This file contains the exercises, hints, and solutions for Chapter 6 of the book "Introduction to the Design and Analysis of Algorithms" 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 6.1

1. Recall that the ***median*** of a list of n numbers is defined as its $\lceil n/2 \rceil$ smallest element. (The median is larger than one half the elements and is smaller than the other half.) Design a presorting-based algorithm for finding the median and determine its efficiency class.

2. Consider the problem of finding the distance between the two closest numbers in an array of n numbers. (The distance between two numbers x and y is computed as $|x - y|$.)

   a. Design a presorting-based algorithm for solving this problem and determine its efficiency class.

   b. Compare the efficiency of this algorithm with that of the brute-force algorithm (see Problem 9 in Exercises 1.2).

3. Let $A = \{a_1,..., a_n\}$ and $B = \{b_1, ..., b_m\}$ be two sets of numbers. Consider the problem of finding their intersection, i.e., the set C of all the numbers that are in both A and B.

   a. Design a brute-force algorithm for solving this problem and determine its efficiency class.

   b. Design a presorting-based algorithm for solving this problem and determine its efficiency class.

4. Consider the problem of finding the smallest and largest elements in an array of n numbers.

   a. Design a presorting-based algorithm for solving this problem and determine its efficiency class.

   b. Compare the efficiency of the three algorithms: (i) the brute-force algorithm, (ii) this presorting-based algorithm, and (iii) the divide-and-conquer algorithm (see Problem 2 in Exercises 4.1).

5. Show that the average-case efficiency of one-time searching by the algorithm that consists of the most efficient comparison-based sorting algorithm followed by binary search is inferior to the average-case efficiency of sequential search.

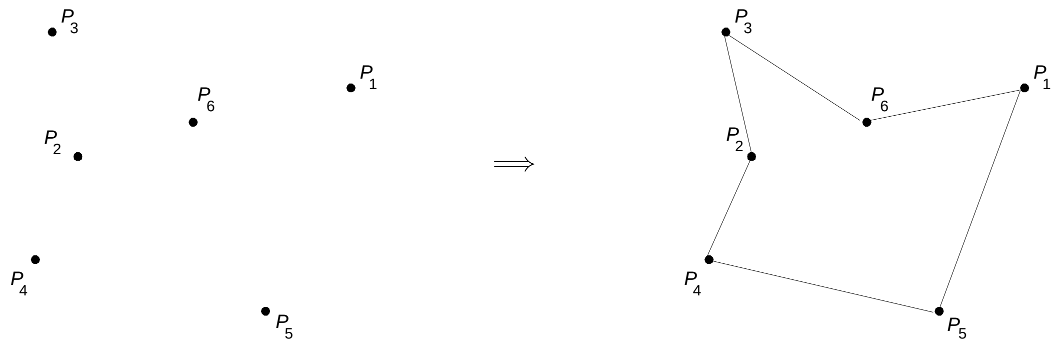
6. Estimate how many searches will be needed to justify time spend on presorting an array of 10^3 elements if sorting is done by mergesort and searching is done by binary search. (You may assume that all searches are for elements known to be in the array.) What about an array of 10^6 elements?

7. To sort or not to sort? Design a reasonably efficient algorithm for solving each of the following problems and determine its efficiency class.

   a. You are given n telephone bills and m checks sent to pay the bills ($n \geq m$). Assuming that telephone numbers are written on the checks, find out who failed to pay. (For simplicity, you may also assume that only one check is written for a particular bill and that it covers the bill in full.)

   b. You have a file of n student records indicating each student's number, name, home address, and date of birth. Find out the number of students from each of the 50 U.S. states.

8. Given a set of $n \geq 3$ points in the x–y coordinate plane, connect them in a simple polygon, i.e., a closed path through all the points so that its line segments (the polygon's edges) do not intersect (except for neighboring edges at their common vertex). For example,



   a. Does the problem always have a solution? Does it always have a unique solution?

   b. Design a reasonably efficient algorithm for solving this problem and indicate its efficiency class.

9. You have an array of n numbers and an integer s. Find out whether the array contains two elements whose sum is s. (For example, for the array 5, 9, 1, 3 and $s = 6$, the answer is yes, but for the same array and $s = 7$, the answer is no.) Design an algorithm for this problem with a better than quadratic time efficiency.

10. a. Design an efficient algorithm for finding all sets of anagrams in a large

file such as a dictionary of English words [Ben00]. For example, *eat, ate,* and *tea* belong to one such a set.

b. Write a program implementing the algorithm.

Hints to Exercises 6.1

1. The algorithm is suggested by the problem's statement. Its analysis is similar to the examples discussed in this section.

2. This problem is similar to one of the examples in this section.

3. a. Compare every element in one set with all the elements in the other.

   b. In fact, you can use presorting in three different ways: sort elements of just one of the sets, sort elements of each of the sets separately, and sort elements of the two sets together.

4. a. How do we find the smallest and largest elements in a sorted list?

   b. The brute-force algorithm and the divide-and-conquer algorithm are both linear.

5. Use the known results about the average-case efficiencies of the algorithms in this question.

6. Assume that sorting requires about $n \log_2 n$ comparisons. Use the known results about the number of comparisons made, on the average, in a successful search by binary search and by sequential search.

7. a. The problem is similar to one of the preceding problems in these exercises.

   b. How would you solve this problem if the student information were written on index cards? Better yet, think how somebody else, who has never taken a course on algorithms but possesses a good dose of common sense, would solve this problem.

8. a. Many problems of this kind have exceptions for one particular configuration of points. As to the question about a solution's uniqueness, you can get the answer by considering a few small "random" instances of the problem.

   b. Construct a polygon for a few small "random" instances of the problem. Try to construct polygons in some systematic fashion.

9. It helps to think about real numbers as ordered points on the real line. Considering the special case of $s = 0$, with a given array containing both negative and positive numbers, might be helpful, too.

10. Use the presorting idea twice.

Solutions to Exercises 6.1

1. Sort the list and then simply return the $\lceil n/2 \rceil$th element of the sorted list. Assuming the efficiency of the sorting algorithm is in $O(n \log n)$, the time efficiency of the entire algorithm will be in

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

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

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

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

3. a. Initialize a list to contain elements of $C = A \cap B$ to empty. Compare every element a_i in A $(1 \leq i \leq n)$ with successive elements of B: if $a_i = b_j$, add this value to the C list and proceed to the next element in A. (In fact, if $a_i = b_j$, b_j need not be compared with the remaining elements in A and may be deleted from B.) In the worst case of input sets with no common elements, the total number of element comparisons will be equal to nm, putting the algorithm's efficiency in $O(nm)$.

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

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

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

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

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

Third solution: Combine the elements of both A and B in a single list and sort it. Then scan this sorted list by comparing pairs of its consecutive elements: if $L_i = L_{i+1}$, add this common value to the C list and increment i by two. If sorting is done with an $n \log n$ algorithm, the total running time will be in

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

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

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

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

5. Since the average-case efficiency of any comparison-based sorting algorithm is known to be in $\Omega(n \log n)$ while the average-case efficiency of binary search is in $\Theta(\log n)$, the average-case efficiency of the searching algorithm combining the two will be in

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

   This is inferior to the average-case efficiency of sequential search, which is linear (see Section 2.1).

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

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

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

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

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

   Note: For large values of n, we can simplify the inequality by eliminating the relatively insignificant term $\log_2 n$ from the inequality's denominator to obtain

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

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

7. a. The following algorithm will beat the brute-force comparisons of the telephone numbers on the bills and the checks: Using an efficient sorting algorithm, sort the bills and sort the checks. (In both cases, sorting has to be done with respect to their telephone numbers, say, in increasing order.) Then do a merging-like scan of the two sorted lists by comparing the telephone numbers b_i and c_j on the current bill and check, respectively: if $b_i < c_j$, add b_i to the list of unpaid telephone numbers and increment i; if $b_i > c_j$, increment j; if $b_i = c_j$, increment both i and j. Stop as soon as one of the two lists becomes empty and add all the remaining numbers on the bill list, if any, to the list of the unpaid ones.

   The time efficiency of this algorithm will be in

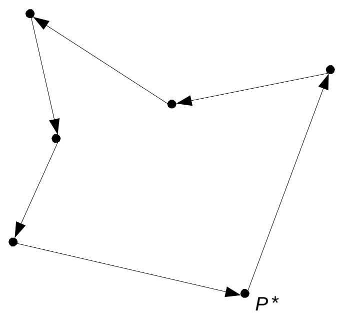
   $$O(n \log n) + O(m \log m) + O(n + m) \underset{n \geq m}{=} O(n \log n).$$

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

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

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

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



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

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

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

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

10. First, attach to every word in the file—as another field of the word's record, for example—its signature defined as the string of the word's letters in alphabetical order. (Obviously, words belong to the same anagram set if and only if they have the same signature.) Sort the records in alphabetical order of their signatures. Scan the list to identify contiguous subsequences, of length greater than one, of records with the same signature.

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

Exercises 6.2

1. Solve the following system by Gaussian elimination

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

2. a. Solve the system of the previous question by the *LU* decomposition method.

   b. From the standpoint of general algorithm design techniques, how would you classify the *LU* decomposition method?

3. Solve the system of Problem 1 by computing the inverse of its coefficient matrix and then multiplying it by the right-hand side vector.

4. Would it be correct to get the efficiency class of the elimination stage of Gaussian elimination as follows?

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

   Since $s_1(n) = \sum_{i=1}^{n-1}(n+2)n \in \Theta(n^3)$, $s_2(n) = \sum_{i=1}^{n-1}(2n+2)i \in \Theta(n^3)$, and $s_3(n) = \sum_{i=1}^{n-1} i^2 \in \Theta(n^3)$, $s_1(n) - s_2(n) + s_3(n) \in \Theta(n^3)$.

5. Write a pseudocode for the back-substitution stage of Gaussian elimination and show that its running time is in $\Theta(n^2)$.

