This file contains the exercises, hints, and solutions for Chapter 5 of the book "Introduction to the Design and Analysis of Algorithms," 2nd 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 5.1

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* 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. [Gar78], p.7.

3. 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.)

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

5. 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?

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

7. 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?

8. 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

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

9. ▷ Binary insertion sort uses binary search to find an appropriate position to insert $A[i]$ among the previously sorted $A[0] \leq ... \leq A[i-1]$. Determine the worst-case efficiency class of this algorithm.

10. 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?

    c. Implement shellsort, straight insertion sort, binary insertion sort, mergesort, and quicksort in the language of your choice and compare their performance on random arrays of sizes 10^2, 10^3, 10^4,and 10^5 as well as on increasing and decreasing arrays of these sizes.

Hints to Exercises 5.1

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. 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.

4. Trace the algorithm as we did in the text for another input (see Fig. 5.4).

5. 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.

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

7. 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.

8. 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.

9. The order of growth of the worst-case number of key comparisons made by binary insertion sort can be obtained from formulas in Section 4.3 and Appendix A. For this algorithm, however, a key comparison is not the operation that determines the algorithm's efficiency class. Which operation does?

10. 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 5.1

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 repeated n times, the problem will be solved after the total of $4n$ trips.

2. 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.

3. 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 5.4):

   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

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

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

5. 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

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

6. 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.

7. 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.

8. 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:

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

   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

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

9. The largest number of key comparisons will be, in particular, for strictly increasing or decreasing arrays:

$$C_{\max}(n) = \sum_{i=1}^{n-1}(\lfloor \log_2 i \rfloor + 1) = \sum_{i=1}^{n-1}\lfloor \log_2 i \rfloor + \sum_{i=1}^{n-1} 1 \in \Theta(n\log n) + \Theta(n) = \Theta(n\log n).$$

It is the number of key *moves*, however, that will dominate the number of key comparisons in the worst case of strictly decreasing arrays. The number of key moves will be exactly the same as for the classic insertion sort, putting the algorithm's worst-case efficiency in $\Theta(n^2)$.

10. 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:

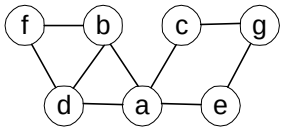
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S_1	H	E_1	L_1	L_2	S_2	O	R	T	I	S_3	U_1	S_4	E_2	F	U_2	L_3
S_1													E_2			
E_2													S_1			
	H													F		
	F													H		
		E_1													U_2	
			L_1													L_3
E_2				L_2												
	F				S_2											
		E_1				O										
			L_1		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

The final pass—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 5.2

1. Consider the graph



   a. Write down the adjacency matrix and adjacency lists specifying this graph. (Assume that the matrix rows and columns and vertices in the adjacency lists follow in the alphabetical order of the vertex labels.)

   b. Starting at vertex a and resolving ties by the vertex alphabetical order, traverse the graph by depth-first search and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices became dead ends (popped off the stack).

2. If we define sparse graphs as graphs for which $|E| \in O(|V|)$, which implementation of DFS will have a better time efficiency for such graphs, the one that uses the adjacency matrix or the one that uses the adjacency lists?

3. Let G be a graph with n vertices and m edges.

   a. True or false: All its DFS forests (for traversals starting at different vertices) will have the same number of trees?

   b. True or false: All its DFS forests will have the same number of tree edges and the same number of back edges?

4. Traverse the graph of Problem 1 by breadth-first search and construct the corresponding breadth-first search tree. Start the traversal at vertex a and resolve ties by the vertex alphabetical order.

5. Prove that a cross edge in a BFS tree of an undirected graph can connect vertices only on either the same level or on two adjacent levels of a BFS tree.

6. a. Explain how one can check a graph's acyclicity by using breadth-first search.

