This file contains the exercises, hints, and solutions for Chapter 9 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 9.1

1. Give an instance of the change-making problem for which the greedy algorithm does not yield an optimal solution.

2. Write a pseudocode of the greedy algorithm for the change-making problem, with an amount n and coin denominations $d_1 > d_2 > ... > d_m$ as its input. What is the time efficiency class of your algorithm?

3. Consider the problem of scheduling n jobs of known durations $t_1, ..., t_n$ for execution by a single processor. The jobs can be executed in any order, one job at a time. You want to find a schedule that minimizes the total time spent by all the jobs in the system. (The time spent by one job in the system is the sum of the time spent by this job in waiting plus the time spent on its execution.)

   Design a greedy algorithm for this problem. ▷ Does the greedy algorithm always yield an optimal solution?

4. Design a greedy algorithm for the assignment problem (see Section 3.4). Does your greedy algorithm always yield an optimal solution?

5. *Bridge crossing revisited* Consider the generalization of the bridge crossing puzzle (Problem 2 in Exercises 1.2) in which we have $n > 1$ people whose bridge crossing times are t_1, $t_2, ..., t_n$. All the other conditions of the problem remain the same: at most two people at the time can cross the bridge (and they move with the speed of the slower of the two) and they must carry with them the only flashlight the group has.

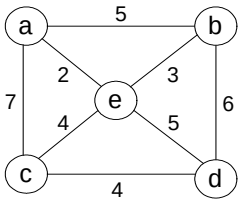
   Design a greedy algorithm for this problem and find how long it will take to cross the bridge by using this algorithm. Does your algorithm yields a minimum crossing time for every instance of the problem? If it does—prove it, if it does not—find an instance with the smallest number of people for which this happens.

6. *Bachet-Fibonacci weighing problem* Find an optimal set of n weights $\{w_1, w_2, ..., w_n\}$ so that it would be possible to weigh on a balance scale any integer load in the largest possible range from 1 to W, provided

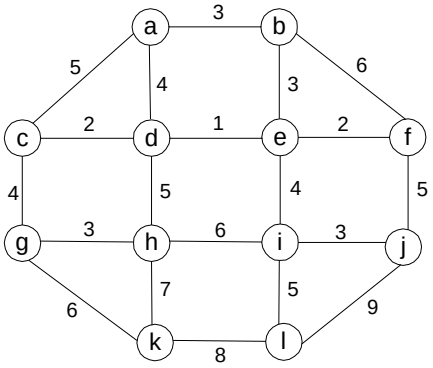
   a.▷ weights can be put only on the free cup of the scale.

   b.► weights can be put on both cups of the scale.

7. a. Apply Prim's algorithm to the following graph. Include in the priority queue all the vertices not already in the tree.



b. Apply Prim's algorithm to the following graph. Include in the priority queue only the fringe vertices (the vertices not in the current tree which are adjacent to at least one tree vertex).



8. The notion of a minimum spanning tree is applicable to a connected weighted graph. Do we have to check a graph's connectivity before applying Prim's algorithm or can the algorithm do it by itself?

9. a. How can we use Prim's algorithm to find a spanning tree of a connected graph with no weights on its edges?

   b. Is it a good algorithm for this problem?

10. ▷ Prove that any weighted connected graph with distinct weights has exactly one minimum spanning tree.

11. Outline an efficient algorithm for changing an element's value in a min-heap. What is the time efficiency of your algorithm?

Hints to Exercises 9.1

1. As coin denominations for your counterexample, you may use, among a multitude of other possibilities, the ones mentioned in the text: $d_1 = 7$, $d_2 = 5$, $d_3 = 1$.

2. You may use integer divisions in your algorithm.

3. Considering the case of two jobs might help. Of course, after forming a hypothesis, you will have to either prove the algorithm's optimality for an arbitrary input or find a specific counterexample showing that it is not the case.

4. You can apply the greedy approach either to the entire cost matrix or to each of its rows (or columns).

5. Simply apply the greedy approach to the situation at hand. You may assume that $t_1 \leq t_2 \leq ... \leq t_n$.

6. For both versions of the problem, it is not difficult to get to a hypothesis about the solution's form after considering the cases of $n = 1, 2$, and 3. It is proving the solutions' optimality that is at the heart of this problem.

7. a. Trace the algorithm for the graph given. An example can be found in the text of the section.

   b. After the next fringe vertex is added to the tree, add all the unseen vertices adjacent to it to the priority queue of fringe vertices.

8. Applying Prim's algorithm to a weighted graph that is not connected should help in answering this question.

9. a. Since Prim's algorithm needs weights on a graph's edges, some weights have to be assigned.

   b. Do you know other algorithms that can solve this problem?

10. Strictly speaking, the wording of the question asks you to prove two things: the fact that at least one minimum spanning tree exists for any weighted connected graph and the fact that a minimum spanning tree is unique if all the weights are distinct numbers. The proof of the former stems from the obvious observation about finiteness of the number of spanning trees for a weighted connected graph. The proof of the latter can be obtained by repeating the correctness proof of Prim's algorithm with a minor adjustment at the end.

11. Consider two cases: the key's value was decreased (this is the case needed for Prim's algorithm) and the key's value was increased.

Solutions to Exercises 9.1

1. Here is one of many such instances: For the coin denominations $d_1 = 7$, $d_2 = 5$, $d_3 = 1$ and the amount $n = 10$, the greedy algorithm yields one coin of denomination 7 and three coins of denomination 1. The actual optimal solution is two coins of denomination 5.

2. **Algorithm** *Change*$(n, D[1..m])$
   //Implements the greedy algorithm for the change-making problem
   //Input: A nonnegative integer amount n and
   // a decreasing array of coin denominations D
   //Output: Array $C[1..m]$ of the number of coins of each denomination
   // in the change or the "no solution" message
   for $i \leftarrow 1$ **to** m **do**
   $C[i] \leftarrow \lfloor n/D[i] \rfloor$
   $n \leftarrow n \bmod D[i]$
   if $n = 0$ **return** C
   else return "no solution"

   The algorithm's time efficiency is in $\Theta(m)$. (We assume that integer divisions take a constant time no matter how big dividends are.) Note also that if we stop the algorithm as soon as the remaining amount becomes 0, the time efficiency will be in $O(m)$.

3. a. Sort the jobs in nondecreasing order of their execution times and execute them in that order.

   b. Yes, this greedy algorithm always yields an optimal solution.
   Indeed, for any ordering (i.e., permutation) of the jobs i_1, i_2, ..., i_n, the total time in the system is given by the formula

   $$t_{i_1} + (t_{i_1} + t_{i_2}) + ... + (t_{i_1} + t_{i_2} + ... + t_{i_n}) = nt_{i_1} + (n-1)t_{i_2} + ... + t_{i_n}.$$

   Thus, we have a sum of numbers n, $n-1$,...,1 multiplied by "weights" t_1, t_2, ...t_n assigned to the numbers in some order. To minimize such a sum, we have to assign smaller t's to larger numbers. In other words, the jobs should be executed in nondecreasing order of their execution times.

   Here is a more formal proof of this fact. We will show that if jobs are executed in some order i_1, i_2, ..., i_n, in which $t_{i_k} > t_{i_{k+1}}$ for some k, then the total time in the system for such an ordering can be decreased. (Hence, no such ordering can be an optimal solution.) Let us consider the other job ordering, which is obtained by swapping the jobs k and $k+1$. Obviously, the time in the systems will remain the same for all but these two

jobs. Therefore, the difference between the total time in the system for the new ordering and the one before the swap will be

$$[(\sum_{j=1}^{k-1} t_{i_j} + t_{i_{k+1}}) + (\sum_{j=1}^{k-1} t_{i_j} + t_{i_{k+1}} + t_{i_k})] - [(\sum_{j=1}^{k-1} t_{i_j} + t_{i_k}) + (\sum_{j=1}^{k-1} t_{i_j} + t_{i_k} + t_{i_{k+1}})]$$
$$= \quad t_{i_{k+1}} - t_{i_k} < 0.$$

4. a. The all-matrix version: Repeat the following operation n times. Select the smallest element in the unmarked rows and columns of the cost matrix and then mark its row and column.

   The row-by-row version: Starting with the first row and ending with the last row of the cost matrix, select the smallest element in that row which is not in a previously marked column. After such an element is selected, mark its column to prevent selecting another element from the same column.

   b. Neither of the versions always yields an optimal solution. Here is a simple counterexample:

$$C = \begin{bmatrix} 1 & 2 \\ 2 & 100 \end{bmatrix}$$

5. Repeat the following step $n-2$ times: Send to the other side the pair of two fastest remaining persons and then return the flashlight with the fastest person. Finally, send the remaining two people together. Assuming that $t_1 \le t_2 \le ... \le t_n$, the total crossing time will be equal to

$$(t_2+t_1)+(t_3+t_1)+...+(t_{n-1}+t_1)+t_n = \sum_{i=2}^{n} t_i+(n-2)t_1 = \sum_{i=1}^{n} t_i+(n-3)t_1.$$

   Note: For an algorithm that always yields a minimal crossing time, see Günter Rote, "Crossing the Bridge at Night," *EATCS Bulletin,* vol. 78 (October 2002), 241–246.

