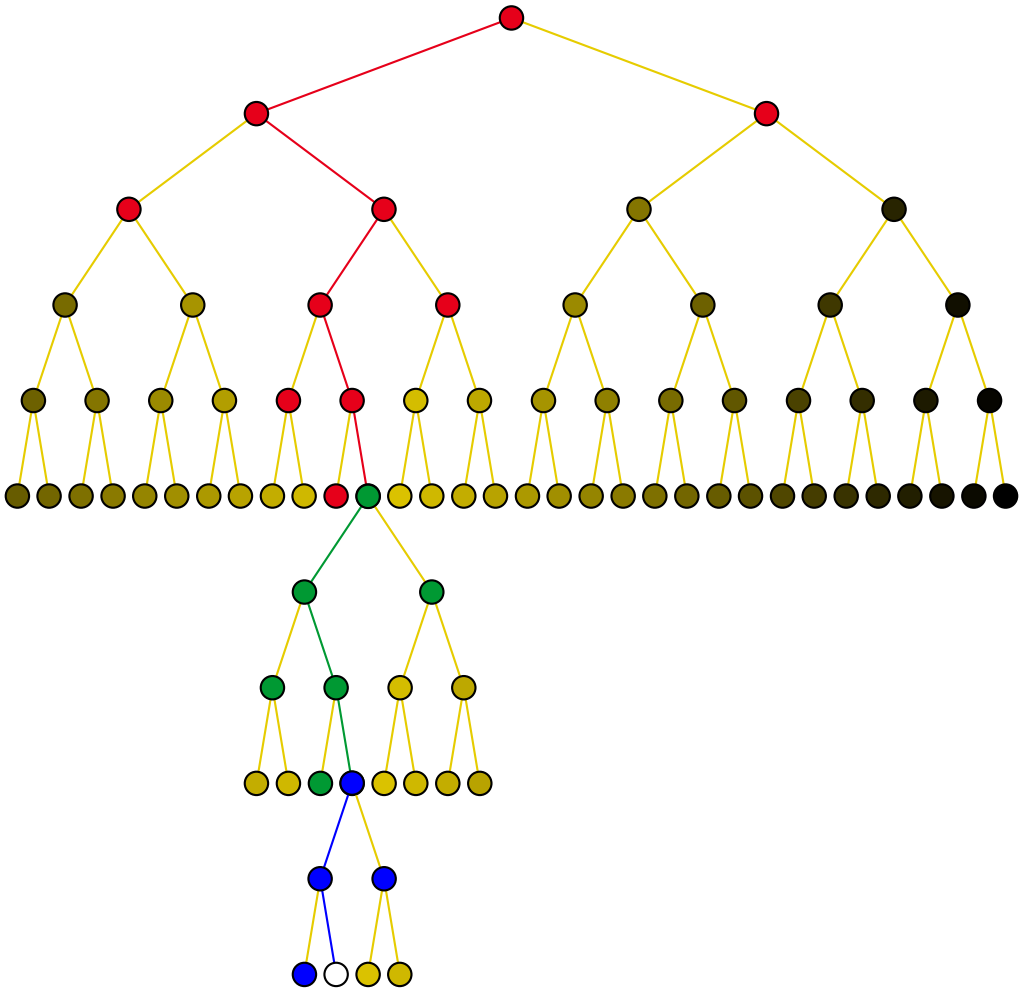
Computer Algorithms, Third Edition, Solutions to Selected Exercises

Sara Baase

Allen Van Gelder



February 25, 2000

INTRODUCTION

This manual contains solutions for the selected exercises in *Computer Algorithms: Introduction to Design and Analysis*, third edition, by Sara Baase and Allen Van Gelder.

Solutions manuals are intended primarily for instructors, but it is a fact that instructors sometimes put copies in campus libraries or on their web pages for use by students. For instructors who prefer to have students work on problems without access to solutions, we have chosen not to include all the exercises from the text in this manual. The included exercises are listed in the table of contents. Roughly every other exercise is solved.

Some of the solutions were written specifically for this manual; others are adapted from solutions sets handed out to students in classes we taught (written by ourselves, teaching assistants, and students).

Thus there is some inconsistency in the style and amount of detail in the solutions. Some may seem to be addressed to instructors and some to students. We decided not to change these inconsistencies, in part because the manual will be read by instructors and students. In some cases there is more detail, explanation, or justification than a student might be expected to supply on a homework assignment.

Many of the solutions use the same pseudocode conventions used in the text, such as:

1. Block delimiters ("{" and "}") are omitted. Block boundaries are indicated by indentation.
2. The keyword **`static`** is omitted from method (function and procedure) declarations. All methods declared in the solutions are **`static`**.
3. Class name qualifiers are omitted from method (function and procedure) calls. For example, **`x = cons(z, x)`** might be written when the Java syntax requires **`x = IntList.cons(z, x)`**.
4. Keywords to control visibility, **`public`**, **`private`**, and **`protected`**, are omitted.
5. Mathematical relational operators "$\neq$," "$\leq$," and "$\geq$" are usually written, instead of their keyboard versions. Relational operators are used on types where the meaning is clear, such as **`String`**, even though this would be invalid syntax in Java.

We thank Chuck Sanders for writing most of the solutions for Chapter 2 and for contributing many solutions in Chapter 14. We thank Luo Hong, a graduate student at UC Santa Cruz, for assisting with several solutions in Chapters 9, 10, 11, and 13.

In a few cases the solutions given in this manual are affected by corrections and clarifications to the text. These cases are indicated at the beginning of each affected solution. The up-to-date information on corrections and clarifications, along with other supplementary materials for students, can be found at these Internet sites:

`ftp://ftp.aw.com/cseng/authors/baase`
`http://www-rohan.sdsu.edu/faculty/baase`
`http://www.cse.ucsc.edu/personnel/faculty/avg.html`

List of Solved Exercises

Chapter 1

Analyzing Algorithms and Problems: Principles and Examples

Section 1.2: *Java as an Algorithm Language*

1.1

It is correct for instance fields whose type is an inner class to be declared before that inner class (as in Figure 1.2 in the text) or after (as here). Appendix A.7 gives an alternative to spelling out all the instance fields in the copy methods (functions).

```
class  Personal
     {
     public static class  Name
          {
          String  firstName;
          String  middleName;
          String  lastName;
          public static Name copy(Name n)
               {
               Name  n2;
               n2.firstName = n.firstName;
               n2.middleName = n.middleName;
               n2.lastName = n.lastName;
               return n2;
               }
          }

     public static class  Address
          {
          String  street;
          String  city;
          String  state;
          public static Address copy(Address a) {/* similar to Name.copy() */ }
          }

     public static class  PhoneNumber
          {
          int  areaCode;
          int  prefix;
          int  number;
          public static PhoneNumber copy(PhoneNumber n) {/* similar to Name.copy() */ }
          }

     Name  name;
     Address  address;
     PhoneNumber  phone;
     String  eMail;

     public static Personal copy(Personal p);
          {
          Personal  p2;
          p2.name = Name.copy(p.name);
          p2.address = Address.copy(p.address);
          p2.phone = PhoneNumber.copy(p.phone);
          p2.eMail = p.eMail;
          return p2;
          }
     }
```

Section 1.3: *Mathematical Background*

1.2
For $0 < k < n$, we have

$$\binom{n-1}{k} = \frac{(n-1)!}{k!(n-1-k)!} = \frac{(n-1)!(n-k)}{k!(n-k)!}$$

$$\binom{n-1}{k-1} = \frac{(n-1)!}{(k-1)!(n-k)!} = \frac{(n-1)!(k)}{k!(n-k)!}$$

Add them giving:

$$\frac{(n-1)!(n)}{k!(n-k)!} = \binom{n}{k}$$

For $0 < n \leq k$ we use the fact that $\binom{a}{b} = 0$ whenever $a < b$. (There is no way to choose more elements than there are in the whole set.) Thus $\binom{n-1}{k} = 0$ in all these cases. If $n < k$, $\binom{n-1}{k-1}$ and $\binom{n}{k}$ are both 0, confirming the equation. If $n = k$, $\binom{n-1}{k-1}$ and $\binom{n}{k}$ are both 1, again confirming the equation. (We need the fact that $0! = 1$ when $n = k = 1$.)

1.4
It suffices to show:

$$\log_c x \log_b c = \log_b x.$$

Consider b raised to each side.

$$b^{\text{left side}} = b^{\log_b c \log_c x} = (b^{\log_b c})^{\log_c x} = c^{\log_c x} = x$$
$$b^{\text{right side}} = b^{\log_b x} = x$$

So left side = right side.

1.6
Let $x = \lceil \lg(n+1) \rceil$. The solution is based on the fact that $2^{x-1} < n+1 \leq 2^x$.

```
x = 0;
twoToTheX = 1;
while (twoToTheX < n+1)
    x += 1;
    twoToTheX *= 2;
return x;
```

The values computed by this procedure for small n and the approximate values of $\lg(n+1)$ are:

n	x	$\lg(n+1)$
0	0	0.0
1	1	1.0
2	2	1.6
3	2	2.0
4	3	2.3
5	3	2.6
6	3	2.8
7	3	3.0
8	4	3.2
9	4	3.3

1.8

$$Pr(S \mid T) = \frac{Pr(S \text{ and } T)}{Pr(T)} = \frac{Pr(S)Pr(T)}{Pr(T)} = Pr(S)$$

The second equation is similar.

1.10
We know $A < B$ and $D < C$. By direct counting:

$$Pr(A < C \mid A < B \text{ and } D < C) = \frac{Pr(A < C \text{ and } A < B \text{ and } D < C)}{Pr(A < B \text{ and } D < C)} = \frac{5/24}{6/24} = \frac{5}{6}$$

$$Pr(A < D \mid A < B \text{ and } D < C) = \frac{Pr(A < D < C \text{ and } A < B)}{Pr(A < B \text{ and } D < C)} = \frac{3/24}{6/24} = \frac{3}{6} = \frac{1}{2}$$

$$Pr(B < C \mid A < B \text{ and } D < C) = \frac{Pr(A < B < C \text{ and } D < C)}{Pr(A < B \text{ and } D < C)} = \frac{3/24}{6/24} = \frac{3}{6} = \frac{1}{2}$$

$$Pr(B < D \mid A < B \text{ and } D < C) = \frac{Pr(A < B < D < C)}{Pr(A < B \text{ and } D < C)} = \frac{1/24}{6/24} = \frac{1}{6}$$

1.12
We assume that the probability of each coin being chosen is 1/3, that the probability that it shows "heads" after being flipped is 1/2 and that the probability that it shows "tails" after being flipped is 1/2. Call the coins *A*, *B*, and *C*. Define the elementary events, each having probability 1/6, as follows.

AH	*A* is chosen and flipped and comes out "heads".
AT	*A* is chosen and flipped and comes out "tails".
BH	*B* is chosen and flipped and comes out "heads".
BT	*B* is chosen and flipped and comes out "tails".
CH	*C* is chosen and flipped and comes out "heads".
CT	*C* is chosen and flipped and comes out "tails".

a) *BH* and *CH* cause a majority to be "heads", so the probability is 1/3.

b) No event causes a majority to be "heads", so the probability is 0.

c) *AH*, *BH*, *CH* and *CT* cause a majority to be "heads", so the probability is 2/3.

1.13
The entry in row *i*, column *j* is the probability that D_i will beat D_j.

$$\begin{pmatrix} - & \frac{22}{36} & \frac{18}{36} & \frac{12}{36} \\ \frac{12}{36} & - & \frac{22}{36} & \frac{16}{36} \\ \frac{18}{36} & \frac{12}{36} & - & \frac{22}{36} \\ \frac{22}{36} & \frac{20}{36} & \frac{12}{36} & - \end{pmatrix}$$

Note that D_1 beats D_2, D_2 beats D_3, D_3 beats D_4, and D_4 beats D_1.

1.15

The proof is by induction on n, the upper limit of the sum. The base case is $n = 0$. Then $\sum_{i=1}^{0} i^2 = 0$, and $\frac{2n^3+3n^2+n}{6} = 0$. So the equation holds for the base case. For $n > 0$, assume the formula holds for $n-1$.

$$\begin{aligned}\sum_{i=1}^{n} i^2 &= \sum_{i=1}^{n-1} i^2 + n^2 = \frac{2(n-1)^3+3(n-1)^2+n-1}{6} + n^2 \\ &= \frac{2n^3-6n^2+6n-2+3n^2-6n+3+n-1}{6} + n^2 \\ &= \frac{2n^3-3n^2+n}{6} + \frac{6n^2}{6} = \frac{2n^3+3n^2+n}{6}\end{aligned}$$

1.18

Consider any two reals $w < z$. We need to show that $f(w) \leq f(z)$; that is, $f(z) - f(w) \geq 0$. Since $f(x)$ is differentiable, it is continuous. We call upon the *Mean Value Theorem* (sometimes called the *Theorem of the Mean*), which can be found in any college calculus text. By this theorem there is some point y, such that $w < y < z$, for which

$$f'(y) = \frac{(f(z) - f(w))}{(z-w)}.$$

By the hypothesis of the lemma, $f'(y) \geq 0$. Also, $(z-w) > 0$. Therefore, $f(z) - f(w) \geq 0$.

1.20

Let $\equiv$ abbreviate the phrase, "is logically equivalent to". We use the identity $\neg\neg A \equiv A$ as needed.

$$\begin{aligned}\neg(\forall x(A(x) \Rightarrow B(x))) &\equiv \exists x \neg(A(x) \Rightarrow B(x)) \qquad \text{(by Eq. 1.24)} \\ &\equiv \exists x \neg(\neg A(x) \vee B(x)) \qquad \text{(by Eq. 1.21)} \\ &\equiv \exists x (A(x) \wedge \neg B(x)) \qquad \text{(by DeMorgan's law, Eq. 1.23).}\end{aligned}$$

Section 1.4: *Analyzing Algorithms and Problems*

1.22

The total number of operations in the worst case is $4n + 2$; they are:

Comparisons involving K:	n
Comparisons involving **`index`**:	$n+1$
Additions:	n
Assignments to **`index`**:	$n+1$

1.23

a)

```
if (a < b)
     if (b < c)
          median = b;
     else if (a < c)
          median = c;
     else
          median = a;
else if (a < c)
     median = a;
else if (b < c)
     median = c;
else
     median = b;
```

b) D is the set of permutations of three items.

c) Worst case = 3; average = $2\frac{2}{3}$.

d) Three comparisons are needed in the worst case because knowing the median of three numbers requires knowing the complete ordering of the numbers.

1.25

Solution 1. Pair up the entries and find the larger of each pair; if n is odd, one element is not examined ($\lfloor n/2 \rfloor$ comparisons). Then find the maximum among the larger elements using Algorithm 1.3, including the unexamined element if n is odd ($\lfloor (n+1)/2 \rfloor - 1$ comparisons). This is the largest entry in the set. Then find the minimum among the smaller elements using the appropriate modification of Algorithm 1.3, again including the unexamined element if n is odd ($\lfloor (n+1)/2 \rfloor - 1$ comparisons). This is the smallest entry in the set. Whether n is odd or even, the total is $\lfloor \frac{3}{2}(n-1) \rfloor$. The following algorithm interleaves the three steps.

```
/** Precondition:  n > 0.  */
if (odd(n))
     min = E[n-1];
     max = E[n-1];
else if (E[n-2] < E[n-1])
     min = E[n-2];
     max = E[n-1];
else
     max = E[n-2];
     min = E[n-1];

for (i = 0; i <= n-3; i = i+2)
     if (E[i] < E[i+1])
          if (E[i] < min) min = E[i];
          if (E[i+1] > max) max = E[i+1];
     else
          if (E[i] > max) max = E[i];
          if (E[i+1] < min) min = E[i+1];
```

Solution 2. When we assign this problem after covering Divide and Conquer sorting algorithms in Chapter 4, many students give the following Divide and Conquer solution. (But most of them cannot show formally that it does roughly $3n/2$ comparisons.)

If there are at most two entries in the set, compare them to find the smaller and larger. Otherwise, break the set in halves, and recursively find the smallest and largest in each half. Then compare the largest keys from each half to find the largest overall, and compare the smallest keys from each half to find the smallest overall.

Analysis of Solution 2 requires material introduced in Chapter 3. The recurrence equation for this procedure, assuming n is a power of 2, is

$$\begin{aligned} W(n) &= 1 \qquad \text{for } n = 2 \\ W(n) &= 2W(n/2) + 2 \qquad \text{for } n > 2 \end{aligned}$$

The recursion tree can be evaluated directly. It is important that the nonrecursive costs in the $n/2$ *leaves* of this tree are 1 each. The nonrecursive costs in the $n/2 - 1$ *internal nodes* are 2 each. This leads to the total of $3n/2 - 2$ for the special case that n is a power of 2. More careful analysis verifies the result $\lceil 3n/2 - 2 \rceil$ for all n. The result can also be proven by induction.

Section 1.5: *Classifying Functions by Their Asymptotic Growth Rates*

1.28

$$\lim_{n\to\infty} \frac{p(n)}{n^k} = \lim_{n\to\infty} \left(a_k + \frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \ldots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} \right) = a_k > 0.$$

1.31

The solution here combines parts (a) and (b). The functions on the same line are of the same asymptotic order.

$\lg\lg n$
$\lg n, \quad \ln n$
$(\lg n)^2$
$\sqrt{n}$
n
$n\lg n$
$n^{1+\varepsilon}$
$n^2, \quad n^2+\lg n$
n^3
$n-n^3+7n^5$
$2^{n-1}, \quad 2^n$
e^n
$n!$

1.33

Let $f(n)=n$. For simplicity we show a counter-example in which a nonmonotonic function is used. Consider the function $h(n)$:

$$h(n) = \begin{cases} n & \text{for odd } n \\ 1 & \text{for even } n \end{cases}$$

Clearly $h(n) \in O(f(n))$. But $h(n) \notin \Omega(f(n))$, so $h(n) \notin \Theta(f(n))$. Therefore, $h(n) \in O(f(n)) - \Theta(f(n))$. It remains to show that $h(n) \notin o(f(n))$. But this follows by the fact that $h(n)/f(n) = 1$ for odd integers.

With more difficulty $h(n)$ can be constructed to be monotonic. For all $k \geq 1$, let $h(n)$ be constant on the interval $k^k \leq n \leq ((k+1)^{k+1}-1)$ and let $h(n)=k^k$ on this interval. Thus when $n=k^k$, $h(n)/f(n)=1$, but when $n=(k+1)^{k+1}-1$, $h(n)/f(n)=k^k/((k+1)^{k+1}-1)$, which tends to 0 as n gets large.

1.35

Property 1: Suppose $f \in O(g)$. There are $c>0$ and n_0 such that for $n \geq n_0$, $f(n) \leq cg(n)$. Then for $n \geq n_0$, $g(n) \geq (1/c)f(n)$. The other direction is proved similarly.

Property 2: $f \in \Theta(g)$ means $f \in O(g) \cap \Omega(g)$. By Property 1, $g \in \Omega(f) \cap O(f)$, so $g \in \Theta(f)$.

Property 3: Lemma 1.9 of the text gives transitivity. Property 2 gives symmetry. Since for any f, $f \in \Theta(f)$, we have reflexivity.

Property 4: We show $O(f+g) \subseteq O(\max(f,g))$. The other direction is similar. Let $h \in O(f+g)$. There are $c>0$ and n_0 such that for $n \geq n_0$, $h(n) \leq c(f+g)(n)$. Then for $n \geq n_0$, $h(n) \leq 2c\max(f,g)(n)$.

1.37

We will use L'Hôpital's Rule, so we need to differentiate 2^n. Observe that $2^n = (e^{\ln 2})^n = e^{n\ln 2}$. Let $c = \ln 2 \approx 0.7$. The derivative of e^n is e^n, so, using the chain rule, we find that the derivative of 2^n is $c2^n$. Now, using L'Hôpital's Rule repeatedly,

$$\lim_{n\to\infty} \frac{n^k}{2^n} = \lim_{n\to\infty} \frac{kn^{k-1}}{c2^n} = \lim_{n\to\infty} \frac{k(k-1)n^{k-2}}{c^2 2^n} = \cdots = \lim_{n\to\infty} \frac{k!}{c^k 2^n} = 0$$

since k is constant.

1.39

$$f(n) = \begin{cases} 1 & \text{for odd } n \\ n & \text{for even } n \end{cases} \qquad g(n) = \begin{cases} n & \text{for odd } n \\ 1 & \text{for even } n \end{cases}$$

There are also examples using continuous functions on the reals, as well as examples using monotonic functions.

Section 1.6: *Searching an Ordered Array*

1.42

The revised procedure is:

```
    int binarySearch(int[] E, int first, int last, int K)
1.      if (last < first)
2.          index = -1;
3.      else if (last == first)
4.          if (K == E[first])
5.              index = first;
6.          else
7.              index = -1;
8.      else
9.          int mid = (first + last) / 2;
10.         if (K ≤ E[mid])
11.             index = binarySearch(E, first, mid, K);
12.         else
13.             index = binarySearch(E, mid+1, last, K);
14.     return index;
```

Compared to Algorithm 1.4 (Binary Search) in the text, this algorithm combines the tests of lines 5 and 7 into one test on line 10, and the left subrange is increased from mid-1 to mid, because mid might contain the key being searched for. An extra base case is needed in lines 3–7, which tests for exact equality when the range shrinks to a single entry.

Actually, if we can assume the precondition first $\leq$ last, then lines 1–2 can be dispensed with. This procedure propagates that precondition into recursive calls, whereas the procedure of Algorithm 1.4 does not, in certain cases.

1.44

The sequential search algorithm considers only whether x is equal to or unequal to the element in the array being examined, so we branch left in the decision tree for "equal" and branch right for "unequal." Each internal node contains the index of the element to which x is compared. The tree will have a long path, with n internal nodes, down to the right, labeled with indexes $1,\ldots,n$. The left child for a node labeled i is an output node labeled i. The rightmost leaf is an output node for the case when x is not found.

1.46

The probability that x is in the array is n/m.

Redoing the average analysis for Binary Search (Section 1.6.3) with the assumption that x is in the array (i.e., eliminating the terms for the gaps) gives an average of approximately $\lceil \lg n \rceil - 1$. (The computation in Section 1.6.3 assumes that $n = 2^k - 1$ for some k, but this will give a good enough approximation for the average in general.)

The probability that x is not in the array is $1 - n/m$. In this case (again assuming that $n = 2^k - 1$), $\lceil \lg n \rceil$ comparisons are done. So the average for all cases is approximately

$$\frac{n}{m}(\lceil \lg n \rceil - 1) + (1 - \frac{n}{m})\lceil \lg n \rceil = \lceil \lg n \rceil - \frac{n}{m} \approx \lceil \lg n \rceil .$$

(Thus, under various assumptions, Binary Search does roughly $\lg n$ comparisons.)

1.47

We examine selected elements in the array in increasing order until an entry larger than x is found; then do a binary search in the segment that must contain x if it is in the array at all. To keep the number of comparisons in $O(\log n)$, the distance between elements examined in the first phase is doubled at each step. That is, compare x to $E[1]$, $E[2]$, $E[4]$, $E[8],\ldots,E[2^k]$. We will find an element larger than x (perhaps **`maxint`**) after at most $\lceil \lg n \rceil + 1$ probes (i.e., $k \leq \lceil \lg n \rceil$). (If x is found, of course, the search terminates.) Then do the Binary Search in the range $E[2^{k-1}+1],\ldots,E[2^k - 1]$ using at most $k-1$ comparisons. (If $E[1] > x$, examine $E[0]$.) Thus the number of comparisons is at most $2k = 2\lceil \lg n \rceil$.

1.48

For $x > 0$, write $-x \ln x$ as $\dfrac{\ln x}{(\frac{-1}{x})}$. By L'Hôpital's rule the limit of this ratio as $x \to 0$ is the same as the limit of $\dfrac{\frac{1}{x}}{(\frac{1}{x^2})} = x$.

1.50

The strategy is to divide the group of coins known to contain the fake coin into subgroups such that, after one more weighing, the subgroup known to contain the fake is as small as possible, no matter what the outcome of the weighing. Obviously, two equal subgroups would work, but we can do better by making *three* subgroups that are as equal as possible. Then weigh two of those subgroups that have an equal number of coins. If the two subgroups that are weighed have equal weight, the fake is in the third subgroup.

Round 1	23, 23, 24	
Round 2	8, 8, 8	(or 8, 8, 7)
Round 3	3, 3, 2	(or 3, 2, 2)
Round 4	1, 1, 1	(or 1, 1, 0)

So four weighings suffice.

Chapter 2
Data Abstraction and Basic Data Structures

Section 2.2: *ADT Specification and Design Techniques*

2.2
Several answers are reasonable, provided that they bring out that looking at the current *implementation* of the ADT is a bad idea.

Solution 1. **`GorpTester.java`** would be preferable because the **`javadoc`** utility (with an html browser) permits the ADT specifications and type declarations to be inspected without looking at the source code of **`Gorp.java`**. Trying to infer what the ADT operations do by looking at the implementation in **`Gorp.java`** is not reliable. However, even if the implementation in **`Gorp.java`** changes, as long as the ADT specifications do not change, then the behavior of **`GorpTester.java`** will remain unchanged, so it is a reliable source of information.

Solution 2. **`Gorp.java`** would be preferable because the javadoc comments in it should contain the preconditions and postconditions for each operation.

Section 2.3: *Elementary ADTs — Lists and Trees*

2.4
The proof is by induction on d, the depth of the node. For a given binary tree, let n_d denote the number of nodes at depth d. For $d = 0$, there is only 1 root node and $1 = 2^0$.

For $d > 0$, assume the formula holds for $d-1$. Because each node at depth $d-1$ has at most 2 children,

$$n_d \leq 2\,n_{d-1} \leq 2\,(2^{d-1}) \leq 2^d$$

2.6
A tree of height $\lceil \lg(n+1) \rceil - 2$ would, by Lemma 2.2, have a t most $2^{\lceil \lg(n+1) \rceil - 1} - 1$ nodes. But

$$2^{\lceil \lg(n+1) \rceil - 1} - 1 < 2^{\lg(n+1)} - 1 = n + 1 - 1 = n$$

Because that expression is less than n, any tree with n nodes would have to have height at least $\lceil \lg(n+1) \rceil - 1$.

2.8
The point of the exercise is to recognize that a binary tree in which all the left subtrees are nil is logically the same as a list. We would give full credit, or almost full credit, to any solution that brings out that idea, without worrying too much about details specific to Java, which can get somewhat complicated. (It is also possible to make all the right subtrees nil and use the left subtrees to contain the elements, but this is less natural.) We will give several solutions to demonstrate the range of possibilities. Solutions 2 and 3 have been compiled and tested in Java 1.2.

Solution 1. This solution merely uses the **`BinTree`** functions and does not even return objects in the **`List`** class, so all the objects are in the class **`BinTree`**. Therefore, this solution doesn't really meet the specifications of the List ADT. Its virtue is simplicity. (In C, the type discrepancy can be solved with a type cast; Java is more strict.)

```
public class List
  {
    public static final BinTree nil = BinTree.nil;

    public static  Object first(BinTree aList) { return BinTree.root(aList); }

    public static  BinTree rest(BinTree aList) { return BinTree.rightSubtree(aList); }

    public static  BinTree cons(Object newElement, BinTree oldList)
      { return BinTree.buildTree(newElement, BinTree.nil, oldList); }
  }
```

Solution 2. This solution creates objects in the List class, but each List object has a single instance field, which is the binary tree that represents that list. This solution uses only Java features covered in the first two chapters of the text. However, it has a subtle problem, which is pointed out after the code.

```
public class List
  {
    BinTree listAsTree;

    public static final List nil = makeNil();

    private static
    List makeNil()
      {
List newL = new List();
newL.listAsTree = BinTree.nil;
return newL;
      }

    public static
    Object first(List aList)
      {
        return BinTree.root(aList.listAsTree);
      }

    public static
    List rest(List aList)
      {
        List newL = new List();
        newL.listAsTree = BinTree.rightSubtree(aList.listAsTree);
        return newL;
      }

    public static
    List cons(Object newElement, List oldList)
      {
        List newL = new List();
        BinTree oldTree;
        if (oldList == List.nil)
          oldTree = BinTree.nil;
        else
          oldTree = oldList.listAsTree;
        newL.listAsTree = BinTree.buildTree(newElement, BinTree.nil,
                                            oldTree);
        return newL;
      }
  }
```

The problem with this implementation is that **`rest`** creates a new object instead of simply returning information about an existing object, so the following postcondition does not hold:

`rest( cons(newElement, oldList)) == oldList.`

There is no reasonable way to get around this problem without using subclasses, which are covered in the appendix.

