., a_n\}$ can be divided into two groups: those that contain a_n and those that do not.

5. The answer is "no."

6. Use the same idea that underlies insertion sort.

7. Trace the algorithm as we did in the text for another input (see Fig. 4.4).

8. a. The sentinel should stop the smallest element from moving beyond the first position in the array.

   b. Repeat the analysis performed in the text for the sentinel version.

9. Recall that we can access elements of a singly linked list only sequentially.

10. Since the only difference between the two versions of the algorithm is in the inner loop's operations, you should estimate the difference in the running times of one repetition of this loop.

11. a. Answering the questions for an array of three elements should lead to the general answers.

    b. Assume for simplicity that all elements are distinct and that inserting $A[i]$ in each of the $i + 1$ possible positions among its predecessors is equally likely. Analyze the sentinel version of the algorithm first.

12. a. Note that it is more convenient to sort sublists in parallel, i.e., compare $A[0]$ with $A[h_i]$, then $A[1]$ with $A[1 + h_i]$, and so on.

    b. Recall that, generally speaking, sorting algorithms that can exchange elements far apart are not stable.

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.

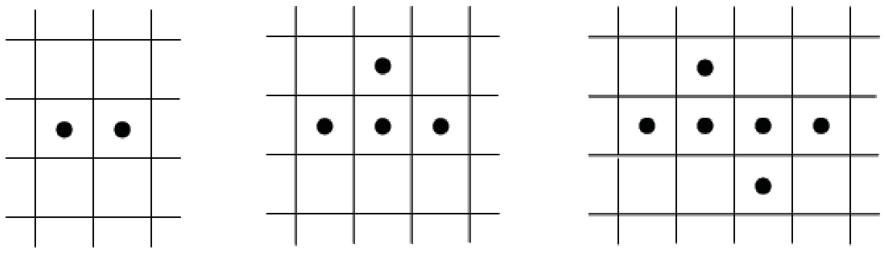
   For an alternative algorithm, see a more general version of the problem in part (b).

   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 $(0 \leq k \leq n)$ 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:

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

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.)

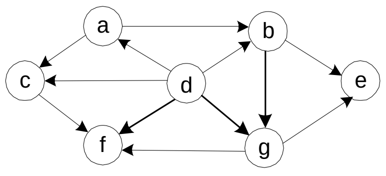
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S_1	H	E_1	L_1	L_2	S_2	O	R	T	I	S_3	U_1	S_4	E_2	F	U_2	L_3
E_2													S_1			
	F													H		
		E_1													U_2	
			L_1													L_3
E_2				L_2												
	F				S_2											
		E_1				O										
			L_1				R									
				L_2				T								
					I		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

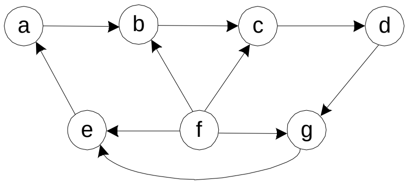
b. Shellsort is not stable. As a counterexample for shellsort with the sequence of step-sizes 4 and 1, consider, say, the array 5, 1, 2, 3, 1. The first pass with the step-size of 4 will exchange 5 with the last 1, changing the relative ordering of the two 1's in the array. The second pass with the step-size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

Exercises 4.2

1. Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs:



(a)



(b)

.

2. a. Prove that the topological sorting problem has a solution for a digraph if and only if it is a dag.

   b. For a digraph with n vertices, what is the largest number of distinct solutions the topological sorting problem can have?

3. a. What is the time efficiency of the DFS-based algorithm for topological sorting?

   b. How can one modify the DFS-based algorithm to avoid reversing the vertex ordering generated by DFS?

4. Can one use the order in which vertices are pushed onto the DFS stack (instead of the order they are popped off it) to solve the topological sorting problem?

5. Apply the source-removal algorithm to the digraphs of Problem 1.

6. a. Prove that a dag must have at least one source.

   b. How would you find a source (or determine that such a vertex does not exist) in a digraph represented by its adjacency matrix? What is the time efficiency of this operation?

   c. How would you find a source (or determine that such a vertex does not exist) in a digraph represented by its adjacency lists? What is the time efficiency of this operation?

7. $\triangleright$ Can you implement the source-removal algorithm for a digraph represented by its adjacency lists so that its running time is in $O(|V| + |E|)$?