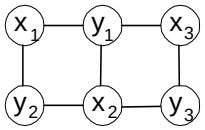
   b. Does either of the two traversals—DFS or BFS—always find a cycle faster than the other? If you answer yes, indicate which of them is better and explain why it is the case; if you answer no, give two examples supporting your answer.

7. Explain how one can identify connected components of a graph by using

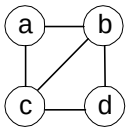
   a. a depth-first search.

   b. a breadth-first search.

8. A graph is said to be ***bipartite*** if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. (We can also say that a graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors; such graphs are also called ***2-colorable***). For example, graph (i) is bipartite while graph (ii) is not.



(i)



(ii)

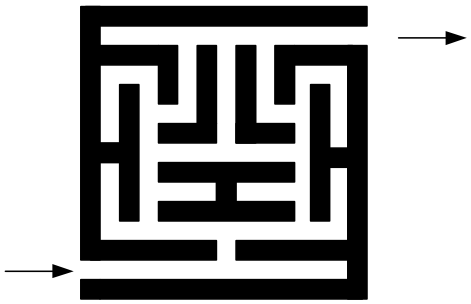
   a. Design a DFS-based algorithm for checking whether a graph is bipartite.

   b. Design a BFS-based algorithm for checking whether a graph is bipartite.

9. Write a program that, for a given graph, outputs
   a. vertices of each connected component;
   b. its cycle or a message that the graph is acyclic.

10. One can model a maze by having a vertex for a starting point, a finishing point, dead ends, and all the points in the maze where more than one path can be taken, and then connecting the vertices according to the paths in the maze.

    a. Construct such a graph for the following maze.



    b. Which traversal—DFS or BFS—would you use if you found yourself in a maze and why?

Hints to Exercises 5.2

1. a. Use the definitions of the adjacency matrix and adjacency lists given in Section 1.4.

   b. Perform the DFS traversal the same way it is done for another graph in the text (see Fig. 5.5).

2. Compare the efficiency classes of the two versions of DFS for sparse graphs.

3. a. What is the number of such trees equal to?

   b. Answer this question for connected graphs first.

4. Perform the BFS traversal the same way it is done in the text (see Fig. 5.6).

5. You may use the fact that the level of a vertex in a BFS tree indicates the number of edges in the shortest (minimum-edge) path from the root to that vertex.

6. a. What property of a BFS forest indicates a cycle's presence? (The answer is similar to the one for a DFS forest.)

   b. The answer is no. Find two examples supporting this answer.

7. Given the fact that both traversals can reach a new vertex if and only if it is adjacent to one of the previously visited vertices, which vertices will be visited by the time either traversal halts (i.e., its stack or queue becomes empty)?

8. Use a DFS forest and a BFS forest for parts (a) and (b), respectively.

9. Use either DFS or BFS.

10. a. Follow the instructions of the problem's statement.

    b. Trying both traversals should lead you to a correct answer very fast.

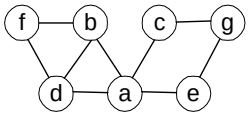
Solutions to Exercises 5.2

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

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

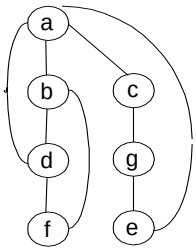
b. See below: (i) the graph; (ii) the traversal's stack (the first subscript number indicates the order in which the vertex was visited, i.e., pushed onto the stack, the second one indicates the order in which it became a dead-end, i.e., popped off the stack); (iii) the DFS tree (with the tree edges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

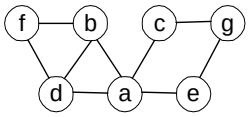
$$|E^{(tree)}| = \sum_{c=1}^{|C|} |E_c^{(tree)}| = \sum_{c=1}^{|C|} (|V_c| - 1) = \sum_{c=1}^{|C|} |V_c| - \sum_{c=1}^{|C|} 1 = |V| - |C|$$

and

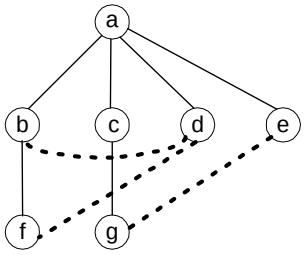
$$|E^{(back)}| = |E| - |E^{(tree)}| = |E| - (|V| - |C|) = |E| - |V| + |C|,$$

