This file contains the exercises, hints, and solutions for Chapter 4 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 4.1

1. a. Write a pseudocode for a divide-and-conquer algorithm for finding the position of the largest element in an array of n numbers.

   b. What will be your algorithm's output for arrays with several elements of the largest value?

   c. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.

   d. How does this algorithm compare with the brute-force algorithm for this problem?

2. a. Write a pseudocode for a divide-and-conquer algorithm for finding values of both the largest and smallest elements in an array of n numbers.

   b. Set up and solve (for $n = 2^k$) a recurrence relation for the number of key comparisons made by your algorithm.

   c. How does this algorithm compare with the brute-force algorithm for this problem?

3. a. Write a pseudocode for a divide-and-conquer algorithm for the exponentiation problem of computing a^n where $a > 0$ and n is a positive integer.

   b. Set up and solve a recurrence relation for the number of multiplications made by this algorithm.

   c. How does this algorithm compare with the brute-force algorithm for this problem?

4. As mentioned in Chapter 2, logarithm bases are irrelevant in most contexts arising in analyzing an algorithm's efficiency class. Is it true for both assertions of the Master Theorem that include logarithms?

5. Find the order of growth for solutions of the following recurrences.

   a. $T(n) = 4T(n/2) + n, \ \ T(1) = 1$

b. $T(n) = 4T(n/2) + n^2, \ T(1) = 1$

c. $T(n) = 4T(n/2) + n^3, \ T(1) = 1$

6. Apply mergesort to sort the list *E, X, A, M, P, L, E* in alphabetical order.

7. Is mergesort a stable sorting algorithm?

8. a. Solve the recurrence relation for the number of key comparisons made by mergesort in the worst case. (You may assume that $n = 2^k$.)

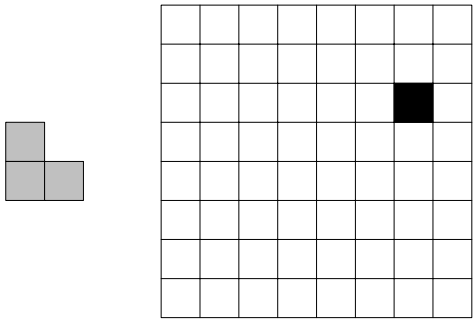
   b. Set up a recurrence relation for the number of key comparisons made by mergesort on best-case inputs and solve it for $n = 2^k$.

   c. Set up a recurrence relation for the number of key moves made by the version of mergesort given in Section 4.1. Does taking the number of key moves into account change the algorithm's efficiency class?

9. Let $A[0..n-1]$ be an array of n distinct real numbers. A pair $(A[i], A[j])$ is said to be an ***inversion*** if these numbers are out of order, i.e., $i < j$ but $A[i] > A[j]$. Design an $O(n \log n)$ algorithm for counting the number of inversions.

10. One can implement mergesort without a recursion by starting with merging adjacent elements of a given array, then merging sorted pairs, and so on. Implement this bottom-up version of mergesort in the language of your choice.

11. *Tromino puzzle* A tromino is an L-shaped tile formed by adjacent 1-by-1 squares. The problem is to cover any 2^n-by-2^n chessboard with one missing square (anywhere on the board) with trominoes. Trominoes should cover all the squares of the board except the missing one with no overlaps.



Design a divide-and-conquer algorithm for this problem.

Hints to Exercises 4.1

1. In more than one respect, this question is similar to the divide-and-conquer computation of the sum of n numbers. Also, you were asked to analyze an almost identical algorithm in Exercises 2.4.

2. Unlike Problem 1, a divide-and-conquer algorithm for this problem can be more efficient by a constant factor than the brute-force algorithm.

3. How would you compute a^8 by solving two exponentiation problems of size 4? How about a^9?

4. Look at the notations used in the theorem's statement.

5. Apply the Master Theorem.

6. Trace the algorithm as it was done for another input in the section.

7. How can mergesort reverse a relative ordering of two elements?

8. a. Use backward substitutions, as usual.

   b. What inputs minimize the number of key comparisons made by mergesort? How many comparisons are made by mergesort on such inputs during the merging stage?

   c. Do not forget to include key moves made both before the split and during the merging.

9. Modify mergesort to solve the problem.

10. n/a

11. A divide-and-conquer algorithm works by reducing a problem's instance to several smaller instances of the *same* problem.

Solutions to Exercises 4.1

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

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

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

c. The recurrence for the number of element comparisons is

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

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

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

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

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

which is the same as the left-hand side.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

b. The recurrence for the number of multiplications is

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

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

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

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

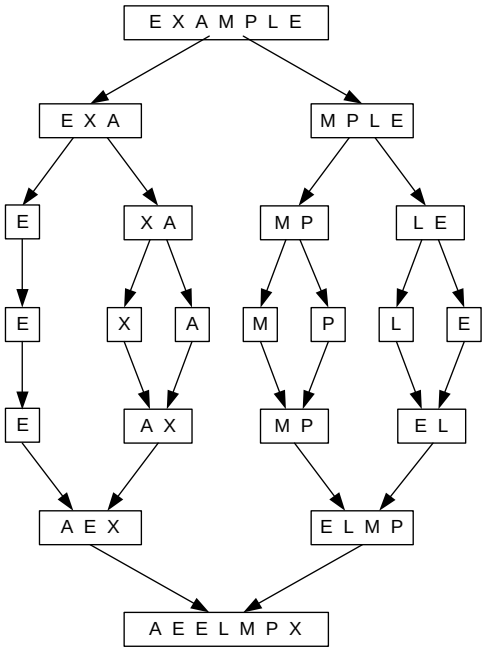
5. The applications of the Master Theorem yield the following.