6. Assuming that division of two real numbers takes three times longer than their multiplication, estimate how much faster *BetterGaussElimination* is than *GaussElimination.* (Of course, you should also assume that a compiler is not going to eliminate the inefficiency in *GaussElimination.*)

7. a. Give an example of a system of two linear equations in two unknowns that has a unique solution and solve it by Gaussian elimination.

   b. Give an example of a system of two linear equations in two unknowns that has no solution and apply Gaussian elimination to it.

   c. Give an example of a system of two linear equations in two unknowns that has infinitely many solutions and apply Gaussian elimination to it.

8. The ***Gauss-Jordan elimination*** method differs from Gaussian elimination in that the elements above the main diagonal of the coefficient matrix are made zero at the same time and by the same use of a pivot row as the elements below the main diagonal.

   a. Apply the Gauss-Jordan method to the system of Problem 1 of these exercises.

   b. What general design technique is this algorithm based on?

   c. In general, how many multiplications are made by this method while solving a system of n equations in n unknowns? How does this compare with the number of multiplications made by the Gaussian elimination method in both its elimination and its back-substitution stages?

9. A system $Ax = b$ of n linear equations in n unknowns has a unique solution if and only if $\det A \neq 0$. Is it a good idea to check this condition before applying Gaussian elimination to a system?

10. a. Apply Cramer's rule to solve the system of Problem 1 of these exercises.

    b. Estimate how many times longer it will take to solve a system of n linear equations in n unknowns by Cramer's rule than by Gaussian elimination. (Assume that all the determinants in Cramer's rule formulas are computed independently by Gaussian elimination.)

Hints to Exercises 6.2

1. Trace the algorithm as we did in solving another system in the section.

2. a. Use the Gaussian elimination results as explained in the text.

   b. It is one of the varieties of the transform-and-conquer technique. Which one?

3. You can either solve simultaneously the system with three right-hand side vectors representing the columns of the 3-by-3 identity matrix or by finding first the LU decomposition of the system's coefficient matrix.

4. Though the final answer is correct, its derivation contains an error you have to find.

5. The pseudocode of this algorithm is quite straightforward. If you are in doubt, see the section's example tracing the algorithm. The order of growth of the algorithm's running time can be estimated by following the standard plan for the analysis of nonrecursive algorithms.

6. Estimate the ratio of the algorithm running times, by using the approximate formulas for the number of divisions and the number of multiplications in both algorithms.

7. a. This is a "normal" case: one of the two equations should not be proportional to the other.

   b. The coefficients of one equation should be the same or proportional to the corresponding coefficients of the other equation while the right-hand sides should not.

   c. The two equations should be either the same or proportional to each other (including the right-hand sides).

8. a. Manipulate the matrix rows above a pivot row the same way the rows below the pivot row are changed.

   b. Are the Gauss-Jordan method and Gaussian elimination based on the same algorithm design technique or on different ones?

   c. Derive the formula for the number of multiplications in the Gauss-Jordan method in the same manner this was done for Gaussian elimination in Section 6.2.

9. How long will it take to compute the determinant compared to the time needed to solve the system?

10. a. Apply Cramer's rule to the system given.

    b. How many distinct determinants are there in the Cramer's rule formulas?

Solutions to Exercises 6.2

1. a. Solve the following system by Gaussian elimination

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

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

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

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

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

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

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

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

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

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

Its solution is

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

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

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

yields the following solution to the system given:

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

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

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

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

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

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

Solving the system with the first right-hand side column

$$\begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix}$$

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

$$\begin{bmatrix} -1 \\ \frac{5}{4} \\ \frac{3}{4} \end{bmatrix}.$$

Solving the system with the second right-hand side column

$$\begin{bmatrix} 0 \\ 1 \\ -2 \end{bmatrix}$$

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

$$\begin{bmatrix} 1 \\ -\frac{1}{2} \\ -\frac{1}{2} \end{bmatrix}.$$

Solving the system with the third right-hand side column

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

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

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

Thus, the inverse of the coefficient matrix is

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

which leads to the following solution to the original system

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

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

5. **Algorithm** *GaussBackSub*$(A[1..n, 1..n+1])$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\left[\begin{array}{rrrr} 1 & 1 & 1 & 2 \\ 2 & 1 & 1 & 3 \\ 1 & -1 & 3 & 8 \end{array}\right]
\begin{array}{l} \\ \text{row } 2 - \frac{2}{1}\text{row } 1 \\ \text{row } 3 - \frac{1}{1}\text{row } 1 \end{array}
\left[\begin{array}{rrrr} 1 & 1 & 1 & 2 \\ 0 & -1 & -1 & -1 \\ 0 & -2 & 2 & 6 \end{array}\right]
\begin{array}{l} \text{row } 1 - \frac{1}{-1}\text{row } 2 \\ \\ \text{row } 3 - \frac{-2}{-1}\text{row } 2 \end{array}$$

$$\left[\begin{array}{rrrr} 1 & 0 & 0 & 1 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 4 & 8 \end{array}\right]
\begin{array}{l} \text{row } 1 - \frac{0}{4}\text{row } 3 \\ \text{row } 2 - \frac{-1}{4}\text{row } 3 \end{array}
\left[\begin{array}{rrrr} 1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & 4 & 8 \end{array}\right]$$

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

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

b. The Gauss-Jordan method is also an example of an algorithm based

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

c. Here is a basic pseudocode for the Gauss-Jordan elimination:

```
**Algorithm** $GaussJordan(A[1..n, 1..n], b[1..n])$
//Applies Gaussian-Jordan elimination to matrix $A$ of a system's
// coefficients, augmented with vector $b$ of the system's right-hand sides
//Input: Matrix $A[1..n, 1, ..n]$ and column-vector $b[1..n]$
//Output: An equivalent diagonal matrix in place of $A$ with the
//corresponding right-hand side values in its $(n+1)$st column
for $i \leftarrow 1$ to $n$ do $A[i, n+1] \leftarrow b[i]$   //augment the matrix
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        if $j \neq i$
           $temp \leftarrow A[j, i]\ /\ A[i, i]$      //assumes $A[i, i] \neq 0$
           for $k \leftarrow i$ to $n+1$ do
               $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$
```

The number of multiplications made by the above algorithm can be computed as follows:

$$\begin{aligned} M(n) &= \sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n}\sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n}(n+1-i+1) = \sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n}(n+2-i) \\ &= \sum_{i=1}^{n}(n+2-i)(n-1) = (n-1)\sum_{i=1}^{n}(n+2-i) \\ &= (n-1)[(n+1)+n+...+2] = (n-1)[(n+1)+n+...+1-1] \\ &= (n-1)[\frac{(n+1)(n+2)}{2}-1] = \frac{(n-1)n(n+3)}{2} \approx \frac{1}{2}n^3. \end{aligned}$$

The total number of multiplications made in both elimination and backward substitution stages of the Gaussian elimination method is equal to

$$\frac{n(n-1)(2n+5)}{6} + \frac{(n-1)n}{2} = \frac{(n-1)n(n+4)}{3} \approx \frac{1}{3}n^3,$$

which is about 1.5 smaller than in the Gauss-Jordan method.

Note: The Gauss-Jordan method has an important advantage over Gaussian elimination: being more uniform, it is more suitable for efficient implementation on a parallel computer.

9. Since the time needed for computing the determinant of the system's coefficient matrix is about the same as the time needed for solving the system (or detecting that the system does not have a unique solution) by Gaussian elimination, computing the determinant of the coefficient matrix to check whether it is equal to zero is not a good idea from the algorithmic point of view.

10. a. Solve the following system by Cramer's rule

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

$$\det A = \det \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & -1 & 3 \end{bmatrix} = 1 \cdot 1 \cdot 3 + 1 \cdot 1 \cdot 1 + 2 \cdot (-1) \cdot 1 - 1 \cdot 1 \cdot 1 - 2 \cdot 1 \cdot 3 - (-1) \cdot 1 \cdot 1 = -4,$$

$$\det A_1 = \det \begin{bmatrix} 2 & 1 & 1 \\ 3 & 1 & 1 \\ 8 & -1 & 3 \end{bmatrix} = 2 \cdot 1 \cdot 3 + 1 \cdot 1 \cdot 8 + 3 \cdot (-1) \cdot 1 - 8 \cdot 1 \cdot 1 - 3 \cdot 1 \cdot 3 - (-1) \cdot 1 \cdot 2 = -4,$$

$$\det A_2 = \det \begin{bmatrix} 1 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 8 & 3 \end{bmatrix} = 1 \cdot 3 \cdot 3 + 2 \cdot 1 \cdot 1 + 2 \cdot 8 \cdot 1 - 1 \cdot 3 \cdot 1 - 2 \cdot 2 \cdot 3 - 8 \cdot 1 \cdot 1 = 4,$$

$$\det A_3 = \det \begin{bmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 1 & -1 & 8 \end{bmatrix} = 1 \cdot 1 \cdot 8 + 1 \cdot 3 \cdot 1 + 2 \cdot (-1) \cdot 2 - 1 \cdot 1 \cdot 2 - 2 \cdot 1 \cdot 8 - (-1) \cdot 3 \cdot 1 = -8.$$

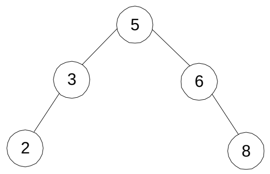
Hence,

$$x_1 = \frac{\det A_1}{\det A} = \frac{-4}{-4} = 1, \quad x_2 = \frac{\det A_2}{\det A} = \frac{4}{-4} = -1, \quad x_3 = \frac{\det A_3}{\det A} = \frac{-8}{-4} = 2.$$

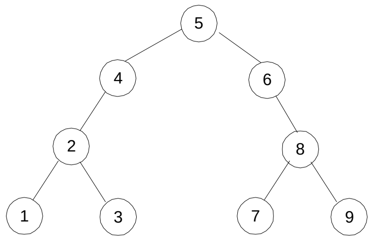
b. Cramer's rule requires computing $n + 1$ distinct determinants. If each of them is computed by applying Gaussian elimination, it will take about $n+1$ times longer than solving the system by Gaussian elimination. (The time for the backward substitution stage was not accounted for in the preceding argument because of its quadratic efficiency vs. cubic efficiency of the triangulation stage.)