8. Implement the two topological sorting algorithms in the language of your choice. Run an experiment to compare their running times.

9. A digraph is called ***strongly connected*** if for any pair of two distinct vertices u and v, there exists a directed path from u to v and a directed path

from v to u. In general, a digraph's vertices can be partitioned into disjoint maximal subsets of vertices that are mutually accessible via directed paths of the digraph; these subsets are called ***strongly connected components***. There are two DFS-based algorithms for identifying strongly connected components. Here is the simpler (but somewhat less efficient) one of the two:

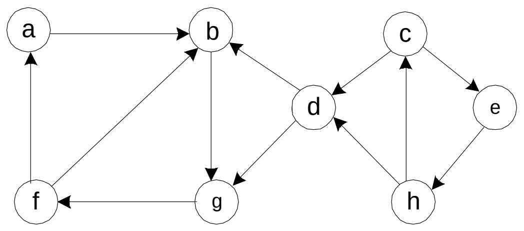
Step 1 Do a DFS traversal of the digraph given and number its vertices in the order that they become dead ends.

Step 2 Reverse the directions of all the edges of the digraph.

Step 3 Do a DFS traversal of the new digraph by starting (and, if necessary, restarting) the traversal at the highest numbered vertex among still unvisited vertices.

The strongly connected components are exactly the subsets of vertices in each DFS tree obtained during the last traversal.

a. Apply this algorithm to the following digraph to determine its strongly connected components.

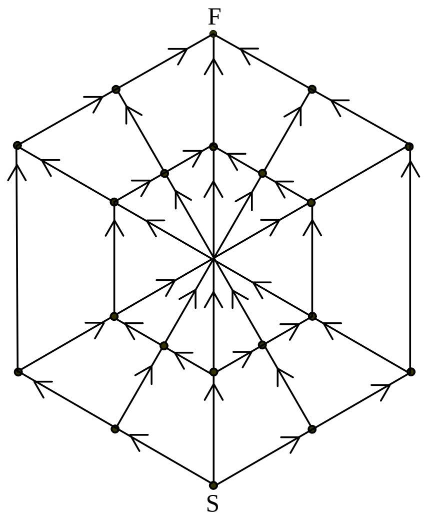


b. What is the time efficiency class of this algorithm? Give separate answers for the adjacency matrix representation and adjacency list representation of an input graph.

c. How many strongly connected components does a dag have?

10. *Spiders's web* A spider sits at the bottom (point S) of its web, while a fly sits at the top (F). How many different ways can the spider reach the fly by moving along the web's lines in the directions indicated by the arrows?

[Kor05]



Hints to Exercises 4.2

1. Trace the algorithm as it is done in the text for another digraph (see Fig. 4.7).

2. a. You need to prove two assertions: (i) if a digraph has a directed cycle, then the topological sorting problem does not have a solution; (ii) if a digraph has no directed cycles, the problem has a solution.

   b. Consider an extreme type of a digraph.

3. a. How does it relate to the time efficiency of DFS?

   b. Do you know the length of the list to be generated by the algorithm? Where should you put, say, the first vertex being popped off a DFS traversal stack for the vertex to be in its final position?

4. Try to do this for a small example or two.

5. Trace the algorithm on the instances given as it is done in the section (see Fig. 4.8).

6. a. Use a proof by contradiction.

   b. If you have difficulty answering the question, consider an example of a digraph with a vertex with no incoming edges and write down its adjacency matrix.

   c. The answer follows from the definitions of the source and adjacency lists.

7. For each vertex, store the number of edges entering the vertex in the remaining subgraph. Maintain a queue of the source vertices.

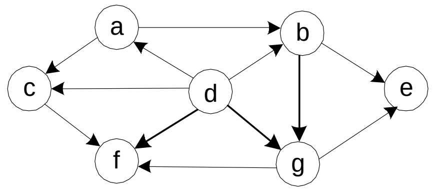
9. a. Trace the algorithm on the input given by following the steps of the algorithm as indicated.

   b. Determine the efficiency for each of the three principal steps of the algorithm and then determine the overall efficiency. Of course, the answers depend on whether a digraph is represented by its adjacency matrix or by its adjacency lists.