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

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

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

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



7. Mergesort is stable, provided its implementation employs the comparison $\leq$ in merging. Indeed, assume that we have two elements of the same value in positions i and j, $i < j$, in a subarray before its two (sorted) halves are merged. If these two elements are in the same half of the subarray, their relative ordering will stay the same after the merging because the elements of the same half are processed by the merging operation in the FIFO fashion. Consider now the case when $A[i]$ is in the first half while $A[j]$ is in the second half. $A[j]$ is placed into the new array either after the first half becomes empty (and, hence, $A[i]$ has been already copied into the new array) or after being compared with some key $k > A[j]$ of the first half. In the latter case, since the first half is sorted before the merging begins, $A[i] = A[j] < k$ cannot be among the unprocessed elements of the first half. Hence, by the time of this comparison, $A[i]$ has been already copied into the new array and therefore will precede $A[j]$ after the merging operation is completed.

8. a. The recurrence for the number of comparisons in the worst case, which was given in Section 4.1, is

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

Solving it by backward substitutions yields the following:

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

b. The recurrence for the number of comparisons on best-case inputs (lists sorted in ascending or descending order) is

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

Thus,

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

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

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

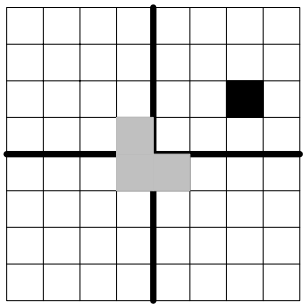
According to the Master Theorem, its solution is in $\Theta(n \log n)$—the same class established by the analysis of the number of key comparisons only.

5. Let *ModifiedMergesort* be a mergesort modified to return the number of inversions in its input array $A[0..n-1]$ in addition to sorting it. Obviously, for an array of size 1, *ModifiedMergesort*$(A[0])$ should return 0. Let

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

10. n/a

11. For $n > 1$, we can always place one L-tromino at the center of the $2^n \times 2^n$ chessboard with one missing square to reduce the problem to four subproblems of tiling $2^{n-1} \times 2^{n-1}$ boards, each with one missing square too. The orientation of this centrally placed piece is determined by the board's quarter with the missing square as shown by the example below.



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

Exercises 4.2

1. Apply quicksort to sort the list

   $$E,\ X,\ A,\ M,\ P,\ L,\ E$$

   in alphabetical order. Draw the tree of the recursive calls made.

2. For the partitioning procedure outlined in Section 4.2:

   a. Prove that if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p.

   b. Prove that when the scanning indices stop, j cannot point to an element more than one position to the left of the one pointed to by i.

   c. Why is it worth stopping the scans after encountering an element equal to the pivot?

3. Is quicksort a stable sorting algorithm?

4. Give an example of an array of n elements for which the sentinel mentioned in the text is actually needed. What should be its value? Also explain why a single sentinel suffices for any input.

5. For the version of quicksort given in the text:

   a. Are arrays made up of all equal elements the worst-case input, the best-case input, or neither?

   b. Are strictly decreasing arrays the worst-case input, the best-case input, or neither?

6. a. For quicksort with the median-of-three pivot selection, are increasing arrays the worst-case input, the best-case input, or neither?

   b. Answer the same question for decreasing arrays.

7. ▶ Solve the average-case recurrence for quicksort.

8. Design an algorithm to rearrange elements of a given array of n real numbers so that all its negative elements precede all its positive elements. Your algorithm should be both time- and space-efficient.

9. ▷ The ***Dutch flag problem*** is to rearrange an array of characters R, W, and B (red, white, and blue are the colors of the Dutch national flag) so that all the R's come first, the W's come next, and the B's come last. Design a linear in-place algorithm for this problem.

10. Implement quicksort in the language of your choice. Run your program on a sample of inputs to verify the theoretical assertions about the algorithm's efficiency.