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:



a b c d e f g



(i) (ii) (iii)

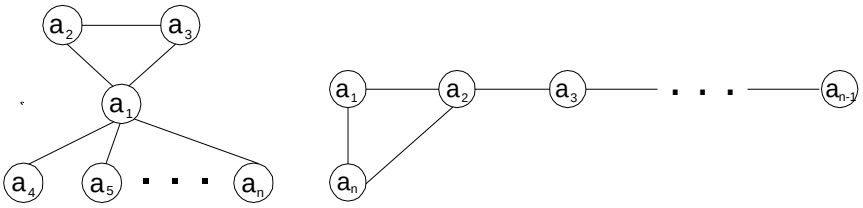
(i) the graph; (ii) the traversal's queue; (iii) the tree (the tree and cross edges are shown with solid and dotted lines, respectively).

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)

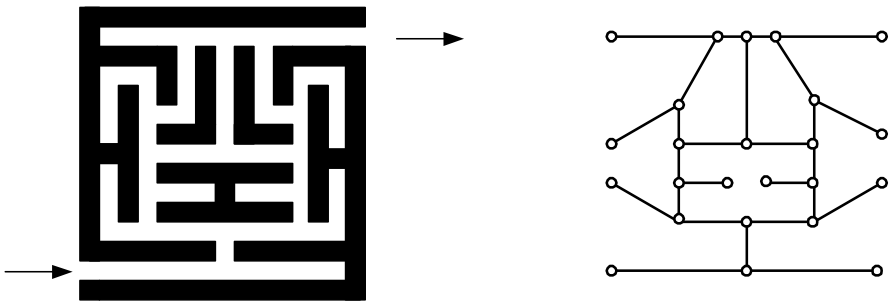
7. Start a DFS (or BFS) traversal at an arbitrary vertex and mark the visited vertices with 1. By the time the traversal's stack (queue) becomes empty, all the vertices in the same connected component as the starting vertex, and only they, will have been marked with 1. If there are unvisited vertices 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

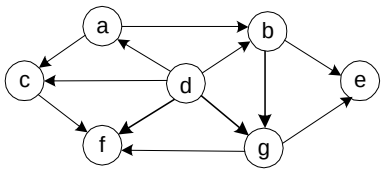
10. a. Here is the maze and a graph representing it:



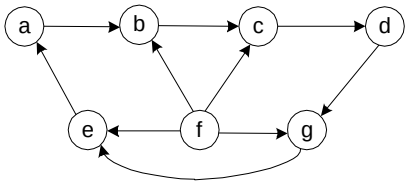
b. DFS is much more convenient for going through a maze than BFS. When DFS moves to a next vertex, it is connected to a current vertex by an edge (i.e., "close nearby" in the physical maze), which is not generally the case for BFS. In fact, DFS can be considered a generalization of an ancient right-hand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

Exercises 5.3

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. ▷ 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:

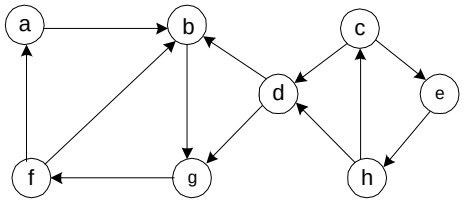
Step 1. Do a DFS traversal of the digraph given and number its vertices in the order that they become dead ends.

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

Hints to Exercises 5.3

1. Trace the algorithm as it is done in the text for another digraph (see Fig. 5.10).

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. 5.11).

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.