Exercises 6.3

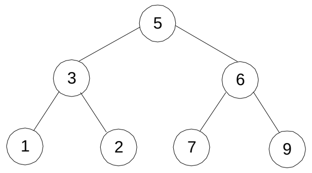
1. Which of the following binary trees are AVL trees?



(a)



(b)



(c)

2. a. For $n = 1, 2, 3, 4$, and 5, draw all the binary trees with n nodes that satisfy the balance requirement of AVL trees.

   b. Draw a binary tree of height 4 that can be an AVL tree and has the smallest number of nodes among all such trees.

3. Draw diagrams of the single *L*-rotation and of the double *RL*-rotation in their general form.

4. For each of the following lists, construct an AVL tree by inserting their elements successively, starting with the empty tree:

   a. 1, 2, 3, 4, 5, 6

   b. 6, 5, 4, 3, 2, 1

   c. 3, 6, 5, 1, 2, 4

5. a. For an AVL tree containing real numbers, design an algorithm for computing the range (i.e., the difference between the largest and smallest numbers) in the tree and determine its worst-case efficiency.

   b. True or false: The smallest and the largest keys in an AVL tree can always be found on either the last level or the next-to-last level.

6. Write a program for constructing an AVL tree for a given list of n distinct integers.

7. a. Construct a 2-3 tree for the list C, O, M, P, U, T, I, N, G. (Use the alphabetical order of the letters and insert them successively starting with the empty tree.)

   b. Assuming that the probabilities of searching for each of the keys (i.e., the letters) are the same, find the largest number and the average number of key comparisons for successful searches in this tree.

8. Let T_B and $T_{2\text{-}3}$ be, respectively, a classical binary search tree and a 2-3 tree constructed for the same list of keys inserted in the corresponding trees in the same order. True or false: Searching for the same key in $T_{2\text{-}3}$ always takes fewer or the same number of key comparisons as searching in T_B.

9. For a 2-3 tree containing real numbers, design an algorithm for computing the range (i.e., the difference between the largest and smallest numbers) in the tree and determine its worst-case efficiency.

10. Write a program for constructing a 2-3 tree for a given list of n integers.

Hints to Exercises 6.3

1. Use the definition of AVL trees. Do not forget that an AVL tree is a special case of a binary search tree.

2. For both questions, it is easier to construct the required trees bottom up, i.e., for smaller values of n first.

3. The single L-rotation and the double RL-rotation are the mirror images of the single R-rotation and the double LR-rotation, whose diagrams can be found in this section.

4. Insert the keys one after another doing appropriate rotations the way it was done in the section's example.

5. a. An efficient algorithm immediately follows from the definition of the binary search tree of which the AVL tree is a special case.

   b. The correct answer is opposite to the one that immediately comes to mind.

6. n/a

7. a. Trace the algorithm for the input given (see Figure 6.8) for an example.

   b. Keep in mind that the number of key comparisons made in searching for a key in a 2-3 tree depends not only on its node's depth but also whether the key is the first or second one in the node.

8. False; find a simple counterexample.

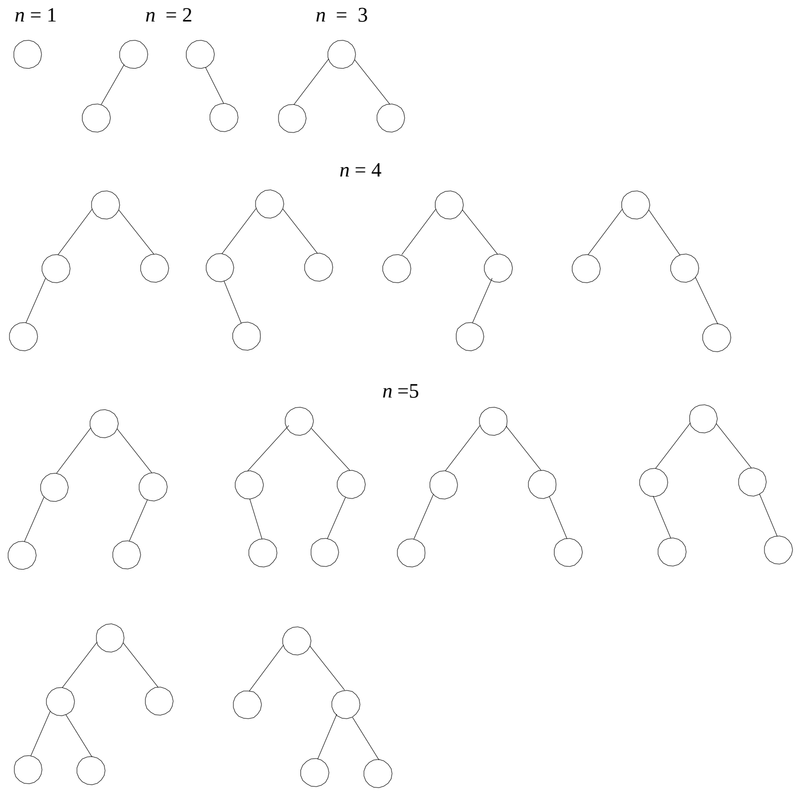
9. Where will the smallest and largest keys be located?

10. n/a

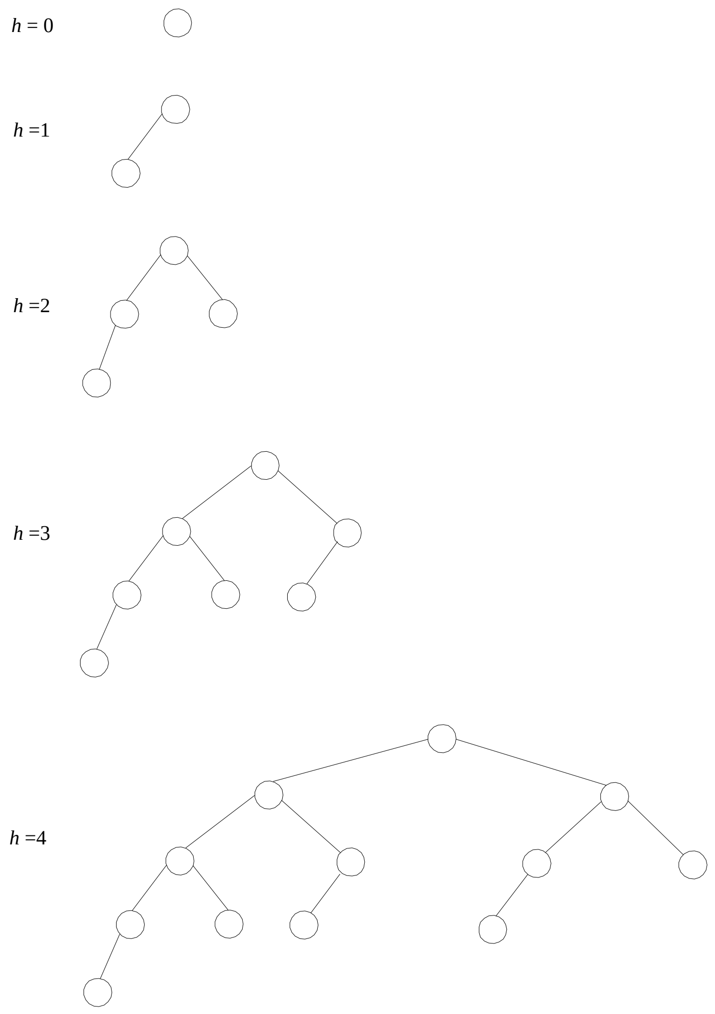
Solutions to Exercises 6.3

1. Only (a) is an AVL tree; (b) has a node (in fact, there are two of them: 4 and 6) that violates the balance requirement; (c) is not a binary search tree because 7 is in the left subtree of 6.

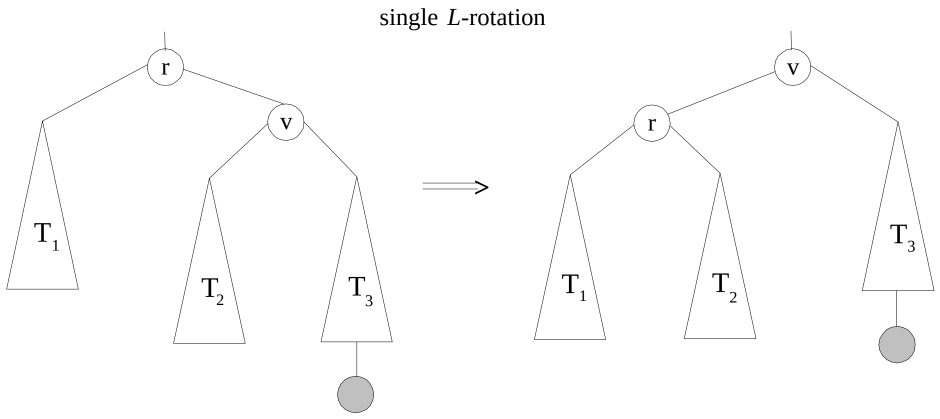
2. a . Here are all the binary trees with n nodes (for $n = 1, 2, 3, 4$, and 5) that satisfy the balance requirement of AVL trees.