Solution 3. This solution is the "most correct" for Java, but the fine points of Java are tangential to the purpose of the exercise. This solution is included mainly to confirm the fact that it *is* possible to do sophisticated data abstraction with type safety, encapsulation, and precise specifications in Java.

The technique follows appendix Section A.6 in which the class **`IntList`** is extended to **`IntListA`**. Here we extend **`BinTree`** to **`List`**. We assume that the **`BinTree`** class has been defined to permit subclasses with the appropriate **`protected`** nondefault constructor, in analogy with the nondefault **`IntList`** constructor in Fig. A.11. The nondefault **`BinTree`** constructor is accessed by **`super`** in the code below.

```
public class List extends BinTree
  {
    public static final List nil = (List)BinTree.nil;

    public static
    Object first(List aList)
      {
        return BinTree.root(aList);
      }

    public static
    List rest(List aList)
      {
        return (List)BinTree.rightSubtree(aList);
      }

    public static
    List cons(Object newElement, List oldList)
      {
        List newList = new List(newElement, oldList);
return newList;
      }

    protected
    List(Object newElement, List oldList)
      {
        super(newElement, BinTree.nil, oldList);
      }
  }
```

2.10

```
Tree t = Tree.buildTree("t", TreeList.nil);

Tree u = Tree.buildTree("u", TreeList.nil);

TreeList uList = TreeList.cons(u, TreeList.nil);

TreeList tuList = TreeList.cons(t, uList);

Tree q = Tree.buildTree("q", tuList);

Tree r = Tree.buildTree("r", TreeList.nil);

Tree v = Tree.buildTree("v", TreeList.nil);

TreeList vList = TreeList.cons(v, TreeList.nil);

Tree s = Tree.buildTree("s", vList);

TreeList sList = TreeList.cons(s, TreeList.nil);

TreeList rsList = TreeList.cons(r, sList);

TreeList qrsList = TreeList.cons(q, rsList);

Tree p = Tree.buildTree("p", qrsList);
```

2.12

A newly created node has one node (itself) in its in-tree, so use 1 for the initial node data.

```
InTreeNode makeSizedNode()
{
   return InTreeNode.makeNode(1);
}
```

When setting the parent of a node, we must first remove it from its current parent's tree. This decreases the size of our parent, our parent's parent, and so on all the way up the tree. Similarly, when attaching to the new parent, we must add to the sizes of the new parent and all of its ancestors.

```
void setSizedParent(InTreeNode v, InTreeNode p)
{
   InTreeNode ancestor = InTreeNode.parent(v);
   while (ancestor != InTreeNode.nil) {
      int ancestorData = InTreeNode.nodeData(ancestor);
      ancestorData -= InTreeNode.nodeData(v);
      InTreeNode.setNodeData(ancestor, ancestorData);
   }

   InTreeNode.setParent(v, p);
   ancestor = InTreeNode.parent(v);
   while (ancestor != InTreeNode.nil) {
      int ancestorData = InTreeNode.nodeData(ancestor);
      ancestorData += InTreeNode.nodeData(v);
      InTreeNode.setNodeData(ancestor, ancestorData);
   }
}
```

Section 2.4: *Stacks and Queues*

2.14

```
public class Stack
{
   List theList;

   public static
   Stack create()
   {
      Stack newStack = new Stack();
      newStack.theList = List.nil;
      return newStack;
   }

   public static
   boolean isEmpty(Stack s)
   {
      return (s.theList == List.nil);
   }

   public static
   Object top(Stack s)
   {
      return List.first(s.theList);
   }

   public static
   void push(Stack s, Object e)
```

```
   {
      s.theList = List.cons(e, s.theList);
   }

   public static
   void pop(Stack s)
   {
      s.theList = List.rest(s.theList);
   }
}
```

Each Stack operation runs in $O(1)$ time because it uses at most 1 List operation plus a constant number of steps.

2.16

a) The precondition for **enqueue** would be that the total number of enqueues minus the total number of dequeues must be less than n.

b) An array of n elements is used because there can never be more than n elements in the queue at a time. The indices for the front and back of the queue are incremented modulo n to circle back to the front of the array once the end is reached. It is also necessary to keep a count of the number of elements used, because **frontIndex == backIndex** both when the queue is empty and when it has n elements in it. It could also be implemented without the count of used elements by making the array size $n+1$ because **frontIndex** would no longer equal **backIndex** when the queue was full.

```
public class Queue
{
   Object[] storage;
   int size, frontIndex, backIndex, used;

   public static
   Queue create(int n)
   {
      Queue newQueue = new Queue();
      newQueue.storage = new Object [n];
      newQueue.size = n;
      newQueue.front = newQueue.back = newQueue.used = 0;
      return newQueue;
   }

   public static
   boolean isEmpty(Queue q)
   {
      return (q.used == 0);
   }

   Object front(Queue q)
   {
      return q.storage[frontIndex];
   }

   public static
   void enqueue(Queue q, Object e)
   {
      storage[backIndex] = e;
      backIndex = (backIndex + 1) % size;
      used++;
   }
```

```
      public static
      void dequeue(Queue q)
      {
         frontIndex = (frontIndex + 1) % size;
         used--;
      }
   }
```

c) Incrementing **`frontIndex`** and **`backIndex`** modulo *n* would be unnecessary, because they could never wrap around. Also, storing **`used`** would be unnecessary, as **`isEmpty`** could just return **`(frontIndex == backIndex)`**.

Additional Problems

2.18

The main procedure, **`convertTree`**, breaks the task up into two subtasks: finding the node degrees and building the out-tree.

```
Tree convertTree(InTreeNode[] inNode, int n)
{
   // initialize remaining to be the number of children for each node
   int[] remaining = getNodeDegrees(inNode, n);

   // construct the out-tree bottom-up, using remaining for bookkeeping
   return buildOutTree(inNode, n, remaining);
}

void getNodeDegrees(InTreeNode[] inNode, int n)
{
   int[] nodeDegrees = new int [n+1];

   for (int i = 1; i <= n; ++i) {
      nodeDegrees[i] = 0;
   }

   // calculate nodeDegrees to be the number of children for each node
   for (int i = 1; i <= n; ++i) {
      if (! InTreeNode.isRoot(inNode[i])) {
         InTreeNode parent = InTreeNode.parent(inNode[i]);
         int parentIndex = InTreeNode.nodeData(parent);
         nodeDegrees[parentIndex]++;
      }
   }
   return nodeDegrees;
}

Tree buildOutTree(InTreeNode[] inNode, int n, int[] remaining)
{
   Tree outputTree;
   TreeList[] subtrees = new TreeList [n+1];

   for (int i = 1; i <= n; ++i) {
      subtrees[i] = TreeList.nil;
   }

   // nodes with no children are already "done" and
   // can be put into the sources stack
   Stack sources = Stack.create();
   for (int i = 1; i <= n; ++i) {
      if (remaining[i] == 0) {
```

```
         Stack.push(sources, i);
      }
   }

   // Process elements from sources, adding them to the TreeList for
   // their parent.  When the stack is empty, all nodes will have
   // been processed and we're done.
   while ( ! Stack.isEmpty(sources) ) {
      int sourceIndex = Stack.top(sources);
      Stack.pop(sources);
      InTreeNode sourceNode = inNode[sourceIndex];
      Tree sourceTree = Tree.buildTree(sourceIndex, subtrees[sourceIndex]);
      if (InTreeNode.isRoot(sourceNode)) {
         // If this source node was the root of the input tree,
         // it is also the root of our output tree.
         outputTree = sourceTree;
      }
      else {
         InTreeNode parentNode = InTreeNode.parent(sourceNode);
         int parentIndex = InTreeNode.nodeData(parentNode);
         subtrees[parentIndex] = TreeList.cons(sourceTree, subtrees[parentIndex]);
         remaining[parentIndex]--;
         if (remaining[parentIndex] == 0)
            Stack.push(sources, parentIndex);
      }
   }

   return outputTree;
}
```

The **buildOutTree** procedure can also be implemented without the auxiliary stack, **sources**, thereby saving some temporary space. It uses a form of the sweep and mark technique that is introduced for graph traversals in Chapter 7.

```
Tree buildOutTree(InTreeNode[] inNode, int n, int[] remaining)
{
   Tree outputTree;
   TreeList[] subtrees = new TreeList [n+1];

   for (int i = 1; i <= n; ++i) {
      subtrees[i] = TreeList.nil;
   }

   // Sweep through remaining[] looking for sources.  When a source
   // is found, it is immediately processed.  If processing a node
   // causes its parent to become a source, then process the parent
   // immediately (instead of pushing it on the stack).
   // Mark processed sources with -1 to prevent possible reprocessing
   // later in the sweep.
   for (int i = 1; i <= n; ++i) {
      int currentNode = i;
      while (remaining[currentNode] == 0) {
         remaining[currentNode] = -1; // mark as done
         InTreeNode sourceNode = inNode[currentNode];
         Tree sourceTree = Tree.buildTree(currentNode, subtrees[currentNode]);
         if (InTreeNode.isRoot(sourceNode)) {
            // If this source node was the root of the input tree,
            // it is also the root of our output tree.
            outputTree = sourceTree;
         }
```

```
            else {
               InTreeNode parentNode = InTreeNode.parent(sourceNode);
               int parentIndex = InTreeNode.nodeData(parentNode);
               subtrees[parentIndex] = TreeList.cons(sourceTree,
                                                     subtrees[parentIndex]);
               remaining[parentIndex]--;
               if (remaining[parentIndex] == 0) {
                  // if the parent is now a source, move up to process it
                  currentNode = parentIndex;
               }
            }
         }
      }

      return outputTree;
   }
```

Chapter 3
Recursion and Induction

***Section 3.2*:** *Recursive Procedures*

3.2
Combining parts 3 and 2 of Lemma 1.4, it suffices to show that the second derivative of $x \lg x$ is ≥ 0 for $x > 0$.

$$\frac{d^2}{dx^2}(x \lg x) \;=\; (\lg e)\frac{d^2}{dx^2}(x \ln x) \;=\; (\lg e)\frac{d}{dx}(\ln x + 1) \;=\; (\lg e)\frac{1}{x} > 0$$

3.4
Statement 2 is the correct one. The proof is by induction on n, the parameter of F. The base cases are $n = 1$ and $n = 2$. For $n = 1$, $F(1) = 1 \geq .01\left(\frac{3}{2}\right)^1 = .015$. For $n = 2$, $F(2) = 1 \geq .01\left(\frac{3}{2}\right)^2 = .0225$.

For $n > 2$, assume that $F(k) \geq .01\left(\frac{3}{2}\right)^k$ for all $1 \leq k < n$. [*Note that the range of k includes the base cases and is strictly less than the main induction variable; k is our* auxiliary variable.] Then letting $k = n-1$ in the first case below and $k = n-2$ in the second case below [*Notice that both these values are in the correct range for k, but would not be if the inductive cases began with* $n = 2$.],

$$F(n-1) \;\geq\; .01\left(\frac{3}{2}\right)^{n-1}$$
$$F(n-2) \;\geq\; .01\left(\frac{3}{2}\right)^{n-2}$$

In this range of n we are given that $F(n) = F(n-1) + F(n-2)$, so

$$\begin{aligned} F(n) \;&\geq\; .01\left(\frac{3}{2}\right)^{n-1} + .01\left(\frac{3}{2}\right)^{n-2} \\ &=\; .01\left(\frac{3}{2}\right)^{n}\left(\frac{2}{3}+\frac{4}{9}\right) \;=\; .01\left(\frac{3}{2}\right)^{n}\left(\frac{10}{9}\right) \;>\; .01\left(\frac{3}{2}\right)^{n}. \end{aligned}$$

So the claim holds for all $n > 2$ also.

***Section 3.5*:** *Proving Correctness of Procedures*

3.6
We give the proof for part (a) with the added statements needed for part (b) enclosed in square brackets.

We use the lexicographic order on pairs of natural numbers (nonnegative integers). That is, $(m,n) < (x,y)$ if $m < x$ or if $m = x$ and $n < y$. The proof is by induction on (m,n) in this order.

The base cases are pairs of natural numbers of the form $(0,n)$, where $n > 0$. (Recall that the precondition of **`gcd`** eliminates $(0,0)$.) In these cases **`gcd(0, n)`** returns n. By the definition of *divisor*, n is a divisor of both 0 and n, so n is a common divisor. [For part (b): No integer greater than n can be a divisor of n, so n is also the *greatest* common divisor.]

For pairs of natural numbers (m,n) with $m > 0$, assume the **`gcd`** function meets its objectives for pairs $(0,0) < (i,j) < (m,n)$ in the lexicographic order. There are two cases in the procedure: $m > n$ and $m \leq n$.

If $m > n$, **`gcd(m, n)`** returns the same value as **`gcd(n, m)`**. But $(0,0) < (n,m) < (m,n)$, so by the inductive hypothesis **`gcd(n, m)`** is a common divisor of m and n. [For part (b): the *greatest* common divisor.] But common divisors and the greatest common divisor don't depend on order, so this value is correct for **`gcd(m, n)`**.

If $m \leq n$, the function returns **`gcd(m, nLess)`**, where **`nLess`** $= n - m \geq 0$. But $(0,0) < (m,$**`nLess`**$) < (m,n)$, so by the inductive hypothesis, **`gcd(m, nLess)`** returns a common divisor of m and **`nLess`**. [For part (b): the *greatest* common divisor.] Define d to be the value returned by **`gcd(m, nLess)`**. Then $n/d =$ **`nLess`**$/d + m/d$ has no remainder, so d is a common divisor of m and n. This concludes the proof for part (a). [For part (b): Let h be *any* common divisor of m and n. It remains to prove that $h \leq d$. But **`nLess`**$/h = n/h - m/h$ also has no remainder, so h is a common divisor of m and **`nLess`**. Therefore, $h \leq d$.]

***Section 3.6*:** *Recurrence Equations*

3.8

Solution 1. Applying the Master Theorem (Theorem 3.17 of the text), we find that $b = 1$, $c = 2$, $E = 0$, and $f(n) = cn \in \Theta(n^1)$. Case 3 applies, so $W(n) \in \Theta(cn) = \Theta(n)$.

Solution 2. Using the assumption that n is a power of 2, we can get an estimate for $W(n)$ that will be of the same asymptotic order. To get the formula for $W(n)$, we look at the recursion tree or expand a few terms to see the pattern.

$$\begin{aligned} W(n) &= cn + W(n/2) = cn + cn/2 + W(n/4) = cn + cn/2 + cn/4 + W(n/8) \\ &= cn \sum_{i=0}^{\lg n - 1} \frac{1}{2^i} + 1 = cn \frac{(\frac{1}{2})^{\lg n} - 1}{\frac{1}{2} - 1} + 1 \quad \text{(using Equation 1.9 of the text)} \\ &= cn \frac{1 - 1/n}{\frac{1}{2}} + 1 = 2cn(1 - 1/n) + 1 = 2c(n-1) + 1 \end{aligned}$$

So $W(n) \in \Theta(n)$.

3.10

Notation is consistent with the Master Theorem (Theorem 3.17 of the text) and/or Exercise 3.9 and Eq. 3.14.

a) $b = 1, c = 2, E = 0$, $f(n) = c \lg n \in \Theta(\log^1 n)$. So use Eq. 3.14 with $a = 1$. $T(n) \in \Theta(\log^2 n)$.

b) Same as Exercise 3.8. $T(n) \in \Theta(n)$.

c) $b = 2$, $c = 2$, $E = 1$, $f(n) = cn \in \Theta(n^1)$. By case 2 of the Master Theorem (or by Eq. 3.14 with $a = 0$), $T(n) \in \Theta(n \log n)$.

d) $b = 2, c = 2, E = 1$, $f(n) = cn \lg n \in \Theta(n^1 \log^1 n)$. So use Eq. 3.14 with $a = 1$. $T(n) \in \Theta(n \log^2 n)$.

e) $b = 2, c = 2, E = 1$, $f(n) = cn^2 \in \Theta(n^2)$. By case 3 of the Master Theorem, $T(n) \in \Theta(n^2)$.

Additional Problems

3.12

The recurrence equation for the number of disk moves is:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \qquad \text{for } n > 0 \\ T(0) &= 0 \end{aligned}$$

Use Eq. 3.12 as the solution of Eq. 3.11 with $b = 2$, $c = 1$, $f(n) = 1$ for $n \geq 1$, and $f(0) = 0$:

$$\begin{aligned} T(n) &= 2^n \sum_{h=1}^{n} \frac{1}{2^h} = 2^n \left(\frac{\frac{1}{2} - \frac{1}{2^{n+1}}}{1 - \frac{1}{2}} \right) \\ &= 2^n - 1 \end{aligned}$$

Alternatively, we can see by inspection that the recursion tree is a complete binary tree of height n, that all the internal nodes have a nonrecursive cost of 1 and the leaves have a nonrecursive cost of 0. So the total nonrecursive cost over the whole tree is $2^n - 1$.

Chapter 4
Sorting

***Section 4.1*:** *Introduction*

4.2

a) A worst case occurs when the keys are in decreasing order; the **`for`** loop is executed for each value of **`numPairs`** from $n-1$ down to 1. The number of key comparisons is $n(n-1)/2$.

b) If the keys are already sorted the **`for`** loop is executed once, doing $n-1$ comparisons and no interchanges.

4.4

a) Suppose the last interchange on some pass is done at the jth and $j+1$st positions. Then the keys in positions $j+1$ through n must be in order. If they were not, there would be a key that immediately preceded a smaller key, and those two would have been interchanged. We need to show that none of these keys belongs in earlier positions in the array. Let x be the largest key from among $E[0],\ldots,E[j+1]$. (If there may be duplicate keys, consider the last instance of x in this range.) In the previous pass through the **`while`** loop, x would always have interchanged with the next position, so it must have reached the $j+1$st position. Hence the smallest key among $E[j+1],\ldots,E[n-1]$ is greater than or equal to all keys among $E[0],\ldots,E[j+1]$, so $E[j+1],\ldots,E[n-1]$ are in their proper positions.

b) The idea is to keep track of the position where the last interchange occurred and use that position to reset **`numPairs`** instead of simply decrementing **`numPairs`** by 1. We can eliminate the variable **`didSwitch`**.

```
last = n-1;
while (last > 0)
     numPairs = last;
     last = -1;
     for (j = 0; j < numPairs; j++)
          if (E[j] > E[j+1])
               Interchange E[j] and E[j+1].
               last = j;
```

c) No, if the keys are in reverse order, the algorithm will still execute the **`for`** loop for each value of **`numPairs`** from $n-1$ down to 1.

***Section 4.2*:** *Insertion Sort*

4.6

1, n, $n-1,\ldots,2$ and n, $n-1,\ldots,3$, 1, 2.

4.9

The average will be lower. The **`while`** loop (in **`shiftVac`**) that finds the place for x stops if it finds a key equal to x.

4.11

This code uses **`insert1`**, which appears in Figure 2.6 of the text; the subroutine **`insert1()`** inserts one new element into a sorted linked list, using partial rebuilding.

```
IntList listInsSort(IntList unsorted)
   IntList sorted, remUnsort;

   sorted = nil;
   remUnsort = unsorted;
   while (remUnsort ≠ nil)
        int newE = first(remUnsort);
        sorted = insert1(newE, sorted);
        remUnsort = rest(remUnsort);
   return sorted;
```

Analysis: Assume **unsorted** has n elements. When **sorted** has k elements in it, then the call to **insert1** has a best-case time of $\Theta(1)$ and average and worst-case times of $\Theta(k)$. (This assumes that **first**, **rest**, and **cons** are implemented in constant time. It also assumes that any garbage collection takes time proportional to the number of garbage (unreferenced) objects collected.) The subroutine **insert1** is called with k varying from 0 to $n-1$. Therefore, for the overall procedure, the best-case time is $\Theta(n)$, and average and worst-case times are $\Theta(n^2)$.

The space usage depends on how many garbage (unreferenced) objects are created during partial rebuilding. With an automatic garbage collector, it is reasonable to assume this is a constant. Otherwise, the number of garbage objects is proportional to the running time, and is in $\Theta(n^2)$ in the worst case. In addition, since **insert1** might have a depth of recursion that is about k and k can be as large as $n-1$, linear space is needed for the recursion stack in the average and worst cases. Note that space required for the output is not counted in this analysis.

With a "mutable" list ADT it would be possible to avoid the creation of garbage objects by changing their instance fields. However, the exercise specified that the list ADT given in the text be used.

Section 4.4: *Quicksort*

4.13

```
/** Postcondition for extendSmallRegion:
   * The leftmost element in E[low], ..., E[highVac-1]
   * whose key is ≥ pivot is moved to E[highVac] and
   * the index from which it was moved is returned.
   * If there is no such element, highVac is returned.
   */
```

4.15

$n(n-1)/2$ key comparisons; $2(n-1)$ element movements (**E[first]** is moved to **pivotElement** and then moved back each time **partition** is called).

4.17

Consider a subrange of k elements to be partitioned. Choosing the pivot requires 3 key comparisons. Now $k-3$ elements remain to be compared to the pivot (it is easy to arrange not to compare elements that were candidates for the pivot again by swapping them to the extremes of the subrange). So k comparisons are done by the time partition is finished. This is one more than if **E[first]** is simply chosen as the pivot.

An example of the worst case might be **E[first]** = 100, **E[(first+last)/2]** = 99, **E[last]** = 98, and all other elements are over 100. Then 99 becomes the pivot, 98 is the only element less than the pivot, and $k-2$ elements are greater than the pivot. The remaining keys can be chosen so the the worst case repeats; that is, we can arrange that the median-of-3 always comes out to be the second-smallest key in the range.

Since the larger subrange is reduced by only 2 at each depth of recursion in **quickSort**, it will require $n/2$ calls to **partition**, with ranges of size n, $n-2$, $n-4$, ... So, using the pairing-up trick mentioned after Eq. 1.5 in the text, the total number of comparisons is about $n^2/4$.

An array that is initially in reverse order also requires $\Theta(n^2)$ comparisons with the median-of-3 strategy for choosing the pivot.

4.19

Note: Students might give several different answers depending on whether certain moves are counted, but we would give full credit (or almost) for any of the reasonable variants that demonstrated a correct understanding of what is happening. The "take home" message is that **`partition`** does a linear number of moves when the initial sequence is in reverse order, whereas **`partitionL`** does a quadratic number of moves in this case.

a) After the first call: 1, 9, 8, 7, 6, 5, 4, 3, 2, 10 (10 was the pivot). One move was done in **`partition`**.
After the second call: the same (1 was the pivot). No moves in **`partition`**.
Total: $n/2$ moves in **`partition`** calls, plus $2(n-1)$ moves for pivots in **`quickSort`** itself.

b) After the first call: 9, 8, 7, 6, 5, 4, 3, 2, 1, 10 (10 was the pivot). 18 moves ($2(n-1)$).
After the second call: 8, 7, 6, 5, 4, 3, 2, 1, 9, 10 (9 was the pivot). 16 moves ($2(n-2)$).
Total: $2\sum_{i=1}^{n-1} i = (n-1)n$ moves. (Half of these are moves "in place," done in Line 5 of **`partitionL`**, and can be avoided by doing a test. Although the test would probably save some time in this worst case, it is far from clear that it would save time in typical cases. As before, **`quickSort`** itself does $2(n-1)$ moves.

c) The only advantage of **`partitionL`** is its simplicity. In most cases it does more element movement than **`partition`**. Parts (a) and (b) illustrated an extreme difference, where **`partitionL`** did 90 moves while **`partition`** did only 5, for the complete run of Quicksort. In both cases **`quickSort`** itself did an additional 20 moves.

4.21

The analysis assumes that all initial permutations are equally likely and all elements are distinct. The question addresses how many moves are done by Quicksort, including its subroutines. The body of Quicksort does two moves of the pivot element in connection with each call to **`partitionL`** or **`partition`**, whichever is used. Since there are at most $n-1$ such calls in the entire run, we shall see that these moves may be neglected in determining the leading term for the average number of moves done by the algorithm; that is, almost all of the moves occur in **`partitionL`** or **`partition`**.

a) Whenever the "unknown" element is less than the pivot, two moves occur in **`partitionL`**. The probability of this event is $\frac{1}{2}$. So $2(k-1)/2$ moves occur on average in partitioning a range of k elements. This is equal to the average number of comparisons. Thus the total number of moves done by Quicksort on average, using **`partitionL`**, is approximately $1.4n\lg n$ plus a small linear term.

b) For **`partition`**, each "unknown" element is examined by **`extendSmallRegion`** or **`extendLargeRegion`** (but not both). In **`extendSmallRegion`** an "unknown" element is moved if it is $\geq$ the pivot, the probability of which is $\frac{1}{2}$. In **`extendLargeRegion`** an "unknown" element is moved if it is $<$ the pivot, the probability of which is $\frac{1}{2}$. So $(k-1)/2$ moves occur on average. This is only half the average number of comparisons. Thus the total number of moves done by Quicksort on average, using **`partition`**, is approximately $.7n\lg n$ plus a small linear term. The relatively small number of element movements is the source of Quicksort's speed in practice.

c) Quicksort with **`partition`** as given in the text does about $1.4n\lg n$ key comparisons but only about $0.7n\lg n$ element movements. Mergesort always does an element movement after each key comparison and so does about $n\lg n$ element movements on average, assuming the optimization described in Exercise 4.27 is done. This number is doubled if all elements to be merged are first copied into a work area, as suggested in the text, before Algorithm 4.5 on page 175. However, if only half of the elements to be merged are copied, as described in Section 4.5.3, then the total is about $1.5n\lg n$.

We also mention that if the average number of comparisons in Quicksort is lower, the average number of moves goes down proportionally, as far as the leading term is concerned. Thus using the median-of-3 method for picking the pivot element reduces both the average number of comparisons and the average number of moves.

***Section 4.5*:** *Merging Sorted Sequences*

4.23

This algorithm is simple and elegant, but it uses linear space in the frame stack (unless the compiler performs tail-recursion optimization).

```
IntList listMerge(IntList in1, IntList in2)
    IntList merged;
    if (in1 == nil)
        merged = in2;
    else if (in2 == nil)
        merged = in1;
    else
        int e1 = first(in1);
        int e2 = first(in2);
        IntList remMerged;
        if (e1 ≤ e2)
            remMerged = listMerge(rest(in1), in2);
            merged = cons(e1, remMerged);
        else
            remMerged = listMerge(in1, rest(in2));
            merged = cons(e2, remMerged);
    return merged;
```

Some people prefer a less verbose style, such as:

```
IntList listMerge(IntList in1, IntList in2)
    { return
        in1 == nil ?  in2 :
        in2 == nil ?  in1 :
        first(in1) ≤ first(in2) ?
            cons(first(in1), listMerge(rest(in1), in2)) :
            cons(first(in2), listMerge(in1, rest(in2)));
    }
```

4.25

Solution 1 (follows hint in text). Let $P(k,m)$ be the number of possible permutations of the merged sequences where the first sequence has k elements and the second has m. Let $n = k+m$. If $A[0] < B[0]$, the subproblem remaining after moving $A[0]$ to the output area has $P(k-1,m)$ possible permutations. If $A[0] > B[0]$, the subproblem remaining after moving $B[0]$ to the output area has $P(k,m-1)$ possible permutations. (If $A[0] = B[0]$, it doesn't matter how the tie is broken; one of the two above subproblems remains.) Therefore, for $k > 0$ and $m > 0$, we have $P(k,m) = P(k-1,m) + P(k,m-1)$. Also, if $k = 0$ or $m = 0$ there is only 1 permutation. By Exercise 1.2 in the text we see that $P(k,m) = \binom{n}{k}$ is a solution to this recurrence equation and satisfies the boundary conditions.

Solution 2. There are $m+k$ slots in the output array. Once the slots for the k keys from the first array are determined, the keys from the second array must fill the remaining slots in order. Thus the number of different output arrangements is the number of ways to choose k slots from $m+k$, i.e., $\binom{m+k}{k}$.

Section 4.6*: *Mergesort

4.26

If the keys are already sorted, merging the two halves will need only $n/2$ comparisons. So the recurrence relation is

$$W(n) = W(\lceil n/2 \rceil) + W(\lfloor n/2 \rfloor) + \lceil n/2 \rceil \quad \text{and} \quad W(1) = 0$$

For simplicity, assume that n is a power of 2. Then, by expanding a few terms it is easy to see that

$$W(n) = \sum_{i=1}^{k} \frac{n}{2} + 2^k W(n/2^k).$$

Let $k = \lg n$; then $W(n) = \frac{n}{2} \lg n$.