11. $\triangleright$ *Nuts and bolts* You are given a collection of n bolts of different widths and n corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However, there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an algorithm for this problem with average-case efficiency in $\Theta(n \log n)$. [Raw91]

Hints to Exercises 4.2

1. We traced the algorithm on an another instance in the section.
2. a. Use the rules for stopping the scans.

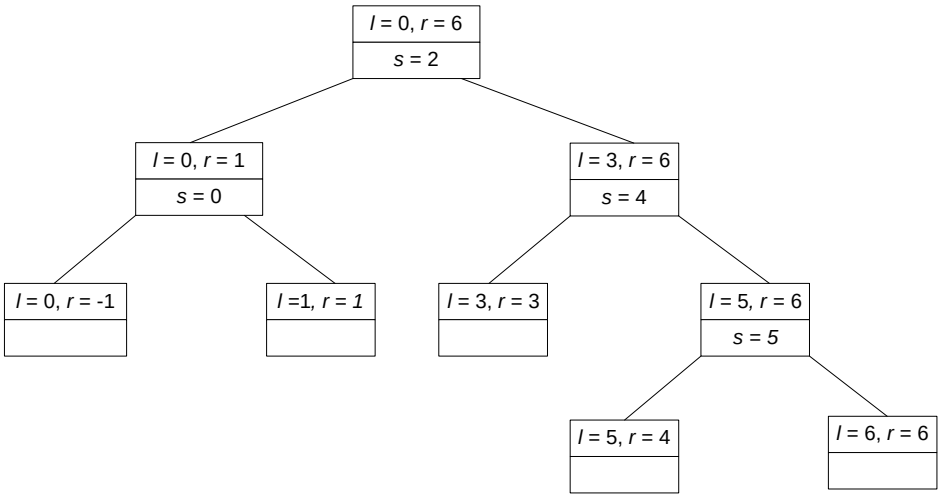
   b. Use the rules for stopping the scans.

   c. Consider an array whose all elements are the same.
3. The definition of *stability* of a sorting algorithm was given in Section 1.3. Generally speaking, algorithms that can exchange elements far apart are not stable.
4. Trace the algorithm to see on which inputs index i gets out of bounds.
5. Study what the text's version of quicksort does on such arrays. You should base your answers on the number of key comparisons, of course.
6. Where will splits occur on the inputs in question?
7. This requires several standard tricks for solving more sophisticated recurrence relations. A solution can be found in most books on the design and analysis of algorithms.
8. Use the partition idea.
9. You may want to solve first the two-color flag problem, i.e., rearrange efficiently an array of R's and B's. (A similar problem is Problem 8 in these exercises.)
10. n/a
11. Use the partition idea.

Solutions to Exercises 4.2

1. Applying the version of quicksort given in Section 4.2, we get the following:

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



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

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

   c. Stopping the scans after encountering an element equal to the pivot tends to yield better (i.e., more equal) splits. For example, if we did otherwise for an array of n equal elements, we would have gotten a split into subarrays of sizes $n - 1$ and 0.

3. Quicksort is not stable. As a counterexample, consider its performance on a two-element array of equal values.

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

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

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

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

7. Here is a solution that follows [Sed88], p. 121:

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

can be rewritten as

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

Since $\sum_{s=0}^{n-1} C(s) = \sum_{s=0}^{n-1} C(n-1-s)$, the equation can be reduced to

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

or

$$nC(n) = n(n+1) + 2\sum_{s=0}^{n-1} C(s).$$

Substituting $n-1$ for n in the last equation yields

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

Subtracting the last equation from the one before yields, after obvious simplifications, the recurrence

$$nC(n) = (n+1)C(n-1) + 2n,$$

which, after dividing both hand sides by $n(n+1)$, becomes

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2}{n+1}.$$

Substituting $B(n) = \frac{C(n)}{n+1}$, we obtain the following recurrence relation:

$$B(n) = B(n-1) + \frac{2}{n+1} \quad \text{for } n \geq 2, \quad B(1) = B(0) = 0.$$

The latter can be solved either by backward substitutions (or by "telescoping") to obtain

$$B(n) = 2\sum_{k=3}^{n+1} \frac{1}{k}.$$

Hence

$$B(n) = 2H_{n+1} - 3, \quad \text{where } H_{n+1} = \sum_{k=1}^{n+1} \frac{1}{k} \approx \ln(n+1) \text{ (see Appendix A).}$$

Thus,

$$C(n) = (n+1)B(n) \approx 2(n+1)\ln(n+1) \approx 2n\ln n.$$

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

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

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

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

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

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

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

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

Algorithm *DutchFlag*($A[0..n-1]$)
//Sorts an array with values in a three-element set
//Input: An array $A[0..n-1]$ of characters from {'R', 'W', 'B'}

```
//Output: Array A[0..n − 1] in which all its R elements precede
//         all its W elements that precede all its B elements
r ← 0;  w ← 0;  b ← n − 1
while w ≤ b do
    if A[w] = 'R'
        swap(A[r], A[w]);  r ← r + 1;  w ← w + 1
    else if A[w] = 'W'
        w ← w + 1
    else //A[w] = 'B'
        swap(A[w], A[b]);  b ← b − 1
```

10. n/a

11. Randomly select a nut and try each of the bolts for it to find the matching bolt and separate the bolts that are smaller and larger than the selected nut into two disjoint sets. Then try each of the unmatched nuts against the matched bolt to separate those that are larger from those that are smaller than the bolt. As a result, we've identified a matching pair and partitioned the remaining nuts and bolts into two smaller independent instances of the same problem. The average number of nut-bolt comparisons $C(n)$ is defined by the recurrence very similar to the one for quicksort in Section 4.2:

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

The solution to this recurrence can be shown to be in $\Theta(n \log n)$ by repeating the steps outlined in the solution to Problem 7.