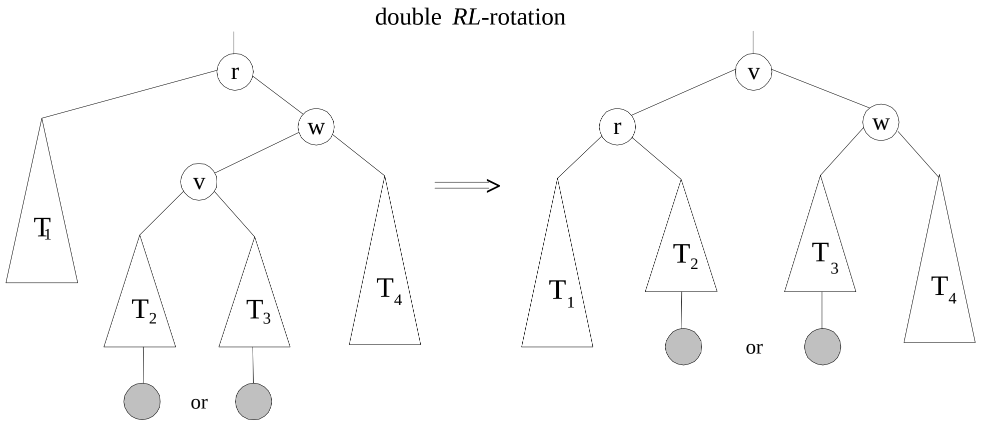
b. A minimal AVL tree (i.e., a tree with the smallest number of nodes) of height 4 must have its left and right subtrees being minimal AVL trees of heights 3 and 2. Following the same recursive logic further, we will find, as one of the possible examples, the following tree with 12 nodes built bottom up:



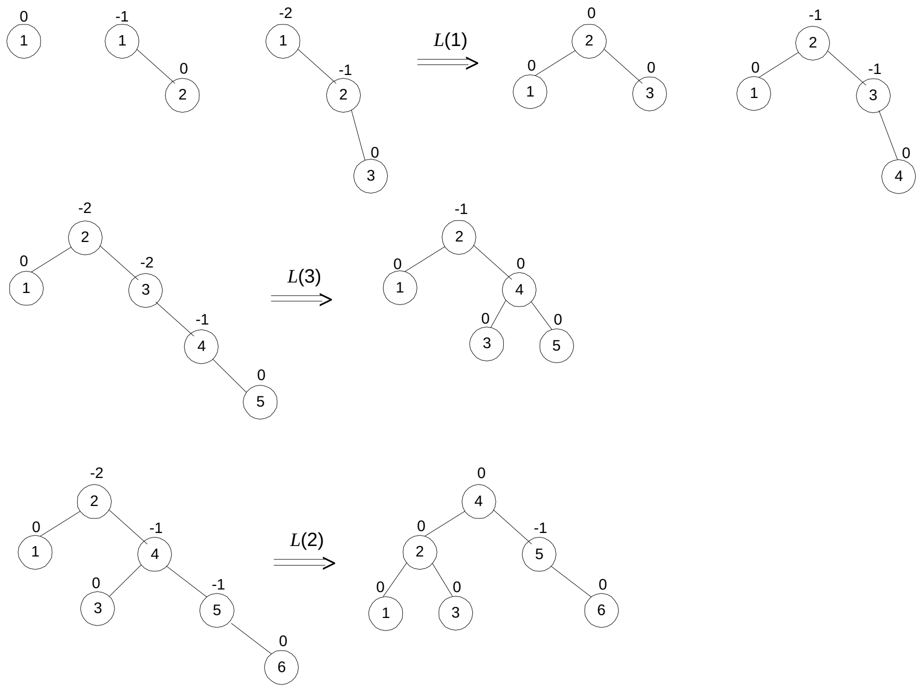
3. a. Here is a diagram of the single L-rotation in its general form



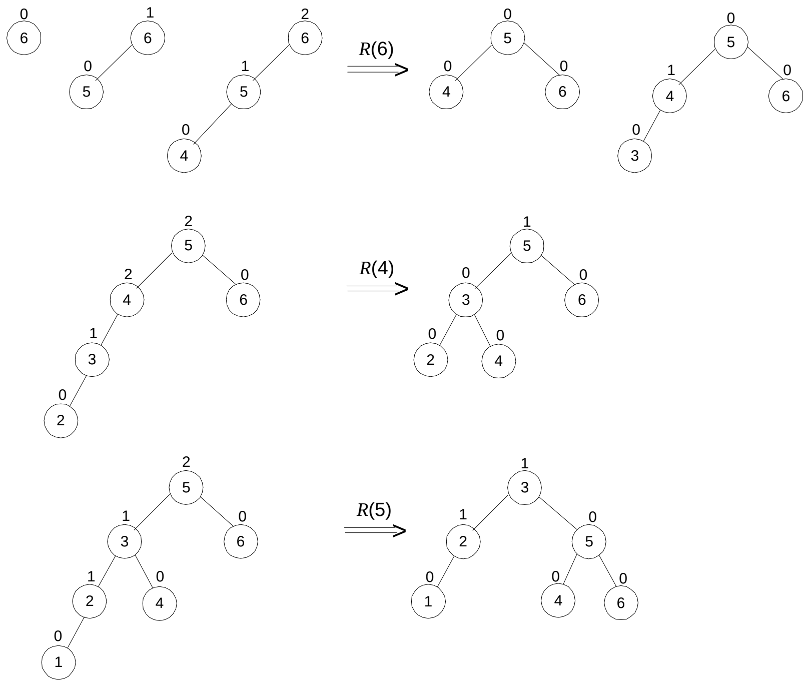
b. Here is a diagram of the double RL-rotation in its general form



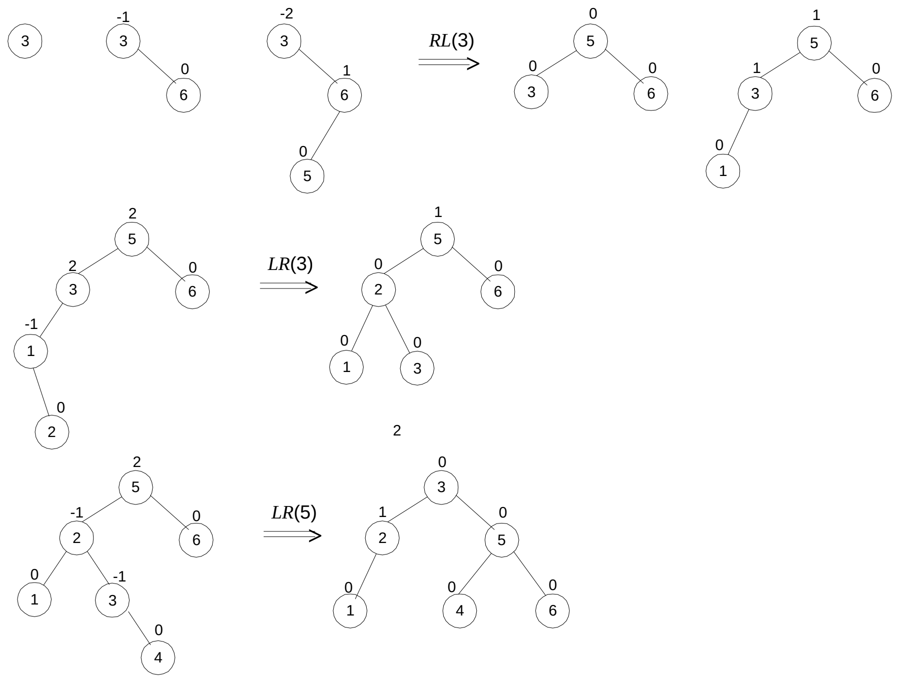
4. a. Construct an AVL tree for the list 1, 2, 3, 4, 5, 6.



b. Construct an AVL tree for the list 6, 5, 4, 3, 2, 1.



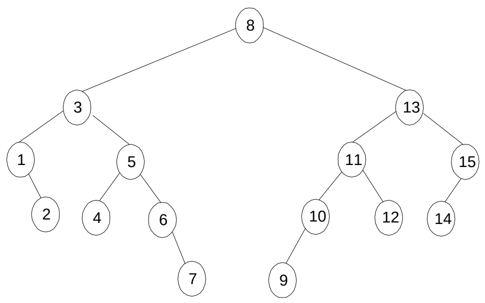
c. Construct an AVL tree for the list 3, 6, 5, 1, 2, 4.



5. a. The simple and efficient algorithm is based on the fact that the smallest and largest keys in a binary search tree are in the leftmost and rightmost nodes of the tree, respectively. Therefore, the smallest key can be found by starting at the root and following the chain of left pointers until a node with the null left pointer is reached: its key is the smallest one in the tree. Similarly, the largest key can be obtained by following the chain of the right pointers. Finally, the range is computed as the difference between the largest and smallest keys found.

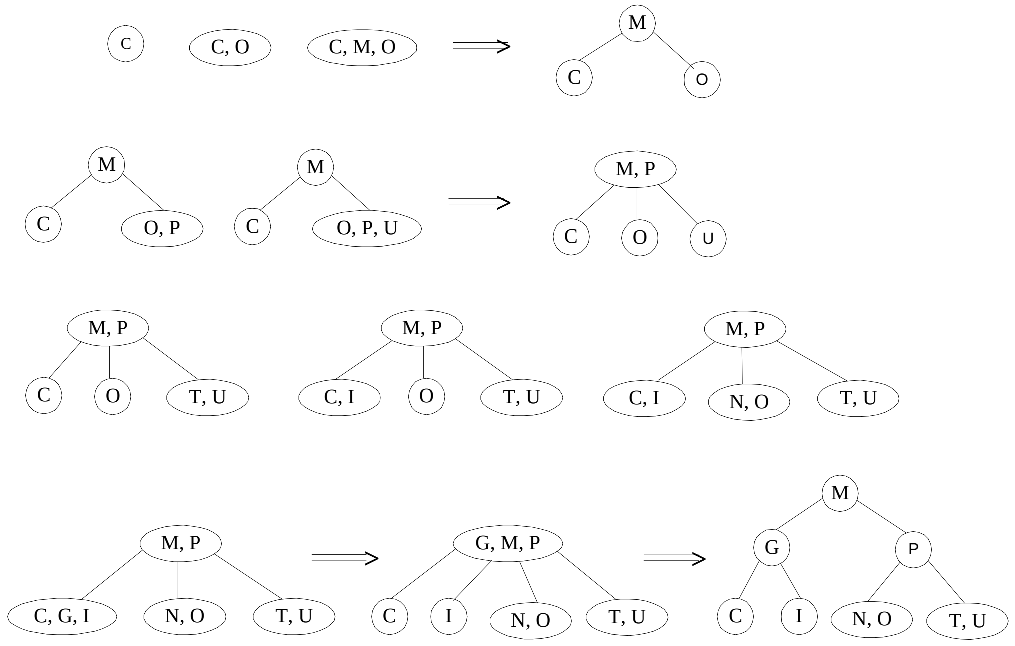
   In the worst case, the leftmost and rightmost nodes will be on the last level of the tree. Hence, the worst-case efficiency will be in $\Theta(\log n) + \Theta(\log n) + \Theta(1) = \Theta(\log n)$.