10. Take advantage of topological sorting and the graph's symmetry.

Solutions to Exercises 4.2

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



```
      f
e     g
b          c
a               d
```

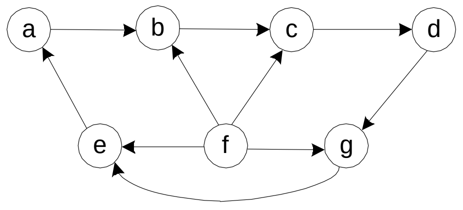
The vertices are popped off the stack in the following order:

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

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:



```
e
g
d
c
b
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.

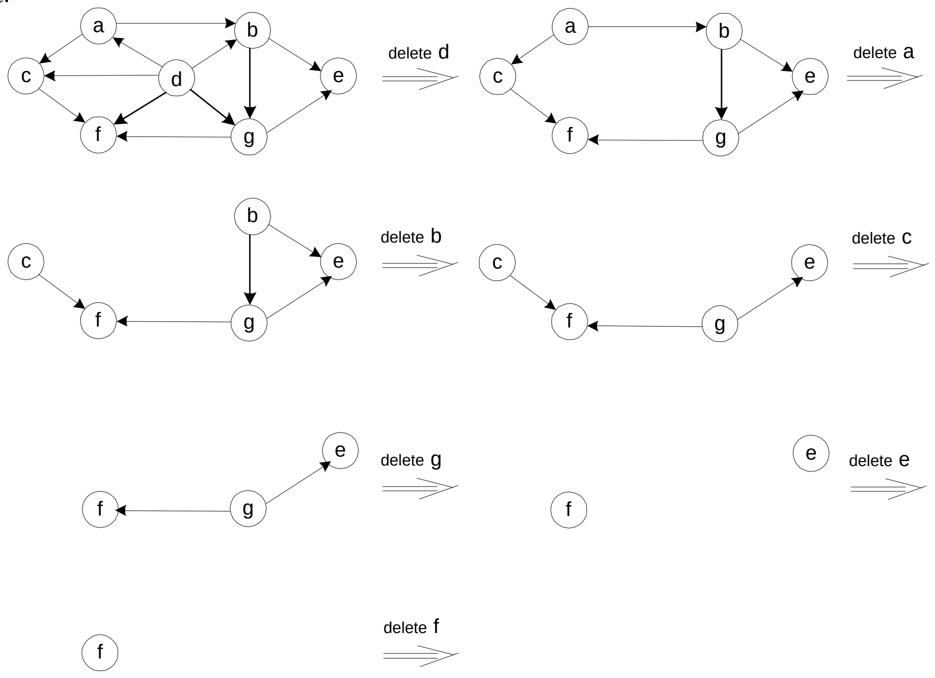
   b. Fill the array of length $|V|$ with vertices being popped off the DFS traversal stack right to left.

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

$$\begin{tikzcd} a \arrow[r] & b & c \arrow[l] \end{tikzcd}$$

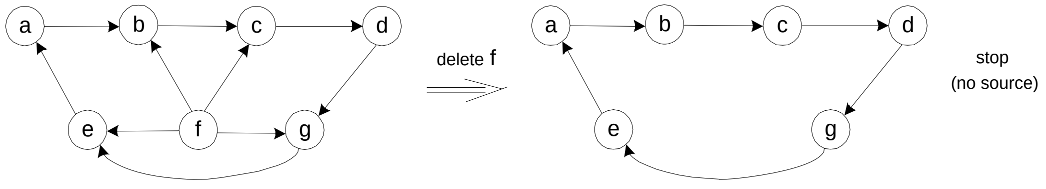
   The DFS traversal that starts at a pushes the vertices on the stack in the order a, b, c, and neither this ordering nor its reversal solves the topological sorting problem correctly.

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.

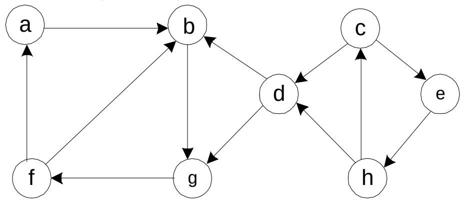
   b. A vertex of a dag is a source if and only if its column in the adjacency matrix contains only 0's. Looking for such a column is a $O(|V|^2)$ operation.

   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