   The solution to the instance of Problem 2 in Exercises 1.2 shows that the greedy algorithm doesn't always yield the minimal crossing time for $n > 3$. No smaller counterexample can be given as a simple exhaustive check for $n = 3$ demonstrates. (The obvious solution for $n = 2$ is the one generated by the greedy algorithm as well.)

6. a. Let's apply the greedy approach to the first few instances of the problem in question. For $n = 1$, we have to use $w_1 = 1$ to balance weight 1. For $n = 2$, we simply add $w_2 = 2$ to balance the first previously unattainable weight of 2. The weights $\{1, 2\}$ can balance every integral weights up to their sum 3. For $n = 3$, in the spirit of greedy thinking, we take the next previously unattainable weight: $w_3 = 4$. The three weights $\{1, 2, 4\}$ allow to weigh any integral load l between 1 and their sum 7, with l's binary expansion indicating the weights needed for load l:

load l	1	2	3	4	5	6	7
l's binary expansion	1	10	11	100	101	110	111
weights for load l	1	2	2+1	4	4+1	4+2	4+2+1

Generalizing these observations, we should hypothesize that for any positive integer n the set of consecutive powers of 2 $\{w_i = 2^{i-1}, i = 1, 2, ...n\}$ makes it possible to balance every integral load in the largest possible range, which is up to and including $\sum_{i=1}^{n} 2^{i-1} = 2^n - 1$. The fact that every integral weight l in the range$1 \le l \le 2^n - 1$ can be balanced with this set of weights follows immediately from the binary expansion of l, which yields the weights needed for weighing l. (Note that we can obtain the weights needed for a given load l by applying to it the greedy algorithm for the change-making problem with denominations $d_i = 2^{i-1}$, $i = 1, 2, ...n$.)

In order to prove that no set of n weights can cover a larger range of consecutive integral loads, it will suffice to note that there are just $2^n - 1$ nonempty selections of n weights and, hence, no more than $2^n - 1$ sums they yield. Therefore, the largest range of consecutive integral loads they can cover cannot exceed $2^n - 1$.

[Alternatively, to prove that no set of n weights can cover a larger range of consecutive integral loads, we can prove by induction on i that if any multiset of n weights $\{w_i, i = 1, ..., n\}$—which we can assume without loss of generality to be sorted in nondecreasing order—can balance every integral load starting with 1, then $w_i \le 2^{i-1}$ for $i = 1, 2, ..., n$. The basis checks out immediately: w_1 must be 1, which is equal to 2^{1-1}. For the general case, assume that $w_k \le 2^{k-1}$ for every $1 \le k < i$. The largest weight the first $i - 1$ weights can balance is $\sum_{k=1}^{i-1} w_k \le \sum_{k=1}^{i-1} 2^{k-1} = 2^{i-1} - 1$. If w_i were larger than 2^i, then this load could have been balanced neither with the first $i - 1$ weights (which are too light even taken together) nor with the weights $w_i \le ... \le w_n$ (which are heavier than 2^i even individually). Hence, $w_i \le 2^{i-1}$, which completes the proof by induction. This immediately implies that no n weights can balance every integral load up to the upper limit larger than $\sum_{i=1}^{n} w_i \le \sum_{i=1}^{n} 2^{i-1} = 2^n - 1$, the limit attainable with the consecutive powers of 2 weights.]

b. If weights can be put on both cups of the scale, then a larger range can

be reached with n weights for $n > 1$. (For $n = 1$, the single weight still needs to be 1, of course.) The weights $\{1, 3\}$ enable weighing of every integral load up to 4; the weights $\{1, 3, 9\}$ enable weighing of every integral load up to 13, and, in general, the weights $\{w_i = 3^{i-1},\ i = 1, 2, ..., n\}$ enable weighing of every integral load up to and including their sum of $\sum_{i=1}^{n} 3^{i-1} = (3^n - 1)/2$. A load's expansion in the ternary system indicates the weights needed. If the ternary expansion contains only 0's and 1's, the load requires putting the weights corresponding to the 1's on the opposite cup of the balance. If the ternary expansion of load l, $l \leq (3^n - 1)/2$, contains one or more 2's, we can replace each 2 by (3-1) to represent it in the form

$$l = \sum_{i=1}^{n} \beta_i 3^{i-1}, \text{ where } \beta_i \in \{0, 1, -1\}, \ \ n = \lceil \log_3(l + 1) \rceil.$$

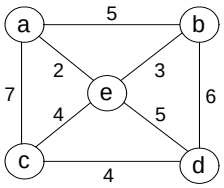
In fact, every positive integer can be uniquely represented in this form, obtained from its ternary expansion as described above. For example,

$$\begin{aligned} 5 &= 12_3 = 1 \cdot 3^1 + 2 \cdot 3^0 = 1 \cdot 3^1 + (3 - 1) \cdot 3^0 = 2 \cdot 3^1 - 1 \cdot 3^0 \\ &= (3 - 1) \cdot 3^1 - 1 \cdot 3^0 = 1 \cdot 3^2 - 1 \cdot 3^1 - 1 \cdot 3^0. \end{aligned}$$

(Note that if we start with the rightmost 2, after a simplification, the new rightmost 2, if any, will be at some position to the left of the starting one. This proves that after a finite number of such replacements, we will be able to eliminate all the 2's.) Using the representation $l = \sum_{i=1}^{n} \beta_i 3^{i-1}$, we can weigh load l by placing all the weights $w_i = 3^{i-1}$ for negative β_i's along with the load on one cup of the scale and all the weights $w_i = 3^{i-1}$ for positive β_i's on the opposite cup.

Now we'll prove that no set of n weights can cover a larger range of consecutive integral loads than $(3^n - 1)/2$. Each of the n weights can be either put on the left cup of the scale, or put on the right cup, or not to be used at all. Hence, there are $3^n - 1$ possible arrangements of the weights on the scale, with each of them having its mirror image (where all the weights are switched to the opposite pan of the scale). Eliminating this symmetry, leaves us with
just $(3^n - 1)/2$ arrangements, which can weight at most $(3^n - 1)/2$ different integral loads. Therefore, the largest range of consecutive integral loads they can cover cannot exceed $(3^n - 1)/2$.

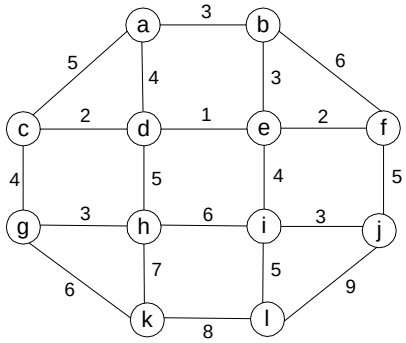
7. a. Apply Prim's algorithm to the following graph:



Tree vertices	Priority queue of remaining vertices
a(-,-)	b(a,5) c(a,7) d(a,∞) **e(a,2)**
e(a,2)	**b(e,3)** c(e,4) d(e,5)
b(e,3)	**c(e,4)** d(e,5)
c(e,4)	**d(c,4)**
d(c,4)	

The minimum spanning tree found by the algorithm comprises the edges *ae*, *eb*, *ec*, and *cd*.

b. Apply Prim's algorithm to the following graph:



Tree vertices	Priority queue of fringe vertices
a(-,-)	**b(a,3)** c(a,5) d(a,4)
b(a,3)	c(a,5) d(a,4) **e(b,3)** f(b,6)
e(b,3)	*c(a,5)* **d(e,1)** f(e,2) i(e,4)
d(e,1)	**c(d,2)** f(e,2) i(e,4) h(d,5)
c(d,2)	**f(e,2)** i(e,4) h(d,5) g(c,4)
f(e,2)	**i(e,4)** h(d,5) g(c,4) j(f,5)
i(e,4)	h(d,5) g(c,4) **j(i,3)** l(i,5)
j(i,3)	h(d,5) **g(c,4)** l(i,5)
g(c,4)	**h(g,3)** l(i,5) k(g,6)
h(g,3)	**l(i,5)** k(g,6)
l(i,5)	**k(g,6)**
k(g,6)	

The minimum spanning tree found by the algorithm comprises the edges *ab*, *be*, *ed*, *dc*, *ef*, *ei*, *ij*, *cg*, *gh*, *il*, *gk*.

8. There is no need to check the graph's connectivity because Prim's algorithm can do it itself. If the algorithm reaches all the graph's vertices (via edges of finite lengths), the graph is connected, otherwise, it is not.

9. a. The simplest and most logical solution is to assign all the edge weights to 1.

b. Applying a depth-first search (or breadth-first search) traversal to get a depth-first search tree (or a breadth-first search tree), is conceptually simpler and for sparse graphs represented by their adjacency lists faster.

10. The number of spanning trees for any weighted connected graph is a positive finite number. (At least one spanning tree exists, e.g., the one obtained by a depth-first search traversal of the graph. And the number of spanning trees must be finite because any such tree comprises a subset of edges of the finite set of edges of the given graph.) Hence, one can always find a spanning tree with the smallest total weight among the finite number of the candidates.