   b. False. Here is a counterexample in which neither the smallest nor the largest keys are on the last, or the next-to-last, level:



6. n/a

7. a. Construct a 2-3 tree for the list C, O, M, P, U, T, I, N, G.



b. The largest number of key comparisons in a successful search will be in the searches for O and U; it will be equal to 4. The average number of key comparisons will be given by the following expression

$$\frac{1}{9}C(\text{C}) + \frac{1}{9}C(\text{O}) + \frac{1}{9}C(\text{M}) + \frac{1}{9}C(\text{P}) + \frac{1}{9}C(\text{U}) + \frac{1}{9}C(\text{T}) + \frac{1}{9}C(\text{I}) + \frac{1}{9}C(\text{N}) + \frac{1}{9}C(\text{G})$$
$$= \frac{1}{9} \cdot 3 + \frac{1}{9} \cdot 4 + \frac{1}{9} \cdot 1 + \frac{1}{9} \cdot 2 + \frac{1}{9} \cdot 4 + \frac{1}{9} \cdot 3 + \frac{1}{9} \cdot 3 + \frac{1}{9} \cdot 3 + \frac{1}{9} \cdot 2 = \frac{25}{9}.$$

8. False. Consider the list B, A. Searching for B in the binary search tree will require 1 comparison while searching for B in the 2-3 tree will require 2 comparisons.

9. The smallest and largest keys will be the first key in the leftmost leaf and the second key in the rightmost leaf, respectively. So searching for them will require following the chain of the leftmost pointers from the root to the leaf and of the rightmost pointers from the root to the leaf. Since the height of a 2-3 tree is always in $\Theta(\log n)$, the time efficiency of the algorithm for all cases will be in $\Theta(\log n) + \Theta(\log n) + \Theta(1) = \Theta(\log n)$.

10. n/a

Exercises 6.4

1. a. Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm.

   b. Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by successive key insertions (top-down algorithm).

   c. Is it always true that the bottom-up and top-down algorithms yield the same heap for the same input?

2. Outline an algorithm for checking whether an array $H[1..n]$ is a heap and determine its time efficiency.

3. a. Find the smallest and the largest number of keys that a heap of height h can contain.

   b.▷ Prove that the height of a heap with n nodes is equal to $\lfloor \log_2 n \rfloor$.

4. ▷ Prove the following equation used in Section 6.4

$$\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)) \quad \text{where } n = 2^{h+1} - 1.$$

5. a. Design an efficient algorithm for finding and deleting an element of the smallest value in a heap and determine its time efficiency.

   b. Design an efficient algorithm for finding and deleting an element of a given value v in a given heap H and determine its time efficiency

6. Sort the following lists by heapsort by using the array representation of heaps:

   a. 1, 2, 3, 4, 5 (in increasing order)

   b. 5, 4, 3, 2, 1 (in increasing order)

   c. S, O, R, T, I, N, G (in alphabetical order)

7. Is heapsort a stable sorting algorithm?

8. What variety of the transform-and-conquer technique does heapsort represent?

9. Implement three advanced sorting algorithms—mergesort, quicksort, and heapsort—in the language of your choice and investigate their performance on arrays of sizes $n = 10^2$, 10^3, 10^4, 10^5, and 10^6. For each of these sizes

consider:

a. randomly generated files of integers in the range $[1..n]$.

b. increasing files of integers $1, 2, ..., n$.

c. decreasing files of integers n, $n-1, ..., 1$.

10. Imagine a handful of uncooked spaghetti, individual rods whose lengths represent numbers that need to be sorted.

    a. Outline a "spaghetti sort"—a sorting algorithm that takes advantage of this unorthodox representation.

    b. What does this example of computer science folklore (see [Dew93]) have to do with the topic of this chapter in general and heapsort in particular?

Hints to Exercises 6.4

1. (a)-(b) Trace the two algorithms outlined in the text on the inputs given..

   c. A mathematical fact cannot be established by checking its validity on a single example.

2. For a heap represented by an array, only the parental dominance requirement needs to be checked.

3. a. What structure does a complete tree of height h with the largest number of nodes have? What about a complete tree with the smallest number of nodes?

   b. Use the results established in part (a).

4. First, express the right-hand side as a function of h. Then prove the obtained equality by either using the formula for the sum $\sum i2^i$ given in Appendix A or by mathematical induction on h.

5. a. Where in a heap should we look for its smallest element?

   b. Deleting an arbitrary element of a heap can be done by generalizing the algorithm for deleting its root.

6. Trace the algorithm on the inputs given (see Figure 6.14 for an example).

7. As a rule, sorting algorithms that can exchange far apart elements are not stable.

8. One can claim that the answers are different for the two principal representations of heaps.

9. n/a

10. Pick the spaghetti rods up in a bundle and place them end-down (i.e., vertically) onto a tabletop.

Solutions to Exercises 6.4

1. a. Constructing a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm (a root of a subtree being heapified is shown in bold):

1	8	**6**	5	3	7	4	$\Rightarrow$	1	8	7	5	3	6	4
1	**8**	7	5	3	6	4								
1	8	7	5	3	6	4	$\Rightarrow$	8	5	7	1	3	6	4

b. Constructing a heap for the list 1, 8, 6, 5, 3, 7, 4 by the top-down algorithm (a new element being inserted into a heap is shown in bold):

1													
1	**8**						$\Rightarrow$	8	1				
8	1	**6**											
8	1	6	**5**				$\Rightarrow$	8	5	6	1		
8	5	6	1	**3**									
8	5	6	1	3	**7**		$\Rightarrow$	8	5	7	1	3	6
8	5	7	1	3	6	**4**							

c. False. Although for the input to questions (a) and (b) the heaps constructed are the same, in general, it may not be the case. For example, for the input 1, 2, 3, the bottom-up algorithm yields 3, 2, 1 while the top-down algorithm yields 3, 1, 2.

2. For $i = 1, 2, ..., \lfloor n/2 \rfloor$, check whether

$$H[i] \geq \max\{H[2i],\ H[2i+1]\}.$$

(Of course, if $2i + 1 > n$, just $H[i] \geq H[2i]$ needs to be satisfied.) If the inequality doesn't hold for some i, stop — the array is not a heap; if it holds for every $i = 1, 2, ..., \lfloor n/2 \rfloor$, it is a heap.

Since the algorithm makes up to $2\lfloor n/2 \rfloor$ key comparisons, its time efficiency is in $O(n)$.

3. a. A complete binary tree of height h with the smallest number of nodes has the maximal number of nodes on levels 0 through $h - 1$ and 1 node on the last level. The total number of nodes in such a tree is

$$n_{\min}(h) = \sum_{i=0}^{h-1} 2^i + 1 = (2^h - 1) + 1 = 2^h.$$

A complete binary tree of height h with the largest number of nodes has the maximal number of nodes on levels 0 through h. The total number of nodes in such a tree is

$$n_{\max}(h) = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

b. The results established in part (a) imply that for any heap with n nodes and height h

$$2^h \leq n < 2^{h+1}.$$

Taking logarithms to base 2 of these inequalities yields

$$h \leq \log_2 n < h + 1.$$

This means that h is the largest integer not exceeding $\log_2 n$, i.e., $h = \lfloor \log_2 n \rfloor$.

4. We are asked to prove that $\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1))$ where $n = 2^{h+1} - 1$.

   For $n = 2^{h+1} - 1$, the right-hand side of the equality in question becomes

   $$2(2^{h+1} - 1 - \log_2(2^{h+1} - 1 + 1)) = 2(2^{h+1} - 1 - (h+1)) = 2(2^{h+1} - h - 2).$$

   Using the formula $\sum_{i=1}^{h-1} i2^i = (h-2)2^h + 2$ (see Appendix A), the left-hand side can be simplified as follows:

   $$\begin{aligned}\sum_{i=0}^{h-1} 2(h-i)2^i &= 2\sum_{i=0}^{h-1}(h-i)2^i = 2[\sum_{i=0}^{h-1} h2^i - \sum_{i=0}^{h-1} i2^i] \\ &= 2[h(2^h - 1) - (h-2)2^h - 2] \\ &= 2(h2^h - h - h2^h + 2^{h+1} - 2) = 2(2^{h+1} - h - 2).\end{aligned}$$

5. a. The parental dominance requirement implies that we can always find the smallest element of a heap $H[1..n]$ among its leaf positions, i.e., among $H[\lfloor n/2 \rfloor + 1], ..H[n]$. (You can easily prove this assertion by contradiction.) Therefore we can find the smallest element by simply scanning sequentially the second half of the array H. Deleting this element can be done by exchanging the found element with the last element $H[n]$, decreasing the heap's size by one, and then, if necessary, sifting up the former $H[n]$ from its new position until it is not larger than its parent.

   The time efficiency of searching for the smallest element in the second half of the array is in $\Theta(n)$; the time efficiency of deleting it after it has been found is in $\Theta(1) + \Theta(1) + O(\log n) = O(\log n)$.

b. Searching for v by sequential search in $H[1..n]$ takes care of the searching part of the question. Assuming that the first matching element is found in position i, the deletion of $H[i]$ can be done with the following three-step procedure (which is similar to the ones used for deleting the root and the smallest element): First, exchange $H[i]$ with $H[n]$; second, decrease n by 1; third, heapify the structure by sifting the former $H[n]$ either up or down depending on whether it is larger than its new parent or smaller than the larger of its new children, respectively.

The time efficiency of searching for an element of a given value is in $O(n)$; the time efficiency of deleting it after it has been found is in $\Theta(1)+\Theta(1)+O(\log n) = O(\log n)$.