8. n/a

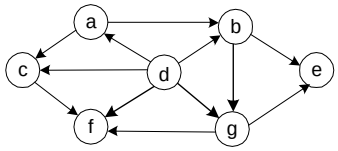
9. a. Trace the algorithm on the input given by following the steps of the algorithm as indicated.

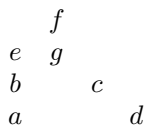
   b. Determine the efficiency for each of the three principal steps of the algorithm and then determine the overall efficiency. Of course, the answers will depend on whether a graph is represented by its adjacency matrix or by its adjacency lists.

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

Solutions to Exercises 5.3

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





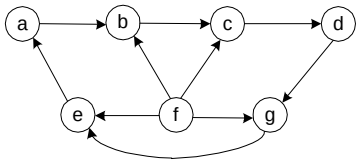
The vertices are popped off the stack in the following order:

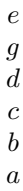
$$e\ f\ g\ b\ c\ a\ d.$$

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.

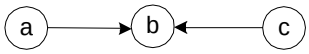
If a digraph has no directed cycles, a solution to the topological sorting problem is fetched by either of the two algorithms discussed in the section. (The correctness of the DFS-based algorithm was explained there; the correctness of the source removal algorithm stems from the assertion of Problem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

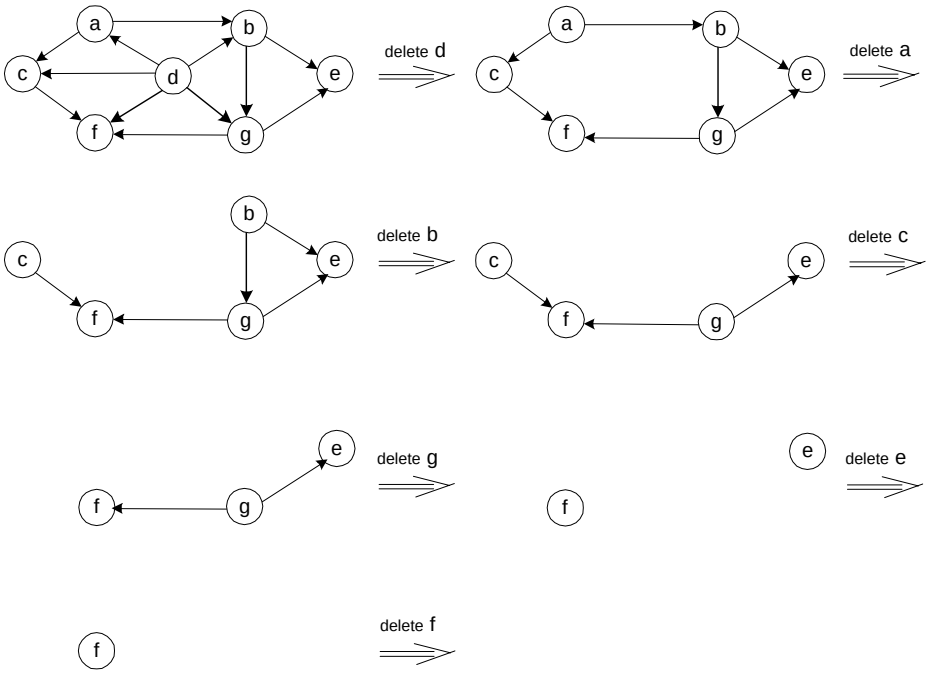
   b. Fill the array of length $|V|$ with vertices being popped off the DFS traversal stack right to left.

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



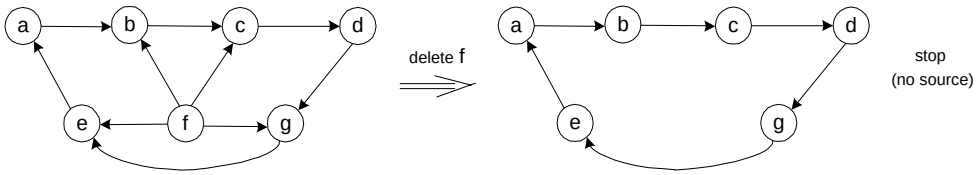
   The DFS traversal that starts at a pushes the vertices on the stack in the order a, b, c, and neither this ordering nor its reversal solves the topological sorting problem correctly.