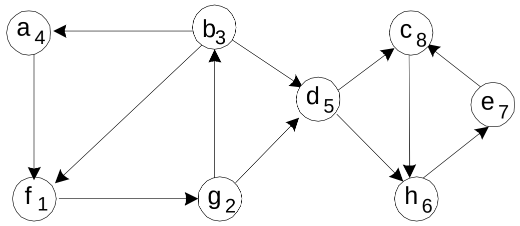


The stack of the first DFS traversal, with a as its starting vertex, will look as follows:

$$\begin{array}{lll} f_1 & & \\ g_2 & & h_6 \\ b_3 & d_5 & e_7 \\ a_4 & c_8 & \end{array}$$

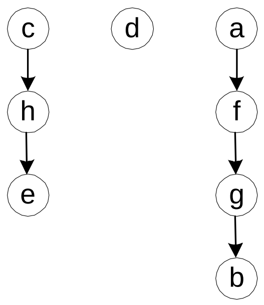
(The numbers indicate the order in which the vertices are popped off the stack.)

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:

$$\begin{array}{lll} & & b \\ e & & g \\ h & & f \\ c & d & a \end{array}$$



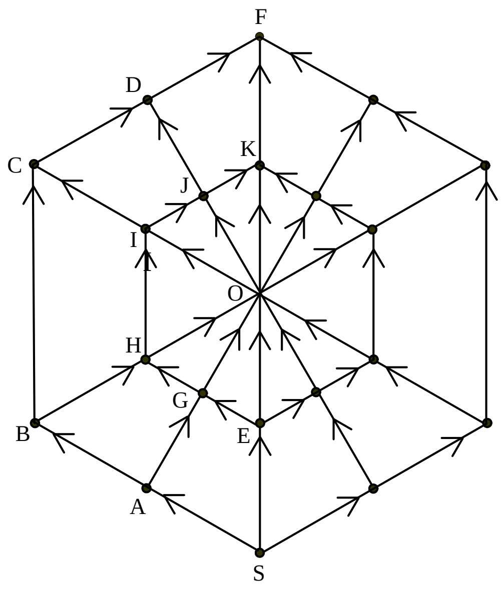
The strongly connected components of the given digraph are:

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

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.

Exercises 4.3

1. Is it realistic to implement an algorithm that requires generating all permutations of a 25-element set on your computer? What about all the subsets of such a set?

2. Generate all permutations of $\{1, 2, 3, 4\}$ by

   a. the bottom-up minimal-change algorithm.

   b. the Johnson-Trotter algorithm.

   c. the lexicographic–order algorithm.

3. Write a program for generating permutations in lexicographic order.

4. ► Consider a simple implementation of the following algorithm for generating permutations discovered by B. Heap [Hea63].