6. a. Sort 1, 2, 3, 4, 5 by heapsort

Heap Construction

1	**2**	3	4	5
1	5	3	4	2
1	5	3	4	2
5	4	3	1	2

Maximum Deletions

5	4	3	1	2
2	4	3	1	| **5**
4	2	3	1	
1	2	3	| **4**	
3	2	1		
1	2	| **3**		
2	1			
1	| **2**			
1				

b. Sort 5, 4, 3, 2, 1 (in increasing order) by heapsort

Heap Construction

5	**4**	3	2	1
5	4	3	2	1

Maximum Deletions

5	4	3	2	1
1	4	3	2	| **5**
4	2	3	1	
1	2	3	| **4**	
3	2	1		
1	2	| **3**		
2	1			
1	| **2**			
1				

c. Sort S, O, R, T, I, N, G (in alphabetic order) by heapsort

Heap Construction

1	2	3	4	5	6	7
S	O	**R**	T	I	N	G
S	**O**	R	T	I	N	G
S	T	R	O	I	N	G
S	T	R	O	I	N	G
T	S	R	O	I	N	G

Maximum Deletions

1	2	3	4	5	6	7
T	S	R	O	I	N	G
G	S	R	O	I	N |	**T**
S	O	R	G	I	N	
N	O	R	G	I |	**S**	
R	O	N	G	I		
I	O	N	G |	**R**		
O	I	N	G			
G	I	N |	**O**			
N	I	G				
G	I |	**N**				
I	G					
G |	**I**					
G						

7. Heapsort is not stable. For example, it sorts $1'$, $1''$ into $1''$, $1'$.

8. If the heap is thought of as a tree, heapsort should be considered a representation-change algorithm; if the heap is thought of as an array with a special property, heapsort should be considered an instance-simplification algorithm.

9. n/a

10. a. After the bunch of spaghetti rods is put in a vertical position on a tabletop, take the tallest rod among the remaining ones out until no more rods are left. This will sort the rods in decreasing order of their lengths.

    b. The method shares with heapsort its principal idea: represent the items to be sorted in a way that makes finding and deleting the largest item a simple task. From a more general perspective, the spaghetti sort is an example, albeit a rather exotic one, of a representation-change algorithm.

Exercises 6.5

1. Consider the following brute-force algorithm for evaluating a polynomial.

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

   Find the total number of multiplications and the number of additions made by this algorithm.

2. Write a pseudocode for the brute-force polynomial evaluation that stems form substituting a given value of the variable into the polynomial's formula and evaluating it from the lowest term to the highest one. Determine the number of multiplications and the number of additions made by this algorithm.

3. a. Estimate how much faster Horner's rule is compared to the "lowest to highest term" brute-force algorithm of Problem 2 if (i) the time of one multiplication is significantly larger than the time of one addition; (ii) the time of one multiplication is about the same as the time of one addition.

   b. Is Horner's rule more time efficient at the expense of being less space efficient than the brute-force algorithm?

4. a. Apply Horner's rule to evaluate the polynomial

$$p(x) = 3x^4 - x^3 + 2x + 5 \text{ at } x = -2.$$

   b. Use the results of Horner's rule application to find the quotient and remainder of the division of $p(x)$ by $x + 2$.

5. Compare the number of multiplications and additions/subtractions needed by the "long division" of a polynomial $p(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$ by $x - c$ where c is some constant with the number of these operations in the "synthetic division."

6. a. Apply the left-to-right binary exponentiation algorithm to compute a^{17}.

   b. Is it possible to extend the left-to-right binary exponentiation algorithm to work for every nonnegative integer exponent?

7. Apply the right-to-left binary exponentiation algorithm to compute a^{17}.

8. Design a nonrecursive algorithm for computing a^n that mimics the right-to-left binary exponentiation but does not explicitly use the binary representation of n.

9. Is it a good idea to use a general-purpose polynomial evaluation algorithm such as Horner's rule to evaluate the polynomial $p(x) = x^n + x^{n-1} + ... + x + 1$?

10. According to the corollary of the Fundamental Theorem of Algebra, every polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_0$$

can be represented in the form

$$p(x) = a_n (x - x_1)(x - x_2)...(x - x_n)$$

where $x_1, ..., x_n$ are the roots of the polynomial (generally, complex and not necessarily distinct). Discuss which of the two representations is more convenient for each of the following operations:

   a. Polynomial evaluation at a given point

   b. Addition of two polynomials

   c. Multiplication of two polynomials

Hints to Exercises 6.5

1. Set up a sum and simplify it by using the standard formulas and rules for sum manipulation. Do not forget to include the multiplications outside the inner loop.

2. Take advantage of the fact that the value of x^i can be easily computed from the previously computed x^{i-1}.

3. a. Use the formulas for the number of multiplications (and additions) for both algorithms.

   b. Does Horner's rule use any extra memory?

4. Apply Horner's rule to the instance given the same way it is applied to another one in the section.

5. If you implement the algorithm for long division by $x - c$ efficiently, the answer might surprise you.

6. a. Trace the left-to-right binary exponentiation algorithm on the instance given the same way it is done for another instance in the section.

   b. The answer is yes: the algorithm can be extended to work for the zero exponent as well. How?

7. Trace the right-to-left binary exponentiation algorithm on the instance given the same way it is done for another instance in the section.

8. Compute and use the binary digits of n "on the fly".

9. Use a formula for the sum of the terms of this special kind of a polynomial.

10. Compare the number of operations needed to implement the task in question.

Solutions to Exercises 6.5

1. The total number of multiplications made by the algorithm can be computed as follows:

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

   The number of additions is obtained as

$$A(n) = \sum_{i=0}^{n} 1 = n+1.$$

2. **Algorithm** *BetterBruteForcePolynomialEvaluation*($P[0..n]$, x)

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

   The number of multiplications made by this algorithm is

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

The number of additions is

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

3. a. If only multiplications need to be taken into account, Horner's rule will be about twice as fast because it makes just n multiplications vs. $2n$ multiplications required by the other algorithm. If one addition takes about the same amount of time as one multiplication, then Horner's rule will be about $(2n+n)/(n+n) = 1.5$ times faster.

   b. The answer is no, because Horner's rule doesn't use any extra memory.

4. a. Evaluate $p(x) = 3x^4 - x^3 + 2x + 5$ at $x = -2$.

coefficients	3	-1	0	2	5
$x = -2$	3	(-2)·3+(-1)= -7	(-2)·(-7)+0=14	(-2)·14+2= -26	(-2)·(-26)+5=57

b. The quotient and the remainder of the division of $3x^4 - x^3 + 2x + 5$ by $x + 2$ are $3x^3 - 7x^2 + 14x - 26$ and 57, respectively.

5. The long division by $x - c$ is done as illustrated below

$$\begin{array}{r|l} & a_nx^{n-1} + ... \\ \cline{2-2} x - c & a_nx^n + a_{n-1}x^{n-1} \qquad +... + a_1x + a_0 \\ & {}^{-}a_nx^n - ca_nx^{n-1} \\ \cline{2-2} & \qquad (ca_n{+}a_{n-1})x^{n-1} + ... + a_1x + a_0 \end{array}$$

This clearly demonstrates that the first iteration—the one needed to get rid of the leading term a_nx^n—requires one multiplication (to get ca_n) and one addition (to add a_{n-1}). After this iteration is repeated $n - 1$ more times, the total number of multiplications and the total number of additions will be n each—exactly the same number of operations needed by Horner's rule. (In fact, it does exactly the same computations as Horner's algorithm would do in computing the value of the polynomial at $x = c$.) Thus, the long division, though much more cumbersome than the synthetic division for hand-and-pencil computations, is actually not less time efficient from the algorithmic point of view.

6. a. Compute a^{17} by the left-to-right binary exponentiation algorithm. Here, $n = 17 = 10001_2$. So, we have the following table filled left-to-right:

binary digits of n	1	0	0	0	1
product accumulator	a	a^2	$(a^2)^2 = a^4$	$(a^4)^2 = a^8$	$(a^8)^2 \cdot a = a^{17}$

b. Algorithm *LeftRightBinaryExponentiation* will work correctly for $n = 0$ if the variable *product* is initialized to 1 (instead of a) and the loop starts with I (instead of $I - 1$).

7. Compute a^{17} by the right-to-left binary exponentiation algorithm. Here, $n = 17 = 10001_2$. So, we have the following table filled right-to-left:

1	0	0	0	1	binary digits of n
a^{16}	a^8	a^4	a^2	a	terms a^{2^i}
$a \cdot a^{16} = a^{17}$				a	product accumulator

8. **Algorithm** *ImplicitBinaryExponentiation*(a, n)

```
//Computes $a^n$ by the implicit right-to-left binary exponentiation
//Input: A number $a$ and a nonnegative integer $n$
//Output: The value of $a^n$
$product \leftarrow 1$;    $term \leftarrow a$
while $n \neq 0$ do
    $b \leftarrow n \bmod 2$;    $n \leftarrow \lfloor n/2 \rfloor$
    if $b = 1$
        $product \leftarrow product * term$
    $term \leftarrow term * term$
return $product$
```

9. Since the polynomial's terms form a geometric series,

$$p(x) = x^n + x^{n-1} + ... + x + 1 = \begin{cases} \frac{x^{n+1}-1}{x-1} & \text{if } x \neq 1 \\ n+1 & \text{if } x = 1 \end{cases}$$

Its value can be computed faster than with Horner's rule by computing the right-hand side formula with an efficient exponentiation algorithm for evaluating x^{n+1}.

10. a. With Horner's rule, we can evaluate a polynomial in its coefficient form with n multiplications and n additions. The direct substitution of the x value in the factorized form will require the same number of operations, though these may be operations on complex numbers even if the polynomial has real coefficients.

b. Addition of two polynomials is incomparably simpler for polynomials in their coefficient forms because knowing the roots of polynomials $p(x)$ and $q(x)$ helps little in deducing the root values of their sum $p(x) + q(x)$.

c. Multiplication of two polynomials is trivial when they are represented in their factorized form. Indeed, if

$$p(x) = a'_n(x - x'_1)...(x - x'_n) \text{ and } q(x) = a''_m(x - x''_1)...(x - x''_m),$$

then

$$p(x)q(x) = a'_n a''_n (x - x'_1)...(x - x'_n)(x - x''_1)...(x - x''_m).$$

To multiply two polynomials in their coefficient form, we need to multiply out

$$p(x)q(x) = (a'_n x^n + ... + a'_0)(a''_m x^m + ... + a''_0)$$

and collect similar terms to get the product represented in this form as well.