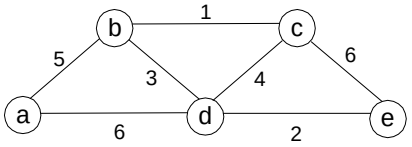
    Let's prove now that the minimum spanning tree is unique if all the weights are distinct. We'll do this by contradiction, i.e., by assuming that there exists a graph $G = (V, E)$ with all distinct weights but with more than one minimum spanning tree. Let $e_1, ..., e_{|V|-1}$ be the list of edges composing the minimum spanning tree T_P obtained by Prim's algorithm with some specific vertex as the algorithm's starting point and let T' be another minimum spanning tree. Let $e_i = (v, u)$ be the first edge in the list $e_1, ..., e_{|V|-1}$ of the edges of T_P which is not in T' (if $T_P \neq T'$, such edge must exist) and let (v, u') be an edge of T' connecting v with a vertex not in the subtree T_{i-1} formed by $\{e_1, ..., e_{i-1}\}$ (if $i = 1$, T_{i-1} consists of vertex v only). Similarly to the proof of Prim's algorithms correctness, let us replace (v, u') by $e_i = (v, u)$ in T'. It will create another spanning tree, whose weight is smaller than the weight of T' because the weight of $e_i = (v, u)$ is smaller than the weight of (v, u'). (Since e_i was chosen by Prim's algorithm, its weight is the smallest among all the weights on the edges connecting the tree vertices of the subtree T_{i-1} and the vertices adjacent to it. And since all the weights are distinct, the weight of (v, u') must be strictly greater than the weight of $e_i = (v, u)$.) This contradicts the assumption that T' was a minimum spanning tree.

11. If a key's value in a min-heap was decreased, it may need to be pushed up (via swaps) along the chain of its ancestors until it is smaller than or equal to its parent or reaches the root. If a key's value in a min-heap was increased, it may need to be pushed down by swaps with the smaller of its current children until it is smaller than or equal to its children or reaches a leaf. Since the height of a min-heap with n nodes is equal to $\lfloor \log_2 n \rfloor$ (by the same reason the height of a max-heap is given by this formula—see Section 6.4), the operation's efficiency is in $O(\log n)$. (Note: The old value of the key in question need not be known, of course. Comparing the new value with that of the parent and, if the min-heap condition holds, with the smaller of the two children, will suffice.)

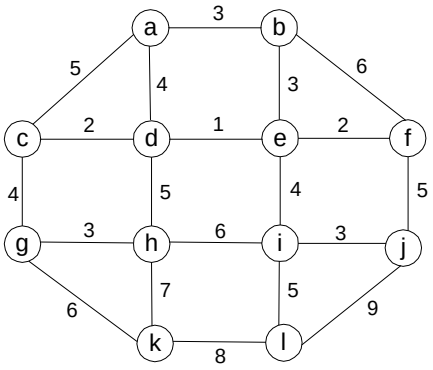
Exercises 9.2

1. Apply Kruskal's algorithm to find a minimum spanning tree of the following graphs.

   a.



   b.



2. Indicate whether the following statements are true or false:

   a. If e is a minimum-weight edge in a connected weighted graph, it must be among edges of at least one minimum spanning tree of the graph.

   b. If e is a minimum-weight edge in a connected weighted graph, it must be among edges of each minimum spanning tree of the graph.

   c. If edge weights of a connected weighted graph are all distinct, the graph must have exactly one minimum spanning tree.

   d. If edge weights of a connected weighted graph are not all distinct, the graph must have more than one minimum spanning tree.

3. What changes, if any, need to be made in algorithm *Kruskal* to make it find a ***minimum spanning forest*** for an arbitrary graph? (A minimum spanning forest is a forest whose trees are minimum spanning trees of the graph's connected components.)

4. Will either Kruskal's or Prim's algorithm work correctly on graphs that have negative edge weights?

5. Design an algorithm for finding a ***maximum spanning tree***—a spanning tree with the largest possible edge weight—of a weighted connected graph.

6. Rewrite the pseudocode of Kruskal's algorithm in terms of the operations of the disjoint subsets' ADT.

7. ▷ Prove the correctness of Kruskal's algorithm.

8. Prove that the time efficiency of $find(x)$ is in $O(\log n)$ for the union-by-size version of quick union.

9. Find at least two Web sites with animations of Kruskal's and Prim's algorithms. Discuss their merits and demerits..

10. Design and conduct an experiment to empirically compare the efficiencies of Prim's and Kruskal's algorithms on random graphs of different sizes and densities.

11. ► ***Steiner tree*** Four villages are located at the vertices of a unit square in the Euclidean plane. You are asked to connect them by the shortest network of roads so that there is a path between every pair of the villages along those roads. Find such a network.

Hints to Exercises 9.2

1. Trace the algorithm for the given graphs the same way it is done for another input in the section.

2. Two of the four assertions are true, the other two are false.

3. Applying Kruskal's algorithm to a disconnected graph should help to answer the question.

4. The answer is the same for both algorithms. If you believe that the algorithms work correctly on graphs with negative weights, prove this assertion; it you believe this is not to be the case, give a counterexample for each algorithm.

5. Is the general trick of transforming maximization problems to their minimization counterparts (see Section 6.6) applicable here?

6. Substitute the three operations of the disjoint subsets' ADT—$makeset(x)$, $find(x)$, and $union(x, y)$—in the appropriate places of the pseudocode given in the section.

7. Follow the plan used in Section 9.1 to prove the correctness of Prim's algorithm.

8. The argument is very similar to the one made in the section for the union-by-size version of quick find.

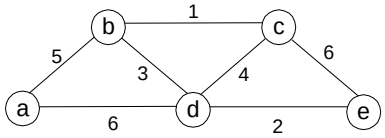
9. You may want to take advantage of the list of desirable characteristics in algorithm visualizations, which is given in Section 2.7.

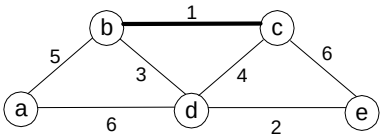
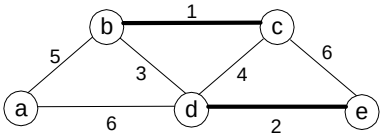
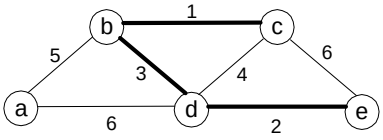
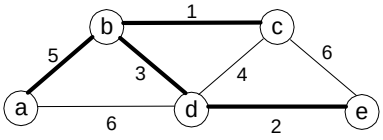
10. n/a

11. The question is not trivial because introducing extra points (called *Steiner points*) may make the total length of the network smaller than that of a minimum spanning tree of the square. Solving first the problem for three equidistant points might give you an indication how a solution to the problem in question could look like.