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.

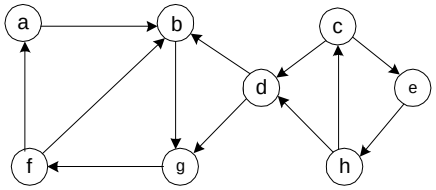
   b. A vertex of a dag is a source if and only if its column in the adjacency matrix contains only 0's. Looking for such a column is a $O(|V|^2)$ operation.

   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

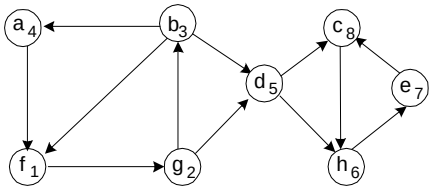


   The stack of the first DFS traversal, with a as its starting vertex, will look as follows:

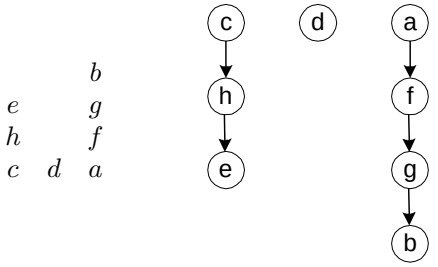
   f_1
   g_2 $\quad$ h_6
   b_3 $\quad$ d_5 $\quad$ e_7
   a_4 $\quad$ c_8

   (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 problem can be solved by a recursive algorithm based on the decrease-by-one strategy. 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 computer implementation of this algorithm can be found, e.g., in Manber's *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, 1989.

Exercises 5.4

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 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 a 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. Use the decrease-by-one technique to generate the binary reflected Gray code for $n = 4$.

   b.▷ Design a general decrease-by-one algorithm for generating the binary reflected Gray code of order n.

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.

Hints to Exercises 5.4

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: $e \approx \sum_{i=0}^{n} \frac{1}{i!}$.

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. A Gray code for $n = 3$ is given at the end of the section. It is not difficult to see how to use it to generate a Gray code for $n = 4$. Gray codes have a useful geometric interpretation based on mapping its bit strings to vertices of the n-dimensional cube. Find such a mapping for $n = 1$, 2, and 3. This geometric interpretation might help you with designing a general algorithm for generating a Gray code of order n.

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.

Solutions to Exercises 5.4

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.)

$$
\begin{array}{cccc}
\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}}\,\overleftarrow{3}\,\overleftarrow{2}\,\overleftarrow{1} & \overleftarrow{3}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{2}\,\overleftarrow{1} & \overleftarrow{3}\,\overleftarrow{2}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{1} & \overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1}\,\overleftarrow{4} \\
\overleftarrow{2}\,\overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{4}} & \overleftarrow{2}\,\overleftarrow{3}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{1} & \overleftarrow{2}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{3}\,\overleftarrow{1} & \overleftarrow{4}\,\overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1} \\
\overrightarrow{\mathbf{4}}\,\overleftarrow{2}\,\overleftarrow{1}\,\overleftarrow{3} & \overleftarrow{2}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{1}\,\overleftarrow{3} & \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{\mathbf{4}}\,\overleftarrow{3} & \overleftarrow{2}\,\overleftarrow{1}\,\overleftarrow{3}\,\overleftarrow{4}
\end{array}
$$

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. n/a

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	$\emptyset$							
1	$\emptyset$	$\{a_1\}$						
2	$\emptyset$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\emptyset$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$
4	$\emptyset$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$
	$\{a_4\}$	$\{a_1, a_4\}$	$\{a_2, a_4\}$	$\{a_1, a_2, a_4\}$	$\{a_3, a_4\}$	$\{a_1, a_3, a_4\}$	$\{a_2, a_3, a_4\}$	$\{a_1, a_2, a_3, a_4\}$