Exercises 6.6

1. a. Prove the equality

$$\text{lcm}(m, n) = \frac{m \cdot n}{\gcd(m, n)}$$

   that underlies the algorithm for computing $\text{lcm}(m, n)$.

   b. Euclid's algorithm is known to be in $O(\log n)$. If it is the algorithm that is used for computing $\gcd(m, n)$, what is the efficiency of the algorithm for computing $\text{lcm}(m, n)$?

2. You are given a list of numbers for which you need to construct a min-heap. (A min-heap is a complete binary tree in which every key is less than or equal to the keys in its children.) How would you use an algorithm for constructing a max-heap (a heap as defined in Section 6.4) to construct a min-heap?

3. Prove that the number of different paths of length $k > 0$ from the ith vertex to the jth vertex of a graph (undirected or directed) equals the (i, j)th element of A^k where A is the adjacency matrix of the graph.

4. a. Design an algorithm with a time efficiency better than cubic for checking whether a graph with n vertices contains a cycle of length 3 [Man89].

   b. Consider the following algorithm for the same problem. Starting at an arbitrary vertex, traverse the graph by depth-first search and check whether its depth-first search forest has a vertex with a back edge leading to its grandparent. If it does, the graph contains a triangle; if it does not, the graph does not contain a triangle as its subgraph. Is this algorithm correct?

5. Given $n > 3$ points $P_1 = (x_1, y_1), ..., P_n = (x_n, y_n)$ in the coordinate plane, design an algorithm to check whether all the points lie within a triangle with its vertices at three of the points given. (You can either design an algorithm from scratch or reduce the problem to another one with a known algorithm.)

6. Consider the problem of finding, for a given positive integer n, the pair of integers whose sum is n and whose product is as large as possible. Design an efficient algorithm for this problem and indicate its efficiency class.

7. The assignment problem introduced in Section 3.4 can be stated as follows. There are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the ith person is assigned to the jth job is a known quantity $C[i, j]$ for each pair $i, j = 1, ..., n$. The problem is to assign the people to the jobs to minimize

the total cost of the assignment. Express the assignment problem as a 0–1 linear programming problem.

8. Solve the instance of the linear programming problem given in Section 6.6

$$\begin{aligned} \text{maximize} \quad & 0.10x + 0.07y + 0.03z \\ \text{subject to} \quad & x + y + z = 100 \\ & x \le \tfrac{1}{3}y \\ & z \ge 0.25(x + y) \\ & x \ge 0, \;\; y \ge 0, \;\; z \ge 0. \end{aligned}$$

9. The graph-coloring problem is usually stated as the vertex-coloring problem: assign the smallest number of colors to vertices of a given graph so that no two adjacent vertices are the same color. Consider the ***edge-coloring*** problem: assign the smallest number of colors possible to edges of a given graph so that no two edges with the same end point are the same color. Explain how the edge-coloring problem can be reduced to a vertex-coloring problem.

10. In the ***jealous-husbands puzzle***, there are n ($n \ge 2$) married couples who need to cross a river. They have a boat that can hold no more than two people at a time. To complicate matters, all the husbands are jealous and will not agree on any crossing procedure that would put a wife on the same bank of the river with another woman's husband without the wife's husband being there too, even if there are other people on the same bank. Can they cross the river under such constraints?

    a. Solve the problem for $n = 2$.

    b. Solve the problem for $n = 3$, which is the classical version of this problem.

    c. Does the problem have a solution for every $n \ge 4$? If it does, explain how and indicate how many river crossings it will take; if it does not, explain why.

Hints to Exercises 6.6

1. a. Use the rules for computing lcm(m, n) and gcd(m, n) from the prime factors of m and n.

   b. The answer immediately follows from the formula for computing lcm(m, n).

2. Use a relationship between minimization and maximization problems.

3. Prove the assertion by induction on k.

4. a. Base your algorithm on the following observation: a graph contains a cycle of length 3 if and only if it has two adjacent vertices i and j that are also connected by a path of length 2.

   b. Do not jump to a conclusion in answering this question.

5. An easier solution is to reduce the problem to another one with a known algorithm. Since we did not discuss many geometric algorithms in the book, it should not be difficult to figure out to which one this problem needs to be reduced.

6. Express this problem as a maximization problem of a function in one variable.

7. Introduce double-indexed variables x_{ij} to indicate an assignment of the ith person to the jth job.

8. Take advantage of the specific features of this instance to reduce the problem to one with fewer variables.

9. Create a new graph.

10. (a)-(b) Create a state-space graph for the problem as it is done for the river-crossing puzzle in the section.

    c. Look at the state obtained after the first six river crossings in the solution to part (b).

Solutions to Exercises 6.6

1. a. Since

$$\begin{aligned}\mathrm{lcm}(m,n) = {} & \text{the product of the common prime factors of } m \text{ and } n\\ & \cdot \text{ the product of the prime factors of } m \text{ that are not in } n\\ & \cdot \text{ the product of the prime factors of } n \text{ that are not in } m\end{aligned}$$

and

$$\gcd(m,n) = \text{the product of the common prime factors of } m \text{ and } n,$$

the product of $\mathrm{lcm}(m,n)$ and $\gcd(m,n)$ is equal to

$$\begin{aligned}&\text{the product of the common prime factors of } m \text{ and } n\\ \cdot\ &\text{the product of the prime factors of } m \text{ that are not in } n\\ \cdot\ &\text{the product of the prime factors of } n \text{ that are not in } m\\ \cdot\ &\text{the product of the common prime factors of } m \text{ and } n.\end{aligned}$$

Since the product of the first two terms is equal to m and the product of the last two terms is equal to n, we showed that $\mathrm{lcm}(m,n) \cdot \gcd(m,n) = m \cdot n$, and, hence,

$$\mathrm{lcm}(m,n) = \frac{m \cdot n}{\gcd(m,n)}.$$

b. If $\gcd(m,n)$ is computed in $O(\log n)$ time, $\mathrm{lcm}(m,n)$ will also be computed in $O(\log n)$ because one extra multiplication and one extra division take only constant time.

2. Replace every key K_i of a given list by $-K_i$ and apply a max-heap construction algorithm to the new list. Then change the signs of all the keys again.

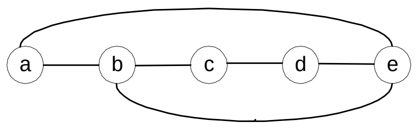
3. The induction basis: For $k = 1$, $A^1[i,j]$ is equal to 1 or 0 depending on whether there is an edge from vertex i to vertex j. In either case, it is also equal to the number of paths of length 1 from i to j.
The general step: Assume that $A^k[i,j]$ is equal to the number of different paths of length k from vertex i to vertex j. Since $A^{k+1} = A^k A$, we have the following equality for the (i,j) element of A^{k+1}:

$$A^{k+1}[i,j] = A^k[i,1]A[1,j] + \ldots + A^k[i,t]A[t,j] + \ldots + A^k[i,n]A[n,j],$$

where $A^k[i,t]$ is equal to the number of different paths of length k from vertex i to vertex t according to the induction hypothesis and $A[t,j]$ is equal to 1 or 0 depending on whether there is an edge from vertex t to vertex j for $t = 1, ..., n$. Further, any path of length $k+1$ from vertex i to vertex j must be made up of a path of length k from vertex i to some intermediate vertex t and an edge from that t to vertex j. Since for different intermediate vertices t we get different paths, the formula above yields the total number of different paths of length $k+1$ from i to j.

4. a. For the adjacency matrix A of a given graph, compute A^2 with an algorithm whose time efficiency is better than cubic (e.g., Strassen's matrix multiplication discussed in Section 4.5). Check whether there exists a nonzero element $A[i,j]$ in the adjacency matrix such that $A^2[i,j] > 0$: if there is, the graph contains a triangle subgraph, if there is not, the graph does not contain a triangle subgraph.

   b. The algorithm is incorrect because the condition is sufficient but not necessary for a graph to contain a cycle of length 3. Consider, as a counterexample, the DFS tree of the traversal that starts at vertex A of the following graph and resolves ties according to alphabetical order of vertices:

   It does not contain a back edge to a grandparent of a vertex, but the graph does have a cycle of length 3: $a - b - e - a$.

5. The problem can be reduced to the question about the convex hull of a given set of points: if the convex hull is a triangle, the answer is yes, otherwise, the answer is no. There are several algorithms for finding the convex hull for a set of points; quickhull, which was discussed in section 4.6, is particularly appropriate for this application.

6. Let x be one of the numbers in question; hence, the other number is $n-x$. The problem can be posed as the problem of maximizing $f(x) = x(n-x)$ on the set of all integer values of x. Since the graph of $f(x) = x(n-x)$ is a parabola with the apex at $x = n/2$, the solution is $n/2$ if n is even and $\lfloor n/2 \rfloor$ (or $\lceil n/2 \rceil$) if n is odd. Hence, the numbers in question can be computed as $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, which works both for even and odd values of n. Assuming that one division by 2 takes a constant time irrespective of n's size, the algorithm's time efficiency is clearly in $\Theta(1)$.