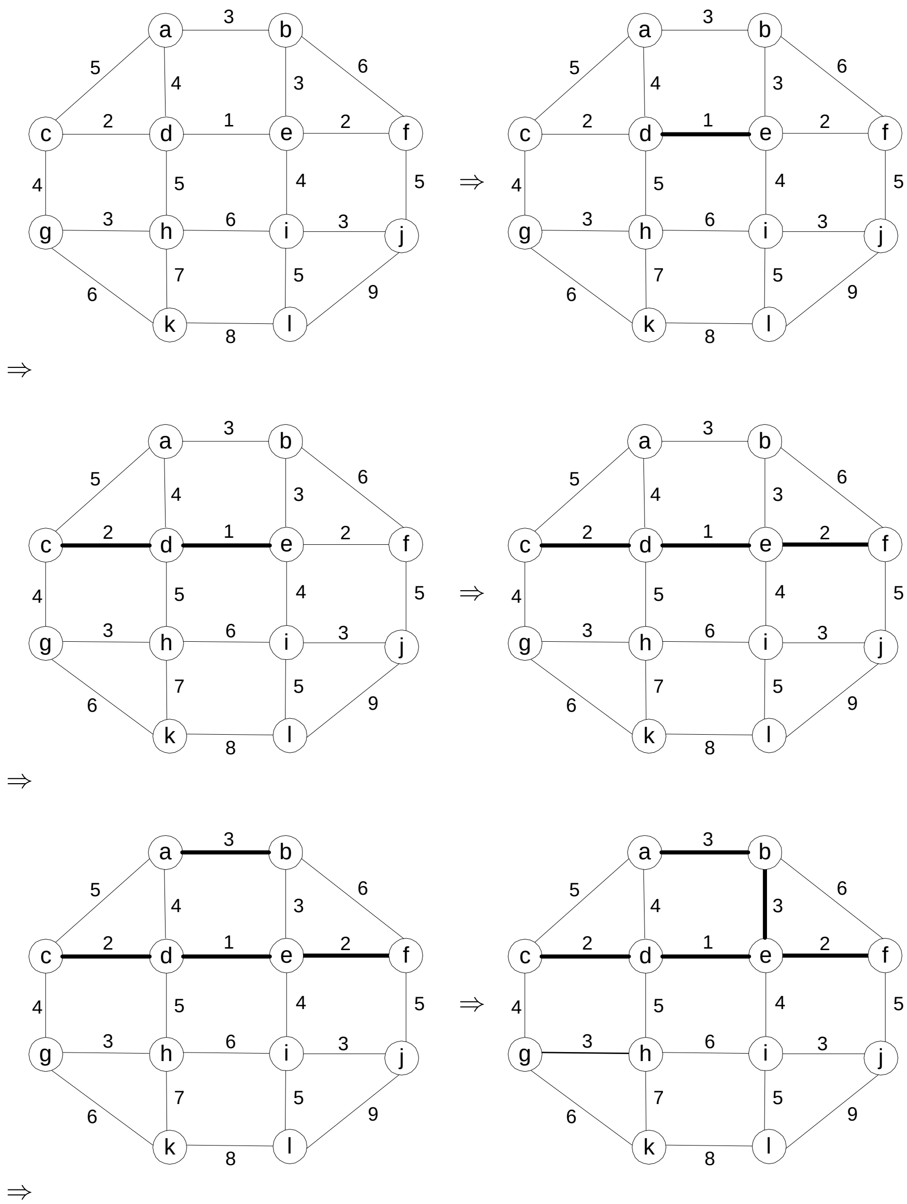
Solutions to Exercises 9.2

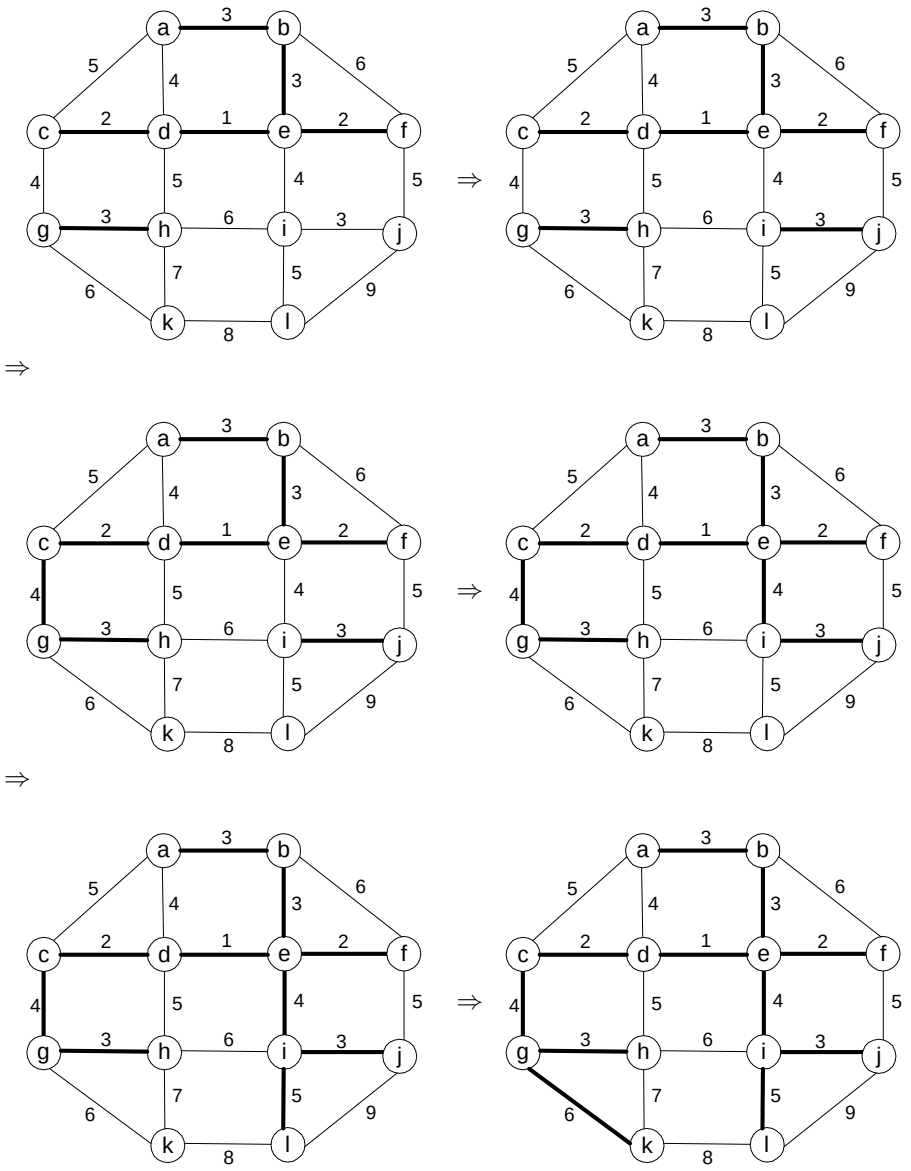
1. a.



Tree edges	Sorted list of edges (selected edges are shown in bold)	Illustration
	bc 1 de 2 bd 3 cd 4 ab 5 ad 6 ce 6	
bc 1	bc 1 **de** 2 bd 3 cd 4 ab 5 ad 6 ce 6	
de 2	bc 1 de 2 **bd** 3 cd 4 ab 5 ad 6 ce 6	
bd 3	bc 1 de 2 bd 3 cd 4 **ab** 5 ad 6 ce 6	
ab 5		

b.





2. a. True. (Otherwise, Kruskal's algorithm would be invalid.)

   b. False. As a simple counterexample, consider a complete graph with three vertices and the same weight on its three edges

   c. True (Problem 10 in Exercises 9.1).

d. False (see, for example, the graph of Problem 1a).

3. Since the number of edges in a minimum spanning forest of a graph with $|V|$ vertices and $|C|$ connected components is equal to $|V| - |C|$ (this formula is a simple generalization of $|E| = |V| - 1$ for connected graphs), *Kruskal*(G) will never get to $|V| - 1$ tree edges unless the graph is connected. A simple remedy is to replace the loop **while** *ecounter* $< |V| - 1$ with **while** $k < |E|$ to make the algorithm stop after exhausting the sorted list of its edges.

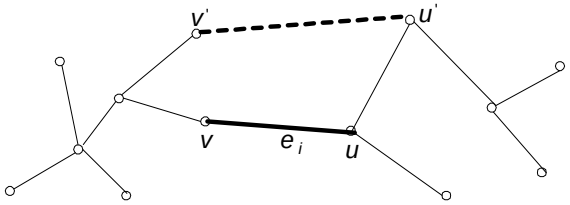
4. Both algorithms work correctly for graphs with negative edge weights. One way of showing this is to add to all the weights of a graph with negative weights some large positive number. This makes all the new weights positive, and one can "translate" the algorithms' actions on the new graph to the corresponding actions on the old one. Alternatively, you can check that the proofs justifying the algorithms' correctness do not depend on the edge weights being nonnegative.

5. Replace each weight $w(u, v)$ by $-w(u, v)$ and apply any minimum spanning tree algorithm that works on graphs with arbitrary weights (e.g., Prim's or Kruskal's algorithm) to the graph with the new weights.

6. **Algorithm** *Kruskal*(G)
   //Kruskal's algorithm with explicit disjoint-subsets operations
   //Input: A weighted connected graph $G = \langle V, E \rangle$
   //Output: E_T, the set of edges composing a minimum spanning tree of G
   sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le ... \le w(e_{i_{|E|}})$
   for each vertex $v \in V$ *make*(v)
   $E_T \leftarrow \emptyset$; *ecounter* $\leftarrow 0$ //initialize the set of tree edges and its size
   $k \leftarrow 0$ //the number of processed edges
   while *ecounter* $< |V| - 1$
   $k \leftarrow k + 1$
   if *find*$(u) \neq$ *find*(v) //u, v are the endpoints of edge e_{i_k}
   $E_T \leftarrow E_T \cup \{e_{i_k}\}$; *ecounter* $\leftarrow$ *ecounter* $+ 1$
   union(u, v)
   return E_T

7. Let us prove by induction that each of the forests F_i, $i = 0, ..., |V| - 1$, of Kruskal's algorithm is a part (i.e., a subgraph) of some minimum spanning tree. (This immediately implies, of course, that the last forest in the sequence, $F_{|V|-1}$, is a minimum spanning tree itself. Indeed, it contains all vertices of the graph, and it is connected because it is both acyclic and has $|V| - 1$ edges.) The basis of the induction is trivial, since F_0 is

made up of $|V|$ single-vertex trees and therefore must be a subgraph of any spanning tree of the graph. For the inductive step, let us assume that F_{i-1} is a subgraph of some minimum spanning tree T. We need to prove that F_i, generated from F_{i-1} by Kruskal's algorithm, is also a part of a minimum spanning tree. We prove this by contradiction by assuming that no minimum spanning tree of the graph can contain F_i. Let $e_i = (v, u)$ be the minimum weight edge added by Kruskal's algorithm to forest F_{i-1} to obtain forest F_i. (Note that vertices v and u must belong to different trees of F_{i-1}—otherwise, edge (v, u) would've created a cycle.) By our assumption, e_i cannot belong to T. Therefore, if we add e_i to T, a cycle must be formed (see the figure below). In addition to edge $e_i = (v, u)$, this cycle must contain another edge (v', u') connecting a vertex v' in the same tree of F_{i-1} as v to a vertex u' not in that tree. (It is possible that v' coincides with v or u' coincides with u but not both.) If we now delete the edge (v', u') from this cycle, we will obtain another spanning tree of the entire graph whose weight is less than or equal to the weight of T since the weight of e_i is less than or equal to the weight of (v', u'). Hence, this spanning tree is a minimum spanning tree, which contradicts the assumption that no minimum spanning tree contains F_i. This completes the correctness proof of Kruskal's algorithm.