```
**Algorithm** *HeapPermute*($n$)
//Implements Heap's algorithm for generating permutations
//Input: A positive integer $n$ and a global array $A[1..n]$
//Output: All permutations of elements of $A$
if $n = 1$
    write $A$
else
    for $i \leftarrow 1$ to $n$ do
        HeapPermute($n - 1$)
        if $n$ is odd
            swap $A[1]$ and $A[n]$
        else swap $A[i]$ and $A[n]$
```

   a. Trace the algorithm by hand for $n = 2$, 3, and 4.

   b. Prove the correctness of Heap's algorithm.

   c. What is the time efficiency of this algorithm?

5. Generate all the subsets of a four-element set $A = \{a_1, a_2, a_3, a_4\}$ by each of the two algorithms outlined in this section.

6. What simple trick would make the bit string–based algorithm generate subsets in squashed order?

7. Write pseudocode for a recursive algorithm for generating all 2^n bit strings of length n.

8. Write a nonrecursive algorithm for generating 2^n bit strings of length n that implements bit strings as arrays and does not use binary additions.

9. a. Generate the binary reflexive Gray code of order 4.

   b. Trace the following nonrecursive algorithm to generate the binary reflexive Gray code of order 4. Start with the n-bit string of all 0's. For $i = 1, 2, ..., 2^{n-1}$, generate the ith bit string by flipping bit b in the previous bit string, where b is the position of the least significant 1 in the binary representation of i.

10. ▶ Design a decrease-and-conquer algorithm for generating all combinations of k items chosen from n, i.e., all k-element subsets of a given n-element set. Is your algorithm a minimal-change algorithm?

11. *Gray code and the Tower of Hanoi*

    (a) ▷ Show that the disk moves made in the classic recursive algorithm for the Tower-of-Hanoi puzzle can be used for generating the binary reflected Gray code.

    (b) ▶ Show how the binary reflected Gray code can be used for solving the Tower-of-Hanoi puzzle.

12. *Fair attraction* In olden days, one could encounter the following attraction at a fair. A light bulb was connected to several switches in such a way that it lighted up only when all the switches were closed. Each switch was controlled by a push button; pressing the button toggled the switch, but there was no way to know the state of the switch. The object was to turn the light bulb on. Design an algorithm to turn on the light bulb with the minimum number of button pushes needed in the worst case for n switches.

Hints to Exercises 4.3

1. Use standard formulas for the numbers of these combinatorial objects. For the sake of simplicity, you may assume that generating one combinatorial object takes the same time as, say, one assignment.

2. We traced the algorithms on smaller instances in the section.

3. See an outline of this algorithm in the section.

4. a. Trace the algorithm for $n = 2$; take advantage of this trace in tracing the algorithm for $n = 3$ and then use the latter for $n = 4$.

   b. Show that the algorithm generates $n!$ permutations and that all of them are distinct. Use mathematical induction.

   c. Set up a recurrence relation for the number of swaps made by the algorithm. Find its solution and the solution's order of growth. You may need the formula: $e \approx \sum_{i=0}^{n} \frac{1}{i!}$ for large values of n.

5. We traced both algorithms on smaller instances in the section.

6. Tricks become boring after they have been given away.

7. This is not a difficult exercise because of the obvious way of getting bit strings of length n from bit strings of length $n - 1$.

8. You may still mimic the binary addition without using it explicitly.

9. Just trace the algorithms for $n = 4$.

10. There are several decrease-and–conquer algorithms for this problem. They are more subtle than one might expect. Generating combinations in a predefined order (increasing, decreasing, lexicographic) helps with both a design and a correctness proof. The following simple property is very helpful. Assuming with no loss of generality that the underlying set is $\{1, 2, ..., n\}$, there are $\binom{n-i}{k-1}$ k-subsets whose smallest element is i, $i = 1, 2, ..., n-k+1$.

11. Represent the disk movements by flipping bits in a binary n-tuple.

12. Thinking about the switches as bits of a bit string could be helpful but not necessary.

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.)

$\overleftarrow{1}\overleftarrow{2}\overleftarrow{3}\overleftarrow{\mathbf{4}}$ $\quad$ $\overleftarrow{1}\overleftarrow{2}\overleftarrow{\mathbf{4}}\overleftarrow{3}$ $\quad$ $\overleftarrow{1}\overleftarrow{\mathbf{4}}\overleftarrow{2}\overleftarrow{3}$ $\quad$ $\overleftarrow{4}\overleftarrow{1}\overleftarrow{2}\overleftarrow{\mathbf{3}}$

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

$\overleftarrow{3}\overleftarrow{1}\overleftarrow{2}\overleftarrow{\mathbf{4}}$ $\quad$ $\overleftarrow{3}\overleftarrow{1}\overleftarrow{\mathbf{4}}\overleftarrow{2}$ $\quad$ $\overleftarrow{3}\overleftarrow{\mathbf{4}}\overleftarrow{1}\overleftarrow{2}$ $\quad$ $\overleftarrow{4}\overleftarrow{3}\overleftarrow{1}\overleftarrow{\mathbf{2}}$

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

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

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

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 $S(n)$:

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

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

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

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

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

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 $i \leftarrow 0$ to $n-1$ do
    $B[i] = 0$
repeat
    print($B$)
    $k \leftarrow n-1$
    while $k \geq 0$ and $B[k] = 1$
        $k \leftarrow k-1$
    if $k \geq 0$
        $B[k] \leftarrow 1$
        for $i \leftarrow k+1$ to $n-1$ do
            $B[i] \leftarrow 0$
until $k = -1$
```