4.27

a) Mergesort is revised to have a work array as an additional parameter. A wrapper might be used to allocate the work array:

```
mergeSortWrap(Element[] E, int first, int last)
    Element[]  W = new Element[E.length];
    mergeSort(E, E, W, first, last);
```

Now the idea is that the revised **mergeSort**, shown below, anticipates that its output needs to be in **out** (its second parameter), and so it arranges for the recursive calls to put their output in two subranges of **work** (its third parameter). Then **mergeSort** calls **merge** to merge from the two subranges of **work** into the desired subrange of **out**.

```
mergeSort(Element[] in, Element[] out, Element[] work, int first, int last)
    if (first == last)
        if (in ≠ out)
            out[first] = in[first];
        // else out[first] already = in[first].
    else
        mid = (first + last) / 2;
        mergeSort(in, work, out, first, mid);
        mergeSort(in, work, out, mid+1, last);
        merge(work, first, mid, last, out);
```

Although there are three parameters (**in**, **out**, **work**), there are only two distinct arrays, **E** and **W**. The changes to **merge**, compared to Algorithm 4.4 in the text, are straightforward.

b) Now **merge** does one move per comparison because its input and output areas are different, so the average for Mergesort, as optimized here, is about $n \lg n$ element movements.

Exercise 4.21 showed that Quicksort does about $0.7 n \lg n$ element movements on average.

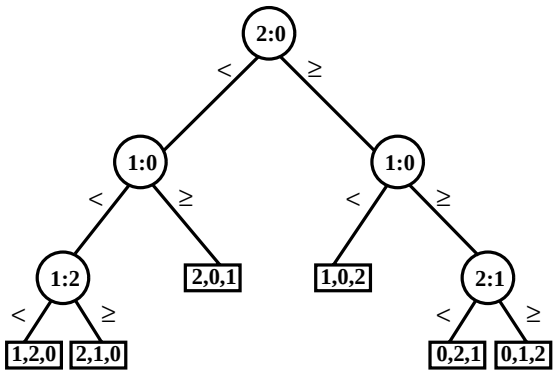
4.29

There are 2^{D-1} nodes at depth $D-1$ in the recursion tree. If B of these have no children, then $(2^{D-1} - B)$ have two children each. So there are $2(2^{D-1} - B)$ leaves (base cases for Mergesort) at depth D. Altogether there are n bases cases for Mergesort, so $n = 2(2^{D-1} - B) + B = 2^D - B$.

***Section 4.7*:** *Lower Bounds for Sorting by Comparison of Keys*

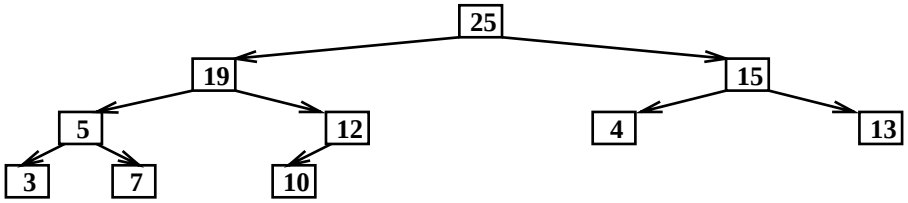
4.31

The comparisons are done in the **partition** algorithm (Algorithm 4.3), actually in its subroutines. In each internal node in the decision tree below, the second index is the index of the pivot element.



***Section 4.8*:** *Heapsort*

4.34
Arranging the elements in a binary tree:



No, it's not a heap, because 5 is smaller than its child 7.

4.35
The array containing the heap, beginning at index 1, is:

Y	T	X	P	O	E	M	I	C	L

14 key comparisons are needed.

4.37

a) 9.

b) Since the keys are in decreasing order, FixHeap does only one or two key comparisons each time it is called; no keys move. If n is even, the last internal node has only one child, so FixHeap does only one key comparison at that node. For each other internal node, two key comparisons are done. If n is odd, each of the $(n-1)/2$ internal nodes has two children, so two key comparisons are done by FixHeap for each one. In both cases, the total is $n-1$.

c) Best case.

4.40
The statement to be added is

```
Interchange E[1] and E[2].
```

The overhead of an extra call to FixHeap is eliminated, but there is no decrease in the number of key comparisons.

4.42

a) $K =$ **H[100]** $= 1$. "Pull out" 100 from **H[1]**.
As **fixHeapFast** begins, **vacant** $= 1$ and $h = 6$.

Compare 99, 98; move 99 to **H[1]**; **vacant** is at **H[2]**.
Compare 97, 96; move 97 to **H[2]**; **vacant** is at **H[4]**.
Compare 93, 92; move 93 to **H[4]**; **vacant** is at **H[8]**.

Compare K with parent of **H[8]**, which contains 93.
K is smaller, so keep going with $h = 3$.

Compare 85, 84; move 85 to **H[8]**; **vacant** is at **H[16]**.
Compare 69, 68; move 69 to **H[16]**; **vacant** is at **H[32]**.

Compare K with parent of **H[32]**, which contains 69.
K is smaller, so so keep going with $h = 1$.

Compare 37, 36; then compare K to 37.
K is smaller, so move 37 to **H[32]** and insert K in **H[64]**.

b) $4+3+2=9$.

c) **`fixHeap`** does 12 comparisons, 2 at each level except the leaf.

4.44

The goal is to show that

$$\lceil \lg(\lfloor \tfrac{1}{2}h \rfloor + 1) \rceil + 1 = \lceil \lg(h+1) \rceil \qquad \text{for all integers } h \geq 1$$

If $h = 2^k - 1$, where $k \geq 1$ is an integer, then $\lfloor \frac{1}{2}h \rfloor = 2^{k-1} - 1$ and the left side $= k$. Also, the right side $= k$. Otherwise,

$$\lceil \lg(\lfloor \tfrac{1}{2}h \rfloor + 1) \rceil + 1 = \lceil \lg(\lfloor \tfrac{1}{2}h + 1 \rfloor) \rceil + 1 = \lceil \lg(2\lfloor \tfrac{1}{2}(h+2) \rfloor) \rceil$$

If h is odd, $2\lfloor \frac{1}{2}(h+2) \rfloor = h+1$, so the last expression equals $\lceil \lg(h+1) \rceil$, as was to be shown.

If h is even the last expression in the above equation equals $\lceil \lg(h+2) \rceil$. But $\lg(h+1)$ is not an exact integer in this case (here we use the fact that $h > 0$ as well as even), so $\lceil \lg(h+2) \rceil = \lceil \lg(h+1) \rceil$, again as was to be shown.

***Section 4.10*:** *Shellsort*

4.45

Suppose the largest of the 5 constants is k. Consider an initial sequence in the reverse of sorted order. The first phase of Shellsort (with increment k) will do about $\frac{1}{2}n^2/k^2$ comparisons to sort elements $0, k, 2k, \ldots$, and an equal number to sort elements $1, k+1, 2k+1, \ldots$, and so on. This phase will cost about $\frac{1}{2}n^2/k$, which is in $\Omega(n^2)$.

***Section 4.11*:** *Radix Sorting*

4.46

The radix (hence the number of buckets) is squared.

Additional Problems

4.48

a) Sort both the bills and the checks by phone number; then check through the piles for bills without corresponding checks. Assuming the number of bills and the number of checks are approximately equal, and letting n be the approximate number, this method does $\Theta(n \log n)$ steps.

   An alternative solution would be to sort just the bills, then do binary search for each check, marking the bills that are paid. One sequential pass through the bills finds those that are unpaid. Again, the number of steps is in $\Theta(n \log n)$.

b) Use an array of counters, one for each publisher. The list of publishers can be alphabetized, or a specialized hash function can be designed to compute the proper counter index given a publisher's name. Scan the list of books and increment the appropriate counter. If there are n books and the list of publishers is sorted and searched by binary search, the number of key comparisons is approximately $30\lceil \lg 30 \rceil - 1.4 * 30 + n\lceil \lg 30 \rceil = 5n + 108$, and the total number of steps is in $\Theta(n)$.

c) Enter the person's name or identification number from each record into an array. Sort this list. Then scan the list ignoring duplicates and count distinct people. (Depending on the sorting algorithm used, it may be convenient to eliminate duplicates during the sort.) The number of steps is in $\Theta(n \log n)$.

4.49

Expand the recurrence to see the pattern of the terms.

$$\begin{aligned} T(n) &= cn + \sqrt{n}T(\sqrt{n}) \\ &= cn + \sqrt{n}\left[c\sqrt{n} + n^{1/4}T(n^{1/4})\right] \\ &= cn + cn + n^{1/2}n^{1/4}T(n^{1/4}) \\ &= cn + cn + n^{1/2+1/4}\left[cn^{1/4} + n^{1/8}T(n^{1/8})\right] \\ &= cn + cn + cn + n^{1/2+1/4+1/8}T(n^{1/8}) \\ &= kcn + n^{1-1/2^k}T(n^{1/2^k}) \end{aligned}$$

The boundary condition is stated for $T(2)$. Thus we solve

$$2 = n^{1/2^k}$$

Raising both sides to the power 2^k gives

$$2^{2^k} = n.$$

Then taking logs twice gives

$$2^k = \lg n \quad \text{and} \quad k = \lg\lg n.$$

So

$$\begin{aligned} T(n) &= cn\lg\lg n + n^{1-1/2^{\lg\lg n}} \\ &= cn\lg\lg n + n^{1-1/\lg n} \quad \in \Theta(n\log\log n). \end{aligned}$$

4.51

a) Insertion Sort: Stable.
b) Maxsort: Not stable: 1, 2, 1, 1.
c) Bubblesort: Stable.
d) Quicksort: Not stable: 2, 3, 3, 1.
e) Heapsort: Not stable: 1, 1.
f) Mergesort: Stable.
g) Shellsort: Not stable: 2, 2, 1, 3, with $h_2 = 2$ and $h_1 = 1$.
h) Radix Sort: Stable.

Note: This exercise was updated in the second printing. Be sure that the errata in the first printing (available via Internet) are applied if you are using the first printing.

4.53

There are several reasonable answers for this open-ended question. What matters most is if the choices are supported by good reasons. Heapsort, Shellsort, and Radix Sort are all good answers.

Heapsort requires some implementation of a binary tree. With considerable trouble it might use the usual binary tree structure with edges directed away from the root. But there is no way to jump long distances in a linked list in constant time, whereas it is possible with an array. So **`fixHeap`**, for example, would require $\Omega(n)$ time in the worst case, rather than $\Omega(\log n)$, if the array were simulated by a linked list in a straightforward manner.

One could argue that the binary tree can be implemented with linked lists, and this is true, but this goes against the spirit of the question. There is no way to use linked lists in a natural manner to implement Heapsort in $\Theta(n\log n)$.

Shellsort cannot be easily adapted to linked lists, and would degrade to a quadratic-time sort in any straightforward adaptation. There is no easy way to jump by the appropriate increment in constant time to find the key to be compared.

However, as with Heapsort, where there's a will there's a way, and Shellsort can be adapted to linked lists within the same asymptotic order, but a sizable increase in the constant factor. To sort subsequences separated by an increment h, one *could* make a pass through the linked list and create h sublists, taking elements in rotation. (For bookkeeping, we would need a list of the sublists.) Then sort each of the sublists. (Actually, using Insertion Sort for the sorting, it can overlap with the creation of the h sublists.) Then go through the sublists in rotation and assemble them into one list, which is now h-sorted. These extra steps take only linear time per increment, so they do not increase the asymptotic order for the algorithm.

Radix Sort also depends intimately on the **`buckets`** array, which in general must have $\Theta(n)$ slots to achieve linear-time sorting. That is, it must be possible to access any element in this array in constant time. Although Radix

Sort uses lists as well as the **buckets** array, if it were required to use a list in place of the **buckets** array, it would degrade to quadratic time.

Insertion Sort (see Exercise 4.11), Quicksort (see Exercise 4.22), Mergesort (see Exercise 4.28), Maxsort, and Bubblesort can be adapted fairly easily for linked lists. One idea to keep in mind is that sorted lists can be constructed from the largest element back to the smallest. Procedures can be found in *Lisp* texts.

Some students who follow the algorithms too literally observe that there are difficulties in using linked lists to exactly imitate the way an algorithm works on arrays. While this is true for **partition**, for example, it is preferable to think of adapting the *ideas*, rather than the code, when switching data structures. Thus the central idea of **partition** is to partition the set into "small" and "large" subsets whose keys do not overlap. This is actually *easier* to do with lists than it is to do in place with arrays.

4.55

This is an extension of the strategy used by **partitionL**.

We assume the elements are stored in the range $1, \ldots, n$. At an intermediate step the elements are divided into four regions: the first contains only **red**s, the second contains only **white**s, the third contains elements of unknown color, and the fourth contains only **blue**s. There are three indexes, described in the comments below.

```
int r; // index of last red
int u; // index of first unknown
int b; // index of first blue

r = 0; u = 1; b = n+1;
while (u < b)
    if (E[u] == red)
            Interchange E[r+1] and E[u].
            r ++;
            u ++;
    else if (E[u] == white)
            u ++;
    else // (E[u] == blue)
            Interchange E[b-1] and E[u].
            b --;
```

With each iteration of the **while** loop either u is incremented or b is decremented, so there are n iterations. (From E. W. Dijkstra, *A Discipline of Programming*.)

4.57

a)

Time step	Instructions
1	CEX 1,2; CEX 3,4
2	CEX 1,3; CEX 2,4
3	CEX 2,3

b) The following algorithm is based on Insertion Sort. Since there are no test-and-branch instructions, enough CEX instructions are done to let each new key filter all the way to the beginning of the array if necessary. The algorithm is shown for $n = 5$; the general form should be clear.

Time step	Instructions			
	Insert x_2	Insert x_3	Insert x_4	Insert x_5
1	CEX 1,2			
2		CEX 2,3		
3		CEX 1,2	CEX 3,4	
4			CEX 2,3	CEX 4,5
5			CEX 1,2	CEX 3,4
6				CEX 2,3
7				CEX 1,2

The last column, insertion of x_n, takes $n-1$ time steps.

There are $n-2$ earlier time steps needed to get the other insertions started. The total is $2n-3$, so Insertion Sort can be implemented in linear time using simultaneous CEX instructions.

There are other solutions. See the odd-even transposition sort in Selim Akl, *Parallel Sorting Algorithms* for another interesting approach.

4.59

This is a good problem to spend some time on in class because it has a variety of solutions that can be used to illustrate many points about developing good algorithms. Many students turn in complicated solutions with a page and a half of code, yet each of the solutions described below can be coded in fewer than ten lines. The problem is discussed at length in Bentley, *Programming Pearls* (Addison-Wesley, 1986). I (Sara Baase) write the following quotation from Antoine de Saint-Exupéry on the board before I start discussing the problem.

> "A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away."

a) **Solution 1:** Consider each starting point, s, and each ending point, e, for a subsequence $(0 \leq s \leq e \leq n-1)$, and compute the sum of each subsequence, always saving the maximum sum found so far. There are $\Theta(n^2)$ subsequences, each with an average of roughly $n/2$ terms to be summed. The number of comparisons done to find the maximum sum is smaller than the number of additions, so this brute-force method takes time in $\Theta(n^3)$.

Solution 2: Fix the starting point s. As e varies from s to $n-1$, compute each new subsequence sum by adding the next element to the current sum, using only one addition, always saving the maximum sum found so far. Do this for each s. For each s, $n-s-1$ additions are done, so the total is in $\Theta(n^2)$. The number of comparisons of sums is roughly equal to the number of additions, so this solution takes time in $\Theta(n^2)$.

Both of the solutions above consider every subsequence explicitly; some students argue that this is necessary and therefore any solution would require time in $\Omega(n^2)$. The next solution is linear, hence shows the subtlety sometimes necessary in lower bounds arguments.

Solution 3: Observe that if an element in a particular position is positive, or if the sum of a subsequence ending at that position is positive, the maximum subsequence can not begin at the next position because including the positive element or subsequence would increase the sum. Thus each possible starting point for a subsequence does not have to be explicitly considered. The only places where a new subsequence must be started is immediately following a position where the sum has become negative. Here's the solution.

```
maxSoFar = 0;
currentSum = 0;
for (i = 0; i < n; i ++)
     currentSum = currentSum + E[i];
     if (currentSum < 0)
          currentSum = 0;
     else if (currentSum > maxSoFar)
          maxSoFar = currentSum;
return maxSoFar;
```

Bentley describes a Divide and Conquer solution that runs in $\Theta(n \log n)$ time. It is more complicated than the solutions described here, but it is also interesting to present in class.

b) Suppose an algorithm does not examine some element, say **E[i]**, and suppose its answer is M. Simply change **E[i]** to $M+1$. The algorithm would give the same response, and it would be wrong. Thus any (correct) algorithm must examine each element, hence do $\Omega(n)$ steps.

4.61

There are many reasonable answers for this open-ended question. What matters most is if the choices are supported by good reasons.

a) There are two basic approaches to this kind of sorting problem. One is to throw everything together and sort once on a combined key of course and student name. Another is to sort each class individually by student name, doing 5000 separate sorts. Because sorting costs grow faster than linearly, many small sorts will be cheaper than one large sort.

Having decided to sort each class individually, and given that most classes have about 30 students, it might be more expedient to use a simple method like Insertion Sort, even though its asymptotics are not favorable. There is no reason to expect the worst case for Insertion Sort, so let's look briefly at some average cases. On a set of 30, Insertion Sort does about 225 comparisons on average, while Mergesort does about 150. Because of the small sizes the space overhead incurred by Mergesort is not a serious drawback. But Mergesort probably has more overhead per comparison than Insertion Sort, so the two might be quite close in time used for sets of 30.

Since the largest class will be 200 at most, Insertion Sort should use about 10,000 comparisons on average. Mergesort would use about 1600 comparisons on average. Even allowing for Mergesort's higher overhead per comparison, it is probably 4 to 5 times faster than Insertion Sort, *for the occasional large classes*.

We might consider Quicksort also. But suppose we frequently have data that is partially sorted, rather than random? This is a reasonable possibility when the data is extracted from another database. In this case, Quicksort would tend to degrade toward its worst case, Mergesort would stay about the same, and Insertion Sort would run *faster* than average. However, if the data is randomly ordered, which it would be if it was in the order in which students registered, then Quicksort might perform slightly better than Mergesort on average because it does fewer moves and the data records are rather sizable.

The exercise implied that the input data was already grouped into classes. If not, then Radix Sort would probably be the best way to group it into classes so that each class can be sorted as a small set, as discussed above.

A final issue to consider is the role of the student records, which are separate from the class records. The problem statement does not make it clear whether information is needed from the student records to compose the class lists. (This is often the case in real-world problems.) If such information is needed, the matching student record for each class record can be looked up after the class records are sorted, using the disk address in the class record. Readers who are familiar with the relative speeds of disk access and internal computations will realize that the time for these disk accesses will dwarf the sorting time, so the chosen sorting method is almost irrelevant in this case. Trying to optimize the pattern of disk accesses goes beyond the scope of the text.

b) The main problem with distributional sorts in a computer is the need to arrange space for buckets that will have an unknown number of items in them. For manual sorting of exams, this is not a problem, as you just pile them up. So simply distribute into 26 main piles based on first letter. If a main pile is too large to sort ad hoc, use the second table to distribute into 26 piles based on second letter. Each of these piles can probably be sorted ad hoc, but if one or two need a further breakdown, the person can shuffle things around to make enough space somewhere.

4.63

One point to this problem is that you can't do any better than the obvious method (if only comparisons are available). See Exercise 5.19(b).

a) Simply sort them with a comparison sort using three-way comparisons. If any three-way comparison is "equal" there are duplicates. A little thought reveals that if there are duplicates, and the sorting procedure is correct, then there must be an "equal" comparison at some point. If three-way comparisons are not convenient, do a post-scan of the sorted array to check for equal adjacent elements.

b) The time is $\Theta(n \log n)$ using Mergesort or Heapsort.

c) Simply count how many have each key value, using an array of $2n+1$ entries to store the counts. (We can quit as soon as any count becomes 2.)

4.65

To simplify the notation let m denote the number of elements in **E2**; m is about $\lg k$ and $n = k + m$, so m is in $\Theta(\log n)$.

For the first solution, sort the unsorted elements in **E2** by themselves with $\Theta(m \log m) = \Theta(\log n \log\log n)$ comparisons. Merge them with the sorted elements, working right-to-left (or high-to-low), using the extra space at the end of **E1** for the initial output area. The idea is similar to text section 4.5.3. The merge step requires $n-1$ comparisons at most, and an equal number of element movements. So the total number of comparisons is in $\Theta(n)$.

The second method will do $o(n)$ comparisons. Since any method requires $\Omega(n)$ element movements in the worst case, it is sort of pointless to try to reduce the number of comparisons below $\Theta(n)$, but it is possible. (A related technique comes up in Chapter 14 where it is called *cross-ranking*.)

After sorting **E2** declare an array **rankE1** with m entries. This array will record where each element in the sorted **E2** array belongs in the final merged sequence. For each j, $0 \leq j < \lg k$, use binary search in **E1** to find out where **E2[j]** belongs in sorted order. If it is discovered that **E2[j]** should immediately precede **E1[r]**, then store r in **rankE1[j]**. This means that when **E1** and **E2** are merged, **E2[j]** will go into **E1[j+r]**. This step requires about $m \lg k$ comparisons. But $m \lg k$ is in $\Theta(\log^2 n)$. This dominates the cost of sorting **E2**.

Once we know where every element of **E2** belongs in the merged array, we can perform the merge without doing any further comparisons. We work right-to-left through **rankE1**, **E2**, and **E1**. For example, for all i such that **rankE1[j]** $\leq i <$ **rankE1[j+1]**, the element initially in **E1[i]** must move to **E1[i+j+1]**. With all this machinery, only n element movements are needed. Actually, the array **rankE1** is not needed, as its elements can be computed on the fly, but it simplifies the description of the algorithm.

For a third method, which is quite simple, simply insert each element from **E2** into **E1**, in the manner of Insertion Sort. If done naively, this requires about $(k + \frac{1}{2}m)m$ comparisons and element movements in the worst case, which is in $\Theta(n \log n)$. But again, binary search in **E1** (see Exercise 4.8) reduces the number of comparisons to $\Theta(\log^2 n)$. However, it does not reduce the number of element movements.

Chapter 5
Selection and Adversary Arguments

Section 5.1: *Introduction*

5.2

a) For each comparison, the adversary considers how many permutations of the keys are consistent with both possible answers. It gives the response consistent with the most permutations. Thus after each comparison, the number of remaining permutations is at least half as many as before the comparison. There are $n!$ possible permutations in all, so the adversary makes any algorithm do at least $\lg n!$ comparisons in the worst case. This is the same result we obtained in Theorem 4.10.

b) The adversary keeps track of all remaining possible arrangements of the keys consistent with the responses it has given to the algorithm and the fact that the two sequences being merged are sorted. It always chooses a response to preserve at least half the remaining possibilities. Exercise 4.25 tells us that the total number of possible permutations is $\binom{n}{n/2}$. Thus the adversary can force an algorithm to do at least $\lg\left(\binom{n}{n/2}\right)$ comparisons. It requires some care to obtain a rigorous lower bound on this quantity.

$$\begin{aligned}\binom{n}{n/2} &= \frac{n!}{(n/2)!(n/2)!}\\ \lg\left(\binom{n}{n/2}\right) &= \lg(n!) - 2\lg((n/2)!)\end{aligned}$$

The difficulty is that the simple lower bound given in Eq. 1.18 cannot be used on the negative terms.

Solution 1. By inspection of Stirling's formula (Eq. 1.19 in the text) we can write

$$\begin{aligned}n! &= \left(\frac{n}{e}\right)^n \Theta\left(\sqrt{n}\right)\\ \lg(n!) &= n(\lg(n) - \lg(e)) + \Theta(\log n)\\ \lg((n/2)!) &= \frac{n}{2}(\lg(n) - 1 - \lg(e)) + \Theta(\log n)\\ \lg\left(\binom{n}{n/2}\right) &= n \pm O(\log n) \qquad \text{for } n \geq 2 \text{ and even.}\end{aligned}$$

There is some abuse of notation to treat $\Theta(\ldots)$ and $O(\ldots)$ as functions, but the meaning is clear here. We wrote "$\pm O(\log n)$" to remind ourselves that we do not know whether this term is positive or negative; also we do not know (from the above analysis) whether it is $\pm\Theta(\log n)$ because the leading terms might cancel.

Theorem 4.4 gave a lower bound of $n-1$ for this problem, which is the best possible. This degree of analysis on the adversary argument gives us a lower bound with the same leading term (n) as that given in Theorem 4.4, but the second order term is not known.

Solution 2. We can carry one more term in our simplification of Stirling's formula and get a more precise bound.

$$\begin{aligned}n! &= \left(\frac{n}{e}\right)^n \sqrt{n}\Theta(1)\\ \lg(n!) &= n(\lg(n) - \lg(e)) + \tfrac{1}{2}\lg(n) + \Theta(1)\\ \lg((n/2)!) &= \frac{n}{2}(\lg(n) - 1 - \lg(e)) + \tfrac{1}{2}\lg(n) + \Theta(1)\\ \lg\left(\binom{n}{n/2}\right) &= n - \tfrac{1}{2}\lg(n) \pm O(1) \qquad \text{for } n \geq 2 \text{ and even.}\end{aligned}$$

Therefore the result given by this adversary argument is very close to the result of Theorem 4.4, but is weaker by a log term.

Solution 3. Applying Stirling's formula carefully gives the following bounds that are often useful in their own

right.

$$\sqrt{\frac{2}{\pi}}\left(1-\frac{4}{11n}\right)\frac{2^n}{\sqrt{n}} < \binom{n}{n/2} < \sqrt{\frac{2}{\pi}}\left(1+\frac{1}{11n}\right)\frac{2^n}{\sqrt{n}}$$

Therefore the "$\pm O(1)$" term in Solution 2 is actually between $-.32$ and -0.62, depending on n. So, finally:

$$\lg\left(\binom{n}{n/2}\right) > n-1-\tfrac{1}{2}\lg(n) \qquad \text{for } n \geq 2 \text{ and even.}$$

Section 5.3: *Finding the Second-Largest Key*

5.4

a) By examination of the **`for`** loop we see that the array positions are filled in reverse order from $n-1$ down to 1 and that each position is assigned a value once and not changed. We use an informal induction argument to show that after **`heapFindMax`** executes, each position contains the maximum of all the leaves in its subtree. The base cases are the leaves, positions n through $2n-1$. The claim is clearly true for them. Suppose the claim is true for all positions greater than k, and consider node k. It is filled when **`last`** $= 2k$. At that time, position k gets the maximum of its children, positions $2k$ and $2k+1$, each of which, by the induction hypothesis, is the maximum of the leaves in its subtree, so position k gets the maximum of the leaves in *its* subtree. Thus, **`E[1]`** will have the maximum of all the leaves, i.e., the max of the original keys.

b) At each internal node, the key in that node is the same as one of its children. The other child lost when that node was filled. Let's assume the keys are distinct. Then, as we move down the tree from the root along the path where each node has a copy of max, the other child of each node we pass contains a key that lost to the max.

c)
```
max = E[1];
secondLargest = −∞;
i = 1;
while (i < n)
     if (E[2*i] == max)
          i = 2*i;
          if (E[i+1] > secondLargest)
               secondLargest = E[i+1];
     else i = 2*i+1;
          if (E[i-1] > secondLargest)
               secondLargest = E[i-1];
return secondLargest;
```

5.6

a) Worst case number of comparisons = 1 (before the **for** loop) + $2(n-2)$ (in the loop) = $2n-3$. The worst case occurs for keys in increasing order.

b) Each time through the loop either one or two comparisons are done. Two comparisons are done if and only if $E[i]$ is one of the two largest keys among the first i keys. The probability for this is $2/i$. The probability that $E[i]$ is one of the $i-2$ smaller keys is $(i-2)/i$. One comparison is done before the loop, so the average number of comparisons is

$$\begin{aligned} A(n) &= 1+\sum_{i=3}^{n}\left(2\cdot\frac{2}{i}+\frac{i-2}{i}\right) = 1+4\sum_{i=3}^{n}\frac{1}{i}+\sum_{i=3}^{n}1-2\sum_{i=3}^{n}\frac{1}{i} = 1+n-2+2\sum_{i=3}^{n}\frac{1}{i} \\ &\approx n-1+2(\ln n-1) = n-3+2\ln n \end{aligned}$$

Equation 1.11 was used to estimate $\displaystyle\sum_{i=3}^{n}\frac{1}{i}$.