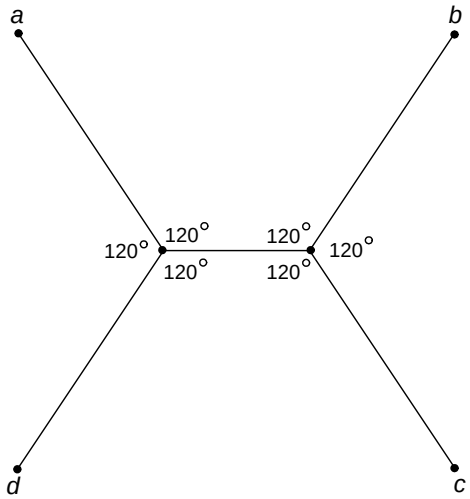


8. In the *union-by-size* version of *quick-union*, each vertex starts at depth 0 of its own tree. The depth of a vertex increases by 1 when the tree it is in is attached to a tree with at least as many nodes during a union operation. Since the total number of nodes in the new tree containing the node is at least twice as much as in the old one, the number of such increases cannot exceed $\log_2 n$. Therefore the height of any tree (which is the largest depth of the tree's nodes) generated by a legitimate sequence of unions will not exceed $\log_2 n$. Hence, the efficiency of *find*(x) is in $O(\log n)$ because *find*(x) traverses the pointer chain from the x's node to the tree's root.

9. n/a

10. n/a

11. The minimum Steiner tree that solves the problem is shown below. (The other solution can be obtained by rotating the figure 90°.)



A popular discussion of Steiner trees can be found in "Last Recreations: Hydras, Eggs, and Other Mathematical Mystifications" by Martin Gardner. In general, no polynomial time algorithm is known for finding a minimum Steiner tree; moreover, the problem is known to be *NP*-hard (see Section 11.3). For the state-of-the-art information, see, e.g., The Steiner Tree Page at http://ganley.org/steiner/.

Exercises 9.3

1. Explain what adjustments if any need to be made in Dijkstra's algorithm and/or in an underlying graph to solve the following problems.

   a. Solve the single-source shortest-paths problem for directed weighted graphs.

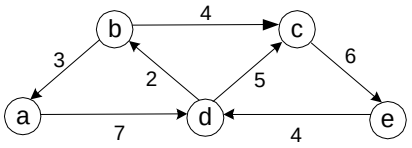
   b. Find a shortest path between two given vertices of a weighted graph or digraph. (This variation is called the ***single-pair shortest-path problem***.)

   c. Find the shortest paths to a given vertex from each other vertex of a weighted graph or digraph. (This variation is called the ***single-destination shortest-paths problem***.)

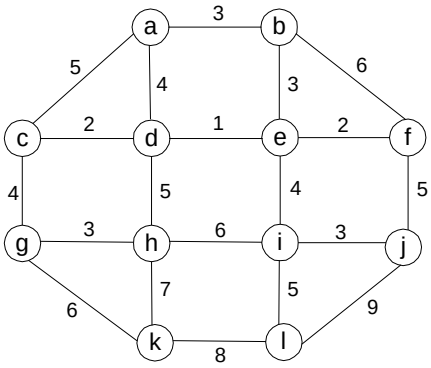
   d. Solve the single-source shortest-path problem in a graph with nonnegative numbers assigned to its vertices (and the length of a path defined as the sum of the vertex numbers on the path).

2. Solve the following instances of the single-source shortest-paths problem with vertex a as the source:

   a.



   b.



3. Give a counterexample that shows that Dijkstra's algorithm may not work for a weighted connected graph with negative weights.

4. Let T be a tree constructed by Dijkstra's algorithm in the process of solving the single-source shortest-path problem for a weighted connected graph G.

   a. True or false: T is a spanning tree of G?

   b. True or false: T is a minimum spanning tree of G?

5. Write a pseudocode of a simpler version of Dijkstra's algorithm that finds only the distances (i.e., the lengths of shortest paths but not shortest paths themselves) from a given vertex to all other vertices of a graph represented by its weight matrix.

6. ▷ Prove the correctness of Dijkstra's algorithm for graphs with positive weights.

7. Design a linear-time algorithm for solving the single-source shortest-paths problem for dags (directed acyclic graphs) represented by their adjacency lists.

8. Design an efficient algorithm for finding the length of a longest path in a dag. (This problem is important because it determines a lower bound on the total time needed for completing a project composed of precedence-constrained tasks.)

9. *Shortest-path modeling* Assume you have a model of a weighted connected graph made of balls (representing the vertices) connected by strings of appropriate lengths (representing the edges).

   a. Describe how you can solve the single-pair shortest-path problem with this model.

   b. Describe how you can solve the single-source shortest-paths problem with this model.

10. Revisit Problem 6 in Exercises 1.3 about determining the best route for a subway passenger to take from one designated station to another in a well-developed subway system like those in Washington, DC and London, UK. Write a program for this task.

Hints to Exercises 9.3

1. One of the questions requires no changes in either the algorithm or the graph; the others require simple adjustments.
2. Just trace the algorithm on the given graphs the same way it is done for an example in the section.
3. The nearest vertex does not have to be adjacent to the source in such a graph.
4. Only one of the assertions is correct. Find a small counterexample for the other.
5. Simplify the pseudocode given in the section by implementing the priority queue as an unordered array and ignoring the parental labeling of vertices.
6. Prove it by induction on the number of vertices included in the tree constructed by the algorithm.
7. Topologically sort the dag's vertices first.
8. Topologically sort the dag's vertices first.
9. Take advantage of the ways of thinking used in geometry and physics.
10. Before you embark on implementing a shortest-path algorithm, you have to decide what criterion determines the "best route". Of course, it would be highly desirable to have a program asking the user which of several possible criteria he or she wants applied.

Solutions to Exercises 9.3

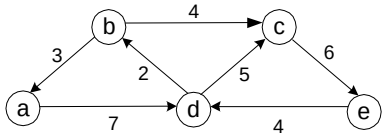
1. a. It will suffice to take into account edge directions in processing adjacent vertices.

   b. Start the algorithm at one of the given vertices and stop it as soon as the other vertex is added to the tree.

   c. If the given graph is undirected, solve the single-source problem with the destination vertex as the source and reverse all the paths obtained in the solution. If the given graph is directed, reverse all its edges first, solve the single-source problem for the new digraph with the destination vertex as the source, and reverse the direction of all the paths obtained in the solution.

   d. Create a new graph by replacing every vertex v with two vertices v' and v'' connected by an edge whose weight is equal to the given weight of vertex v. All the edges entering and leaving v in the original graph will enter v' and leave v'' in the new graph, respectively. Assign the weight of 0 to each original edge. Applying Dijkstra's algorithm to the new graph will solve the problem.

2. a.

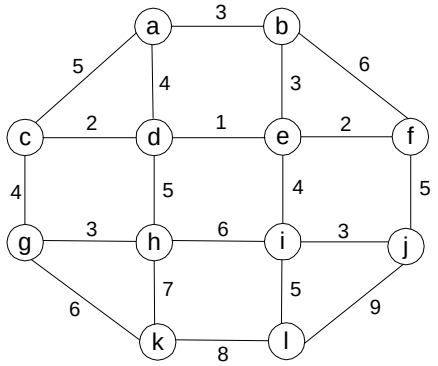


Tree vertices	Remaining vertices
a(-,0)	b(-,∞) c(–,∞) **d(a,7)** e(-,∞)
d(a,7)	**b(d,7+2)** c(d,7+5) e(-,∞)
b(d,9)	**c(d,12)** e(-,∞)
c(d,12)	**e(c,12+6)**
e(c,18)	

The shortest paths (identified by following nonnumeric labels backwards from a destination vertex to the source) and their lengths are:

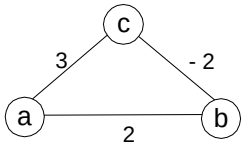
from a to d:	$a - d$	of length 7
from a to b:	$a - d - b$	of length 9
from a to c:	$a - d - c$	of length 12
from a to e:	$a - d - c - e$	of length 18

b.