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

000 001 011 010 110 111 101 100.

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 $j \leftarrow i$ to $n - k + 1$ do
        $A[k] \leftarrow j$
        $Choose(j + 1, k - 1)$
```

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:

$$121312141213121.$$

Exercises 4.4

1. A stick n inches long needs to be cut into n 1-inch pieces. Outline an algorithm that performs this task with the minimum number of cuts if several pieces of the stick can be cut at the same time. Also give a formula for the minimum number of cuts.

2. Design a decrease-by-half algorithm for computing $\lfloor \log_2 n \rfloor$ and determine its time efficiency.

3. a. What is the largest number of key comparisons made by binary search in searching for a key in the following array?

3	14	27	31	39	42	55	70	74	81	85	93	98

   b. List all the keys of this array that will require the largest number of key comparisons when searched for by binary search.

   c. Find the average number of key comparisons made by binary search in a successful search in this array. Assume that each key is searched for with the same probability.

   d. Find the average number of key comparisons made by binary search in an unsuccessful search in this array. Assume that searches for keys in each of the 14 intervals formed by the array's elements are equally likely.

4. Estimate how many times faster an average successful search will be in a sorted array of one million elements if it is done by binary search versus sequential search.

5. The time efficiency of sequential search does not depend on whether a list is implemented as an array or as a linked list. Is it also true for searching a sorted list by binary search?

6. a. Design a version of binary search that uses only two-way comparisons such as $\leq$ and $=$. Implement your algorithm in the language of your choice and carefully debug it: such programs are notorious for being prone to bugs.

   b. Analyze the time efficiency of the two-way comparison version designed in part a.

7. A version of the popular problem-solving task involves presenting people with an array of 42 pictures—seven rows of six pictures each—and asking them to identify the target picture by asking questions that can be answered yes or no. Further, people are then required to identify the picture with as few questions as possible. Suggest the most efficient algorithm for this problem and indicate the largest number of questions that may be necessary.

8. Consider ***ternary search***—the following algorithm for searching in a sorted array $A[0..n-1]$. If $n = 1$, simply compare the search key K with the single element of the array; otherwise, search recursively by comparing K with $A[\lfloor n/3 \rfloor]$, and if K is larger, compare it with $A[\lfloor 2n/3 \rfloor]$ to determine in which third of the array to continue the search.

   a. What design technique is this algorithm based on?

   b. Set up a recurrence for the number of key comparisons in the worst case. You may assume that $n = 3^k$.

   c. Solve the recurrence for $n = 3^k$.

   d. Compare this algorithm's efficiency with that of binary search.

9. An array $A[0..n-2]$ contains $n-1$ integers from 1 to n in increasing order. (Thus one integer in this range is missing.) Design the most efficient algorithm you can to find the missing integer and indicate its time efficiency.

10. a. Write a pseudocode for the divide-into-three algorithm for the fake-coin problem. Make sure that your algorithm handles properly all values of n, not only those that are multiples of 3.

    b. Set up a recurrence relation for the number of weighings in the divide-into-three algorithm for the fake-coin problem and solve it for $n = 3^k$.

    c. For large values of n, about how many times faster is this algorithm than the one based on dividing coins into two piles? Your answer should not depend on n.

11. a. Apply the Russian peasant algorithm to compute $26 \cdot 47$.

    b. From the standpoint of time efficiency, does it matter whether we multiply n by m or m by n by the Russian peasant algorithm?

12. a. Write pseudocode for the Russian peasant multiplication algorithm.

    b. What is the time efficiency class of Russian peasant multiplication?

13. Find $J(40)$—the solution to the Josephus problem for $n = 40$.

14. Prove that the solution to the Josephus problem is 1 for every n that is a power of 2.

15. ► For the Josephus problem,

    a. compute $J(n)$ for $n = 1, 2, ..., 15$.

b. discern a pattern in the solutions for the first fifteen values of n and prove its general validity.

c. prove the validity of getting $J(n)$ by a one-bit cyclic shift left of the binary representation of n.

Hints to Exercises 4.4

1. Take care of the length of the longest piece present.

2. If the instance of size n is to compute $\lfloor \log_2 n \rfloor$, what is the instance of size $n/2$? What is the relationship between the two?

3. a. Take advantage of the formula that gives the immediate answer.

   (b)–(d) The most efficient prop for answering such questions is a binary search tree that mirrors the algorithm's operations in searching for an arbitrary search key.

4. Estimate the ratios of the number of key comparisons made by sequential search to the average number made by binary search in successful searches.

5. How would you reach the middle element in a linked list?

6. a. Use the comparison $K \leq A[m]$ where $m \leftarrow \lfloor (l+r)/2 \rfloor$ until $l = r$. Then check whether the search is successful or not.

   b. The analysis is almost identical to that of the text's version of binary search.

7. Number the pictures and use this numbering in your questions.

8. The algorithm is quite similar to binary search, of course. In the worst case, how many key comparisons does it make on each iteration and what fraction of the array remains to be processed?

9. Start by comparing the middle element $A[m]$ with $m + 1$.

10. It is obvious how one needs to proceed if $n \bmod 3 = 0$ or $n \bmod 3 = 1$; it is somewhat less so if $n \bmod 3 = 2$.

11. a. Trace the algorithm for the numbers given as it is done in the text for another input (see Figure 4.11b).

    b. How many iterations does the algorithm perform?

12. You may implement the algorithm either recursively or nonrecursively.

13. The fastest way to the answer the question is to use the formula that exploits the binary representation of n, which is mentioned at the end of the section.

14. Use the binary representation of n.

15. a. Use forward substitutions (see Appendix B) into the recurrence equations given in the text.

    b. On observing the pattern in the first 15 values of n obtained in part (a), express it analytically. Then prove its validity by mathematical induction.

    c. Start with the binary representation of n and translate into binary the formula for $J(n)$ obtained in part (b).

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 ⌊log_2 n⌋
if n = 1 return 0
else return LogFloor(⌊n/2⌋) + 1
```