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	$\emptyset$	$\{a_4\}$	$\{a_3\}$	$\{a_3, a_4\}$	$\{a_2\}$	$\{a_2, a_4\}$	$\{a_2, a_3\}$	$\{a_2, a_3, a_4\}$
bit strings	1000	1001	1010	1011	1100	1101	1110	1111
subsets	$\{a_1\}$	$\{a_1, a_4\}$	$\{a_1, a_3\}$	$\{a_1, a_3, a_4\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_4\}$	$\{a_1, a_2, a_3\}$	$\{a_1, a_2, a_3, a_4\}$

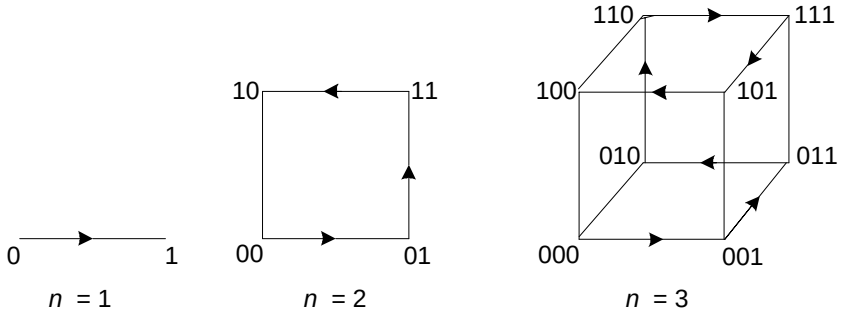
6. Establish the correspondence between subsets of $A = \{a_1, ..., a_n\}$ and bit strings $b_1...b_n$ of length n by associating bit i with the presence or absence of element a_{n-i+1} for $i = 1, ..., n$.

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. As mentioned in the hint to this problem, binary Gray codes have a useful geometric interpretation based on mapping their bit strings to vertices of the n-dimensional cube. Such a mapping is shown below for $n = 1, 2$, and 3.



The list of bit strings in the binary reflexive Gray code for $n = 3$ given in the section is obtained by traversing the vertices of the three-dimensional cube by starting at 000 and following the arrows shown:

000 001 011 010 110 111 101 100.

We can obtain the binary reflexive Gray code for $n = 4$ as follows. Make two copies of the list of bit strings for $n = 3$; add 0 in front of each bit string in the first copy and 1 in front of each bit sting in the second copy and then append the second list to the first in reversed order to obtain:

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

(Note that the last bit string differs from the first one by a single bit, too.)

b. The mechanism employed in part (a) can be used for constructing

the binary reflexive Gray code for an arbitrary $n \geq 1$: If $n = 1$, return the list 0, 1. If $n > 1$, generate recursively the list of bit strings of size $n - 1$ and make a copy of this list; add 0 in front of each bit string in the first list and add 1 in front of each bit string in the second list; then append the second list in reversed order to the first list.

Note: The correctness of the algorithm stems from the fact that it generates 2^n bit strings and all of them are distinct. (Both these assertions are very easy to check by mathematical induction.)

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.

Exercises 5.5

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

2. 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 relation 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.

3. 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.)

4. Apply multiplication à la russe to compute $26 \cdot 47$.

5. a. From the standpoint of time efficiency, does it matter whether we multiply n by m or m by n by the multiplication à la russe algorithm?

   b. What is the efficiency class of multiplication à la russe?

6. Write a pseudocode for the multiplication à-la-russe algorithm.

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

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

9. ▶ 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 5.5

1. 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?

2. 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?

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

4. Trace the algorithm for the numbers given as it is done in the text for another input (see Figure 5.14b).

5. How many iterations does the algorithm do?

6. You can implement the algorithm either recursively or nonrecursively.

7. The fastest way to the answer is to use the formula that exploits the binary representation of n, which is mentioned at the end of Section 5.5.