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

Exercises 4.3

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

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

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

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

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

2. Solve the recurrence $C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor)+1$ for $n > 1$, $C_{worst}(1) = 1$, for $n = 2^k$ by backward substitutions.

3. a.▷ Prove the equality

$$\lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil \quad \text{for } n \geq 1.$$

   b.▷ Prove that $C_{worst}(n) = \lfloor \log_2 n \rfloor + 1$ satisfies equation (4.2) for every positive integer n.

4. Estimate how many times faster an average successful search will be in a sorted array of 100,000 elements if it is done by binary search versus sequential search.

5. Sequential search can be used with about the same efficiency whether a list is implemented as an array or as a linked list. Is it also true for binary search? (Of course, we assume that a list is sorted for binary search.)

6. How can one use binary search for range searching, i.e., for finding all the elements in a sorted array whose values fall between two given values L and U (inclusively), $L \leq U$? What is the worst-case efficiency of this algorithm?

7. Write a pseudocode for a recursive version of binary search.

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

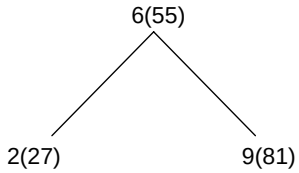
9. Analyze the time efficiency of the two-way comparison version designed in Problem 8.

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

Hints to Exercises 4.3

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

   (b)–(d) The most efficient prop for answering such questions is a binary tree that mirrors the algorithm's operations in searching for an arbitrary search key. The first three nodes of such a tree for the instance in question will look as follows:

   (The first number inside a node is the index m of the array's element being compared with a search key; the number in the parentheses is the value of the element itself, i.e., $A[m]$.)

2. If you need to refresh your memory, look up Section 2.4, where we solved an almost identical recurrence relation, and Appendix B.

3. a. Use the fact that n is bounded below and above by some consecutive powers of 2, i.e., $2^k \le n < 2^{k+1}$.

   b. The case of an even n ($n = 2i$) was considered in the section. For an odd n ($n = 2i + 1$), substitute the function into both hand sides of the equation and show their equality. The formula of part (a) might be useful. Do not forget to verify the initial condition, too.

4. Estimate the ratio of the average number of key comparisons made in a successful search by sequential search to that for binary search.

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

6. Find separately the elements that are greater than or equal to L and those that are smaller than or equal to U. Do not forget that neither L nor U have to be among the array values.

7. You may find the diagram of binary search in the text helpful.

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

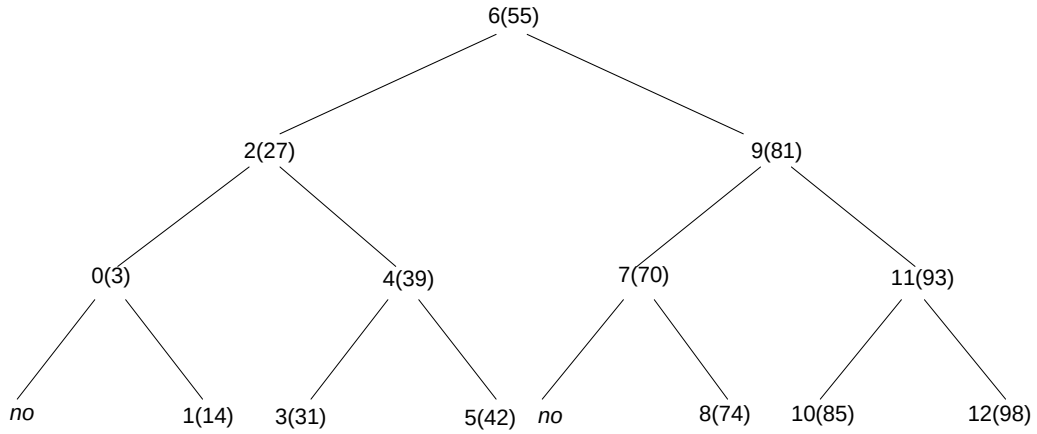
9. The analysis is almost identical to that of the text's version of binary search.

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

Solutions to Exercises 4.3

1. a. According to formula (4.4), $C_{worst}(13) = \lceil \log_2(13+1) \rceil = 4$.