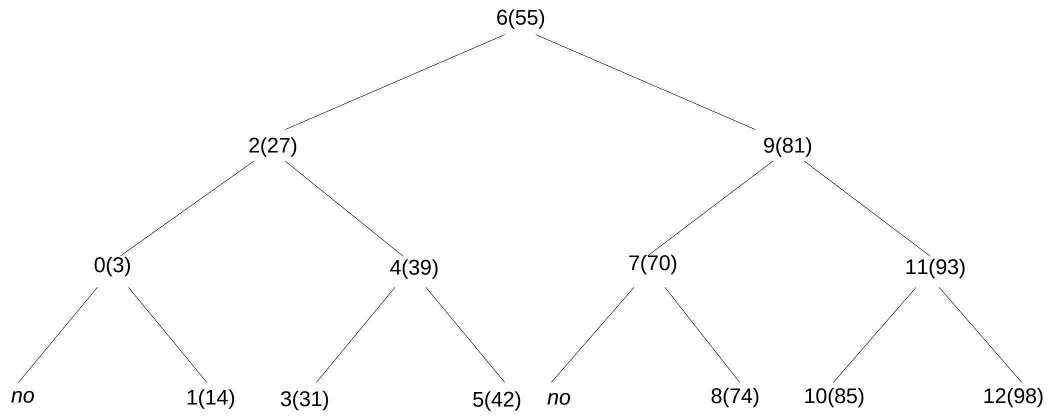
The algorithm is almost identical to the algorithm for computing the number of binary digits, which was investigated in Section 2.4. The recurrence relation for the number of additions is

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

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 $K \leq A[m]$
            $r \leftarrow m$
        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** $n \bmod 2 = 1$   $p \leftarrow p + m$
    $n \leftarrow \lfloor n/2 \rfloor$
    $m \leftarrow 2 * m$
**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 $J(2^k) = J(1\underbrace{0...0}_{\substack{2 \\ k \text{ zeros}}})$ $= 1$.

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

$2^k \leq n < 2^{k+1}$ $(k = 0, 1, 2, 3)$ 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

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

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

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

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 $0 \leq j < 2^k$. 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.

Exercises 4.5

1. a. If we measure an instance size of computing the greatest common divisor of m and n by the size of the second number n, by how much can the size decrease after one iteration of Euclid's algorithm?

   b. Prove that an instance size will always decrease at least by a factor of two after two successive iterations of Euclid's algorithm.

2. Apply quickselect to find the median of the list of numbers 9, 12, 5, 17, 20, 30, 8.

3. Write pseudocode for a nonrecursive implementation of quickselect.

4. Derive the formula underlying interpolation search.

5. ▷ Give an example of the worst-case input for interpolation search and show that the algorithm is linear in the worst case.

6. a. Find the smallest value of n for which $\log_2 \log_2 n + 1$ is greater than 6.

   b. Determine which, if any, of the following assertions are true:

   i. $\log \log n \in o(\log n)$ ii. $\log \log n \in \Theta(\log n)$ iii. $\log \log n \in \Omega(\log n)$.

7. a. Outline an algorithm for finding the largest key in a binary search tree. Would you classify your algorithm as a variable-size-decrease algorithm?

   b. What is the time efficiency class of your algorithm in the worst case?

8. a. Outline an algorithm for deleting a key from a binary search tree. Would you classify this algorithm as a variable-size-decrease algorithm?

   b. What is the time efficiency class of your algorithm?