Tree vertices	Fringe vertices	Shortest paths from a
a(-,0)	**b(a,3)** c(a,5) d(a,4)	to b: $a-b$ of length 3
b(a,3)	c(a,5) **d(a,4)** e(b,3+3) f(b,3+6)	to d: $a-d$ of length 4
d(a,4)	**c(a,5)** e(d,4+1) f(a,9) h(d,4+5)	to c: $a-c$ of length 5
c(a,5)	**e(d,5)** f(a,9) h(d,9) g(c,5+4))	to e: $a-d-e$ of length 5
e(d,5)	**f(e,5+2)** h(d,9) g(c,9) i(e,5+4)	to f: $a-d-e-f$ of length 7
f(e,7)	**h(d,9)** g(c,9) i(e,9) j(f,7+5)	to h: $a-d-h$ of length 9
h(d,9)	**g(c,9)** i(e,9) j(f,12) k(h,9+7))	to g: $a-c-g$ of length 9
g(c,9)	**i(e,9)** j(f,12) k(g,9+6)	to i: $a-d-e-i$ of length 9
i(e,9)	**j(f,12)** k(g,15) l(i,9+5)	to j: $a-d-e-f-j$ of length 12
j(f,12)	k(g,15) **l(i,14)**	to l: $a-d-e-i-l$ of length 14
l(i,14)	**k(g,15)**	to k: $a-c-g-k$ of length 15
k(g,15)		

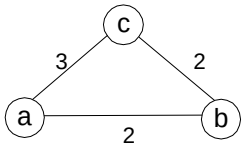
3. Consider, for example, the graph



As the shortest path from a to b, Dijkstra's algorithm yields $a-b$ of length 2, which is longer than $a-c-b$ of length 1.

4. a. True: On each iteration, we add to a previously constructed tree an edge connecting a vertex in the tree to a vertex that is not in the tree. So, the resulting structure must be a tree. And, after the last operation, it includes all the vertices of the graph. Hence, it's a spanning tree.

b. False. Here is a simple counterexample:



With vertex a as the source, Dijkstra's algorithm yields, as the shortest path tree, the tree composed of edges (a, b) and (a, c). The graph's minimum spanning tree is composed of (a, b) and (b, c).

5. **Algorithm** *SimpleDijkstra*$(W[0..n-1, 0..n-1], s)$
   //Input: A matrix of nonnegative edge weights W and
   // integer s between 0 and $n-1$ indicating the source
   //Output: An array $D[0..n-1]$ of the shortest path lengths
   // from s to all vertices
   for $i \leftarrow 0$ **to** $n-1$ **do**
   $D[i] \leftarrow \infty$; $treeflag[i] \leftarrow$ **false**
   $D[s] \leftarrow 0$
   for $i \leftarrow 0$ **to** $n-1$ **do**
   $dmin \leftarrow \infty$
   for $j \leftarrow 0$ **to** $n-1$ **do**
   if not $treeflag[j]$ **and** $D[j] < dmin$
   $jmin \leftarrow j$; $dmin \leftarrow D[jmin]$
   $treeflag[jmin] \leftarrow$ **true**
   for $j \leftarrow 0$ **to** $n-1$ **do**
   if not $treeflag[j]$ **and** $dmin + W[jmin, j] < \infty$
   $D[j] \leftarrow dmin + W[jmin, j]$
   return D

6. We will prove by induction on the number of vertices i in tree T_i constructed by Dijkstra's algorithm that this tree contains i closest vertices to source s (including the source itself), for each of which the tree path from s to v is a shortest path of length d_v. For $i = 1$, the assertion is obviously true for the trivial path from the source to itself. For the general step, assume that it is true for the algorithm's tree T_i with i vertices. Let v_{i+1} be the vertex added next to the tree by the algorithm. All the vertices on a shortest path from s to v_{i+1} preceding v_{i+1} must be in T_i because they are closer to s than v_{i+1}. (Otherwise, the first vertex on the path from s to v_{i+1} that is not in T_i would've been added to T_i instead of v_{i+1}.) Hence, the $(i+1)$st closest vertex can be selected as the algorithm does: by minimizing the sum of d_v (the shortest distance from s to $v \in T_i$ by the assumption of the induction) and the length of the edge from v to an adjacent vertex not in the tree.

7. **Algorithm** *DagShortestPaths*(G, s)
   //Solves the single-source shortest paths problem for a dag
   //Input: A weighted dag $G = \langle V, E \rangle$ and its vertex s
   //Output: The length d_v of a shortest path from s to v and
   // its penultimate vertex p_v for every vertex v in V
   topologically sort the vertices of G
   for every vertex v **do**
   $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
   $d_s \leftarrow 0$
   for every vertex v taken in topological order **do**
   for every vertex u adjacent to v **do**
   if $d_v + w(v, u) < d_u$
   $d_u \leftarrow d_v + w(v, u)$; $p_u \leftarrow v$

   Topological sorting can be done in $\Theta(|V|+|E|)$ time (see Section 5.3). The distance initialization takes $\Theta(|V|)$ time. The innermost loop is executed for every edge of the dag. Hence, the total running time is in $\Theta(|V|+|E|)$.

8. **Algorithm** *DagLongestPath*(G)
   //Finds the length of a longest path in a dag
   //Input: A weighted dag $G = \langle V, E \rangle$
   //Output: The length of its longest path *dmax*
   topologically sort the vertices of G
   for every vertex v **do**
   $d_v \leftarrow 0$ //the length of the longest path to v seen so far
   for every vertex v taken in topological order **do**
   for every vertex u adjacent to v **do**
   if $d_v + w(v, u) > d_u$
   $d_u \leftarrow d_v + w(v, u)$
   $dmax \leftarrow 0$
   for every vertex v **do**
   if $d_v > dmax$
   $dmax \leftarrow d_v$
   return *dmax*

9. a. Take the two balls representing the two singled out vertices in two hands and stretch the model to get the shortest path in question as a straight line between the two ball-vertices.

   b. Hold the ball representing the source in one hand and let the rest of the model hang down: The force of gravity will make the shortest path to each of the other balls be on a straight line down.

10. n/a

Exercises 9.4

1. a. Construct a Huffman code for the following data:

character	A	B	C	D	_
probability	0.4	0.1	0.2	0.15	0.15

   b. Encode the text ABACABAD using the code of question a.

   c. Decode the text whose encoding is 100010111001010 in the code of question a.

2. For data transmission purposes, it is often desirable to have a code with a minimum variance of the codeword lengths (among codes of the same average length). Compute the average and variance of the codeword length in two Huffman codes that result from a different tie breaking during a Huffman code construction for the following data:

character	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4

3. Indicate whether each of the following properties are true for every Huffman code.

   a. The codewords of the two least frequent characters have the same length.

   b. The codeword's length of a more frequent character is always smaller than or equal to the codeword's length of a less frequent one.

4. What is the maximal length of a codeword possible in a Huffman encoding of an alphabet of n characters?

5. a. Write a pseudocode for the Huffman tree construction algorithm.

   b. What is the time efficiency class of the algorithm for constructing a Huffman tree as a function of the alphabet's size?

6. Show that a Huffman tree can be constructed in linear time if the alphabet's characters are given in a sorted order of their frequencies.

7. Given a Huffman coding tree, which algorithm would you use to get the codewords for all the characters? What is its time-efficiency class as a function of the alphabet's size?

8. Explain how one can generate a Huffman code without an explicit generation of a Huffman coding tree.

9. a. Write a program that constructs a Huffman code for a given English text and encode it.

b. Write a program for decoding an English text which has been encoded with a Huffman code.

c. Experiment with your encoding program to find a range of typical compression ratios for Huffman's encoding of English texts of, say, 1000 words.

d. Experiment with your encoding program to find out how sensitive the compression ratios are to using standard estimates of frequencies instead of actual frequencies of character occurrences in English texts.

10. *Card guessing* Design a strategy that minimizes the expected number of questions asked in the following game [Gar94], #52. You have a deck of cards that consists of one ace of spades, two deuces of spades, three threes, and on up to nine nines, making 45 cards in all. Someone draws a card from the shuffled deck, which you have to identify by asking questions answerable with yes or no.

Hints to Exercises 9.4

1. See the example given in the section.

2. After combining the two nodes with the lowest probabilities, resolve the tie arising on the next iteration in two different ways. For each of the two Huffman codes obtained, compute the mean and variance of the codeword length.

3. You may base your answers on the way Huffman's algorithm works or on the fact that Huffman codes are known to be optimal prefix codes.

4. The maximal length of a codeword relates to the height of Huffman's coding tree in an obvious fashion. Try to find a set of n specific frequencies for an alphabet of size n for which the tree has the shape yielding the longest codeword possible.

5. a. What is the most appropriate data structure for an algorithm whose principal operation is finding the two smallest elements in a given set, deleting them, and then adding a new item to the remaining ones?

   b. Identify the principal operations of the algorithm, the number of times they are executed, and their efficiencies for the data structure used.