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



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

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

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

2. For $n = 2^k$ (and omitting the subscript to simplify the notation), we get the recurrence relation

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

   By making backward substitutions, we obtain the following:

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

   Thus, we end up with

$$C(2^k) = C(1) + k = 1 + k$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$C(n) = \log_2 n + 1.$$

3. a. Every positive integer n is bounded below and above by some consecutive powers of 2, i.e.,

$$2^k \le n < 2^{k+1},$$

where k is a nonnegative integer (uniquely defined by the value of n). Taking the base-2 logarithms, we obtain

$$k \le \log_2 n < k + 1.$$

Hence $k \le \lfloor \log_2 n \rfloor$ and, since $\lfloor \log_2 n \rfloor \le \log_2 n < k + 1$, we have the inequality

$$k \le \lfloor \log_2 n \rfloor < k + 1,$$

which implies that $\lfloor \log_2 n \rfloor = k$. Similarly, $2^k < n + 1 \le 2^{k+1}$, and an argument analogous to the one just used to show that $\lfloor \log_2 n \rfloor = k$, we obtain that $\lceil \log_2(n + 1) \rceil = k + 1$. Hence, $\lceil \log_2(n + 1) \rceil = \lfloor \log_2 n \rfloor + 1$.

b. The case of even n was considered in the text of Chapter 4.3. Let $n > 1$ be odd, i.e., $n = 2i + 1$, where $i > 0$. The left-hand side is:

$C_w(n) = \lfloor \log_2 n \rfloor + 1 = \lfloor \log_2(2i+1) \rfloor + 1 = \lceil \log_2(2i+2) \rceil = \lceil \log_2 2(i+1) \rceil$

$= \lceil \log_2 2 + \log_2(i+1) \rceil = 1 + \lceil \log_2(i+1) \rceil = 1 + (\lfloor \log_2 i \rfloor + 1) = \lfloor \log_2 i \rfloor + 2.$

The right-hand side is:

$C_w(\lfloor n/2 \rfloor) + 1 = C_w(\lfloor (2i + 1)/2 \rfloor) + 1 = C_w(\lfloor i + 1/2 \rfloor) + 1 = C_w(i) + 1 = (\lfloor \log_2 i \rfloor + 1) + 1 = \lfloor \log_2 i \rfloor + 2,$

which is the same as the left-hand side.

The initial condition is verified immediately: $C_w(1) = \lfloor \log_2 1 \rfloor + 1 = 1$.

4. The ratio in question can be estimated as follows:

$$\frac{C_{avg}^{seq.}(n)}{C_{avg}^{bin.}(n)} \approx \frac{n/2}{\log_2 n} = (\text{for } n = 10^5)\frac{10^5/2}{\log_2 10^5} = \frac{1}{2 * 5}\frac{10^5}{\log_2 10} = \frac{10^4}{\log_2 10} \approx 3000.$$

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

6. Step 1: Check whether $A[0] \le U$ and $A[n-1] \le L$. If this is not true, stop: there are no such elements.
   Step 2: Search for L using the text's version of binary search. If the search was successful, record the value m of the index returned by the algorithm; if the search was unsuccessful, record the value of l on the exit from the algorithm.
   Step 3: Search for U using the text's version of binary search. If the search was successful, record the value m of the index returned by the algorithm; if the search was unsuccessful, record the value of r on the exit from the algorithm.
   The final answer (if the problem has a solution) is the range of the array indices between l and r (inclusively), where l and r are the values recorded in Steps 2 and 3, respectively.

7. Call $BSR(A[0..n-1], K)$ where

   Algorithm *BSR(* $A[l..r]$, K)
   //Implements binary search recursively.
   //Input: A sorted (sub)array $A[l..r]$ and a search key K
   //Output: An index of the array's element equal to K
   // or -1 if there is no such element.
   if $l > r$ **return** -1
   else $m \leftarrow \lfloor (l+r)/2 \rfloor$
   if $K = A[m]$ **return** m
   else if $K < A[m]$ **return** $BSR(A[l..m-1], K)$
   else return $BSR(A[m+1..r], K)$

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

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

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

Exercises 4.4

1. Design a divide-and-conquer algorithm for computing the number of levels in a binary tree. What is the efficiency class of your algorithm?

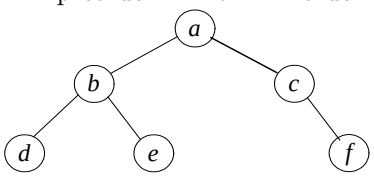
2. The following algorithm seeks to compute the number of leaves in a binary tree.

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

   Is this algorithm correct? If it is, prove it; if it is not, make an appropriate correction.

3. Prove equality (4.5) by mathematical induction.

4. Traverse the following binary tree
   a.. in preorder. b. in inorder. c. in postorder.



.

5. Write a pseudocode for one of the classic traversal algorithms (preorder, inorder, and postorder) for binary trees. Assuming that your algorithm is recursive, find the number of recursive calls made.

6. Which of the three classic traversal algorithms yields a sorted list if applied to a binary search tree? Prove this property.

7. a. Draw a binary tree with ten nodes labeled.0, 1, 2, ..., 9 in such a way that the inorder and postorder traversals of the tree yield the following lists: 9, 3, 1, 0, 4, 2, 7, 6, 8, 5 (inorder) and 9, 1, 4, 0, 3, 6, 7, 5, 8, 2 (postorder).

   b. Give an example of two permutations of the same n labels $0, 1, 2, .., n-1$ that cannot be inorder and postorder traversal lists of the same binary tree.

   c. Design an algorithm that constructs a binary tree for which two given lists of n labels $0, 1, 2, .., n-1$ are generated by the inorder and postorder traversals of the tree. Your algorithm should also identify inputs for which the problem has no solution.