Section 5.4: *The Selection Problem*

5.8

a) The idea is that, whereas Quicksort must recursively sort both segments after partitioning the array, **`findKth`** need search only one segment.

```
/** Precondition:  first ≤ k ≤ last.  */
Element findKth(Element[] E, int first, int last, int k)
    Element ans;
    if (first == last)
        ans = E[first];
    else
        Element pivotElement = E[first];
        Key pivot = pivotElement.key;
        int splitPoint = partition(E, pivot, first, last);
        E[splitPoint] = pivotElement;
        if (k < splitPoint)
            ans = findKth(E, first, splitPoint - 1, k);
        else if (k > splitPoint)
            ans = findKth(E, splitPoint + 1, last, k);
        else
            ans = pivotElement;
    return ans;
```

b) As with Quicksort, we have a worst case if each call to **`partition`** eliminates only one key. If the input is sorted, **`findKth(E, 1, n, `**$\lceil n/2 \rceil$**`)`**, which is the call that would be used to find the median, will generate successive calls with **`first`** = 1, 2, …, $\lceil n/2 \rceil$ and **`last`** = n. For all of these calls the subrange that is partitioned has at least $n/2$ elements, so the number of comparisons is in $\Omega(n^2)$. Also, **`findKth`** never does *more* comparisons than Quicksort, so its worst case is in $\Theta(n^2)$.

c) **Solution 1**. A recurrence for the average number of comparisons when there are n elements and rank k is to be selected is:

$$T(n,k) = n-1+\frac{1}{n}\left(\sum_{i=0}^{k-1} T(n-1-i,k-1-i)+\sum_{i=k+1}^{n-1} T(i-1,k)\right) \qquad \text{for } n>1,\ 0\le k<n$$

$$T(1,0) = 0$$

The running time will be of the same asymptotic order as the number of comparisons.

Solution 2. To avoid a 2-parameter recurrence we might settle for an upper bound expression instead of an exact recurrence. Let $T(n)$ denote the average number of comparisons when there are n elements and k has its most unfavorable value. In other words, $T(n) = \max_k T(n,k)$. Then

$$T(n) \le n-1+\frac{1}{n}\left(\sum_{i=0}^{k-1} T(n-1-i)+\sum_{i=k+1}^{n-1} T(i-1)\right)$$

d) Working with Solution 1 above, we will guess that the solution is $T(n,k) \le cn$, where c is a constant to be chosen, and try to verify the guess. The same technique works for Solution 2. The right-hand side of the recurrence, with the guessed solution substituted in, becomes:

$$\begin{aligned} n-1+\frac{1}{n}\left(\sum_{i=0}^{k-1} c(n-1-i)+\sum_{i=k+1}^{n-1} c(i-1)\right) &= n-1+\frac{1}{n}\left(\tfrac{1}{2}k(n-1+n-k)c+\tfrac{1}{2}(n-1-k)(k+n-2)c\right) \\ &= n-1+\frac{1}{2n}\left(c(n^2+2kn-2k^2-3n+2)\right) \\ &= n(1+\tfrac{1}{2}c)+k(c-c(k/n))+c(-3/2+\tfrac{1}{2}(k/n)+1/n)-1 \end{aligned}$$

We need to choose c so that this expression is at most cn when $0 \le k < n$. Since $k(c-c(k/n)) \le cn/4$, we choose $c = 4$, and the desired inequality holds. This method of solution also works for the recurrence for $T(n)$.

5.10
Building the heap takes at worst $2n$ comparisons. *DeleteMax* uses *FixHeap* which does approximately $2\lg m$ comparisons if there are m keys in the heap. Thus the **for** loop does approximately $\sum_{i=1}^{k} 2(\lg n - i)$ comparisons. The sum is bounded by $2k\lg n$, so k may be as large as $n/\lg n$, and the total time will still be linear in n.

Section 5.5: *A Lower Bound for Finding the Median*

5.12
At least $n-1$ crucial comparisons must be done whether n is even or odd. The adversary follows the strategy summarized in Table 5.4; it may give status S to at most $n/2-1$ keys and status L to at most $n/2$. Since each comparison described in Table 5.4 creates at most one L key, and each creates at most one S key, the adversary can force any algorithm to do at least $n/2-1$ non-crucial comparisons. The modified version of Theorem 5.3 is: Any algorithm to find the median of n keys for even n, where the median is the $n/2$th smallest key, by comparison of keys, must do at least $3n/2-2$ comparisons in the worst case.

Section 5.6: *Designing Against an Adversary*

5.14
Any key that is found to be larger than three other keys or smaller than three other keys may be discarded. For simplicity in presenting the solution, we use the CEX (compare-exchange) instructions described in Exercise 4.56. CEX *i*,*j* compares the keys with indexes *i* and *j* and interchanges them if necessary so that the smaller key is in the position with the smaller index.

```
CEX 1,2
CEX 3,4
CEX 1,3
If an interchange occurs here, also interchange E[2] and E[4].
// E[1] is smaller than three other keys; it can be rejected.
// We know E[3] < E[4].
CEX 2,5
CEX 2,3
If an interchange occurs here, also interchange E[4] and E[5].
// E[2] is smaller than three other keys; it can be rejected.
// We know E[3] < E[4].
CEX 3,5
// Now E[3] is the smallest of the three remaining keys; it is the median.
```

Additional Problems

5.16
The array indexes holding keys are $0, \ldots, n-1$. The search key is K and the array name is E. The adversary maintains variables that are initialized and have meanings as follows. The subscript on the variable indicates how many questions have been answered.

$L_0 = 0$	(low)	smallest index whose key might equal K.
$H_0 = n-1$	(high)	largest index whose key might equal K.
$M_0 = (n-1)/2$	(midpoint)	$(L+H)/2$, with any fractional part.
$R_0 = n$	(range)	$(H+1-L)$, the number of slots that might hold K.

M and R are recomputed whenever L or H is updated. We'll assume the algorithm does 3-way compares.

When the algorithm compares K to $E[i]$ (asks the q-th question), the adversary responds and updates L and H as follows:

1. If $i < L_{q-1}$, answer $K > E[i]$. No updates (i.e., $L_q = L_{q-1}$ and $H_q = H_{q-1}$).
2. If $i > H_{q-1}$, answer $K < E[i]$. No updates (i.e., $L_q = L_{q-1}$ and $H_q = H_{q-1}$).
3. If $L_{q-1} \le i < M_{q-1}$, answer $K > E[i]$. Update $L_q = i+1$, $H_q = H_{q-1}$.
4. If $M_{q-1} \le i \le H_{q-1}$, answer $K < E[i]$. Update $H_q = i-1$, $L_q = L_{q-1}$.

The essential invariant is $R_q + 1 \geq (R_{q-1} + 1)/2$. For example, in case 4,

$$\begin{aligned} H_q &= i - 1 \geq M_{q-1} - 1 = (H_{q-1} - L_{q-1})/2 - 1 + L_{q-1} = (R_{q-1} + 1)/2 - 2 + L_{q-1} \\ L_q &= L_{q-1} \\ (R_q + 1) &= H_q - L_q + 2 \geq (R_{q-1} + 1)/2 \end{aligned}$$

Case 3 is similar.

If the algorithm stops before $R_q = 0$, then there are still at least two possible outcomes of the search. For example, if $L_q = H_q$, so that $R_q = 1$, it is possible that $K = E[L_q]$ and it is possible that K is not in the array.

Now assume the algorithm stops when $R_q = 0$. By the recurrence for $R_q + 1$ and the base case $R_0 + 1 = n + 1$, we have $R_q + 1 \geq \left(\frac{1}{2}\right)^q (n+1)$. So $q \geq \lg(n+1)$. Since q must be an integer, $q \geq \lceil \lg(n+1) \rceil$.

5.19

a) Sort the keys using an $\Theta(n \log n)$ sort, but stop and report that a key is duplicated if any comparison comes out "equal". If sorting completes without any "equal" occurring, then all keys are distinct. (Every pair of keys that are adjacent in the final sequence must have been compared directly, or the algorithm would be unsound.) The total time is in $\Theta(n \log n)$.

   If a standard sorting algorithm is used (without a test for $=$), one pass through the sorted array can be used to detect duplicates in $O(n)$ time, so the total is $\Theta(n \log n)$ in the worst case.

b) Consider a list where the keys are distinct. Let $x_1 < x_2 < \ldots < x_n$ be the keys in order, (not necessarily the way the algorithm sees them). If the algorithm knows the relative order of each pair x_i and x_{i+1}, then it has enough information to sort the keys.

   Suppose there are two keys x_i and x_{i+1} whose relative order is not known by the algorithm.

   Then the algorithm does not have enough information to determine whether $x_i = x_{i+1}$ since no comparisons of these keys with others in the list would provide that information. If the algorithm says there are two equal keys, it is wrong. If the algorithm says the keys are all distinct, the adversary can change x_i to make it equal to x_{i+1} without affecting the results of any other comparisons, so the algorithm would give the wrong answer for that input.

   Thus, to determine whether the keys are distinct, the algorithm must collect enough information to sort the keys. The three-way comparison is more powerful than the two-way comparisons for which we established the $\Omega(n \log n)$ lower bound for sorting, but if the keys are distinct, the "=" test in the comparison does not provide any extra information, so in the worst case, sorting needs at $\Omega(n \log n)$ three-way comparisons. Thus determining if the keys are distinct also needs $\Omega(n \log n)$ comparisons.

5.21

To establish the minimum number of cells needed requires both an algorithm and an argument that no other algorithm can solve the problem using fewer cells.

a) The algorithm uses two cells **`maxSoFar`** and **`next`**.

   The first key is read into **`maxSoFar`**; each remaining key is read into **`next`**, compared with **`maxSoFar`**, and copied into **`maxSoFar`** if it is larger.

   The problem can't be solved with fewer than two cells, because if there were only one, no comparisons could be done.

b) For simplicity we assume n is odd. The problem can be solved with $(n+1)/2 + 1$ cells. The algorithm's strategy is to save the $(n+1)/2$ smallest keys it has seen so far, in a sorted array, using the extra cell for "scratch work." We use **`shiftVac`** from Insertion Sort (Section 4.2.2) to insert the next key into the correct position if it is among the smallest keys. After all keys have been read, the largest key retained by the algorithm is the median. More specifically:

```
Read the first (n+1)/2 keys into E[0], ..., E[(n+1)/2 - 1].
Sort these keys (in place).
for (i = (n+1)/2; i < n; i ++)
     Read the next key into scratch;
     if (scratch < E[(n+1)/2 - 1])
          // Insert the new key in its proper place;
          // the key that was in E[(n+1)/2 - 1] is discarded.
          vacant = shiftVac(E, (n+1)/2 - 1, scratch);
          E[vacant] = scratch;
return E[(n+1)/2 - 1];
```

Now we show that the problem can't be solved with fewer than $(n+1)/2+1$ cells. Suppose an algorithm used fewer cells; we show that an adversary can make it give a wrong answer.

Case 1: The algorithm gives its answer before overwriting a key in one of the cells.

Then, at the time it determines its answer, the algorithm has read no more than $(n+1)/2$ keys. If it chooses any but the largest of these, an adversary can make all the remaining keys, unseen by the algorithm, larger than the key chosen. There are at least $(n-1)/2$ unseen keys and at least one key seen by the algorithm is larger than the key it chose, so there are too many larger keys, and the algorithm's answer is wrong.

If the algorithm chooses the largest of the keys it has seen, the adversary can make all the remaining keys smaller, so again, the one chosen by the algorithm is not the median.

Case 2: Before choosing its answer, the algorithm reads a key into a cell that was used earlier, discarding the key that was there.

The discarded key cannot be the algorithm's output. We will show that an adversary can make the destroyed key be the median.

Suppose the first key that is overwritten by the algorithm is the *i*th smallest of the keys it has seen up to that point.

There are at least $(n-1)/2$ keys it hasn't seen (at that time). The adversary makes $(n-1)/2-(i-1)$ of the unseen keys smaller than the discarded key and the other unseen keys larger. The total number of keys smaller than the discarded keys is $i-1$ (seen by the algorithm) + $(n-1)/2-(i-1)$ (not yet seen by the algorithm) = $(n-1)/2$, so the discarded key was the median.

5.22

a) Find the minimum of $E[k]$ through $E[n]$ with $n-k$ comparisons.

b) Follow the adversary strategy in the proof of Theorem 5.2 in the text, except that the replies are reversed.

1. $w(i)=1$ for all keys initially.
2. If the algorithm compares $E[i]$ and $E[j]$ and $w(i)>w(j)$, then reply that $E[i]<E[j]$, and update new $w(i)=$ prior $(w(i)+w(j))$; new $w(j)=0$.
3. If the algorithm compares $E[i]$ and $E[j]$ and $w(i)=w(j)>0$, then reply that $E[i]<E[j]$, and update new $w(i)=$ prior $(w(i)+w(j))$; new $w(j)=0$.
4. If the algorithm compares $E[i]$ and $E[j]$ and $w(i)<w(j)$, then reply that $E[i]>E[j]$, and update new $w(j)=$ prior $(w(i)+w(j))$; new $w(i)=0$.
5. If the algorithm compares $E[i]$ and $E[j]$ and $w(i)=w(j)=0$, then reply consistently with earlier replies and do not update any w.

A key has *won* a comparison if and only its weight is zero. (Recall, the winner of a comparison is the larger key, so in this problem we are looking for the champion loser.) The sum of all weights is always n. As in the proof of Theorem 5.2, imagine that the adversary builds a tree to represent the ordering relations between the keys, except that the smallest key is at the root. Two trees are combined only if neither root won before, and the loser is the new root. Whenever i is the root of a tree, $w(i)$ is the number of nodes in that tree.

Suppose, when the algorithm stops, there is no key with weight $>n-k$. Then whatever output the algorithm makes, the adversary can demonstrate a consistent set of keys for which the output is wrong. Specifically, if the

output is m and $E[m]$ is in tree T then make all keys not in tree T less than all keys that are in tree T. Since at least k keys are not in tree T, this refutes the algorithm.

Therefore, a correct algorithm can stop only when the weight of some key is at least $n-k+1$. But each question creates at most one new edge in the set of trees that the adversary builds. A tree of $n-k+1$ nodes must have $n-k$ edges. So the lower bound is $n-k$ questions.

5.24

The adversary will force an algorithm to examine each entry on the diagonal from the upper right corner $(M[0][n-1])$ to the lower left corner $(M[n-1][0])$, and on the diagonal just below this one $(M[1][n-1]$ to $M[n-1][1])$. There are $2n-1$ keys on these diagonals. So our argument will show that any algorithm for this problem must examine at least $2n-1$ matrix entries.

The first diagonal is characterized by the property that $i+j$ (the sum of the row and column numbers of an element) is $n-1$; elements on the lower diagonal satisfy $i+j=n$. Observe that the known ordering of the rows and columns in the matrix implies nothing about the ordering of the keys on each diagonal.

Whenever an algorithm compares x to an element $M[i][j]$ the adversary responds as follows until there is only one element on the two diagonals described above that has not been examined by the algorithm:

if $i+j \leq n-1$, then $x > M[i][j]$
if $i+j > n-1$, then $x < M[i][j]$

The adversary's answers are consistent with the given ordering of the rows and columns and with each other. If the algorithm says x is in M, the adversary can fill all slots with keys unequal to x, so the algorithm would be wrong. If the algorithm says x is not in M without examining all $2n-1$ keys on the two diagonals, the adversary puts x in an unexamined position on one of the diagonals.

Chapter 6
Dynamic Sets and Searching

Section 6.2: *Array Doubling*

6.1

The geometric series for array quadrupling is n, $n/4$, $n/16$, ..., and sums to at most $(4n/3)t$, compared to $(2n)t$ for array doubling. However, the amount of space allocated might be almost four times what is necessary instead of only twice what is necessary.

Section 6.3: *Amortized Time Analysis*

6.2

a) Worst case cost is $\Omega(n)$. Say the stack size is N_0 and $n = N_0 - 1$. Do 2 pushes, forcing an array double, for actual cost $2 + tN_0$. The new size is $2N_0$. Now do 2 pops, reducing the number of elements to the original n, which is less than $2N_0/2$. So this forces a shrink, at a cost of $2 + t(N_0 - 1)$. The cost of 4 operations is $4 + 2tN_0 - t \approx 2n$. This sequence can be repeatedly indefinitely.

b) Use accounting costs as follows:

   1. **push**, no array double: $2t$.
   2. **push**, array double from n to $2n$: $-nt + 2t$.
   3. **pop**, no shrink: $2t$.
   4. **pop**, leaving n elements and causing a shrink: $-nt + 2t$. Note that the array size is at least $n + 1$ after the shrink and was at least $4n + 1$ before being shrunk.

   Then the amortized cost is $1 + 2t$ for **push** and $1 + 2t$ for **pop**. The cumulative accounting cost is always $\geq (4n_h = 2n - N)t$ when the stack had n_h elements at high water since the last size change. A proof by induction on i, the number of the operation, is straightforward. If the i-th operation was a **pop** that caused a shrink with accounting cost $-tn$, then $N/2 + 1 \geq n/2 - 1$ and the array size was $4n + 4 \geq N \geq 4n + 1$. So, by the inductive hypothesis the cumulative accounting cost was at least $(2n)t$ before the shrink and at least $(n)t$ afterwards. The new $n_h = n$. The new array size is $n + 1$ or n, so the formula still holds.

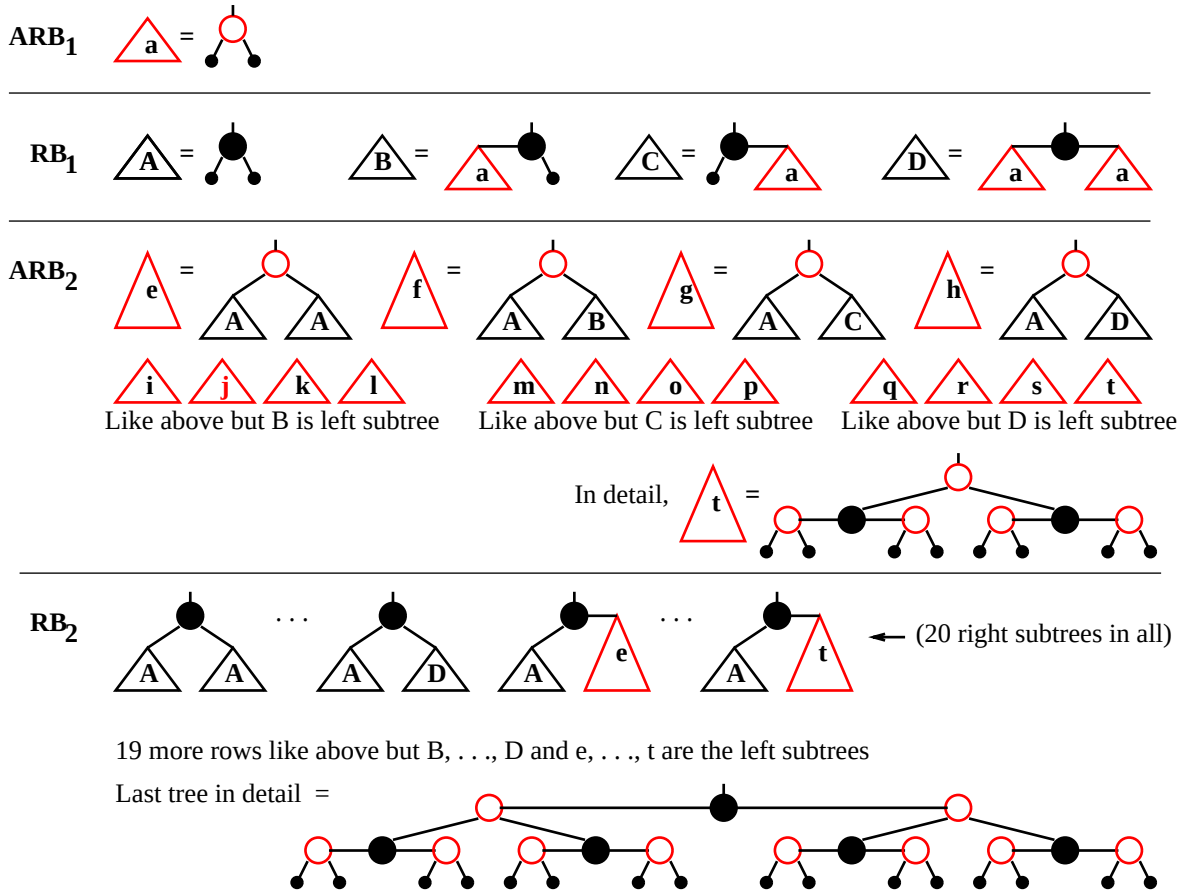
   Note that an accounting cost of $1t$ on **pop** is not enough. Suppose $N_0 = 10$. Do 41 pushes, then 22 pops, then 2 pushes, and the cumulative accounting cost is negative.

c) The accounting costs for **pop** are t and $-nt + t$. The cumulative accounting cost is $\geq (3n_h - n - N)t$. The analysis is similar to part (b) except that before a shrink the cumulative accounting cost is $\geq nt$, and after it is ≥ 0. But the new array size $= 2n$ or $2n + 1$ so the formula holds.

d) Expand the array by a factor $E > 2$ and use accounting cost $\left(\frac{E}{E-1}\right) t$ instead of $2t$ for **push**. For **pop**s, shrink by a factor of E when $n < N/E^2$. The accounting cost for **pop** is $\left(\frac{1}{E-1}\right) t$ when there is no shrink and $-nt + \left(\frac{1}{E-1}\right) t$ when there is a shrink.

Section 6.4: *Red-Black Trees*

6.4



6.6

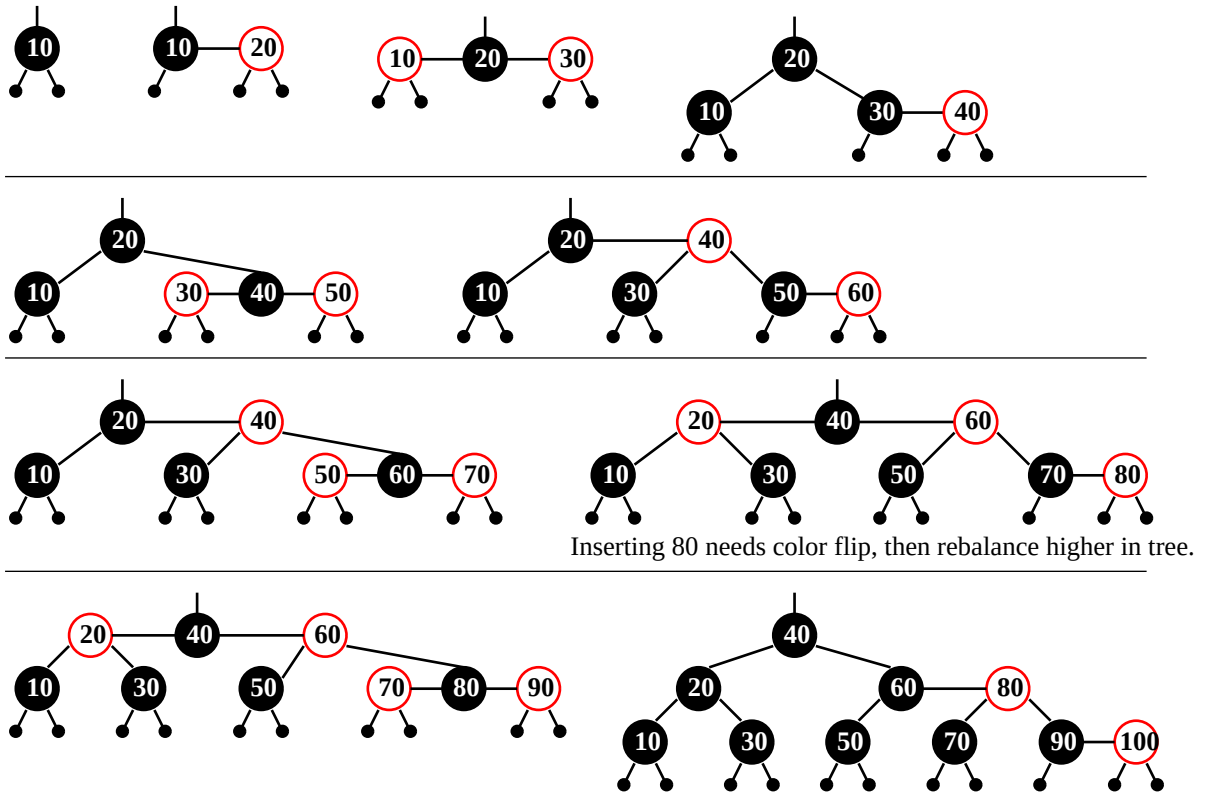
The proof is by induction on h, the black height of the RB_n or ARB_n tree. The base case is $h = 0$. An RB_0 tree consists of 1 external node, so parts 1-3 hold for this RB_0 tree. An ARB_0 does not exist, so parts 1-3 for ARB_0 trees hold vacuously.

For $h > 0$ assume the lemma holds for black heights $0 \leq j < h$. The principal subtrees of an ARB_h tree are both RB_{h-1} trees. An ARB_h tree has a minimum number of internal black nodes when both principal subtrees do. So this is $2(2^{h-1} - 1) = 2^h - 2$, and part 1 holds for ARB_h trees. Similarly, for part 2, an ARB_h tree has a maximum number of nodes when both principal subtrees do. So this is $1 + 2(4^{h-1} - 1) = \frac{1}{2}(4^h) - 1$.

An RB_h tree has a minimum number of internal black nodes when both principal subtrees are RB_{h-1} trees. But $1 + 2(2^{h-1} - 1) = 2^h - 1$, so part 1 holds. for RB_h trees. An RB_h tree has a maximum number of nodes when both principal subtrees are ARB_h trees. We showed above that each has at most $\frac{1}{2}(4^h) - 1$ nodes, so the RB_h tree has at most $1 + 2(\frac{1}{2}(4^h) - 1) = 4^h - 1$ internal nodes, and part 2 holds for RB_h trees.

For part 3, any path from the root to a black node has at least as many black nodes as red nodes, not counting the root. So black depth $\geq \frac{1}{2}$ depth.

6.8



Inserting 80 needs color flip, then rebalance higher in tree.

Inserting 100 needs color flip, then color flip higher in tree.

6.10

a)
```
int colorOf(RBtree t)
    if (t == nil)
        return black;
    else
        return t.color;
```

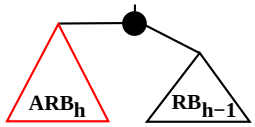
b)
```
void colorFlip(RBtree t)
    t.leftSubtree.color = black;
    t.rightSubtree.color = black;
    t.color = red;
```

6.12

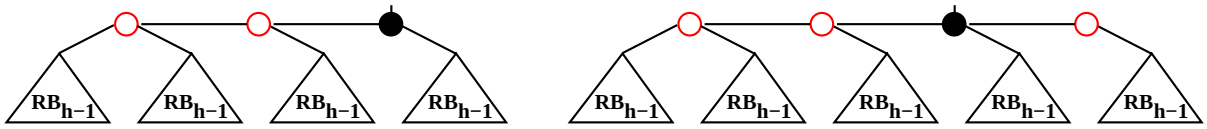
The proof is by induction on h, where **`oldRBtree`** is either an RB_h tree or an ARB_{h+1} tree. The base case is $h = 0$. If **`oldRBtree`** is an RB_0 tree, it is **`nil`** and case 3 of the lemma is true. If **`oldRBtree`** is an ARB_1 tree, it is a red node with both children **`nil`**. Then **`rbtIns`** is called on one of these these children, and case 3 applies, so the recursive call returns status **`brb`**. This status is passed to either **`repairLeft`** or **`repairRight`**. For definiteness, assume **`repairLeft`** was called; the case of **`repairRight`** is symmetrical. Then case 4 of the lemma applies (case 5 after **`repairRight`**).