8. Use the binary representation of n.

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

   b. On observing the pattern in the first fifteen 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 5.5

1. **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 $A(n) = \lfloor \log_2 n \rfloor \in \Theta(\log n)$.

2. 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.

3. 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

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

4. 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

5. a. The number of divisions multiplication à la russe needs for computing $n \cdot m$ and $m \cdot n$ is $\lfloor \log_2 n \rfloor$ and $\lfloor \log_2 m \rfloor$, respectively.

b. Its time efficiency is in $\Theta(\log n)$ where n is the first factor of the product. As a function of b, the number of binary digits of n, the time efficiency is in $\Theta(b)$.

6. **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*$((n-1)/2, 2m) + m$

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

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

8. 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 $J(2^k) = J(1\underbrace{0...0}_{k \text{ zeros}}) = 1$.

9. 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

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

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 $0 \leq j < 2^k$. 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 9 are from [Gra94].

Exercises 5.6

1. a. If we measure the size of an instance of the problem of computing the greatest common divisor of m and n by the size of the second parameter 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 2 after two successive iterations of Euclid's algorithm.

2. a. Apply the partition-based algorithm to find the median of the list of numbers 9, 12, 5, 17, 20.

   b. Show that the worst-case efficiency of the partition-based algorithm for the selection problem is quadratic.

3. a. Write a pseudocode for a nonrecursive implementation of the partition-based algorithm for the selection problem.

   b. Write a pseudocode for a recursive implementation of this algorithm.

4. Derive the formula underlying interpolation search.

5. $\triangleright$ 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. *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.

10. ▷a. *Moldy chocolate* Two payers take turns by breaking an m-by-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.

11. ▷*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 sack 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.

Hints to Exercises 5.6

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. a. Trace the algorithm on the input given, as was done in the section for another input.

   b. Since the algorithm in question is based on the same partitioning idea as quicksort is, it is natural to expect the worst-case inputs to be similar for these algorithms.

3. You should have difficulties with neither implementation of the algorithm outlined in the text.

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 will be the worst-case input for your algorithm? How many key comparisons will it make on such an input?

8. a. Consider separately three cases: (1) the key's node is a leaf; (2) the key's node has one child; (3) the key's node has two children.

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

9. Follow the plan used in Section 5.6 for analyzing the standard version of the game.

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

11. 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.

Solutions to Exercises 5.6

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. a. Since $n = 5$, $k = \lceil 5/2 \rceil = 3$. For the given list 9, 12, 5, 17, 20, with the first element as the pivot, we obtain the following partition

9	12	5	17	20
5	**9**	12	17	20

   Since $s = 2 < k = 3$, we proceed with the right part of the list:

12	17	20
12	17	20

   Since $s = k = 3$, 12 is the median of the list given.

   b. Consider an instance of the selection problem with $k = n$ and a strictly increasing array. The situation is identical to the worst-case analysis of quicksort (see Section 4.2).

3. a. **Algorithm** *Selection*$(A[0..n-1], k)$
   //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

$i \leftarrow l; \quad j \leftarrow r + 1$
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 $j > k - 1 \quad r \leftarrow j - 1$
else if $j < k - 1 \quad l \leftarrow j + 1$
else return $A[k - 1]$

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

Algorithm *SelectionRec*($A[l..r]$, 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 kth smallest element in $A[l..r]$
$s \leftarrow$ *Partition*($A[l..r]$) //see Section 4.2; must return l if $l = r$
if $s > l + k - 1$ *SelectionRec*($A[l..s - 1]$, k)
else if $s < l + k - 1$ *SelectionRec*($A[s + 1..r]$, $k - 1 - s$)
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 (5.4) 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

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

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 5.6).

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. 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.

10. 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 $m \ne n$. 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 $k \le n$, 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.

11. 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.)