8. ▷ The ***internal path length*** I of an extended binary tree is defined as the sum of the lengths of the paths—taken over all internal nodes—from the root to each internal node. Similarly, the ***external path length*** E of an extended binary tree is defined as the sum of the lengths of the paths—taken over all external nodes—from the root to each external node. Prove that $E = I + 2n$ where n is the number of internal nodes in the tree.

9. Write a program for computing the internal path length of a binary search tree. Use it to investigate empirically the average number of key comparisons for searching in a randomly generated binary search tree.

10. *Chocolate bar puzzle* Given an n-by-m chocolate bar, you need to break it into nm 1-by-1 pieces. You can break a bar only in a straight line, and only one bar can be broken at a time. Design an algorithm that solves the problem with the minimum number of bar breaks. What is this minimum number? Justify your answer by using properties of a binary tree.

Hints to Exercises 4.4

1. The problem is almost identical to the one discussed in this section.
2. Trace the algorithm on a small input.
3. Use strong induction on the number of internal nodes.
4. This is a standard exercise that you have probably done in your data structures course. With the traversal definitions given at the end of this section, you should be able to trace them even if you have never seen them before.
5. The pseudocodes can simply mirror the traversals' descriptions.
6. If you do not know the answer to this important question, you may want to check the results of the traversals on a small binary search tree. For a proof, answer the question: what can be said about two nodes with keys k_1 and k_2 if $k_1 < k_2$?
7. Find the root's label of the binary tree first and then identify the labels of the nodes in its left and right subtrees.
8. Use strong induction on the number of internal nodes.
9. n/a
10. Breaking the chocolate bar can be represented by a binary tree.

Solutions to Exercises 4.4

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

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

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

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

3. Here is a proof of equality (4.5) by strong induction on the number of internal nodes $n \geq 0$. The basis step is true because for $n = 0$ we have the empty tree whose extended tree has 1 external node by definition. For the inductive step, let us assume that

   $$x = k + 1$$

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

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

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

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

   which completes the proof.

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

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

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

5. Here is a pseudocode of the preorder traversal:

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

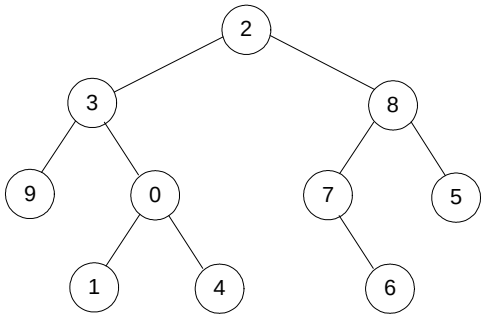
   The number of calls, $C(n)$, made by the algorithm is equal to the number of nodes, both internal and external, in the extended tree. Hence, according to the formula in the section,

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

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

7. a. The root's label is listed last in the postorder tree: hence, it is 2. The labels preceding 2 in the order list—9,3,1,0,4—form the inorder traversal list of the left subtree; the corresponding postorder list for the left subtree traversal is given by the first four labels of the postorder list: 9,1,4,0,3. Similarly, for the right subtree, the inorder and postorder lists are, respectively, 7,6,8,5 and 6,7,5,8. Applying the same logic recursively to each of

the subtrees yields the following binary tree:



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

c. The problem can be solved by a recursive algorithm based on the following observation: There exists a binary tree with inorder traversal list $i_0, i_1, ..., i_{n-1}$ and postorder traversal list $p_0, p_1, ..., p_{n-1}$ if and only if $p_{n-1} = i_k$ (the root's label), the sets formed by the first k labels in both lists are the same: $\{i_0, i_1, ..., i_{k-1}\} = \{p_0, p_1, ..., p_{k-1}\}$ (the labels of the nodes in the left subtree) and the sets formed by the other $n - k - 1$ labels excluding the root are the same: $\{i_{k+1}, i_{k+2}, ..., i_{n-1}\} = \{p_k, p_{k+1}, ..., p_{n-2}\}$ (the labels of the nodes in the right subtree).
Algorithm *Tree*$(i_0, i_1, ..., i_{n-1}, p_0, p_1, ..., p_{n-1})$
//Construct recursively the binary tree based on the inorder and postorder traversal lists
//Input: Lists $i_0, i_1, ..., i_{n-1}$ and $p_0, p_1, ..., p_{n-1}$ of inorder and postorder traversals, respectively
//Output: Binary tree T, specified in preorder, whose inorder and postorder traversals yield the lists given or
// -1 if such a tree doesn't exist
Find element i_k in the inorder list that is equal to the last element p_{n-1} of the postorder list.
if the previous search was unsuccessful **return** -1
else *print*(i_k)
Tree$(i_0, i_1, ..., i_{k-1}, p_0, p_1, ..., p_{k-1})$
Tree$(i_{k+1}, i_{k+2}, ..., i_{n-1}, p_k, p_{k+1}, ..., p_{n-2})$

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

For the general case of induction, we assume that

$$E = I + 2k$$

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

$$E_L = I_L + 2n_L,$$

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

$$E_R = I_R + 2n_R,$$

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

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

where x is the number of external nodes in T. Since $x = n+1$ (see Section 4.4), we finally obtain the desired equality:

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

9. n/a

10. We can represent operations of any algorithm solving the problem by a full binary tree in which parental nodes represent breakable pieces and leaves represent 1-by-1 pieces of the original bar. The number of the latter is nm; and the number of the former, which is equal to the number of the bar breaks, is one less, i.e., $nm - 1$, according to equation (4.5) in Section 4.4. (Note: This elegant solution was suggested to the author by Simon Berkovich, one of the book's reviewers.)