7. Let x_{ij} be a 0-1 variable indicating an assignment of the ith person to the jth job (or, in terms of the cost matrix C, a selection of the matrix element from the ith row and the jth column). The assignment problem can then be posed as the following integer (in fact, 0–1) linear programming problem:

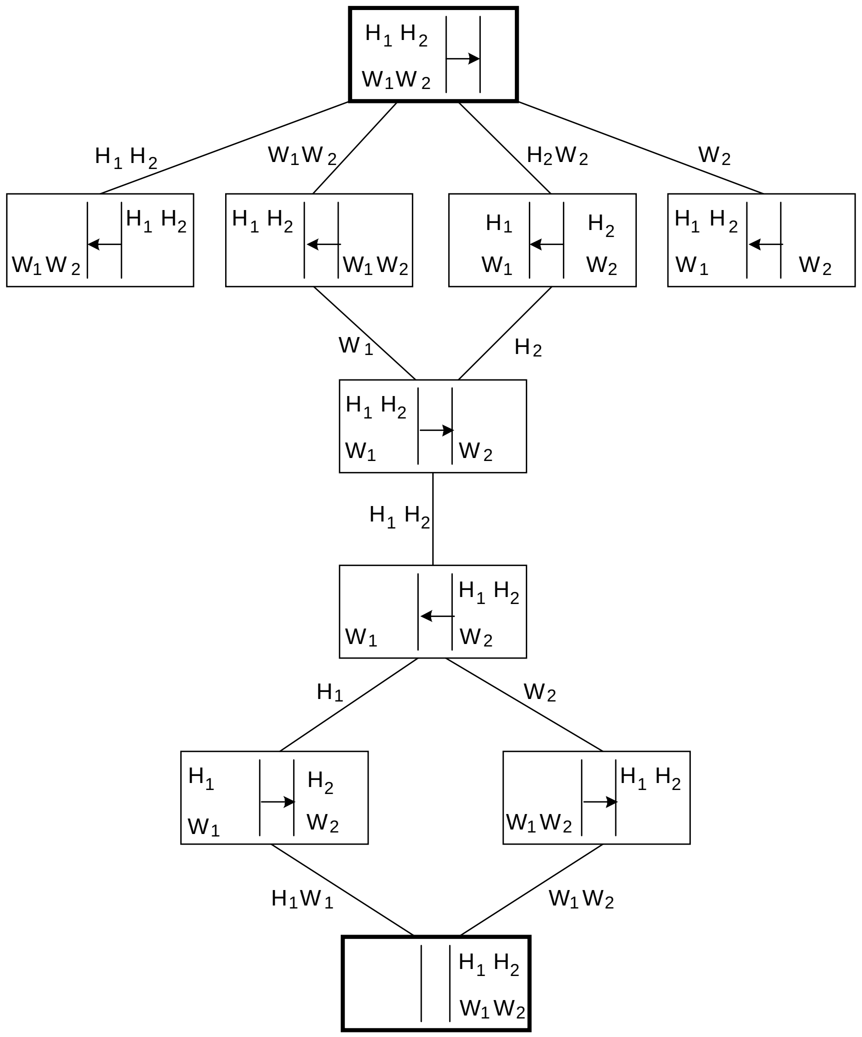
$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n}\sum_{i=1}^{n} c_{ij}x_{ij} \quad\quad\quad\quad \text{(the total assignment cost)} \\ \text{subject to} & \sum_{j=1}^{n} x_{ij} = 1 \text{ for } i = 1,...n \text{ (person } i \text{ is assigned to one job)} \\ & \sum_{i=1}^{n} x_{ij} = 1 \text{ for } j = 1,...n \text{ (job } j \text{ is assigned to one person)} \\ & x_{ij} \in \{0,1\} \text{ for } i = 1,...,n \text{ and } j = 1,...,n \end{array}$$

8. We can exploit the specific features of the instance in question to solve it by the following reasoning. Since the expected return from cash is the smallest, the value of cash investment needs to be minimized. Hence, $z = 0.25(x + y)$ in an optimal solution. Substituting $z = 0.25(x + y)$ into $x + y + z = 100$, yields $x + y = 80$ and hence $z = 20$. Similarly, since the expected return on stocks is larger than that of bonds, the amount invested in stocks needs to be maximized. Hence, in an optimal allocation $x = y/3$. Substituting this into $x + y = 80$ yields $y = 60$ and $x = 20$. Thus, the optimal allocation is to put 20 million in stocks, 60 millions in bonds, and 20 million in cash.

   Note: This method should not be construed as having a power beyond this particular instance. Generally speaking, we need to use general algorithms such as simplex method for solving linear programming problems with three or more unknowns. A special technique applicable to instances with only two variables is illustrated in the solution to Problem 9 in Exercises 3.3.

9. Create a new graph whose vertices represent the edges of the graph given and connect two vertices in the new graph by an edge if and only if these vertices represent two edges with a common endpoint in the original graph. A solution of the vertex-coloring problem for the new graph solves the edge-coloring problem for the original graph.

10. a. Here is a state-space graph for the two jealous husbands puzzle: H_i, W_i denote the husband and wife of couple i ($i = 1, 2$), respectively; the two bars | | denote the river; the arrow indicates the possible direction of the next trip, which is defined by the boat's location. (For the sake of simplicity, the graph doesn't include crossings that differ by obvious index substitutions such as starting with the first couple H_1W_1 crossing the river instead of the second one H_2W_2.) The vertices corresponding to the initial and final states are shown in bold.

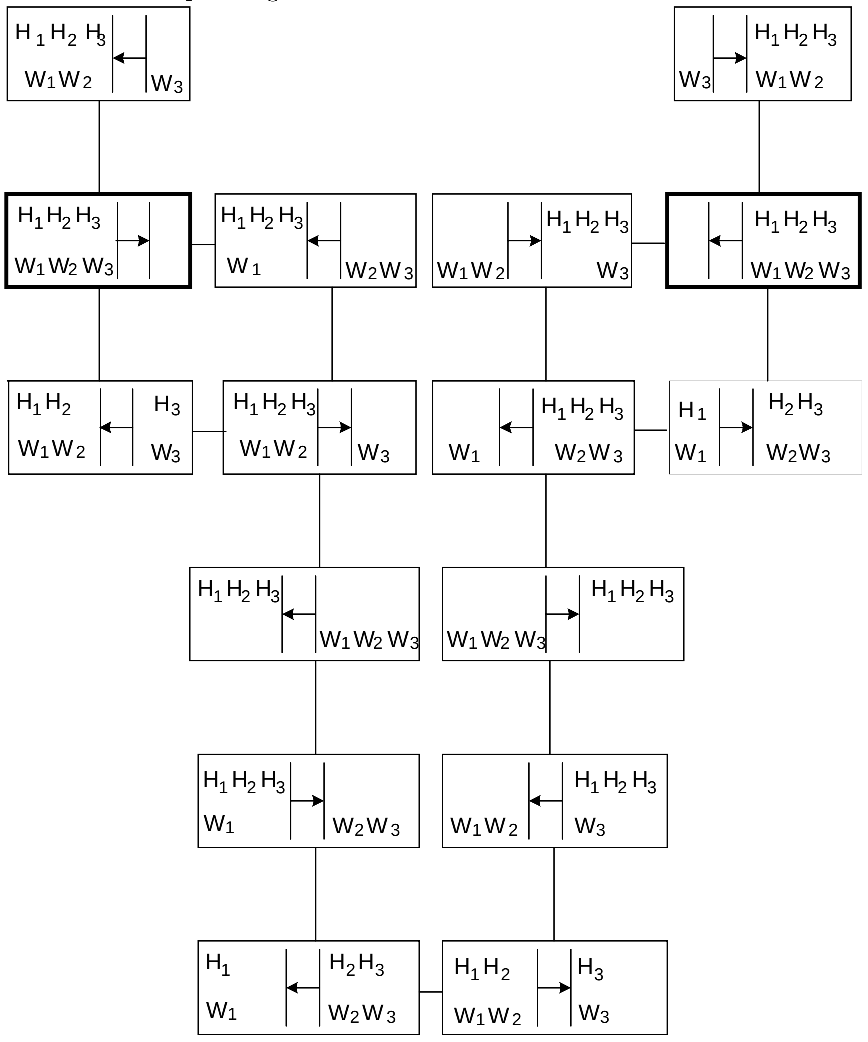


There are four simple paths from the initial-state vertex to the final-state vertex, each five edges long, in this graph. If specified by their edges, they are:

W_1W_2 W_1 H_1H_2 H_1 H_1W_1
W_1W_2 W_1 H_1H_2 W_2 W_1W_2
H_2W_2 H_2 H_1H_2 H_1 H_1W_1
H_2W_2 H_2 H_1H_2 W_2 W_1W_2

Hence, there are four (to within obvious symmetric substitutions) optimal solutions to this problem, each requiring five river crossings.

b. Here is a state-space graph for the three jealous husbands puzzle: H_i, W_i denote the husband and wife of couple i ($i = 1, 2, 3$), respectively; b stands for the boat; the two bars $|\ |$ denote the river; the arrow indicates the possible direction of the next trip, which is defined by the boat's location. (For the sake of simplicity, the graph doesn't include crossings that differ by obvious index substitutions such as starting with the first or second couple crossing the river instead of the third one $\mathrm{H}_3\mathrm{W}_3$.) The vertices corresponding to the initial and final states are shown in bold.



There are four simple paths from the initial-state vertex to the final-state vertex, each eleven edges long, in this graph. If specified by their edges, they are:

W_2W_3	W_2	W_1W_2	W_1	H_2H_3	H_2W_2	H_1H_2	W_3	W_2W_3	W_2	W_1W_2
W_2W_3	W_2	W_1W_2	W_1	H_2H_3	H_2W_2	H_1H_2	W_3	W_2W_3	H_1	H_1W_1
H_3W_3	H_3	W_1W_2	W_1	H_2H_3	H_2W_2	H_1H_2	W_3	W_2W_3	W_2	W_1W_2
H_3W_3	H_3	W_1W_2	W_1	H_2H_3	H_2W_2	H_1H_2	W_3	W_2W_3	H_1	H_1W_1

Hence, there are four (to within obvious symmetric substitutions) optimal solutions to this problem, each requiring eleven river crossings.

c. After the first six river crossings (see the solution to part (b)), the number of couples that need to be ferried over is reduced by one. This observation leads to the following recurrence relation for the number of river crossings $C(n)$ for the instance of n couples:

$$\begin{aligned} C(n) &= 6 + C(n-1) \quad \text{for } n > 2, \\ C(2) &= 5. \end{aligned}$$

Its solution, that can be obtained either by backward substitutions or by noting that $C(n)$ is an arithmetic progression, is

$$C(n) = 5 + 6(n-2) \quad \text{for } n \geq 2.$$