For $h > 0$, assume the lemma holds for $0 \le j < h$. Suppose **`oldRBtree`** is an RB_h tree. Then each principal subtree is either an RB_{h-1} tree or an ARB_h tree, and the inductive hypothesis applies to whichever principal subtree is passed as the parameter in the recursive call to **`rbtIns`**. For definiteness, assume recursion goes into the left subtree. The case of the right subtree is symmetrical. There are 5 cases to consider for this recursive call.

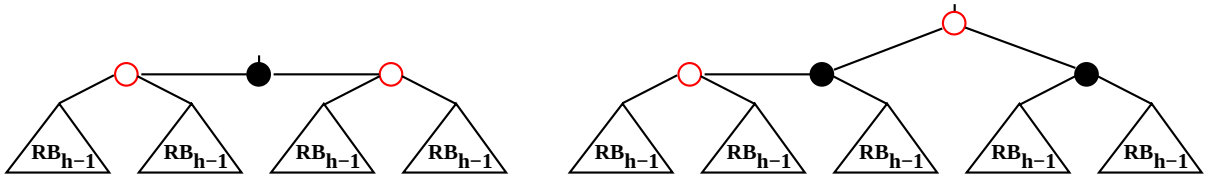
1. The recursive call returns **ansLeft.status** = **ok** and **ansLeft.newTree** is RB_{h-1} or ARB_h. Then **repairLeft** attaches **ansLeft.newTree** as the new principal left subtree and returns **status** = **ok**, which is case 1 of the lemma.
2. The recursive call returns **ansLeft.status** = **rbr** and **ansLeft.newTree** is RB_{h-1}. This is similar to case 1 above.
3. The recursive call returns **ansLeft.status** = **brb** and **ansLeft.newTree** is ARB_h. This is similar to case 1 above.
4. The recursive call returns **ansLeft.status** = **rrb** and **ansLeft.newTree** looks like this, where white circles denote red nodes:



After attaching **ansLeft.newTree** as the left subtree of **oldRBtree** we have one of these configurations, where the root is **oldRBtree**:



In the left case **rebalLeft** is called and in the right case **colorFlip** is called. The outcomes of each, now rooted at **ans.newTree**, are:



ans.status = **ok**
ans.status = **brb**

The left case is again case 1 of the lemma, while the right case is case 3.

5. This is the mirror image of case 4 above and leads to either case 1 or case 3 of the lemma.

So all returns from the recursive call of **rbtIns** lead to one of the cases stated in the lemma. This completes the proof for **oldRBtree** being an RB_h tree.

Now suppose **oldRBtree** is an ARB_{h+1} tree, so each of its principal subtrees is an RB_h tree. Then a recursive call to **rbtIns** is made with one of these principal subtrees as a parameter, and one of the cases described above applies to the value returned. Again we assume the recursion goes into the left subtree and consider the possible values the might be returned into **ansLeft**. The case for the right subtree is symmetrical. Notice that only cases 1 and 3 of the lemma occurred when the parameter of **rbtIns** was an RB_h tree. In (recursive) case 1 of the lemma, no further repair is needed, and case 1 of the lemma applies. In (recursive) case 3 of the lemma, **ansLeft.status** = **brb** and **ansLeft.newTree** is ARB_{h+1}. Now **oldRBtree.color** = **red**, so **ans.status** = **rrb** and **ans.newTree** = **oldRBtree** and **ans.newTree.leftSubtree** = **ansLeft.newTree**, which is ARB_{h+1}. Further, **ans.newTree.rightSubtree** is RB_h, the same as **oldRBtree.rightSubtree**. So all details of case 4 of the lemma are confirmed.

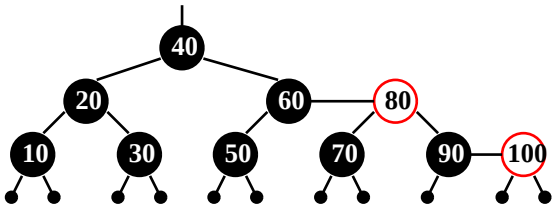
For **oldRBtree** being an ARB_{h+1} tree and the recursion going to the right subtree, either case 1 or case 5 of the lemma applies, by arguments that are symmetrical to those above.

Notice that case 2 of the lemma never applied. Indeed, **rbr** is never assigned as a status in any of the procedures as given. It could have been assigned after **rebalLeft** was called.

6.14

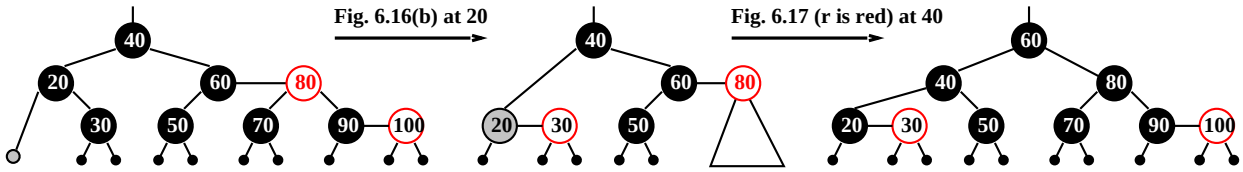
Note for instructor: The initial tree is not given explicitly in the text, but is the result of Exercise 6.8, which is solved above.

The starting point for all parts is:



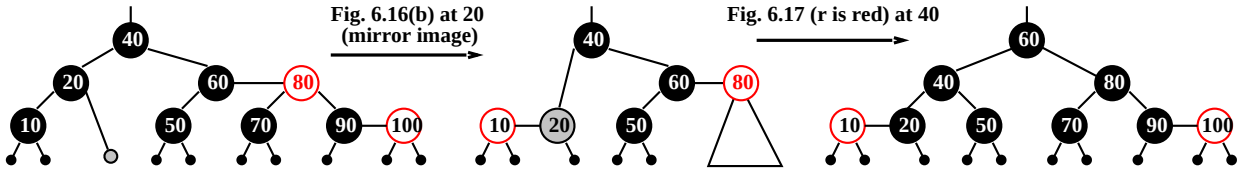
Circles with white backgrounds denote red nodes and gray circles are gray nodes.

a) Delete 10:



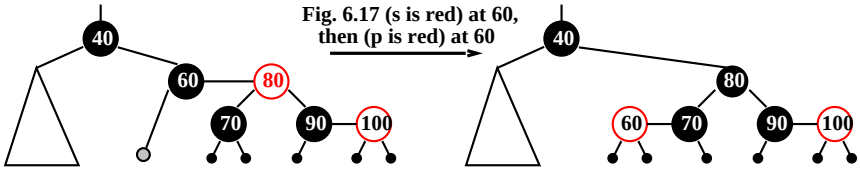
To delete 20, move 30 into 20's node and delete 30's prior node (see next).

Delete 30:



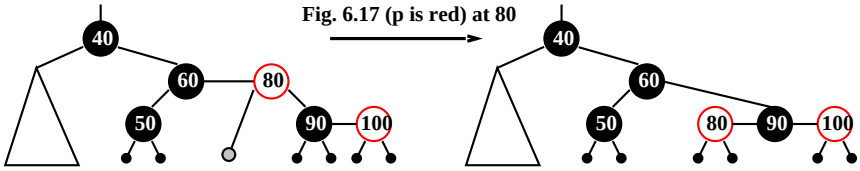
To delete 40, move 50 into 40's node and delete 50's prior node (see next).

Delete 50:



To delete 60, move 70 into 60's node and delete 70's prior node (see next).

Delete 70:

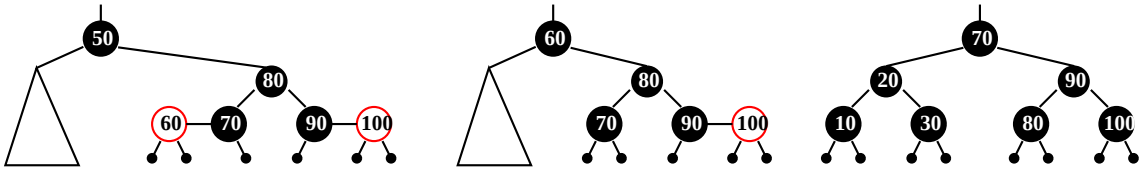


To delete 80, move 90 into 80's node and delete 90's prior node (see next).

To delete 90, simply attach its right subtree, rooted at 100, as the new right subtree of its parent, since its left subtree is **`nil`**. Recolor the right subtree to black.

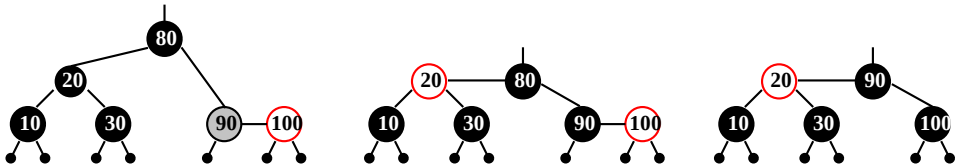
To delete 100, simply attach its right subtree, which is **`nil`**, as the new right subtree of its parent, since its left subtree is **`nil`**.

b) Delete 40 as in part (a), giving the tree on the left, below. Then delete 50 (now in the root) by moving 60 into its place and deleting 60's prior node, giving the tree in the middle. Next, delete 60 (now in the root) by moving 70 into its place and deleting 70's prior node. This leads to the case "*r* is red" in Fig. 6.17 of the text. The result is shown in the right tree.



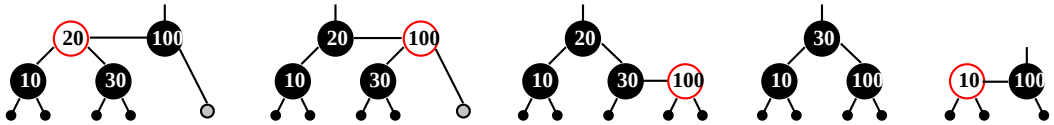
To delete 70 (now in the root), move 80 into its place and delete 80's prior node. After applying the transformation of Fig. 6.16(b) in the text, the result is shown on the left below. In this configuration it is necessary to apply the mirror image of the same transformation, with 90 now as the gray node, leading to the middle tree below. Although 80 is initially gray after the transformation, since it is the root of the entire tree, it is recolored to black, terminating the operation.

80 is deleted without difficulty in the right tree.

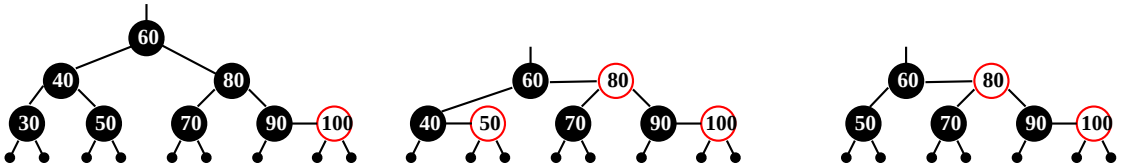


To delete 90 (now in the root), move 100 into its place and delete 100's node, giving the result shown on the left below. Apply the transformation for the mirror image of the case "*s* is red" in Fig. 6.17, leading to the second tree. Then apply the transformation for the mirror image of the case "*p* is red," to the third tree, which completes the operation.

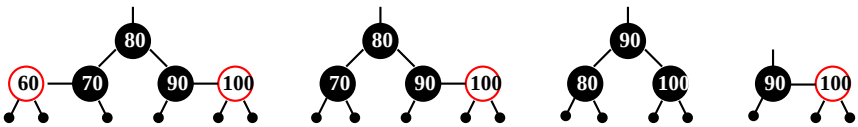
The fourth tree shows 20 being deleted. The fifth tree shows 30 being deleted. Deletion of the last two nodes is simple.



c) Deleting 10 is seen in part (a). Then deleting 20 is simple, and leads to the left tree below. Deleting 30 is similar to deleting 70 in part (b), and leads to the middle tree. Then deleting 40 is simple, and leads to the right tree.



To delete 50, apply the transformation for the case "*s* is red" in Fig. 6.17, then apply the case "*p* is red," leading to the left tree below. Then deleting 60 is simple, shown in the second tree. Deleting 70 uses the case ""*r* is red" in Fig. 6.17, shown in the third tree. Deleting 80 uses Fig. 6.16(b), shown in the fourth tree. The rest is simple.



6.16

- ℓ is red: **rightRotate(**ℓ**,** s**)**, then **leftRotate(**p**,** ℓ**)**. The combination is also called a right double rotation.
- p is red, r is red, or s is red: **leftRotate(**p**,** s**)**.

Section 6.5: *Hashing*

6.18

The operations **hashDelete** and **hashFind** terminate in failure if they encounter an empty cell, whereas **hashInsert** terminates with success in that case.

The operations **hashDelete** and **hashFind** continue over an **obsolete** cell, whereas **hashInsert** terminates with success at such a cell.

The operations **hashDelete** and **hashFind** terminate with success at a cell whose key matches the search key, whereas **hashInsert** continues over it without checking for a match, assuming that duplicate keys are allowed. (We assume that **hashDelete** deletes just one occurrence if the key has multiple matches.)

Two counters, **full** and **empty**, are maintained. $h -$ **full** $-$ **empty** is the number of **obsolete** cells. If **hashInsert** inserts into an empty cell, it increments **full** and decrements **empty**; however, if it inserts into an **obsolete** cell, it only increments **full**. As shown below, **hashDelete** decrements **full**.

If **empty** gets too small, the efficiency of **hashDelete** and **hashFind** will degrade. If **full** is also large, say $\geq \frac{1}{2}h$, then reorganization into a larger array is indicated. However, if **full** is $< \frac{1}{2}h$, then reorganization into an array of the same size might be satisfactory, because all **obsolete** cells are discarded during reorganization.

The following code shows how **obsolete** is handled in **hashDelete**. Code for the other operations follows the same pattern, with the differences mentioned in the paragraphs above.

```
/** empty > 0 */
void  hashDelete(Key K)
     i = hashCode(K);
     while (true)
          if (H[i] == emptyCell)
               // K not there
               break;
          if (H[i] ≠ obsolete)
               if (H[i].key == K)
                    H[i] = obsolete;
                    full --;
                    break;
          i = ((i + 1) % h
          // Continue loop.
```

Section 6.6: *Dynamic Equivalence Relations and Union-Find Programs*

6.20

The point of this exercise is to show that a straightforward matrix implementation is not efficient. The reader should be able to find ways to improve the algorithm below, but the order of the worst case time will still be high compared to the techniques developed in the text.

Use an $n \times n$ matrix A where $A[i][j] = 1$ if i and j are in the same equivalence class, and $A[i][j] = 0$ otherwise. Implementation of an IS instruction is trivial; just examine the corresponding matrix entry. MAKE $i \equiv j$ could be implemented as follows.

```
// Assign to row i the union of rows i and j.
for (k = 1; k ≤ n; k ++)
     if (A[j][k] == 1)
          A[i][k] = 1;
// Copy this union into each row k such that k is in the union.
for (k = 1; k ≤ n; k ++)
     if (A[i][k] == 1)
          Copy row i into row k.
```

The number of matrix entries operated on when implementing MAKE $i \equiv j$ is proportional to $n + pn$ where p is the number of elements in the new equivalence class (the union of i's class and j's class). Consider the following program.

```
MAKE 1 ≡ 2
MAKE 3 ≡ 2
MAKE 4 ≡ 3
  ...
MAKE n ≡ n-1
```

The number of steps is proportional to $\sum_{p=2}^{n}(n+pn) = n(n-1) + n^2(n+1)/2 - n \in \Theta(n^3)$.

6.22

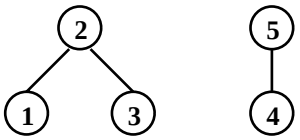
a) Assume the trees are represented by arrays **parent** and **height** where **height[i]** is the height of the tree rooted at i if i is a root (and is irrelevant otherwise).

```
void hUnion(int t, int u)
    if (height[t] > height[u])
        parent[u] = t;
    else
        parent[t] = u;
        if (height[t] == height[u])
            height[u] ++;
```

b) The instructions

```
Union(1,2)
Union(2,3)
Union(4,5)
```

produce the two trees shown in the following figure, no matter whether the height or the number of nodes is used as the weight.



However, if the next instruction is **union(2,5)**, then **wUnion** will produce a tree with root 2, and **hUnion** will produce a tree with root 5.

c) The worst case is in $\Theta(n \log n)$.

We just need to modify the proof of Lemma 6.6 for **hUnion**; Theorem 6.7 completes the argument without modification.

In the proof of Lemma 6.6 modified for **hUnion**, if u was made a child of t, then either $h = h_1$ (because $h_1 > h_2$) or $h_1 = h_2$ and $h = h_1 + 1 = h_2 + 1$ (in this case, u must have been the first argument of the **hUnion**). The proof for the first case is the same as in the text. For the second case, using the induction hypothesis,

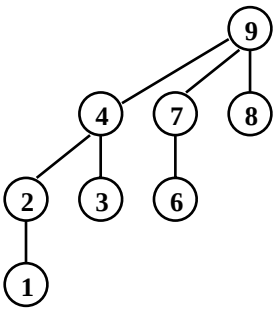
$$h = h_1 + 1 \le \lfloor \lg k_1 \rfloor + 1 \qquad \text{and} \qquad h = h_2 + 1 \le \lfloor \lg k_2 \rfloor + 1$$

So

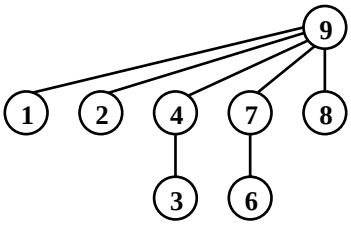
$$h \le \lfloor \lg(2\min(k_1, k_2)) \rfloor \le \lfloor \lg(k_1 + k_2) \rfloor = \lfloor \lg k \rfloor.$$

6.24

Tree before any finds:

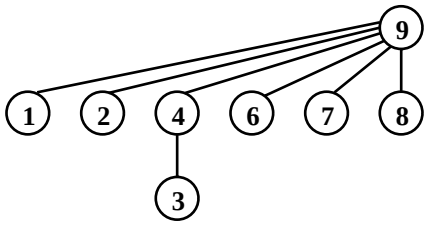


find(1): 4 **parent** accesses.
Resulting tree:



find(4): 2 **parent** accesses.
No change in tree.

find(6): 3 **parent** accesses.
Resulting tree:



find(2): 2 **parent** accesses.
find(1): 2 **parent** accesses.
No changes in tree.

6.26

First we show that an S_k tree can be decomposed as described. The proof is by induction on k. The base case is $k = 1$. An S_1 tree has two nodes. The root r_0 is an S_0 tree, and the handle v is the child of r_0.

Now let T be an S_k tree for $k > 1$ and suppose that any S_{k-1} tree has the described decomposition. T was constructed by attaching the root of one S_{k-1} tree U to the root of another V. The handle of T, which is v, is the handle of U. By the inductive hypothesis, U can be decomposed into $v, T_0, \ldots, T_{k-2}$ as described. Let $T_{k-1} = V$; it satisfies the requirements of the decomposition.

Conversely, if a tree can be decomposed as described, then it is an S_k tree. This can also be proved by a simple induction argument. The base case is $k = 1$. Then T_0 is the only tree in the decomposition and is an S_0 tree, so it consists of one node, r_0. Attaching v to r_0 yields an S_1 tree.

For $k > 1$, suppose that the tree is decomposed into v and trees $\{T_0, \ldots, T_{k-1}\}$ with roots $\{r_0, \ldots, r_{k-1}\}$, respectively, such that T_i is an S_i tree. By the inductive hypothesis, attaching v to r_0, then attaching r_i to r_{i+1} for $0 \le i < k-2$ yields an S_{k-1} tree. (The sequence of attachments indexed by i is empty if $k = 2$.) It is given that T_{k-1} is an S_{k-1} tree. So attaching r_{k-2} to r_{k-1} creates an S_k tree.

6.28

Suppose that T is an S_k tree properly embedded in a tree U, and let r be the root of U. By the decomposition described in Exercise 6.26 (see Fig. 6.29 in the text), the path from T's handle v to the root of T consists of roots of S_i trees for $0 \le i \le k-1$.

cFind($f(v)$ **)** in U causes all these roots to become children of r. That is, the **parent** of each of these roots is r in U'. Then these S_i trees, along with r, satisfy the requirements of the decomposition described in Exercise 6.27 (see Fig. 6.30 in the text), so there is an S_k tree initially embedded in U'. (Notice that v is not part of this S_k tree.)

6.30

Since **wUnion**s take constant time, a worst-case program must have some **cFind**s. Let T be a tree constructed by the **wUnion**s. We will use the name T to mean this tree as it appeared before any **cFind**s are done.

Let T' be the tree that results from **cFind(v)**; in T' all nodes on the path from v to the root of T are now children of the root.

Suppose that **cFind(w)** is done later. If the path in T from w to the root meets the path in T from v to the root, say at x, then **cFind(w)** follows only one link for the overlapping path segment since in T' x is a child of the root. Thus the total amount of work done by the **cFind**s is bounded by a multiple of the number of distinct branches in T. Since each **wUnion** introduces one new branch, the total amount of work done by the **cFind**s is in $O(n)$.

6.32

We will use a Dictionary ADT to keep track of variable and array names and to assign successive integers 1, 2, ..., to them when they are first encountered. So the rest of the program can treat them as vertices in a graph, and can store information about them in arrays, indexed by the vertex number. We initialize a Union-Find ADT with **`create(0)`**.

For each vertex (i.e., variable or array declaration), we keep track of its *leader* and its *offset* from its leader. The *offset* gives the relative memory location of the vertex from its leader. Conceptually,

$$\texttt{offset(u)} = \texttt{address(u)} - \texttt{address(uLdr)}$$

although the appropriate addresses will be determined later by a compiler; our algorithm only works with differences between addresses.

When a declaration is first encountered, it is its own leader and its offset is 0. At this point we execute **`makeSet`** on the new vertex.

When an **`equivalence`** declaration is encountered, say "**`equivalence`** (d_1, d_2, ..., d_k)," we treat it as the sequence of *binary* **`equivalence`** declarations,

`equivalence (`d_1**`,`** d_2**`)`**
`equivalence (`d_1**`,`** d_3**`)`**
...
`equivalence (`d_1**`,`** d_k**`)`**

and we think of each binary **`equivalence`** as an edge between the two vertices whose names occur in the expressions d_i. Assume the underlying vertices are u and v. The exact form of d_1 and d_i determines a necessary offset between u and v, which we'll call **`newOffset(u,v)`**. There are three possibilities for the new offset:

1. The offset is new information.
2. The offset is consistent with an offset between u and v that was determined earlier.
3. The offset is inconsistent with the earlier offset.

Which of these cases applies is determined as follows. First we execute **`find(u)`** and **`find(v)`** to get their current leaders, call them **`uLdr`** and **`vLdr`**.

1. If the leaders are different, this offset is new information.

   We combine the following conceptual equations:

   $$\begin{aligned} \texttt{address(u)} - \texttt{address(uLdr)} &= \texttt{offset(u)} \\ \texttt{address(v)} - \texttt{address(vLdr)} &= \texttt{offset(v)} \\ \texttt{address(u)} - \texttt{address(v)} &= \texttt{newOffset(u,v)} \end{aligned}$$

   to get the new relationship:

   $$\begin{aligned} \texttt{newOffset(uLdr, vLdr)} &= \texttt{address(uLdr)} - \texttt{address(vLdr)} \\ &= \texttt{offset(u)} - \texttt{offset(v)} + \texttt{newOffset(u,v)} \end{aligned}$$

   Notice that we never need any actual addresses to compute **`newOffset(uLdr, vLdr)`**.

   Now we execute **`union(uLdr, vLdr)`**. If the result is that **`vLdr`** becomes the leader of **`uLdr`**, then **`offset(uLdr)`** is updated to become **`newOffset(uLdr, vLdr)`**. If the result is that **`uLdr`** becomes the leader of **`vLdr`**, then **`offset(vLdr)`** is updated to become **`-newOffset(uLdr, vLdr)`**.

2. If the leaders are the same and

   $$\texttt{newOffset(u,v)} = \texttt{offset(u)} - \texttt{offset(v)},$$

   then the new offset is consistent with the earlier offset. No more updates are required for this **`equivalence`**.

3. If the leaders are the same and

   $$\texttt{newOffset(u,v)} \neq \texttt{offset(u)} - \texttt{offset(v)},$$

   then the new offset is inconsistent with the earlier offset. This **`equivalence`** should be flagged as an error.

As usual with the Union-Find ADT, the leader information that is stored for a vertex might be out of date until a **`find`** is executed for that vertex. Similarly, the **`offset`** will be out of date in these cases. It is necessary to update offsets during a **`find`** according to the rule

```
offset[v] = offset[v] + offset[parent[v]];
```

In **cFind**, this statement would be placed right before the statement that updates **parent[v]** in Fig. 6.22 of the text.

Section 6.7: *Priority Queues with a Decrease Key Operation*

6.34

The point of this exercise is to show how **xref** is updated in tandem with the steps of the usual heap deletion procedure. Elements are shown in the format id(priority) for clarity.

First we save $K = 7(6.4)$, set **xref[7] = 0**, and reduce **heapSize** to 4. Next, **vacant = 1** and **xref[4] = 0** to delete element 4(3.2).

Then 3(4.5) is compared to 1(7.4) and 3(4.5) is compared to K and 3(4.5) moves up to **vacant**. This requires the update **xref[3] = 1**. Now **vacant = 2**.

Then 5(5.4) has no sibling to compare with, so 5(5.4) is compared to K and 5(5.4) moves up to **vacant**. This requires the update **xref[5] = 2**. Now **vacant = 4**.

Since **vacant** is a leaf now, we store K in **vacant** and update **xref[7] = 4**.

6.36

Yes, it is always optimal for $n = 2^k$. The n inserts create n trees of one node each and use no comparisons. The **getMax** reorganizes the elements into one tree in which the root has $\lg(n)$ children. It uses $n-1$ comparisons. The **deleteMax** removes the root and leaves a forest in which the children of the former root are now roots, so there are $\lg(n)$ trees at this point. The second **getMax** performs $\lg(n)-1$ comparisons. The total number of comparisons is $n - \lg(n) - 2$. Since $\lg(n)$ is an integer, this agrees with Theorem 5.2.

If n is not an exact power of 2, then the above argument still holds, replacing $\lg(n)$ by $\lceil \lg(n) \rceil$, but this was not required for the exercise. In this case, depending on the outcomes of comparisons, the total might be one less than the worst case, just as it would be in the tournament method of Section 5.3.2.

6.37

An obsolete node is no longer referenced by the **xref** array. It occurs somewhere in a list of trees. The main idea is that it might be expensive to remove it from the list at the time it becomes obsolete, so we wait until we are traversing the list and come across it. Luckily, the first time we come across it we will be able to delete it cheaply. This exercise fleshes out the details.

a) To see that a lot of obsolete nodes can become roots of trees, suppose we start with one tree whose root r has m children, $v_1, \ldots, v_m$.

Execute **decreaseKey** on each v_m creating m obsolete nodes in the children of r, ensuring that all of the keys become less than r. Do additional **decreaseKey**s on all the descendants of the v_i, making them less than r. Now do a series of **getMax** and **deleteMax** until r is again the root of the only tree, and the obsolete nodes are still children of r.

Now do **deleteMax** and the m obsolete nodes are roots. Now **isEmpty** will require time in $\Theta(m)$.

The code for **isEmpty** could be:

```
boolean isEmpty(PairingForest pq)
    pq.forest = bypass(pq.forest);
    return (pq.forest == nil);

TreeList bypass(TreeList remForest)
    if (remForest == nil)
        return (remForest)
    else if (root(first(remForest)).id ≠ -1)
        return (remForest);
    else
        return bypass(rest(remForest));
```

b) The argument that **pairForest** does not run in $O(k)$ is similar to part (a). Create a forest with k genuine trees and m obsolete trees. The cost is $\Theta(k+m)$ but m can be arbitrarily large in relation to k. The code, using **bypass** from part (a), could be:

```
TreeList  pairForest(TreeList oldForest)
    TreeList  newForest, remForest;
    newForest = nil; remForest = bypass(oldForest);
    while (remForest ≠ nil)
        Tree  t1 = first(remForest);
        remForest = bypass(rest(remForest));
        if (remForest == nil)
            newForest = cons(t1, newForest);
        else
            Tree  t2 = first(remForest);
            remForest = bypass(rest(remForest));
            Tree  winner = pairTree(t1, t2);
            newForest = cons(winner, newForest);
        // Continue loop.
    return newForest;
```

c) Charge **decreaseKey** an accounting cost of +2 for creating an obsolete node, in additional to the actual cost of 1. Charge the **bypass** subroutine an accounting cost of –2 for each obsolete node that it encounters (and discards, since it returns a list without that node), in addition to the actual cost of 2 for testing and discarding the node. Thus **bypass** runs in amortized time $O(1)$. To verify that the obsolete nodes really are discarded we note that any list passed to **bypass** is never accessed again, either because that variable is never accessed again or because the variable is overwritten with a new list from which the obsolete nodes seen by **bypass** have been removed. There are only four places to check, so this verification is straightforward. Therefore the cumulative accounting cost can never be negative.