    Alternatively, we can reason as follows: Since only one bar can be broken at a time, any break increases the number of pieces by 1. Hence, $nm - 1$

breaks are needed to get from a single n-by-m piece to nm one-by-one pieces, which is obtained by *any* sequence of $nm-1$ allowed breaks. (The same argument can be made more formally by mathematical induction.)

Exercises 4.5

1. What are the smallest and largest numbers of digits the product of two decimal n-digit integers can have?

2. Compute $2101 * 1130$ by applying the divide-and-conquer algorithm outlined in the text.

3. a. Prove the equality $a^{\log_b c} = c^{\log_b a}$, which was used twice in Section 4.5.

   b. Why is $n^{\log_2 3}$ better than $3^{\log_2 n}$ as a closed-form formula for $M(n)$?

4. a. Why did we not include multiplications by 10^n in the multiplication count $M(n)$ of the large-integer multiplication algorithm?

   b. In addition to assuming that n is a power of 2, we made, for the sake of simplicity, another, more subtle, assumption in setting up a recurrence relation for $M(n)$ which is not always true (it does not change the final answer, however.) What is this assumption?

5. How many one-digit additions are made by the pen-and-pencil algorithm in multiplying two n-digit integers? (You may disregard potential carries.)

6. Verify the formulas underlying Strassen's algorithm for multiplying 2-by-2 matrices.

7. Apply Strassen's algorithm to compute

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

   exiting the recursion when $n = 2$, i.e., computing the products of 2-by-2 matrices by the brute-force algorithm.

8. Solve the recurrence for the number of additions required by Strassen's algorithm. (Assume that n is a power of 2.)

9. V. Pan [Pan78] has discovered a divide-and-conquer matrix multiplication algorithm that is based on multiplying two 70-by-70 matrices using 143,640 multiplications. Find the asymptotic efficiency of Pan's algorithm (you can ignore additions) and compare it with that of Strassen's algorithm.

10. Practical implementations of Strassen's algorithm usually switch to the brute-force method after matrix sizes become smaller than some "crossover point". Run an experiment to determine such crossover point on your computer system.

Hints to Exercises 4.5

1. You might want to answer the question for $n = 2$ first and then generalize it.

2. Trace the algorithm on the input given. You will have to use it again in order to compute products of two-digit numbers as well.

3. a. Take logarithms of both sides of the equality.

   b What did we use the closed-form formula for?

4. a. How do we multiply by powers of 10?

   b. Try to repeat the argument for, say, $98 * 76$.

5. Counting the number of one-digit additions made by the pen-and-pencil algorithm in multiplying, say, two four-digit numbers, should help answering the general question.

6. Check the formulas by simple algebraic manipulations.

7. Trace Strassen's algorithm on the input given. (It takes some work, but it would have been much more of it if you were asked to stop the recursion when $n = 1$.) It is a good idea to check your answer by multiplying the matrices by the brute-force (i.e., definition-based) algorithm, too.

8. Use the method of backward substitutions to solve the recurrence given in the text.

9. The recurrence for the number of multiplications in Pan's algorithm is similar to that for Strassen's algorithm. Use the Master Theorem to find the order of growth of its solution.

10. n/a

Solutions to Exercises 4.5

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

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

2. For $2101 * 1130$:

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

   For $21 * 11$:

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

   For $01 * 30$:

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

   For $22 * 41$:

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

   Hence

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

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

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

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

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

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

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

   Hence, the total number of additions without carries will be

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

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

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

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

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

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

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

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

where

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

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

Therefore,

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

Accordingly,

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

That is,

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

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

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

Solving it by backward substitutions yields the following:

$$\begin{aligned}
A(2^k) &= 7A(2^{k-1}) + \frac{9}{2}4^k \\
&= 7[7A(2^{k-2}) + \frac{9}{2}4^{k-1}] + \frac{9}{2}4^k = 7^2A(2^{k-2}) + 7 \cdot \frac{9}{2}4^{k-1} + \frac{9}{2}4^k \\
&= 7^2[7A(2^{k-3}) + \frac{9}{2}4^{k-2}] + 7 \cdot \frac{9}{2}4^{k-1} + \frac{9}{2}4^k \\
&= 7^3A(2^{k-3}) + 7^2 \cdot \frac{9}{2}4^{k-2} + 7 \cdot \frac{9}{2}4^{k-1} + \frac{9}{2}4^k \\
&= ... \\
&= 7^kA(2^{k-k}) + \frac{9}{2}\sum_{i=0}^{k-1} 7^i4^{k-i} = 7^k \cdot 0 + \frac{9}{2}4^k \sum_{i=0}^{k-1} \left(\frac{7}{4}\right)^i \\
&= \frac{9}{2}4^k \frac{(7/4)^k - 1}{(7/4) - 1} = 6(7^k - 4^k).
\end{aligned}$$