6. Maintain two queues: one for given frequencies, the other for weights of new trees.

7. It would be natural to use one of the standard traversal algorithms.

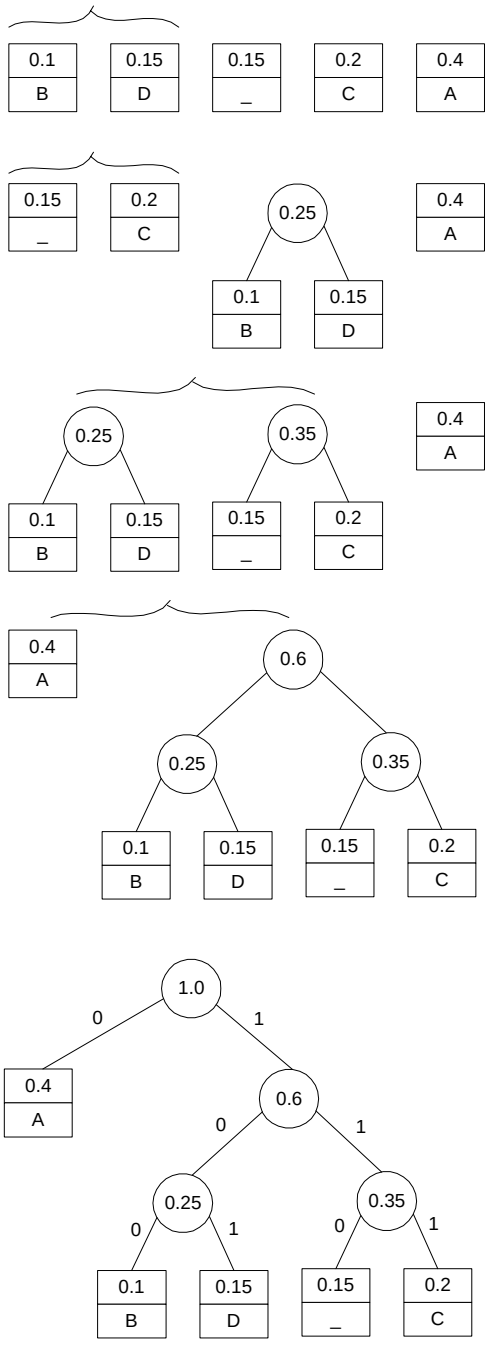
8. Generate the codewords right to left.

9. n/a

10. A similar example was discussed at the end of Section 9.4. Construct Huffman's tree and then come up with specific questions that would yield that tree. (You are allowed to ask questions such as: Is this card the ace, or a seven, or an eight?)

Solutions to Exercises 9.4

1. a.



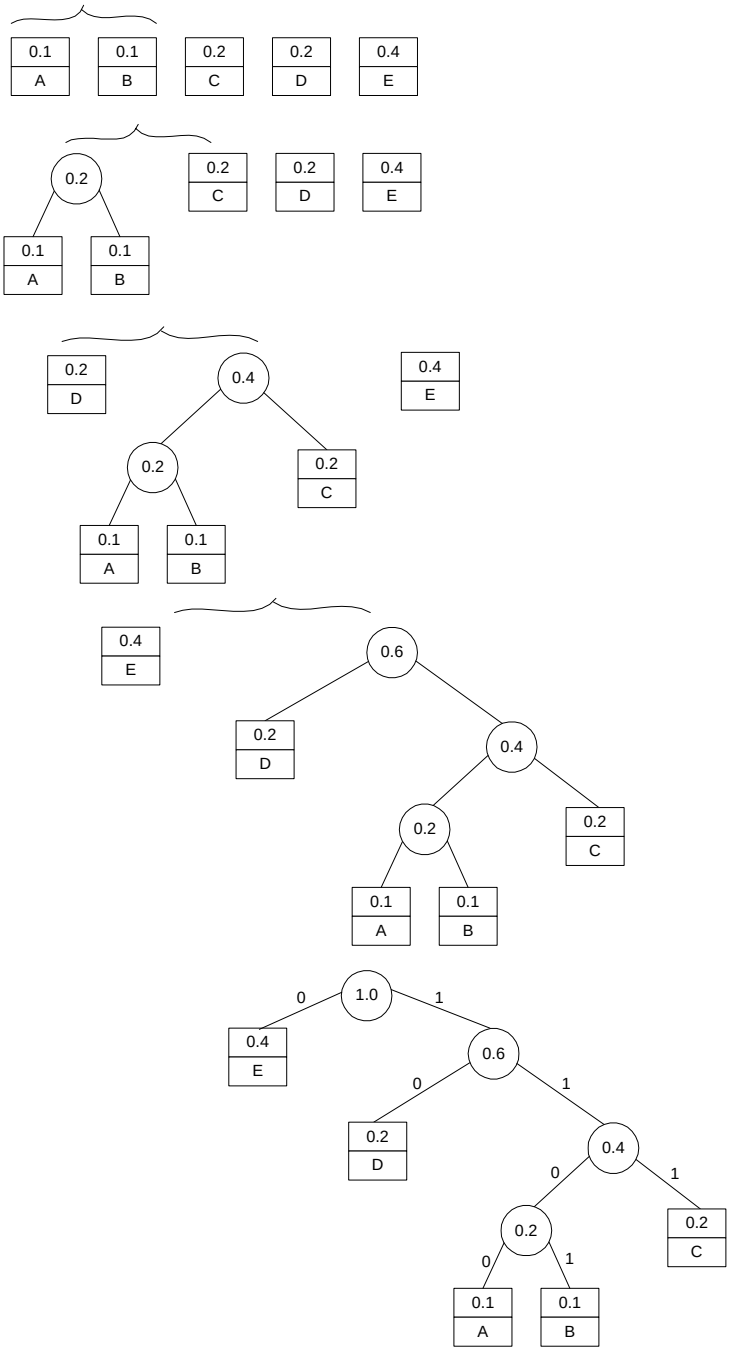
character	A	B	C	D	_
probability	0.4	0.1	0.2	0.15	0.15
codeword	0	100	111	101	110

b. The text ABACABAD will be encoded as 0100011101000101.

c. With the code of part a, 100010111001010 will be decoded as

$$\underset{B}{100}|\underset{A}{0}|\underset{D}{101}|\underset{\_}{110}|\underset{A}{0}|\underset{D}{101}|\underset{A}{0}$$

2. Here is one way:



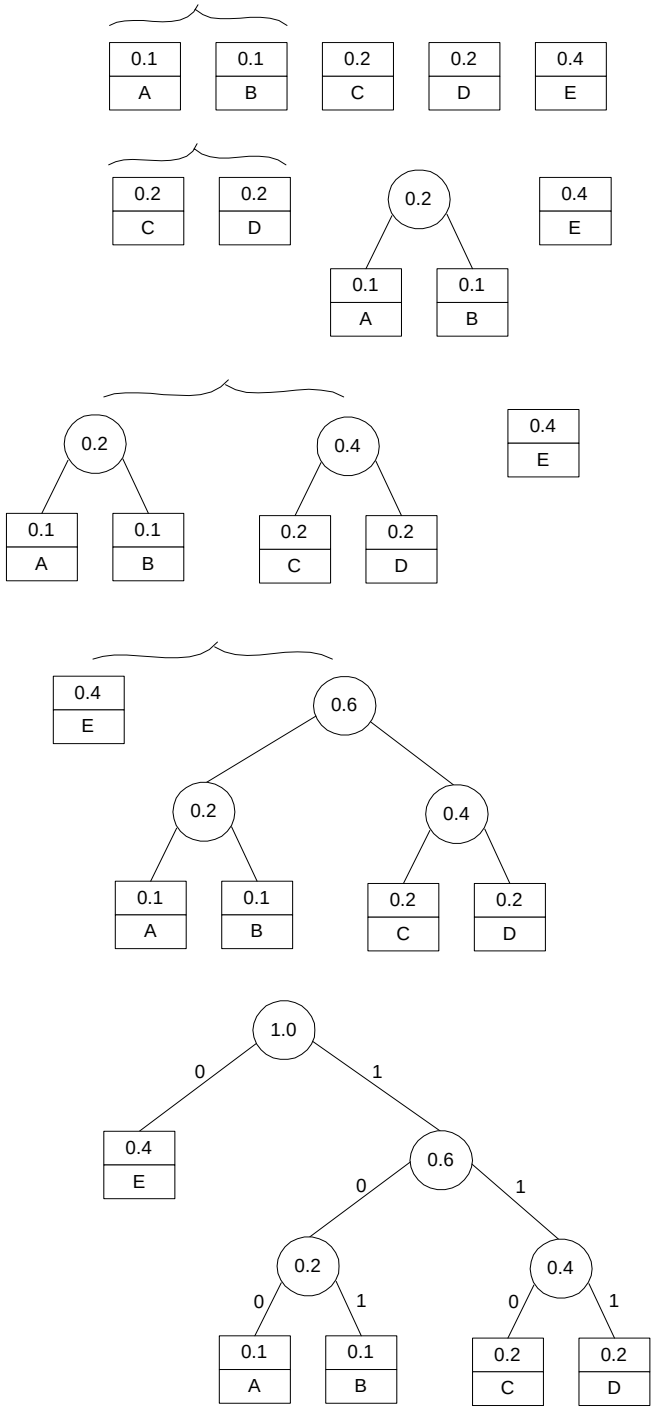
character	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4
codeword	1100	1101	111	10	0
length	4	4	3	2	1

Thus, the mean and variance of the codeword's length are, respectively,

$$\bar{l} = \sum_{i=1}^{5} l_i p_i = 4 \cdot 0.1 + 4 \cdot 0.1 + 3 \cdot 0.2 + 2 \cdot 0.2 + 1 \cdot 0.4 = 2.2 \quad \text{and}$$

$$Var = \sum_{i=1}^{5} (l_i - \bar{l})^2 p_i = (4\text{-}2.2)^2 0.1 + (4\text{-}2.2)^2 0.1 + (3\text{-}2.2)^2 0.2 + (2\text{-}2.2)^2 0.2 + (1\text{-}2.2)^2 0.4 = 1.36.$$