9. Outline a variable-size-decrease algorithm for constructing an Eulerian circuit in a connected graph with all vertices of even degrees.

10. *Misere one-pile Nim* Consider the so-called ***misere version*** of the one-pile Nim, in which the player taking the last chip looses the game. All the other conditions of the game remain the same, i.e., the pile contains n chips and on each move a player takes at least one but no more than m chips. Identify the winning and loosing positions (for the player to move) in this game.

11. ▷a. *Moldy chocolate* Two payers take turns by breaking an $m \times n$ chocolate bar, which has one spoiled 1-by-1 square. Each break must be a single straight line cutting all the way across the bar along the boundaries between the squares. After each break, the player who broke the bar last

eats the piece that does not contain the spoiled corner. The player left with the spoiled square loses the game. Is it better to go first or second in this game?

b. Write an interactive program to play this game with the computer. Your program should make a winning move in a winning position and a random legitimate move in a loosing position.

12. $\triangleright$ *Flipping pancakes* There are n pancakes all of different sizes that are stacked on top of each other. You are allowed to slip a flipper under one of the pancakes and flip over the whole stack above the flipper. The purpose is to arrange pancakes according to their size with the biggest at the bottom. (You can see a visualization of this puzzle on the *Interactive Mathematics Miscellany and Puzzles* site [Bog].) Design an algorithm for solving this puzzle.

13. $\triangleright$ You need to search for a given number in an $n \times n$ matrix in which every row and every column is sorted in increasing order. Can you design a $O(n)$ algorithm for this problem? [Laa10]

Hints to Exercises 4.5

1. a. The answer follows immediately from the formula underlying Euclid's algorithm.

   b. Let $r = m \bmod n$. Investigate two cases of r's value relative to n's value.

2. Trace the algorithm on the input given, as is done in the section for another input.

3. The nonrecursive version of the algorithm was applied to a particular instance in the section's example.

4. Write an equation of the straight line through the points $(l, A[l])$ and $(r, A[r])$ and find the x coordinate of the point on this line whose y coordinate is v.

5. Construct an array for which interpolation search decreases the remaining subarray by one element on each iteration.

6. a. Solve the inequality $\log_2 \log_2 n + 1 > 6$.

   b. Compute $\lim_{n\to\infty} \frac{\log \log n}{\log n}$. Note that to within a constant multiple, you can consider the logarithms to be natural, i.e., base e.

7. a. The definition of the binary search tree suggests such an algorithm.

   b. What is the worst-case input for your algorithm? How many key comparisons does it make on such an input?

8. a. Consider separately three cases: (i) the key's node is a leaf, (ii) the key's node has one child, (iii) the key's node has two children.

   b. Assume that you know a location of the key to be deleted.

9. Starting at an arbitrary vertex of the graph, traverse a sequence of its untraversed edges until either all the edges are traversed or no untraversed edge is available.

10. Follow the plan used in the section for analyzing the normal version of the game.

11. Play several rounds of the game on the graph paper to become comfortable with the problem. Considering special cases of the spoiled square's location should help you to solve it.

12. Do yourself a favor: try to design an algorithm on your own. It does not have to be optimal, but it should be reasonably efficient.

13. Start by comparing the search number with the last element in the first row.

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 $n \bmod r \leq n/2$. Consider two cases: $r \leq n/2$ and $n/2 < r < n$. If $r \leq n/2$, then

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

   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 $k$ $(1 \leq k \leq n)$
//Output: The value of the $k$th smallest element in $A[0..n-1]$
$l \leftarrow 0$;   $r \leftarrow n-1$
$A[n] \leftarrow \infty$   //append sentinel
while $l \leq r$ do
        $p \leftarrow A[l]$     //the pivot
        $i \leftarrow l$;   $j \leftarrow r+1$
        repeat
                repeat $i \leftarrow i+1$ until $A[i] \geq p$
                repeat $j \leftarrow j-1$ until $A[j] \leq p$ do
                swap($A[i], A[j]$)
        until $i \geq j$
        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 $k$ $(1 \leq k \leq r-l+1)$
//Output: The value of the $k$th 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

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

and

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

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 ($1 \leq k < n$) 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

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

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

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 $2 \leq n \leq m+1$, 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 $m+3 \leq n \leq 2m+2$ (i.e., $2+(m+1) \leq n \leq 2(m+1)$), 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 $O(n)$.