Since **bypass** runs in amortized time $O(1)$, **isEmpty** does also. Moreover, **pairForest** runs in amortized time $O(k)$ because its loop executes about $k/2$ times and each pass runs in amortized time $O(1)$. Here we are using the facts that **pairTree**, **cons**, **first**, and **rest** all run in time $O(1)$.

Additional Problems

6.40

It is straightforward for red-black trees to implement an elementary priority queue with each operation running in time $O(\log n)$. To find the minimum element simply follow left subtrees to the internal node whose left subtree is **nil**. The usual method can be used to delete this node. The usual method can be used to insert new elements and test for emptiness.

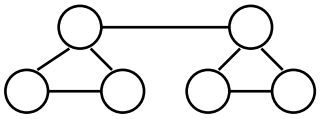
For a full priority queue use an **xref** array similar to the one described in the text except that **xref[i]** refers to the subtree whose root holds element i. Whenever the red-black tree undergoes any structural change, as in rebalancing, the **xref** array has to be kept up to date. But each node contains the element's id, as well as its priority, and the id is the index in **xref**, so updating **xref** costs $O(1)$ per structural change. There is no problem doing **getPriority** in $O(1)$, through **xref**. Treat **decreaseKey** as a delete (locating the node through **xref** instead of by a BST search), followed by an insert of the same id with the new priority. Thus **decreaseKey** runs in $O(\log n)$.

Thus all operations have the same asymptotic order as they would for the binary-heap implementation of priority queues.

Chapter 7
Graphs and Graph Traversals

***Section 7.2*:** *Definitions and Representations*

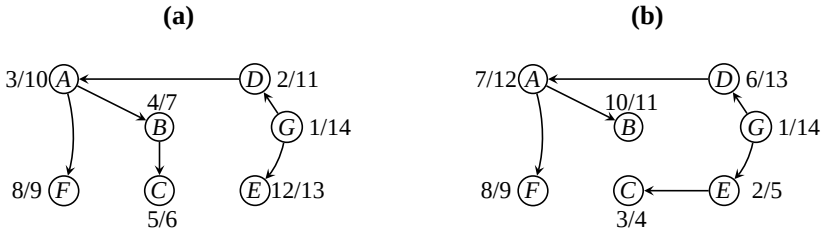
7.1



7.3
If v and w are any two distinct vertices of G and there is a path from v to w, then there is an edge vw in G.

***Section 7.3*:** *Traversing Graphs*

7.4
The diagrams show discovery and finishing times, which were not asked, but which help to interpret the DFS tree.



7.6
It is always true that $height(T_D) \geq height(T_B)$. This follows from the following lemma.

Lemma:
Let v be the root of a breadth-first search tree for a graph G. For any vertex x in the tree, the path from v to x using tree edges is a shortest path from v to x in the graph. (The length of a path from v to x is simply the number of edges in the path, and the length of the shortest path is called the *distance* from v to x.)

Proof The proof is by induction on k, the distance of a vertex x from v. The base case is $k = 0$. The empty path is the shortest path from v to v and is the tree path from v to v.

Suppose the distance of a vertex x from v in G is $k > 0$. Then x cannot already be in the BFS tree at a depth of $k-1$ or less. Let u be the next-to-last vertex on a shortest path from v to x in G. Then the distance from v to u is $k-1 \geq 0$, so by the inductive hypothesis, u is at depth $k-1$ in the breadth-first search tree. Since x is adjacent to u, x will be put on depth k of the tree, so its distance from v in the tree is k. □

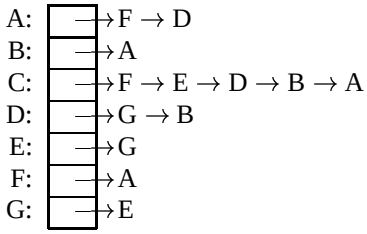
Thus the height of a breadth-first search tree is the distance of the farthest vertex from the root. No spanning tree of G with the same root can have smaller height.

***Section 7.4*:** *Depth-First Search on Directed Graphs*

7.8
The algorithm goes through the adjacency lists of the given graph and constructs a new array of adjacency lists for the transpose. For each vertex w found on vertex v's list, it puts v on w's list in the new structure.

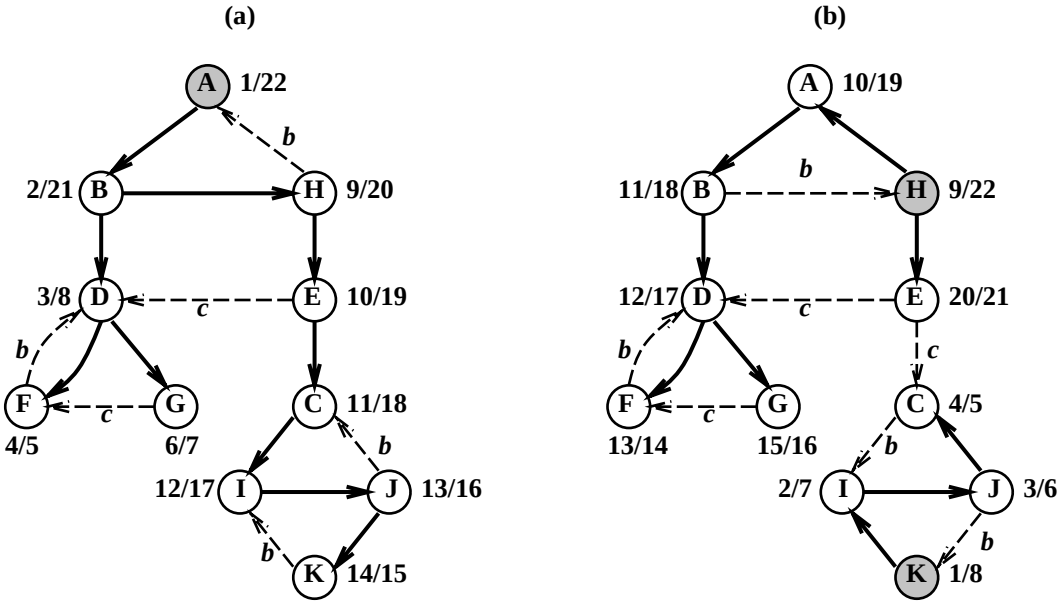
a)
```
IntList[] transpose(IntList[] adjVerts, int n)
   IntList[] trAdjVerts = new IntList[];

   for (v=1; v ≤ n; v++)
       trAdvVerts[v] = nil;
   for (v=1; v ≤ n; v++)
       remAdj = adjVerts[v];
       while (remAdj ≠ nil)
           int w = first(remAdj);
           trAdjVerts[w] = cons(v, trAdjVerts[w]);
           remAdj = rest(remAdj);
   return trAdjVerts;
```

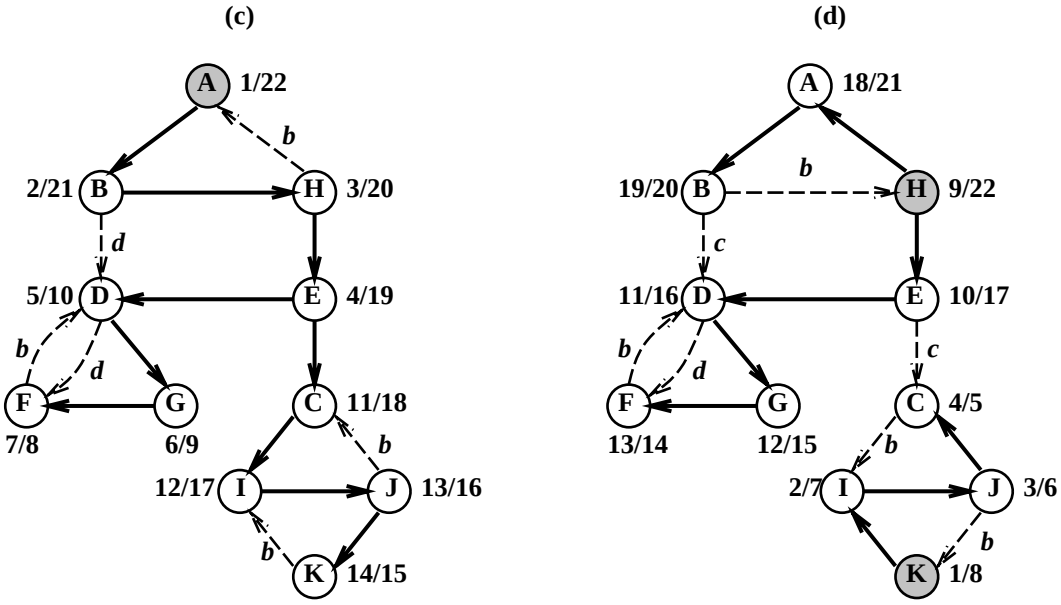
b) Assume the *array* of adjacency lists for G indexes vertices in alphabetical order. The adjacency lists in the transpose graph will be in reverse alphabetical order because **cons** puts new nodes at the beginning of a list. The order of the actual adjacency lists for G is irrelevant. The adjacency lists of G^T are as follows:



7.10
Gray.

7.12
Roots of the DFS trees are shaded. Heavy solid edges are trees edges; dotted edges are nontree edges; their labels are *b* for back edge, *c* for cross edge, and *d* for descendant (forward) edge.





Notes for instructor: Note that only part (d) has all four edge types. Small changes can cause all four edge types to appear in the other parts without having much effect on the solution.

- For part (a), reversing the *JC* edge makes it a descendant edge.
- For part (b), reversing the *JK* edge makes it a descendant edge.
- For part (c), reversing the *EC* edge makes it a cross edge, makes a second DFS tree, and revises the times as follows: E(4/11), H(3/12), B(2/13), A(1/14), C(15/22), I(16/21), J(17/20), K(18/19).

We would probably only assign one part of this exercise for homework and use another part for an exam, each time the course is taught, and shuffle them around from term to term.

7.14

Solution 1 (follows the hint). Let P_1 and P_2 be the unique simple paths from the root to v and w, respectively. Let c be the last vertex on path P_1 that also appears in P_2. Node c exists because at least the root of the tree is on both paths, and c is a common ancestor of v and w. By the definition of c, no node following c on path P_1 appears on P_2 at all, so in particular, it does not appear on the part of P_2 that follows c. Therefore the part of P_1 following c is disjoint from the part of P_2 following c.

Solution 2. Treat the tree as a directed graph, do a depth-first search, and use the results about *active* intervals in Theorem 7.1 of the text. The active intervals of v and w are disjoint, and are contained in the active interval of the root. Let c be node with the smallest active interval that contains the active intervals of both v and w; there can't be a tie because active intervals are either nested or disjoint—they can't partially overlap.

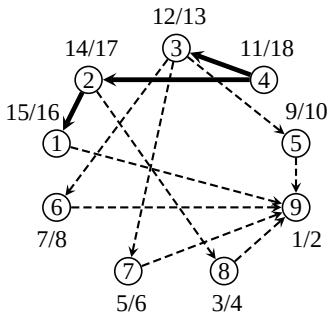
The active intervals of the children of c are disjoint, one of them contains the active interval of v and some other one contains the active interval of w. So the paths from c to v and c to w are disjoint except for c.

7.16

Modify the directed DFS skeleton, Algorithm 7.3, as follows. At "Exploratory processing" (line 7), insert a statement to output the edge vw. (Lines 2, 9, 11, and 13 and **ans** are not needed.) Insertion at line 9 would also work.

7.18

There are six trees; the first five have only one vertex each.



vertex	1	2	3	4	5	6	7	8	9
topo	7	8	6	9	5	4	3	2	1

7.20

a) There must be at least one vertex with an empty adjacency list. (Otherwise there would be a cycle in the digraph.) If there is more than one vertex with an empty adjacency list, neither can reach the other, so the DAG is not a lattice. So, go through the adjacency list array and determine if there is exactly one vertex, say v whose list is empty.

We continue only if we found a unique vertex v with an empty list. The vertex v can be reached by following edges from each other vertex because there are no cycles and no other dead ends.

Now construct the adjacency lists for the transpose digraph. (See Exercise 7.8.) This takes $\Theta(n+m)$ time. Then check if there is exactly one vertex in the transpose graph with an empty adjacency list. If so, that vertex can reach all others in the original digraph.

b) Checking the adjacency list arrays for empty lists takes $\Theta(n)$ time. Construction of the adjcency lists for the transpose digraph takes $\Theta(n+m)$ time. The total is in $\Theta(n+m)$.

c) There is exactly one vertex with an empty adjacency list, vertex 9. If the edges are all reversed, only vertex 4 will have an empty adjacency list. (This may be easier to see by looking at the adjacency lists in Example 7.14.) Thus the digraph in Example 7.15 is a lattice.

***Section 7.5*:** *Strongly Connected Components of a Directed Graph*

7.22

Let G be a directed graph, and suppose $s_0, s_1, \ldots, s_k (= s_0)$ is a cycle in the condensation of G. We can assume $k \geq 2$ because a condensation has no edges of the form vv. Let S_i be the strong component of G that corresponds to the vertex s_i in the condensation ($0 \leq i \leq k$, $S_k = S_0$). For each i ($0 \leq i \leq k-1$), since there is an edge $s_i s_{i+1}$ in the condensation, there is an edge $v_i w_i$ in G where v_i is in S_i and w_i is in S_{i+1}. Since each S_i is strongly connected there is a path in G from w_i to v_{i+1}, which we will indicate by $w_i \to v_{i+1}$ (where we let $v_k = v_0$).

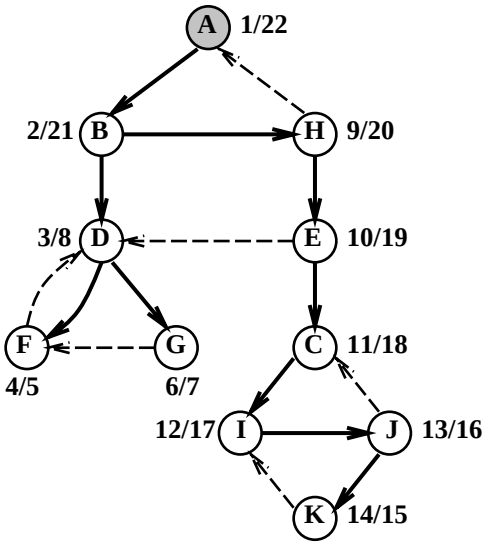
Thus we can form the following cycle in G: $v_0, w_0 \to v_1, w_1 \to \ldots \to v_{k-1}, w_{k-1} \to v_k (= v_0)$. But then all the vertices in this cycle should have been in the same strong component, so the original cycle assumed to be in the condensation of G cannot exist.

7.24

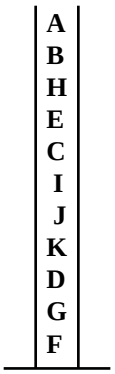
The strong components are the same, of course, for each part of the problem. They are $\{A, B, H\}$, $\{D, F, G\}$, $\{E\}$, and $\{C, I, J, K\}$ (with the appropriate edges).

In the diagrams the roots of the DFS trees are shaded, tree edges are heavy solid, and nontree edges are dashed. The **`finishStack`** at the end of Phase 1 is shown. The DFS trees in the transpose graph each encompass one SCC. We show the algorithm's output, the **`scc`** array, containing the leaders for the components, under each diagram.

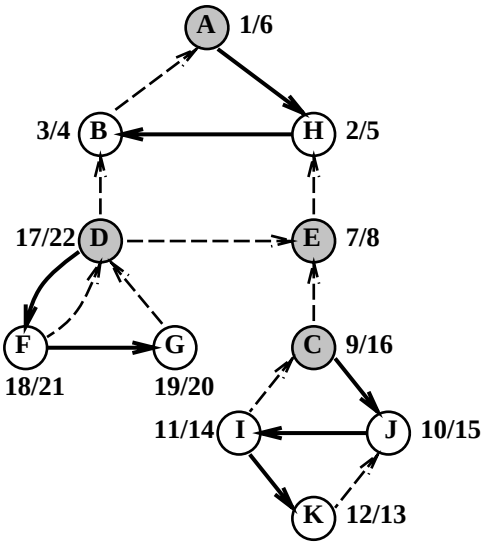
(a)



Original graph



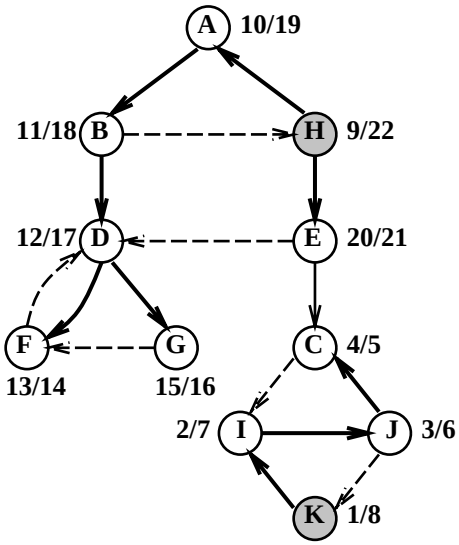
finishStack



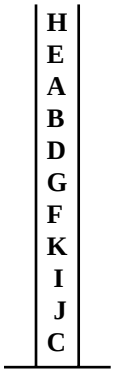
Transpose graph

	A	B	C	D	E	F	G	H	I	J	K
scc:	A	A	C	D	E	D	D	A	C	C	C

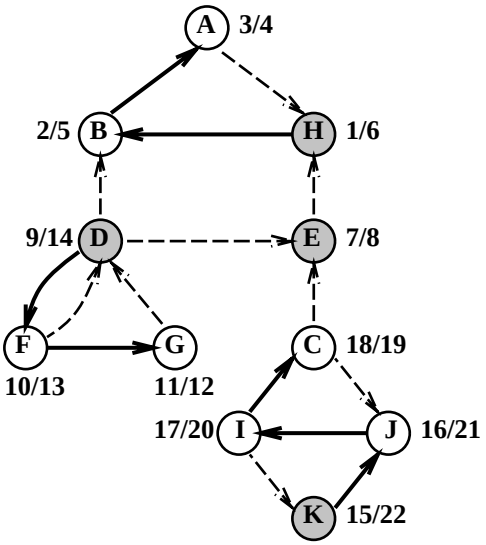
(b)



Original Graph



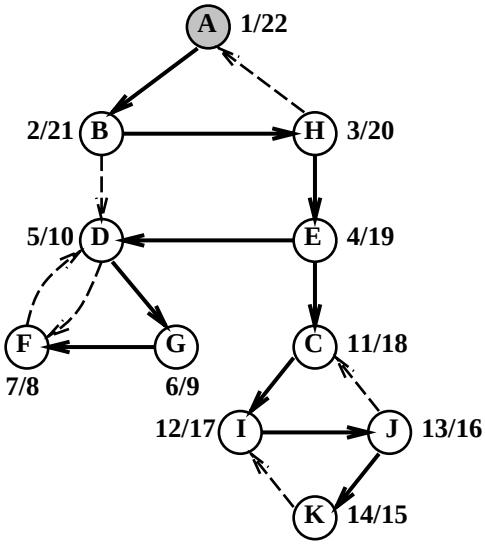
finishStack



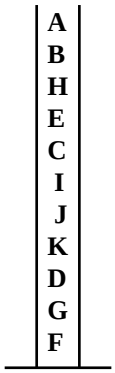
Transpose Graph

	A	B	C	D	E	F	G	H	I	J	K
scc:	H	H	K	D	E	D	D	H	K	K	K

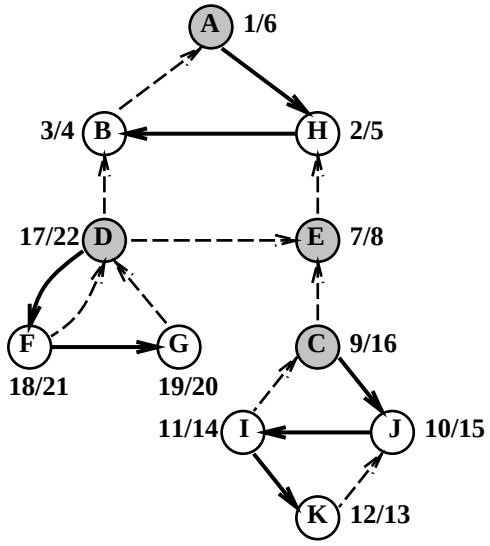
(c)



Original Graph



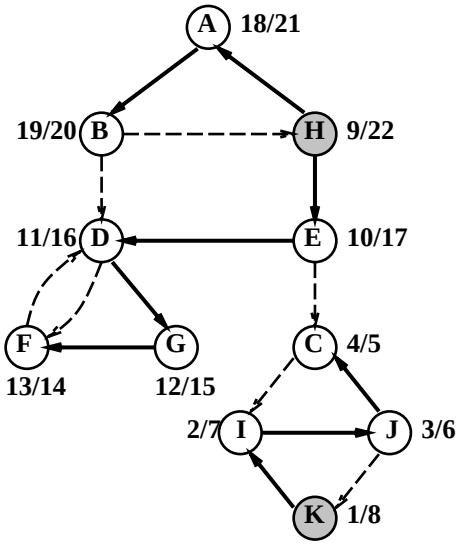
`finishStack`



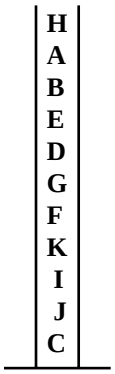
Transpose Graph

scc:	A	B	C	D	E	F	G	H	I	J	K
	A	A	C	D	E	D	D	A	C	C	C

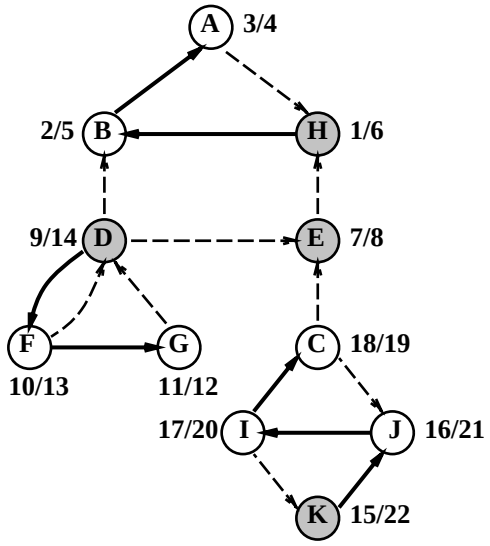
(d)



Original Graph



`finishStack`



Transpose Graph

scc:	A	B	C	D	E	F	G	H	I	J	K
	H	H	K	D	E	D	D	H	K	K	K

Note for instructor: Like Exercise 7.12, we would probably only assign one part of this exercise for homework and use another part for an exam, each time the course is taught, and shuffle them around from term to term.

Section 7.6: *Depth-First Search on Undirected Graphs*

7.27

Modify the undirected DFS skeleton, Algorithm 7.8, as follows. At "Exploratory processing" (line 7) and "Checking" (line 11) insert statements to output the edge vw. (Lines 2, 9, and 13 and **ans** are not needed.)

Notice that outputting the edge vw as soon as it is obtained from the adjacency list (i.e., after line 5 of the skeleton) will not work for undirected graphs, although it would work for directed graphs. This shows the value of working with the skeleton and inserting code only in the indicated places.

7.28

Let vw be an edge. Without loss of generality, we assume that v is encountered before w in the depth-first search.

Case 1: The edge vw is first encountered while traversing the adjacency list of v.

At this point w could not yet have been visited because, by the recursive nature of the algorithm, if w had been visited while traversing subtrees of v, a complete depth-first search from w would have been completed before returning to v and the edge $wv(=vw)$ would have been encountered first from w. Since w has not been visited, it will be visited immediately as a child of v and vw will be a tree edge.

Case 2: The edge $wv(=vw)$ is first encountered while traversing the adjacency list of w.

Since v was visited before w, at the time when wv is first encountered, the depth-first search from v must still be in progress; otherwise vw would have already been encountered from v. Thus v is an ancestor of w, so wv is a back edge.

7.30

Use the connected component algorithm (Algorithm 7.2) modified to quit with a negative answer if any back edge is encountered or if any vertices remain unmarked after the first call to **ccDFS**.

If it is known that G is connected, then G is a tree if and only if $m = n - 1$. (If m is not given as input, count the edges.)

Section 7.7: *Biconnected Components of an Undirected Graph*

7.32

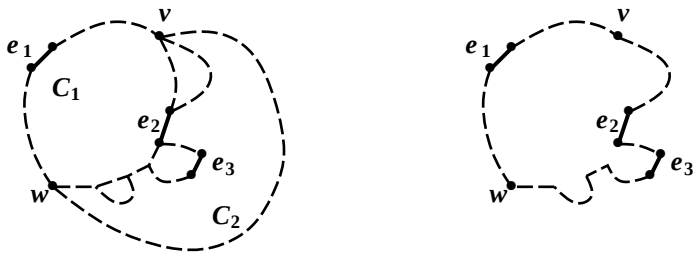
D, A, I.

7.34

a) The reflexive property is satisfied because we are given that $e_1 R e_2$ if $e_1 = e_2$. The symmetric property is satisfied because if a cycle contains e_1 and e_2, then it contains e_2 and e_1. For the transitive property we must show that if there is a simple cycle containing e_1 and e_2 and a simple cycle containing e_2 and e_3, then there is a simple cycle containing e_1 and e_3. Since we require a simple cycle, this is not trivial.

Let C_1 be a simple cycle containing e_1 and e_2.

If e_3 is in C_1 also, we are done. Suppose it is not. Let C_2 be a cycle containing e_2 and e_3. We will show that there is a simple cycle C that contains e_1 and e_3. C is constructed by combining a piece of C_1 with a piece of C_2. The following figure illustrates the construction described in the next paragraph.



In C_1, begin at e_1, and continue around C_1 until reaching the first vertex, say v, that is also in C_2. Continue around C_1 to find the last vertex, say w, that C_1 and C_2 have in common. (They must have at least two common vertices because e_2 is on both cycles.) Now break C_1 at v and w. It falls into two pieces; let C_{11} be the piece that contains e_1. Break C_2 at v at w. It also falls into two pieces; let C_{23} be the piece that contains e_3. Attach C_{11} and C_{23} at v and w to form the cycle C.

C is a simple cycle because no vertex of C_2 appears on C_{11} (except the endpoints v and w).

b) The graph in the diagram in the text has two equivalence classes, the two triangles.

c) Let H be the subgraph based on such an equivalence class, i.e., any two edges in H are in some simple cycle, or H contains only one edge. We will use the facts that H contains at least two vertices and every vertex in H is incident upon *some* edge in H. We show first that H is biconnected, then that it is a maximal biconnected subgraph (i.e., a biconnected component).

If H contains only one edge, it is biconnected by definition, so assume that H has more than one edge for the rest of this paragraph. H is connected because for any two distinct vertices v and w in H, v and w are incident on edges in a cycle. Because the cycle is simple, there are two disjoint paths between v and w (disjoint other than the endpoints v and w, of course), so the loss of any one other vertex leaves a path between v and w. Thus H is biconnected.

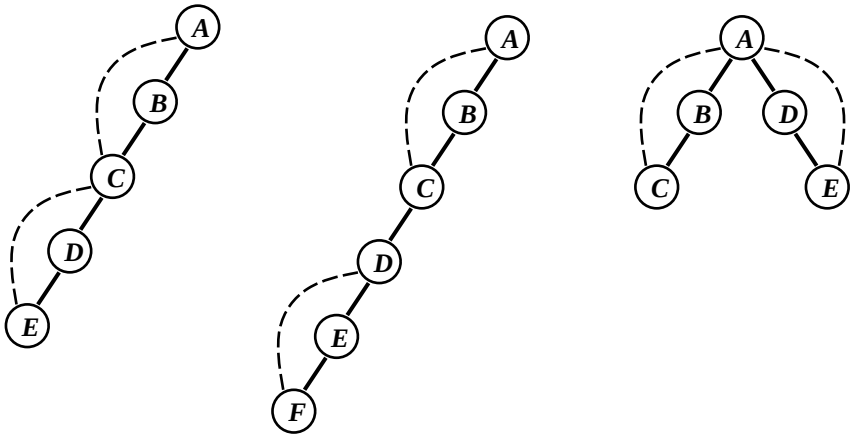
Now suppose H is properly contained within J, a larger subgraph of G that is also biconnected. J must include an edge that is not in H (since if it contained only additional vertices, it would not be connected).