Here is another way:



character	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4
codeword	100	101	110	111	0
length	3	3	3	3	1

Thus, the mean and variance of the codeword's length are, respectively,

$$\bar{l} = \sum_{i=1}^{5} l_i p_i = 2.2 \quad \text{and} \quad Var = \sum_{i=1}^{5} (l_i - \bar{l})^2 p_i = 0.96.$$

3. a. Yes. This follows immediately from the way Huffman's algorithm operates: after each of its iterations, the two least frequent characters that are combined on the first iteration are always on the same level of their tree in the algorithm's forest. An easy formal proof of this obvious observation is by induction.
(Note that if there are more than two least frequent characters, the assertion may be false for some pair of them, e.g., $A(\frac{1}{3})$, $B(\frac{1}{3})$, $C(\frac{1}{3})$.)

   b. Yes. Let's use the optimality of Huffman codes to prove this property by contradiction. Assume that there exists a Huffman code containing two characters c_i and c_j such that $p(c_i) > p(c_j)$ and $l(w(c_i)) > l(w(c_j))$, where $p(c_i)$ and $l(w(c_i))$ are the probability and codeword's length of c_i, respectively, and $p(c_j)$ and $l(w(c_j))$ are the probability and codeword's length of c_j, respectively. Let's create a new code by simply swapping the codewords of c_1 and c_2 and leaving the codewords for all the other characters the same. The new code will obviously remain prefix-free and its expected length $\sum_{k=1}^{n} l(w(c_k))p(c_k))$ will be smaller than that of the initial code. This contradicts the optimality of the initial Huffman code and, hence, proves the property in question.

4. The answer is $n-1$. Since two leaves corresponding to the two least frequent characters must be on the same level of the tree, the tallest Huffman coding tree has to have the remaining leaves each on its own level. The height of such a tree is $n-1$. An easy and natural way to get a Huffman tree of this shape is by assuming that $p_1 \le p_2 < ... < p_n$ and having the weight W_i of a tree created on the ith iteration of Huffman's algorithm, $i = 1, 2, ..., n-2$, be less than or equal to p_{i+2}. (Note that for such inputs, $W_i = \sum_{k=1}^{i+1} p_k$ for every $i = 1, 2, ..., n-1$.)

   As a specific example, it's convenient to consider consecutive powers of 2:

$$p_1 = p_2 \quad \text{and} \quad p_i = 2^{i-n-1} \text{ for } i = 2, ..., n.$$

   (For, say, $n = 4$, we have $p_1 = p_2 = 1/8$, $p_3 = 1/4$ and $p_4 = 1/2$.)

Indeed, $p_i = 2^i/2^{n+1}$ is an increasing sequence as a function of i. Further, $W_i = p_{i+2}$ for every $i = 1, 2, ..., n-2$, since

$$W_i = \sum_{k=1}^{i+1} p_k = p_1 + \sum_{k=2}^{i+1} p_k = 2^2/2^{n+1} + \sum_{k=2}^{i+1} 2^k/2^{n+1} = \frac{1}{2^{n+1}}(2^2 + \sum_{k=2}^{i+1} 2^k)$$
$$= \frac{1}{2^{n+1}}(2^2 + (2^{i+2} - 4)) = \frac{2^{i+2}}{2^{n+1}} = p_{i+2}.$$

5. a. The following pseudocode is based on maintaining a priority queue of trees, with the priorities equal the trees' weights.

Algorithm *Huffman*$(W[0..n-1])$
//Constructs Huffman's tree
//Input: An array $W[0..n-1]$ of weights
//Output: A Huffman tree with the given weights assigned to its leaves
initialize priority queue Q of size n with one-node trees and priorities equal to the elements of $W[0..n-1]$
while Q has more than one element **do**
    $T_l \leftarrow$ the minimum-weight tree in Q
    delete the minimum-weight tree in Q
    $T_r \leftarrow$ the minimum-weight tree in Q
    delete the minimum-weight tree in Q
    create a new tree T with T_l and T_r as its left and right subtrees
            and the weight equal to the sum of T_l and T_r weights
    insert T into Q
return T

Note: See also Problem 6 for an alternative algorithm.

b. The algorithm requires the following operations: initializing a priority queue, deleting its smallest element $2(n-1)$ times, computing the weight of a combined tree and inserting it into the priority queue $n-1$ times. The overall running time will be dominated by the time spent on deletions, even taking into account that the size of the priority queue will be decreasing from n to 2. For the min-heap implementation, the time efficiency will be in $O(n \log n)$; for the array or linked list representations, it will be in $O(n^2)$. (Note: For the coding application of Huffman trees, the size of the underlying alphabet is typically not large; hence, a simpler data structure for the priority queue might well suffice.)

6. The critical insight here is that the weights of the trees generated by Huffman's algorithm for nonnegative weights (frequencies) form a nondecreasing sequence. As the hint to this problem suggests, we can then maintain

two queues: one for given frequencies in nondecreasing order, the other for weights of new trees. On each iteration, we do the following: find the two smallest elements among the first two (ordered) elements in the queues (the second queue is empty on the first iteration and can contain just one element thereafter); add their sum to the second queue; and then delete these two elements from their queues. The algorithm stops after $n-1$ iterations (where n is the alphabet's size), each of which requiring a constant time.

7. Use one of the standard traversals of the binary tree and generate a bit string for each node of the tree as follows:. Starting with the empty bit string for the root, append 0 to the node's string when visiting the node's left subtree begins and append 1 to the node's string when visiting the node's right subtree begins. At a leaf, print out the current bit string as the leaf's codeword. Since Huffman's tree with n leaves has a total of $2n-1$ nodes (see Sec. 4.4), the efficiency will be in $\Theta(n)$.

8. We can generate the codewords right to left by the following method that stems immediately from Huffman's algorithm: when two trees are combined, append 0 in front of the current bit strings for each leaf in the left subtree and append 1 in front of the current bit strings for each leaf in the right subtree. (The substrings associated with the initial one-node trees are assumed to be empty.)

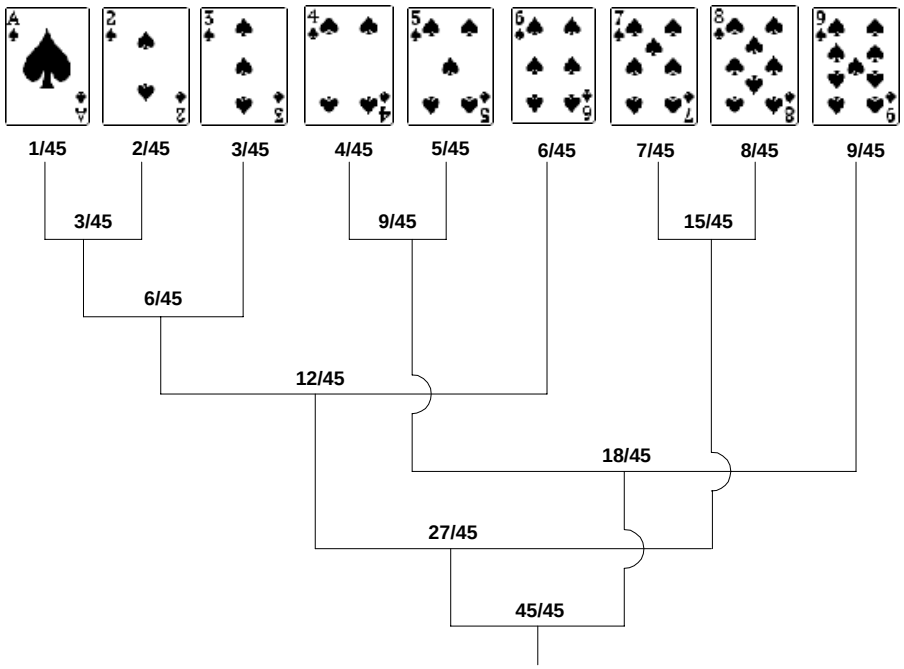
9. n/a

10. See the next page

10. The probabilities of a selected card be of a particular type is given in the following table:

card	ace	deuce	three	four	five	six	seven	eight	nine
probability	1/45	2/45	3/45	4/45	5/45	6/45	7/45	8/45	9/45

Huffman's tree for this data looks as follows:



The first question this tree implies can be phrased as follows: "Is the selected card a four, a five, or a nine?" . (The other questions can be phrased in a similar fashion.)

The expected number of questions needed to identify a card is equal to the weighted path length from the root to the leaves in the tree:

$$\bar{l} = \sum_{i=1}^{9} l_i p_i = \frac{5 \cdot 1}{45} + \frac{5 \cdot 2}{45} + \frac{4 \cdot 3}{45} + \frac{3 \cdot 5}{45} + \frac{3 \cdot 6}{45} + \frac{3 \cdot 7}{45} + \frac{3 \cdot 8}{45} + \frac{2 \cdot 9}{45} = \frac{135}{45} = 3.$$