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

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

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

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

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

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

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

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

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

10. n/a

Exercises 4.6

1. a. For the one-dimensional version of the closest-pair problem, i.e., for the problem of finding two closest numbers among a given set of n real numbers, design an algorithm that is directly based on the divide-and-conquer technique and determine its efficiency class.

   b. Is it a good algorithm for this problem?

2. Consider the version of the divide-and-conquer two-dimensional closest-pair algorithm in which we simply sort each of the two sets C_1 and C_2 in ascending order of their y coordinates on each recursive call. Assuming that sorting is done by mergesort, set up a recurrence relation for the running time in the worst case and solve it for $n = 2^k$.

3. Implement the divide-and-conquer closest-pair algorithm, outlined in this section, in the language of your choice.

4. Find a visualization of an algorithm for the closest-pair problem on the Web. What algorithm does this visualization represent?

5. The ***Voronoi polygon*** for a point P of a set S of points in the plane is defined to be the perimeter of the set of all points in the plane closer to P than to any other point in S. The union of all the Voronoi polygons of the points in S is called the ***Voronoi diagram*** of S.

   a. What is the Voronoi diagram for a set of three points?

   b. Find a visualization of an algorithm for generating the Voronoi diagram on the Web and study a few examples of such diagrams. Based on your observations, can you tell how the solution to the previous question is generalized to the general case?

6. Explain how one can find point $P_{\max}$ in the quickhull algorithm analytically.

7. What is the best-case efficiency of quickhull?

8. Give a specific example of inputs that make the quickhull algorithm run in quadratic time.

9. Implement the quickhull algorithm in the language of your choice.

10. *Shortest path around* There is a fenced area in the two-dimensional Euclidean plane in the shape of a convex polygon with vertices at points $P_1(x_1, y_1)$, $P_2(x_2, y_2), ..., P_n(x_n, y_n)$ (not necessarily in this order). There are two more points, $A(x_A, y_A)$ and $B(x_B, y_B)$, such that $x_A < \min\{x_1, x_2, ..., x_n\}$ and $x_B > \max\{x_1, x_2, ..., x_n\}$. Design a reasonably efficient algorithm for computing the length of the shortest path between A and B. [ORo98], p.68

Hints to Exercises 4.6

1. a. How many points need to be considered in the combining-solutions stage of the algorithm?

   b. Design a simpler algorithm in the same efficiency class.

2. Recall (see Section 4.1) that the number of comparisons made by mergesort in the worst case is $C_{worst}(n) = n \log_2 n - n + 1$ (for $n = 2^k$). You may use just the highest-order term of this formula in the recurrence you need to set up.

3. n/a

4. n/a

5. The answer to part (a) comes directly from a textbook on plane geometry.

6. Use the formula relating the value of a determinant with the area of a triangle.

7. It must be in $\Omega(n)$, of course. (Why?)

8. Design a sequence of n points for which the algorithm decreases the problem's size just by one on each of its recursive calls.

9. n/a

10. The path cannot cross inside the fenced area but it can go along the fence.

Solutions to Exercises 4.6

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

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

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

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

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

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

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

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

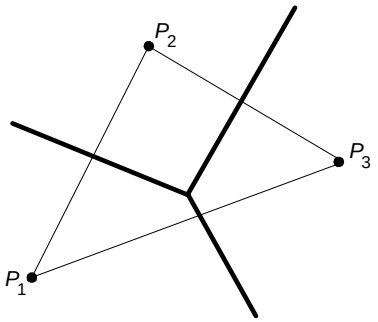
   Thus, $T(2^k) = 2T(2^{k-1}) + 2^k(k-1)$. Solving it by backward substitutions yields the following:

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

3. n/a

4. n/a

5. a. The Voronoi diagram of three points not on the same line is formed by the perpendicular bisectors of the sides of the triangle with vertices at P_1, P_2, and P_3:



(If P_1, P_2, and P_3 lie on the same line, with P_2 between P_1 and P_3, the Voronoi diagram is formed by the perpendicular bisectors of the segments with the endpoints at P_1 and P_2 and at P_2 and P_3.)

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

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

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

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

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

8. Among many possible answers, one can take two endpoints of the horizontal diameter of some circumference as points P_1 and P_n and obtain the other points P_i, $i = 2, ..., n-1$, of the set in question by placing them successively in the middle of the circumference's upper arc between P_{i-1} and P_n.

9. n/a

10. Find the upper and lower hulls of the set $\{A,\ B,\ P_1, ..., P_n\}$ (e.g., by quickhull), compute their lengths (by summing up the lengths of the line segments making up the polygonal chains) and return the smaller of the two.