Case 1: All vertices in J are also in H. Let xy be an edge in J that is not in H. The vertices x and y are in H, which is connected, so there is a path between x and y in H. We can append the edge xy to form a simple cycle in G, so xy should have been in H.

Case 2: J has at least one vertex not in H. J is connected, so let xy be an edge with one vertex, x, in H and the other, y, in J but not H. Let v be any other vertex in H adjacent to x. Because J is biconnected, there must be a simple path from v to y in G that does not use x. This path forms a simple cycle when the edges xv and xy are included, so again xy should have been in H.

7.35

Consider the following undirected graphs, in which the solid lines are edges in a DFS tree rooted at A and dashed lines are back edges.



a) In the leftmost depth-first search tree, $old_1[B] = A = old_1[D]$, but B and D are not in the same biconnected component.

In the middle tree, C and D (and the edge CD) form one bicomponent, but $old_1[C] = A$ and $old_1[D] = D$.

b) In the rightmost DFS tree, $old_2[B] = A = old_2[D]$, but B and D are not in the same biconnected component.

In the middle tree, C and D (and the edge CD) form one bicomponent, but $old_2[C] = A$ and $old_2[D] = D$.

7.37
The root of a depth-first search tree for a graph is an articulation point if and only if the root has at least two children.

Proof. If there is only one child, the subtree rooted at that child contains all the other vertices, and since it is a tree, there is a path between each pair of vertices (other than the root) that does not go through the root. So the root is not an articulation point. If the root has two (or more) children, there can be no path from the first child to the second child that does not go through the root. If there were, the second child of the root would have been encountered while doing the depth-first search from the first child and would have been put in the first subtree. Thus, if there is more than one child, the root is an articulation point.

A similar exercise is: Can a leaf in a depth-first search tree for a connected graph be an articulation point?

7.39
Consider the leftmost depth-first search tree shown in the solution of Exercise 7.35, and assume the vertices on each adjacency list are in alphabetical order. If an edge is stacked on **`edgeStack`** each time it is encountered, then after the depth-first search backs up from E, **`edgeStack`** contains

(bottom) *AB, BA, BC, CA, CB, CD, DC, DE, EC, ED* (top).

When the search backs up from D to C, it detects the bicomponent containing the vertices C, D, and E, and it pops the top five entries from the stack. Only then, when the search is ready to branch out again from C is the edge CE encountered and stacked. It will be popped later as part of the other bicomponent.

7.40
No. Consider the leftmost graph shown in the solution of Exercise 7.35. Let $v = C$ and $w = D$. At the time the algorithm backs up from D to C, **`back`** = 1 (the "back" value for C), so the test suggested in the exercise would fail to detect the bicomponent consisting of C, D, and E.

7.41

a) True except for a graph consisting of exactly one edge. Such a graph is biconnected, but removing the edge would leave two isolated vertices. If a biconnected graph has more than one edge, every edge is part of a cycle, so removal of any one edge can not disconnect the graph.

b) Not true. Consider a graph consisting of two triangles that share one vertex (the diagram in Exercise 7.34(b) in the text). The shared vertex is an articulation point, but the graph is edge-biconnected.

Additional Problems

7.43
The general strategy of both solutions below is to first eliminate all but one vertex as possible hypersinks, then check whether the remaining vertex is actually a hypersink. The first solution is a little bit easier to follow and to write out; the second is a little faster. They both are linear in n.

Let $A = (a_{ij})$ be the adjacency matrix ($1 \le i, j \le n$). Observe that for $i \ne j$, if $a_{ij} = 0$, then v_j cannot be a hypersink, and if $a_{ij} = 1$, then v_i cannot be a hypersink. So each probe in the matrix can eliminate one vertex as a possible hypersink. (The definition of *hypersink* does not specify anything about edges of the form *ss*, so the solution given allows *ss* to be present or not. With minor changes *ss* can be considered to be required or prohibited.)

Solution 1. Let v_s be a possible hypersink. We move down column s as long as we see 1's. When a 0 is encountered, we jump right to the column of the next possible hypersink (the row number where we encountered the 0) and continue moving down the column.

```
boolean hsink;
int s;   // possible hypersink
s = 1;
while (s < n)
    int i;
    hsink = true; // s is a candidate for the hypersink
    for (i = s+1; i ≤ n; i++)
        if (a[i][s] == 0)
            hsink = false;
            break;
        // Continue inner loop.
    if (hsink)
        // All vertices except s were eliminated.
        break;

    // No vertex with index < i (including s) can be a hypersink.
    s = i;
    // Continue outer loop.

// Now check whether s is indeed a hypersink.
hsink = true;
for (i = 1; i ≤ n; i++)
    if (i ≠ s && a[i][s] == 0)
        hsink = false;
    if (i ≠ s && a[s][i] == 1)
        hsink = false;
return hsink;
```

The elimination phase of the algorithm examines $n-1$ matrix entries (one from each row other than the first row). The verification phase examines $2n-2$ entries. The total is $3n-3$.

Solution 2. The elimination phase can be conducted like a tournament (where examining an entry a_{ij} is a "match" between v_i and v_j, and the vertex that is not eliminated is the "winner." In Section 5.3.2 we saw that the overall winner of a tournament (in this problem, the one possible hypersink) will have competed directly against roughly $\lg n$ other vertices. The relevant matrix entries do not have to be reexamined in the verification phase. So the problem can be solved by examining at most $3n - \lceil \lg n \rceil - 3$ matrix entries. (For details see King and Smith-Thomas, "An optimal algorithm for sink-finding," *Information Processing Letters*, May 1982.)

7.45

For simplicity we will output the edges of the desired path in sequence. Modify the undirected DFS skeleton, Algorithm 4.8, as follows.

- At "Exploratory processing" (line 7) insert a statement to output the edge *vw*.
- At "Backtrack processing" (line 9) insert a statement to output the edge *wv*.
- At "Checking" (line 11) insert statements to output the edge *vw*, then the edge *wv*.

For example, if the graph has edges (1,2), (1,3) and (2,3) and the DFS starts at 1, the outputs would be 12 23 31 13 32 21.

7.47

This problem is a good example of the power of strongly connected component analysis. First find the SCCs of the directed graph G and create the condensation graph (denoted $G\downarrow$ in the text), using $\Theta(n+m)$ time. Now take the transpose (reverse the edges) of the condensation graph, giving $G\downarrow^T$. In $G\downarrow^T$, determine how many vertices have no outgoing edges. If there is just one such vertex, say s, then it (or any vertex in its SCC) can reach all vertices in the original graph.

The explanation is that every vertex can reach s in $G\downarrow^T$ because the graph is acyclic and has no other dead end. So s can reach every vertex in $G\downarrow$. So any vertex in the SCC of s can reach every vertex in G.

7.49

Use depth-first search. For each vertex v, keep a count of how many descendants of v (including v itself) have been visited. This can be done by initializing v's descendant counter to 1 when v is first visited and adding the descendant counter for each of its children when the search backs up from the child to v. Each time the search backs up to v test if $count[v] \geq n/2$. If the test fails, another, unvisited, subtree of v could have more than $n/2$ vertices, so we cannot conclude that v satisfies the required property, and the traversal continues. If the test is satisfied, the algorithm outputs v and halts. We will show that v satisfies the property.

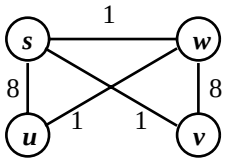
When v is removed, each subtree of v will be a separate component, and the ancestors of v along with their descendants (other than v and its subtrees) will be one other component. Since we have already visited at least $n/2$ descendants of v, each subtree of v that has not yet been visited has at most $n/2$ vertices, and the component that contains ancestors of v has at most $n/2$ vertices.

Let w be the root of a subtree of v that has been visited. When the search backed up to w, `count`$[w]$ was found to be less than $n/2$ (otherwise the algorithm would have quit there and output w), so the subtree rooted at w has fewer than $n/2$ vertices. Thus, when v is removed, each separate component has at most $n/2$ vertices.

Chapter 8
Graph Optimization Problems and Greedy Algorithms

Section 8.2: *Prim's Minimum Spanning Tree Algorithm*

8.1
Start at *s*.



8.3
Add xy to T_2 forming a cycle. By the MST property all edges in the cycle have weights $\leq W(xy)$. At least one edge in the cycle is not in T_1, say zw. But zw is in T_2. So $W(xy) \geq W(zw) \geq W(uv)$.

8.5
Note: Part (a) of this exercise was updated in the second printing. Be sure that the errata in the first printing (available via Internet) are applied if you are using the first printing.

a) Let G_n be a simple cycle of n vertices; that is, there is an edge between the ith and $i+1$st vertices, for $1 \leq i \leq n-1$ and an edge between 1 and n. Then there are at most two candidate edges to choose from at one time, so each of the n **`getMin`**s requires at most one comparison of edge weights. (We adopt the convention given in part (b) of the exercise, that a comparison of an edge weight with ∞ does not count.) The **`updateFringe`** subroutine only does an edge-weight comparison for the last vertex to enter the fringe.

   For a more general class, let C be some constant, and consider the class of connected undirected graphs consisting of the union of at most C simple cycles. Then there are at most two fringe vertices per simple cycle at any one time and **`updateFringe`** does at most one edge-weight comparison per simple cycle. If the graph has n vertices, the total is about $2Cn$.

b) Let G_n be a simple chain of n vertices; that is, there is an edge between the ith and $i+1$st vertices, for $1 \leq i \leq n-1$. Let the start vertex be 1.

Remarks: Focusing on weight comparisons, as is done in this exercise and in several others in the chapter, requires some justification. If the weights are floating point, weight comparisons are substantially more expensive than integer or boolean tests. But this is only a partial justification, because it addresses a constant factor, not the asymptotic order of the algorithm.

In fact, as given in the text, Prim's algorithm does "bookkeeping" work that requires time in $Theta(n^2)$, so even if the number of weight comparisons is of a lower order, the bookkeeping work will dominate for large enough n.

More sophisticated implementations can make the bookkeeping time proportional to the number of weight comparisons plus n. (We can't omit "plus n" because of pathological examples which the number of weight comparisons is $o(n)$, such as we saw in part (b) above.) There are three reasons why the text does not give one of these more sophisticated implementations for the "standard" version of Prim's algorithm or Dijkstra's algorithm:

1. We preferred to present the main ideas as simply as possible.
2. There is no broad class of graphs for which we know that the average size of the fringe is in $o(n)$. (We mean the average over the course of the algorithm, not the average over some set of inputs.) If the average size of the fringe is in $\Omega(n)$, then any implementation of **`getMin`** that checks all of the fringe vertices will put the overall cost in $\Theta(n^2)$, and we can only hope for improving the constant factor.
3. Section 8.2.8 outlines the best ways that are known to improve the performance of Prim's and Dijkstra's algorithms, using a Pairing Forest or a Fibonacci Heap, which in turn are discussed in Section 6.7.2. These methods don't check all of the fringe vertices.

With that said, we can briefly mention a few ways to reduce the bookkeeping in **`getMin`** to be proportional to the size of the fringe. With one of these alternatives in place, evaluating the time of the algorithms by considering only edge-weight comparisons (plus a linear term to cover pathological cases) would be fully justified.

One alternative is to use a (mutable) linked list to hold the fringe vertices. It is also possible to use the List ADT in the text, which is immutable, by incorporating the idea of obsolete nodes described in Section 6.7.2. We believe that the overhead of lists and complications of implementing all the cases correctly make these alternatives less desirable than the alternative described below.

Another alternative is to maintain the set of fringe vertices in an array, say **`inFringe`**. A counter, **`fringeSize`**, keeps track of how many fringe vertices there are at any moment, and they are stored in the range $1, \ldots,$ **`fringeSize`** of **`inFringe`**. Then **`findMin`** finds the vertex in **`inFringe`** with the minimum **`fringWgt`** and swaps it with the vertex in **`inFringe[fringeSize]`**. Later, **`deleteMin`** can simply subtract 1 from **`fringeSize`**. Finally, **`insert`** adds new vertices at the end, incrementing **`fringeSize`**.

It would require extensive experimentation to determine under what circumstances these techniques produce improved performance. Limited empirical results indicate that the last technique would be profitable when the average fringe size is $n/20$ or less, but we do not know what graphs tend to have this property.

8.7

All $n-1$ edges immediately become candidate edges, but each time a candidate edge is selected, every candidate edge is involved in a weight comparison to find the candidate with minimum weight. The total number of weight comparisons is $(n-2)(n-1)/2$.

8.8

a) Each time a vertex is taken into the tree (after the start vertex), its edge to every vertex not yet in the tree is involved in a weight comparison in **`updateFringe`**. This accounts for $(n-2)+(n-3)+\cdots+1$ weight comparisons, or $(n-1)(n-2)/2$ in all. In addition, **`getMin`** does $(n-2)+(n-3)+\cdots+1$ weight comparisons over the course of the algorithm. So the total is $(n-1)(n-2)$.

b) All of them.

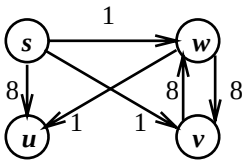
8.10

In Algorithm 8.1 delete "**`if (pq.status[w] == unseen) insert(pq, w, v, newWgt); else`**" as the "if" will always be false. In the **`create`** function, initialize **`pq.numPQ = n`**, **`status[v] = fringe`** and **`fringeWgt[v] =`** ∞ for $1 \leq v \leq n$. The asymptotic order is unchanged.

***Section 8.3*:** *Single-Source Shortest Paths*

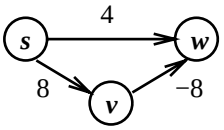
8.12

The source vertex is s.



8.14

Dijkstra's shortest path algorithm will not work correctly with negative weights. Consider the following figure.



The algorithm, with s as the start vertex, will pull w into the tree with the path of length 4, and then pull v into the tree, so the edge with negative weight, which gives a shorter path to w, never becomes a candidate.

Another exercise: Review the proof of Theorem 8.6 (the correctness argument for Dijkstra's algorithm) to find where it uses the hypothesis that the weights are nonnegative.

8.16

We use the notation of **`dijkstraSSSP`** and Section 8.3.2. Suppose that Dijkstra's algorithm adds vertices to its tree in the sequence $v_0(=s), v_1, \ldots, v_{n-1}$. (If some vertices are unreachable from s they are added arbitrarily to the end of the sequence.) We will show that after k times through the main loop the algorithm has computed the correct shortest path distances for the vertices $v_0, \ldots, v_k$.

The proof is by induction on k, the index of this sequence. The base case is $k = 0$. The shortest path from s to s is 0 and $d(s,s) = 0$ before the main loop.

For $k > 0$, assume the theorem holds for $k-1$. Then the hypotheses of Theorem 8.6 hold with v_k in the role of z and $\{v_0, \ldots, v_{k-1}\}$ in the role of V'. Therefore, if the candidate edge that is taken in the k-th pass through the main loop is tv_k, then $d(s,t) + W(tv_k)$ is the shortest path distance from s to v_k in G. This is the value that the algorithm stores for $d(s,v_k)$, so the algorithm has computed the shortest paths for $v_0, \ldots, v_k$.

If there are no candidate edges remaining at the beginning of the k-th pass through the main loop, then the algorithm terminates. At this point, shortest path distances have been computed correctly for $\{v_0, \ldots, v_{k-1}\}$ (by the inductive hypothesis) and they are ∞ for $\{v_k, \ldots, v_{n-1}\}$.

8.18

We use the notation of **`dijkstraSSSP`** and Section 8.3.2. Use an array **`count`** such that for any vertex u, **`count[u]`** is the number of distinct shortest paths from s to u known so far. Initialize **`count[s] = 1`**. Note that two paths are called *distinct* if they have any diffence at all; they may partially overlap. We assume there are no edges of weight 0; otherwise there can be an infinite number of shortest paths.

When a vertex x is added to the tree, x's adjacency list is traversed. Let y be a vertex on the list. There are three cases to handle.

1. y was an unseen vertex.

   There were no paths to y known previously. Now each best path to x gives a best path to y, so

   ```
   count[y] = count[x];
   ```

   This is where the count is initialized for each vertex other than s.

2. y is a fringe vertex, and $d(s,x) + W(xy) < d(s,y)$.

   The edge xy replaces the current candidate to y; the paths that had been the best paths to y are discarded, but each best path to x gives a best path to y, so again

   ```
   count[y] = count[x];
   ```

3. y is a fringe vertex, and $d(s,x) + W(xy) = d(s,y)$.

   The current candidate edge to y is not replaced, but all shortest paths to x with xy appended on the end give paths to y as short as the best already known. Therefore **`count[y]`** is updated as follows:

   ```
   count[y] += count[x];
   ```

8.20

We use the notation of **`dijkstraSSSP`** and Section 8.3.2.

a) This solution uses an $n \times n$ weight matrix W to represent the graph. The status array is replaced by a **`boolean`** array **`inFringe`**.

```
void sssp(float[][] W, float[] dist, int[] parent, int s, int n)
    int x, y, treeSize;
    boolean[] inFringe = new boolean[n+1];
    // Initialization
    for (y = 1; y ≤ n; y++)
        dist[y] = ∞;
        parent[y] = -1;
        inFringe[y] = true;

    // Main loop
    dist[s] = 0;
    for (treeSize = 0; treeSize < n; treeSize++)
        x = findMin(dist, inFringe);
        inFringe[x] = false;
        updateFringe(W, dist, parent, n, x);
    return;

/** Scan vertices with inFringe[x] == true to find one for which
  dist[x] is minimum; return this vertex.  */
int findMin(float[] dist, boolean[] inFringe)
    int x, y;
    float best;
    x = -1; best = ∞;
    for (y = 1; y ≤ n; y ++)
        if (inFringe[y] && dist[y] < best)
            best = dist[y];)
            x = y;
        // Continue loop.
    return x;

/** Replace appropriate candidate edges with edges from x.  */
void updateFringe(float[][] W, float[] dist, int[] parent, int n, int x)
    int y;
    for (y = 1; y ≤ n; y ++)
        float newDist = dist[x] + W[x][y];
        if (newDist < dist[y])
            parent[y] = x;
            dist[y] = newDist;
        // We did not have to explicitly test whether y is in the fringe or the
        // tree because if it is in the tree, the current path to y (of length
        // dist[y]) cannot have higher weight than the path through x to y.
        // Continue loop.
```

b) The procedure in part (a) is clearly simpler than Algorithm 8.2. However, since all vertices are in the fringe at the beginning, and only one is removed during each pass of the main loop, this algorithm will always do a total of about $\frac{1}{2}n^2$ weight comparisons when searching the fringe vertices to find the candidate edge with lowest weight. Algorithm 8.2 will do substantially fewer weight comparisons on some graphs, although we do not know any general class of graphs for which this is true.

Focusing on weight comparisons requires some justification because the bookkeeping in both algorithms takes time in $\Theta(n^2)$. As discussed in the solution of Exercise 8.5, more sophisticated implementations can make the bookkeeping time proportional to the number of weight comparisons. Also, if the weights are floating point, then weight comparisons are substantially more expensive than integer or boolean tests.

Like any algorithm that treats the graph as a complete graph, this one uses $\Theta(n^2)$ space for the weight matrix. If the actual graph is sparse, there is a lot of wasted space.

***Section 8.4*:** *Kruskal's Minimum Spanning Tree Algorithm*

8.22

If e forms a cycle with edges in F, then there must be a path from v to w in F, so v and w are in the same connected component of F.

On the other hand, if v and w are in the same connected component of F, there is a path between them that does not use e (since e is not in F). Adjoining e to the path gives a cycle.

8.24

The equivalence classes other than singletons are shown after each union.

Take AB, (AB).
Take EF, (AB, EF).
Take EK, (AB, EFK).
Drop FK.
Take GH, (AB, EFK, GH).
Take GL, (AB, EFK, GHL).
Take GM, $(AB, EFK, GHLM)$.
Drop HL.
Take BC, $(ABC, EFK, GHLM)$.
Take CM, $(ABCGHLM, EFK)$.
Take DJ, $(ABCGHLM, DJ, EFK)$.
Take FG, $(ABCGHLMEFK, DJ)$.
Take JM, $(ABCGHLMEFKDJ)$.
Drop LM, AH, CD, CJ, HM.
Take AI, $(ABCGHLMEFKDJI)$.

This is a spanning tree now so remaining edges can be ignored.

Additional Problems

8.26

Dijkstra's shortest path algorithm could be used with all edge weights assigned 1, but there is a lot of unneeded overhead. The simplest solution is to use breadth-first search starting at s. When w is encountered, the path in the breadth-first search tree is a shortest path from s to w. There is no straightforward way to use depth-first search.

8.27

Use a dynamic equivalence relation (Section 6.6 of the text), where two vertices are in the same equivalence class if and only if they are in the same connected component of the graph consisting of the edges read in so far.

```
Read (n, m);
count = 0;
for (i = 1; i ≤ m; i ++)
     Read (v, w);
     if (IS(v, w) == false)
          MAKE v ≡ w;
          count ++;

if (count == n-1)
     connected = true;
else
     connected = false;
```

Each IS is implemented with two **cFind**s and each MAKE uses two **cFind**s and one **wUnion**. Thus this algorithm includes a *Union-Find* program with at most $2m + 3(n-1)$ instructions. The worst-case running time is in

$$O\left((n+m)\lg^*(n+m)\right),$$

where $\lg^*$ is the slowly-growing function defined in Definition 6.9 (Section 6.6).

Chapter 9

Transitive Closure, All-Pairs Shortest Paths

Section 9.2: *The Transitive Closure of a Binary Relation*

9.2

The point of this exercise is to see that depth-first search does not give a particularly good solution to the problem, so we will not give a complete solution. Since the two solutions given here in part (a) are shown not to be very good in parts (b) and (c), some students may come up with different solutions; the two given here should be taken with a grain of salt.

a) We would expect students to try the following procedure (to be called repeatedly to start new depth-first searches from vertices that have not yet been visited), perhaps with the embellishment described after it. We will see when we look at the example in part (c) that this is not enough; some paths will be missed. An alternative second solution is sketched briefly, but part (c) shows that it also misses some paths.

Solution 1. R is the (global) reachability matrix being computed. The algorithm is written recursively for clarity, but it accesses the vertices on the recursion stack (the vertices on the path from the root to the current vertex). This could be implemented by not using built-in recursion and manipulating the stack directly, but we use a linked list, named **`pathList`** to store these vertices. The list operates like a stack except that all elements can be inspected without modifying the list. Assume the usual initializations done in **`dfsSweep`**, and **`pathList = nil`**.

```
void reachFrom(int v)
     int w;
     IntList remAdj;

     // Number v and fill matrix entries for its ancestors.
     color[v] = gray;
     time ++; discoverTime[v] = time;
     For all u in pathList:  R[u][v] = 1;
     pathList = cons(v, pathList);  // "push" v.

     // Consider vertices adjacent to v.
     remAdj = adjacencyList[v];
     while (remAdj ≠ nil)
          w = first(remAdj);
          if (color[w] == white)
               reachFrom(w);
          else if (discoverTime[w] < discoverTime[v])
               // vw is a back edge or cross edge
               For all u in pathList down to w (or to the end):  R[u][w] = 1;
          remAdj = rest(remAdj);
     time ++; finishTime[v] = time;
     pathList = rest(pathList);  // "pop" v.
     color[v] = black;
```

Solution 2. Initialize R to zeros, except 1's on the main diagonal (in time $\Theta(n^2)$). Perform a depth-first search according to the DFS skeleton, Algorithm 7.3. Except for the postorder processing, to be described in a moment, this take time in $\Theta(n+m)$.

At postorder time for vertex v (line 13 in the DFS skeleton) we compute row v of the matrix R as follows: In a loop through v's adjacent list, for every vertex w that is adjacent to v (via an out-edge), "or" row w of R into row v of R. That is, if $R[w][i] = 1$ set $R[v][i]$ to 1 for $1 \leq i \leq n$. The cost of this "step" for v is n times the out-degree of v; the cost for the whole graph of these "steps" is in $\Theta(nm)$. (We put "step" in quotes because it is a **`for`** loop within a **`while`** loop—quite a substantial "step.")

b) For solution 1, there are never more than n vertices on the stack, so the worst case time is in $O(nm)$, where n is the number of vertices and m is the number of edges.

For solution 2, the times for various parts were given in part (a), and the total is $\Theta(n(n+m))$.

c) For solution 1, if **`reachFrom`** is called with v_1 as the root, the depth-first search tree will contain all five vertices, but the paths from v_2, v_3 and v_4 to v_5 will not be recorded in R. For solution 2, *any* path that requires a back edge will be missed.

The simple way to overcome the bugs in these solutions is to start a depth-first search from *each* vertex. This obviously will do some repeated work, but it is about the best solution known for sparse graphs and has the same asymptotic order on dense graphs ($O(n^3)$) as Warshall's algorithm, which is studied in this chapter. Warshall's algorithm and other methods studied in this chapter introduce algorithmic themes that are useful in other situations. The themes of this chapter are revisited in Chapters 12 and 14.

***Section 9.3*:** *Warshall's Algorithm for Transitive Closure*

9.4

Note to instructor: It is surprisingly difficult, if not impossible, to find a graph for which Algorithm 8.1 requires more than four passes through the **`while`** loop, including the pass to detect that nothing changed. Regardless of how large n is, the algorithm seems to find long paths very quickly.

Note that a path beginning at any vertex, followed by zero or more vertices in *descending* order, followed by zero or more vertices in *ascending* order will be found in one pass of the **`while`** loop, no matter how long the path is.

It is not easy to predict the course of the algorithm "by eye." Unless students check their constructions by programming and running Algorithm 8.1, they are likely to jump to the wrong conclusions (as both authors have, on various tries).

For purposes of making Algorithm 8.1 run for as many passes as possible through its **`while`** loop, it is sufficient to consider graphs whose directed edges form a single simple path through all n vertices. Such a graph is specified by merely giving a permutation of $1, \ldots, n$.

Solution 1. Here are the edges for a directed graph with 7 vertices that forces Algorithm 8.1 to make four passes through the **`while`** loop, including a last pass in which nothing changes:

$$3 \to 5 \to 7 \to 2 \to 4 \to 6 \to 1$$

The final R matrix is:

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

As we mentioned, the authors have not been able to find a graph (with any number of vertices) that requires five or more passes.

Solution 2. Program Algorithm 8.1, but instead of reading in a graph, have a procedure to generate one. For a sequence of p beginning at 2 and going as high as your computer can stand it, set $n = 2^p$ and generate a random permutation of vertices 1 through n. Make $n-1$ directed edges between adjacent vertices in the random sequence, from left to right. Run Algorithm 8.1 on this graph and report how many passes through the **`while`** loop were used. Do about 10 such runs for each p to get an idea of what is happening. Then do more runs for values where you hope to find a higher maximum than you found in your small sample. Plot the maximum number of passes observed for each p. Form a conjecture about the general case, noting that $p = \lg n$.

Solution 3. Let q be about $\sqrt{n}$. Make directed edges as follows. $(1, q+1)$, (q+1, 2q+1),. . . ,((q-2)q+1, (q-1)q+1), *then*((q-1)q+1, 2)*towraparound*, *then*(2, q+2), $(q+2, 2q+2)$, . . . , $((q-2)q+2, (q-1)q+2)$, then $((q-1)q+2, 3)$ to wrap around, and so on. (Solution 1 uses a variation of this idea, and there are many other variations.)

How many passes will the **`while`** loop take in Algorithm 8.1? It can be shown, in this case, that at most four passes are needed, for arbitrarily large n.

Section 9.4: *All-Pairs Shortest Paths in Graphs*

9.6

Let $V = \{1,2,3,4\}$, let $E = \{12, 24, 43\}$, and let the weight of each edge be 1.

$$\begin{array}{ccc} 1 & \rightarrow & 2 \\ & & \downarrow \\ 3 & \leftarrow & 4 \end{array}$$

Initially $D[2][3] = D[1][3] = D[1][4] = \infty$. The algorithm, with the modification described in the exercise, computes $D[1][3]$ before it computes $D[2][3]$ and $D[1][4]$. It will incorrectly leave $D[1][3]$ with the value ∞.

9.7

$$D = \begin{pmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & 7 & 3 \\ 5 & 4 & 0 & 3 \\ 4 & 1 & 4 & 0 \end{pmatrix}$$

9.8

a) The matrices below show the steps to calculate the distance matrix. That is, the first shows the matrix after all iterations of the inner **for** loops for $k = 1$, the second after the inner loops for $k = 2$, etc. The last matrix is the distance matrix.

$$\begin{bmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & 7 & 3 \\ 5 & 7 & 0 & -3 \\ \infty & -1 & 4 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & 7 & 3 \\ 5 & 7 & 0 & -3 \\ 2 & -1 & 4 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 2 & 4 & 1 \\ 3 & 0 & 7 & 3 \\ 5 & 7 & 0 & -3 \\ 2 & -1 & 4 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 4 & 1 \\ 3 & 0 & 7 & 3 \\ -1 & -4 & 0 & -3 \\ 2 & -1 & 4 & 0 \end{bmatrix}$$

b) As long as there are no negative-weight cycles, there is always a *simple* shortest path between any pair of nodes. Algorithm 9.4 is guaranteed to find the shortest simple path, regardless of whether weights are negative or not.

9.10

The following algorithm computes the routing table **go** as well as the distance matrix D. Initially, D contains the weight matrix for the graph.

```
void routeTable(float[][] D, int[][] go, int n)
    int i, j, k;


// Initialize go:  If edge ij exists, first step from i is to j.
    for (i = 1; i ≤ n; i++)
        for (j = 1; j ≤ n; j++)
            if (D[i][j] == ∞)
                go[i][j] = -1;
            else
                go[i][j] = j;


// Find shortest paths and record best first step in go.
    for (k = 1; k ≤ n; k++)
        for (i = 1; i ≤ n; i++)
            for (j = 1; j ≤ n; j++)
                if (D[i][j] > D[i][k] + D[k][j])
                    D[i][j] = D[i][k] + D[k][j];
                    go[i][j] = go[i][k];
```

9.12

The shortest nonempty path from one vertex to itself is the shortest cycle. In Algorithm 9.4, if we initialize $D[i][i] = \infty$ for $1 \leq i \leq n$ (after copying W into D), then run this algorithm, the final value in $D[i][i]$ is the length of a shortest cycle containing i, as long as there are no negative-weight cycles. There is a negative-weight cycle if and only if some $D[i][i] < 0$.

If the graph is undirected, the algorithm will incorrectly interpret an edge vw as a cycle v, w, v.

Although the exercise did not explicitly require discussion of negative-weight cycles, it is worth knowing that they can be dealt with fairly straightforwardly.

Should a negative-weight cycle be discovered when Algorithm 9.4 terminates, then other nodes might be in non-simple negative-weight cycles that have not been discovered yet. One simple way to find all such nodes is to set $D[i][i]$ to $-\infty$ wherever $D[i][i] < 0$ and use this matrix as input for a second run of Algorithm 9.4. (This time, of course, do not set $D[i][i] = \infty$ before starting. We might need to evaluate $\infty + (-\infty)$; the appropriate value is ∞, denoting that the path doesn't exist.)

***Section 9.5*:** *Computing Transitive Closure by Matrix Operations*

9.14

Let $P = v_0(= v), v_1, \ldots, v_k(= w)$ be a path from v to w with $k > n$. Since there are only n vertices in the graph, some vertex must appear more than once on the path. Suppose $v_i = v_j$ where $i < j$. Then $v_0, \ldots, v_i, v_{j+1}, \ldots, v_k$ is also a path from v to w, but its length is strictly smaller than k. If the length of the new path is still larger than n, the same shortening process (removing loops in the path) can be used until the length is less than n.

***Section 9.6*:** *Multiplying Bit Matrices—Kronrod's Algorithm*

9.16

Element $j \in C_i$ if and only if $\bigvee_{k=1}^{n} a_{ik} b_{kj} = 1$ if and only if there is a k such that $a_{ik} = 1$ and $b_{kj} = 1$ if and only if there is a $k \in A_i$ such that $j \in B_k$ if and only if $j \in \bigcup_{k \in A_i} B_k$.

Additional Problems

9.18

If the graph is undirected, compute the Boolean matrix product A^3 and see if there is a 1 on the main diagonal.

If the graph is directed and we interpret a triangle as a *simple* cycle with three edges, we must avoid using self-loops. So copy the adjacency matrix to a new matrix T, replace all main-diagonal entries with zero, compute T^3, and see if there is a 1 on the main diagonal.

If straightforward matrix multiplication is used, the number of operations is in $\Theta(n^3)$.

Chapter 10
Dynamic Programming

Section 10.1: *Introduction*

10.2

```
if (n ≤ 1) return n;
prev2 = 0; prev1 = 1;
for (i = 2; i ≤ n; i ++)
     cur = prev1 + prev2;
     prev2 = prev1;
     prev1 = cur;
return cur;
```

Section 10.3: *Multiplying a Sequence of Matrices*

10.4

Let $n = \texttt{high} - \texttt{low}$ be the problem size. When $k = \texttt{low} + 1$, line 7 of the **mmTry2** procedure in the text makes a recursive call with problem size $n-1$. When $k = \texttt{high} - 1$, line 6 makes another recursive call with problem size $n-1$. The loop beginning at line 5 executes $n-1$ times, and makes 2 recursive calls in each pass, for a total of $2n-2$. For $n \geq 2$, we have $2n-2 \geq n$. Therefore, letting $T(n)$ denote the number of recursive calls made by **mmTry2** on a problem of size n:

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

For an easy lower bound, we claim $T(n) \geq 2^n - 2$ for $n \geq 1$. The proof is simple by induction, since $2^1 - 2 = 0$ and $2\left(2^{n-1}-2\right)+n = 2^n + n - 4 \geq 2^n - 2$ for $n \geq 2$.

10.5

$$\texttt{cost} = \begin{bmatrix} . & 0 & 600 & 2800 & 1680 \\ . & . & 0 & 1200 & 1520 \\ . & . & . & 0 & 2400 \\ . & . & . & . & 0 \\ . & . & . & . & . \end{bmatrix} \qquad \texttt{last} = \begin{bmatrix} . & -1 & 1 & 1 & 1 \\ . & . & -1 & 2 & 3 \\ . & . & . & -1 & 3 \\ . & . & . & . & -1 \\ . & . & . & . & . \end{bmatrix} \qquad \texttt{multOrder} = \begin{bmatrix} . & 2 & 3 & 1 \end{bmatrix}$$

10.7

Let the dimensions of A, B, and C be 100×1, 1×100, and 100×1, respectively.

Section 10.4: *Constructing Optimal Binary Search Trees*

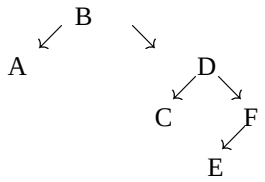
10.9

a) In the matrix below, expressions $p+q=w$ mean that $p = p(i,j)$, the weight of elements i through j, and q is the minimum weighted cost of two subtrees, for various choices of root from i through j, so the w is the optimum weighted cost for subproblem (i,j).

$$\texttt{cost} = \begin{bmatrix} . & . & . & . & . & . & . \\ 0 & 0.20 & \begin{matrix}0.44+0.20\\=0.64\end{matrix} & \begin{matrix}0.60+0.36\\=0.96\end{matrix} & \begin{matrix}0.88+0.80\\=1.68\end{matrix} & \begin{matrix}0.92+0.88\\=1.80\end{matrix} & \begin{matrix}1.00+1.08\\=2.08\end{matrix} \\ . & 0 & 0.24 & \begin{matrix}0.40+0.16\\=0.56\end{matrix} & \begin{matrix}0.68+0.52\\=1.20\end{matrix} & \begin{matrix}0.72+0.60\\=1.32\end{matrix} & \begin{matrix}0.80+0.72\\=1.52\end{matrix} \\ . & . & 0 & 0.16 & \begin{matrix}0.44+0.16\\=0.60\end{matrix} & \begin{matrix}0.48+0.20\\=0.68\end{matrix} & \begin{matrix}0.56+0.32\\=0.88\end{matrix} \\ . & . & . & 0 & 0.28 & \begin{matrix}0.32+0.04\\=0.36\end{matrix} & \begin{matrix}0.40+0.16\\=0.56\end{matrix} \\ . & . & . & . & 0 & 0.04 & \begin{matrix}0.12+0.04\\=0.16\end{matrix} \\ . & . & . & . & . & 0 & 0.08 \\ . & . & . & . & . & . & 0 \end{bmatrix}$$

$$\texttt{root} = \begin{bmatrix} . & . & . & . & . & . & . \\ -1 & 1 & 2 & 2 & 2 & 2 & 2 \\ . & -1 & 2 & 2 & 3 & 3 & 4 \\ . & . & -1 & 3 & 4 & 4 & 4 \\ . & . & . & -1 & 4 & 4 & 4 \\ . & . & . & . & -1 & 5 & 6 \\ . & . & . & . & . & -1 & 6 \\ . & . & . & . & . & . & -1 \end{bmatrix}$$

b)



10.10

We build a binary search tree where each node has a key field K_k. The **buildBST** function is called with

```
T = buildBST(1, n).
```

Here's the function **buildBST**. To simplify the notation, we let the matrix **root** be global.

```
/** buildBST constructs the tree in postorder using the global root matrix.
  The current subtree being constructed will contain the keys
  K_first, ..., K_last.
  */
BinTree buildBST (int first, int last)
    BinTree ans;
    if (last < first);
        ans = nil;
    else
        int k = root[first][last]; // index of the root's key
        BinTree leftSubtree = buildBST(first, k-1);
        BinTree rightSubtree = buildBST(k+1, last);
        ans = buildTree(K_k, leftSubtree, rightSubtree);
    return ans;
```

Since there is no loop within the procedure and each call generates one new node, the running time is in $\Theta(n)$.

10.12

For keys in the range **first** through **last**, choose the most probable key in the range as the root of the BST. Suppose its index is k. Then recursively build the left subtree for the range **first** through $k-1$ and build the right subtree for the range $k+1$ through **last** (using the same greedy strategy).

The strategy is nonoptimal for the input $A(.41)$, $B(.40)$, $C(.20)$ because it will make a stringy tree to the right with weighted cost $0.41+2*0.40+3*0.20=1.81$. The balanced BST with root B has weighted cost $2*0.41+0.40+2*0.20=1.62$.

***Section 10.5*:** *Separating Words into Lines*

10.14

a)

words	X	penalty
Those who	7	343
cannot remember	1	1
the past are	4	64
condemned to	4	64
repeat it	0	0
		472

b) The subproblems that are evaluated are those beginning at words 11, 10, 9, 8, ..., 1, including the main problem. Each subproblem is evaluated once in the DP version, so the total is 11.

c) In the cost calculations below for the non-DP recursive version, "Tree k" refers to the subproblem tree rooted at subproblem k and the cost is the number of nodes in the tree. Each node represents one activation frame.

Tree 11 costs 1.
Tree 10 costs 1.
Tree 9 costs 1.
Tree 8 costs 1 + Tree 9 + Tree 10 = 3.
Tree 7 costs 1 + Tree 8 + Tree 9 + Tree 10 = 6.
Tree 6 costs 1 + Tree 7 + Tree 8 = 10.
Tree 5 costs 1 + Tree 6 + Tree 7 + Tree 8 = 20.
Tree 4 costs 1 + Tree 5 + Tree 6 = 31.
Tree 3 costs 1 + Tree 4 + Tree 5 = 52.
Tree 2 costs 1 + Tree 3 + Tree 4 = 84.
Tree 1 costs 1 + Tree 2 + Tree 3 + Tree 4 = 168.

Additional Problems

10.16

It is not hard to see that if $k > n$ then $C(n,k) = 0$. We use this fact in some of the solutions. However, we are given the precondition that $n \geq k$, so other cases are computed only at the algorithm's convenience.

1. For Method 1, the recursive function definition is:

```
int C(int n, int k)
    if (n ≥ 0 && k == 0)
        ans = 1;
    else if (n == 0)
        ans = 0;
    else
        ans = C(n-1, k-1) + C(n-1, k);
    return ans;
```

As with the Fibonacci numbers, this is a very inefficient computation; the number of recursive calls to C (and the number of additions) will in general be exponential in n. This is true because $C(n,k) > (n/k)^k$, and there will be more than $C(n,k)$ leaves in the recursive call tree.

Since the depth of the call tree will be n, $\Theta(n)$ stack space is used.

2. For Method 2, we could use a table $C[0\ldots n][0\ldots k]$ with all entries in the lefthand column initialized to 1, and all entries in the top row, other than the first, initialized to 0. To compute an entry $C[i][j]$ only two neighbors in the previous row are needed.

   So the table can easily be filled until $C[n][k]$ is computed. At most nk entries are computed, so at most nk additions are done.

   Since the computation of each entry needs only entries in the previous row, at most two rows need be stored, so this method used $O(k)$ space.

   A little thought shows that this solution can be improved. All entries above the diagonal consisting of $C[0][0]$, ..., $C[k][k]$ are equal to 0 and are not needed in the computation. All entries on the diagonal are equal to 1; these can be initialized. Entries below the diagonal consisting of $C[n-k][0],\ldots,C[n][k]$ are not needed in the computation. Thus only entries in a strip of $n-k+1$ diagonals need be computed. That includes $k(n-k+1)$ entries, hence $k(n-k+1)$ additions. With careful indexing, the two partial rows the algorithm is using at any one time can be stored in $2(n-k+1)$ cells.

   This method is clearly superior to the first one.

3. Method 3 could be implemented with one loop in $\Theta(k)$ time as follows:

```
if (k > n/2) k = n-k;
C = 1;
for (i = 0; i < k; i ++)
     C = (C * (n-i)) / (i+1);
```

   Note the parenthesization. After j passes through the loop, $C = \binom{n}{j}$, which is an integer, so the integer division is precise. Because k is made less than $n/2$, the value of C increases monotonically. However, intermediate values will be about k times as large as C and may cause an integer overflow that might have been avoided in the first two methods.

   This method is clearly faster than the first two and uses less space (essentially none). The only drawback is that for a small range of inputs it will overflow when the first two methods do not. The critical range is

$$\frac{\texttt{maxint}}{k} < \binom{n}{k} \leq \texttt{maxint}.$$

   The moral is that mathematical transformations may pay bigger dividends than clever programming.

4. Method 4 could be implemented with simple loops in $O(n+k)$ time as follows:

```
// Compute n!
numerator = 1;
for (i = 2; i ≤ n; i ++)
     numerator = numerator * i;
// Compute k!  * (n-k)!
denominator = 1;
for (i = 2; i ≤ k; i ++)
     denominator = denominator * i;
for (i = 2; i ≤ n-k; i ++)
     denominator = denominator * i;
// Compute C
C = numerator / denominator;
```

   However, if n and k are large, computing **numerator** could cause integer overflow, even when the quantity $C(n,k)$ is well within the range of values that can be represented on the machine. If floating point arithmetic is used there is the possibility that the answer will be inaccurate and might not be an integer. Floating point numbers even overflow their *exponent* parts for $n!$ when n has fairly modest values.

10.18

Many students try to solve this problem by a brute force method using index variables to keep track of the current location in each string (more or less generalizing Algorithm 11.1, the brute force string matching algorithm). All such

solutions that I have seen were wrong; they missed some valid matches. The worst case time for a correct brute force algorithm would probably be cubic.

Our first solution (dynamic programming) uses a table $S[0\ldots n][0\ldots m]$ where $S[i][j]$ = the length of the longest common substring ending at x_i and y_j. The top row and lefthand column are initialized to contain 0; the remaining entries are computed as follows:

$$S[i][j] = \begin{cases} S[i-1][j-1]+1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

The table entries can be computed row-by-row; only two rows need actually be stored. The answer to the problem is the largest entry; it is easy to keep track of the largest value computed so far while filling the table. Computing each table entry and updating the maximum if necessary clearly takes constant time so the total time in the worst case is in $\Theta(nm)$.

An alternative method uses a table $M[1\ldots n][1\ldots m]$ where

$$M[i][j] = \begin{cases} 1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

After filling the table, the algorithm scans all diagonals (slanting downward to the right) for the longest string of contiguous 1's. Each such string corresponds to a common substring. This method also takes $\Theta(nm)$ time, but it uses more space (the whole $n \times m$ table).

10.19

We can construct a common subsequence by a variant of the merge procedure of Section 4.5 in the text. If the first elements of both sequences are equal, remove both, adding their common value to the common subsequence. If they are unequal, remove one or the other. Then continue with the remaining subsequences. To find, the *longest* common subsequence, we need to backtrack over both choices of which element to remove when the front elements are unequal.

Define problem (i, j) to be the problem of finding the longest common subsequence of $A[i\ldots n]$ and $B[j\ldots n]$. Actually, in the first version we will only find the *length* of the longest common subsequence, not its contents. When we convert into the DP version we will also record the decisions that allow us to construct it. The outline of a recursive solution is

```
int findLCS(i, j)
    if (i > n) return 0;
    if (j > n) return 0;
    // point 1
    if (A[i] == B[j])
        action = 'T';
        len = 1 + findLCS(i+1, j+1);
    else
        lenA = findLCS(i+1, j);
        lenB = findLCS(i, j+1);
        if (lenA > lenB)
            action = 'A';
            len = lenA;
        else
            action = 'B';
            len = lenB;
    // point 2
    return len;
```

The significance of point 1 and point 2 is explained below.

To convert into DP, define a dictionary with the id (i, j). It can be implemented as a two-dimensional array **`lcsLength[n+1][n+1]`**. However, we also want to record the decisions that led to the longest common subsequence, so we define a second two-dimensional array **`lcsAction[n+1][n+1]`**. The values of **`lcsAction[i][j]`** will be one of the characters **`T`**, **`A`**, or **`B`**, meaning:

- **`T`** *Take* the element at both $A[i]$ and $B[j]$; increment i and j.
- **`A`** Discard the element at $A[i]$ and increment i.
- **`B`** Discard the element at $B[j]$ and increment j.

In the above procedure, **action** was set accordingly, but not preserved after the procedure returned.

Initialize the entries of **lcsLength** to –1 to denote that the solution is unknown for that entry. A wrapper initializes the dictionary and calls **findLCS(1,1)**.

Modify the recursive **findLCS** to check whether **lcsLength[i][j]** is nonnegative at point 1. If so, return its value immediately; otherwise, continue as before. If point 2 is reached, store **len** and **action** in **lcsLength[i][j]** and **lcsAction[i][j]** before returning.

There are about n^2 subproblems. Each subproblem calls two other subproblems so the subproblem graph has about $2n^2$ edges. The processing per "vertex" and per "edge" is in $O(1)$, so depth-first search of the subproblem graph requires time in $O(n^2)$. The dictionary requires space in $O(n^2)$.

10.21

If $\sum_{i=1}^{n} s_i$ is odd, there is no solution. Suppose $\sum_{i=1}^{n} s_i$ is even. Let $H = \frac{1}{2}\sum_{i=1}^{n} s_i$. Use a table $P[1\ldots n][0\ldots H]$ where

$$P[i][j] = \begin{cases} true & \text{if some subset of } s_1,\ldots,s_i \text{ sums to } j \\ false & \text{otherwise} \end{cases}$$

All entries in column 0 are initialized to **true** (the empty set has sum 0). The rest of the first row is initialized with $P[1][s_1]$ = **true** and all other entries **false**. In general, to determine if a sum j can be achieved using some subset of $\{s_1,\ldots,s_i\}$, we consider subsets that *do not* contain s_i, for which we can find the answer in $P[i-1][j]$, and subsets that *do* contain s_i, where the rest of the subset must sum to $j - s_i$, so we find the answer in $P[i-1][j-s_i]$. So

$$P[i][j] \;=\; P[i-1][j] \textbf{ or } P[i-1][j-s_i]$$

where the second term is retrieved only if $j \geq s_i$ and is considered false otherwise.

If any entry in the last column is found to be **true**, the answer to the problem is "yes."

As we have seen with many dynamic programming problems, the algorithm does not have to store the whole table; computation of each entry uses only the previous row.

Computation of each table entry takes constant time, so the total time used in the worst case is in $\Theta(nH) = \Theta(nS)$.

Remember that H is computed from the data; it is approximately the magnitude of the data, not the number of data. S may be very large relative to n. In the terminology of Chapter 13, this is not a polynomial-bounded algorithm. Other algorithms whose running time varies with the magnitude of the data are considered in Section 13.2.5.

10.23

a) Use as many of the largest coin as possible, then as many of the next largest as possible, and so on. For example, to make change for \$1.43 (U.S), use two half-dollars, one quarter, one dime, one nickel, and three pennies.

b) An optimal way to make change must use at most four pennies (five could be replaced by one nickel), one nickel (two could be replaced by one dime), two dimes (three could be replaced by a quarter and a nickel), and one quarter. So half-dollars would be used for all but at most 54 cents (the value of four pennies, one nickel, two dimes, and one quarter). By considering all the remaining cases, it is easy to see that the greedy algorithm is optimal.

c) Let the coin denominations be 25, 10, and 1, and try to make change for 30 cents.

d) **Solution 1**. There is a solution very similar to the solution for the Partition Problem, Exercise 10.21. Let the table be $M[1\ldots n][0\ldots a]$, where $M[i][j]$ is the minimum number of coins needed to make change for j cents using any of the coins $c_1,\ldots,c_i$. Then

$$M[i][j] \;=\; \min(M[i-1][j],\; 1+M[i][j-c_i])$$

where the second term is considered to be ∞ if $j < c_i$.

The answer will be in $M[n][a]$.

Computing each entry takes constant time, so the worst case time is in $\Theta(na)$.

Solution 2. The second solution is similar to the solutions for the problems discussed in Section 10.3. Let $M[j]$ be the minimum number of coins needed to make change for j cents. After deciding on the first coin, we would make change for the remaining amount in the optimal way. We don't know what the first coin should be, so we

try all possibilities.

$$M[j] = \min_{1 \le i \le n} (1 + M[j - c_i])$$

where $M[0] = 0$ and terms with index less than zero are ignored (or treated as ∞). Note that only a one-dimensional table is used here, but computing an entry may take as many as $\Theta(n)$ steps.

Thus the worst case time is in $\Theta(na)$.

10.26

We use a table $T[1 \ldots 500][1 \ldots 500]$ as described in the hint in the exercise. Only the entries on and above the main diagonal are needed. We define a summing operation # (to compute the amount of time a sequence of songs takes up on 60-minute disks) as follows:

$$a \# b = \begin{cases} a + b & \text{if } (a \bmod 60) + b \le 60 \\ 60k + b & (a \bmod 60) + b > 60 \text{ and } \lceil a/60 \rceil = k \end{cases}$$

The collection of i songs whose total time is represented in $T[i][j]$ may or may not include ℓ_j, so

$$T[i][j] = \min(T[i][j-1],\ T[i-1][j-1] \# \ell_j)$$

The entries can be computed column-by-column (only down as far as the main diagonal), and only two columns need be stored. The top row and the main diagonal can be initialized by

```
T[1][1] = ℓ1;
for (j = 2; j ≤ 500; j ++)
    T[1][j] = min(T[1][j-1], ℓj);
for (i = 2; i ≤ 500; i ++)
    T[i][i] = T[i-1][i-1] # ℓi
```

To find the answer, search the last column from the bottom up for the first entry (the largest i) such that $T[i, 500] \le 300$. At most i songs can fit on the five 60-minute disks satisfying the constraints of the problem.

In general, let n be the number of songs. The initialization and search for the answer take $\Theta(n)$ steps. Computing each table entry takes constant time, so the total time is in $\Theta(n^2)$.

Chapter 11
String Matching

***Section 11.2*:** *A Straightforward Solution*

11.1
In Algorithm 11.1, replace "**`match = i`**" with "**`match = j - m`**" (on the line with the comment "***`Match found`***"). Delete all other references to *i*. Since *j* and *k* are incremented together when characters from the pattern and text match, when the algorithm terminates with $k = m + 1$, $j - m$ must be where the match began in the text.

11.2
The idea is that *i*, *j*, and *k* become (references to) linked lists and "**`++`**" becomes the **`rest`** operation. The pattern length *m* is not needed because the list will be empty when the pattern is exhausted. Because the invariant

$$i = j - k + 1$$

is satisfied in Algorithm 11.1, the statement "**`j = j - backup`**" has the same effect as "**`j = i`**" in that algorithm. Also, "**`k = k - backup`**" has the same effect as "**`k = 1`**."

```
IntList listScan(IntList P, IntList T)
    IntList match; // value to return
    IntList i, j, k;
    // i is the current guess at where P begins in T;
    // j begins at the current character in T;
    // k begins at the current character in P.
    match = nil;
    j = T; k = P;
    i = j;
    while (j ≠ nil)
        if (k == nil)
            match = i; // Match found
            break;

        if (first(j) == first(k))
            j = rest(j); k = rest(k);
        else
            // Back up over matched characters.
            j = i; k = P;
            // Slide pattern forward, start over.
            j = rest(j);
            i = j;
        // Continue loop.
    return match;
```

11.4
Answers can vary considerably for this open-ended question, depending on the software-engineering style.

a) Looking ahead to part (c) we let elements of OpenQueue be type **`char`** instead of **`Object`**.

```
OpenQueue create(int n)
    precondition:  n > 0.
    postconditions:  If q = create(n)
        1.  q refers to a new object.
        2.  maxLength(q) = n.
        3.  length(q) = 0.

int maxLength(OpenQueue q)
    precondition:  q was created by create.
```

```
int length(OpenQueue q)
    precondition:  q was created by create.

char getElmt(Queue q, int position)
    preconditions:  length(q) > 0; 0 ≤ position < length(q).

void enq(OpenQueue q, char e)
    Notation:  /q/ denotes the queue before the operation.
    precondition:  length(/q/) < maxLength(/q/).
    postconditions:  Let k = length(/q/).
        1.  length(q) = k + 1.
        2.  For 0 ≤ p < k, getElmt(q, p+1) = getElmt(/q/, p).

void deq(OpenQueue q)
    Notation:  /q/ denotes the queue before the operation.
    precondition:  length(q) > 0.
    postcondition:  Let k = length(/q/).
        1.  length(q) = k - 1.
        2.  For 0 ≤ p < k-1, getElmt(q, p) = getElmt(/q/, p).
```

b) All methods in **`OpenQueue`** are **`public static`**.

```
class OpenQueue
    int front;  // index of the first element in the queue
    int rear;  // index of the last element in the queue
    int count;  // number of elements in the queue
    char[] elements;  // circular array of elements

    void create(int n)
        OpenQueue q = new OpenQueue();
        q.elements = new char[n];
        q.front = 0;
        q.rear = n-1;
        q.count = 0;

    int length(OpenQueue q)
        return q.count;

    int maxLength(OpenQueue q)
        return q.elements.length;

    char getElmt(OpenQueue q, int position)
        int n = q.elements.length;
        int index = (q.rear - position + n) % n;
        return (q.elements[index]);

    void enq(OpenQueue q, char e)
        int n = q.elements.length;
        q.rear = (q.rear + 1) % n;
        q.count ++;
        q.elements[q.rear] = e;

    void deq(OpenQueue q)
        int n = q.elements.length;
        q.front = (q.front + 1) % n;
        q.count --;
```

c) Note that **`position`** counts backward (to the left) from the most recently read text character, whose position is 0. Although a method **`getPosition`** is provided for completeness, none of the pattern-matching algorithms need it. All methods in **`Text`** are **`public static`**.

```
class Text
    OpenQueue q;
    int position;
    Reader fp;
    boolean eof;
    /** m should be the pattern length.  */
    Text create(Reader input, int m)
        Text T = new Text();
        T.q = OpenQueue.create(m+1);  // add 1 for safety.
        T.position = 0;
        T.fp = input;
        T.eof = false;
        return T;

    /** endText(T) is true if an earlier advanceText(T) encountered end-of-file.  */
    boolean endText(Text T)
        return T.eof;

    int getPosition(Text T)
        return T.position;

    /** Notation:  /T/ denotes T before the operation.
      * Postcondition for advanceText(T, forward):
      * If forward ≤ getPosition(/T/), then getPosition(T) = getPosition(/T/) - forward.
      * Otherwise, getPosition(T) = 0 and either (forward - getPosition(/T/))
      * new characters were read from T.fp or endText(T) is true.
      */
    void advanceText(Text T,int forward)
        int  newpos;
        newpos = T.position - forward;
        while (newpos < 0)
            int  c = T.fp.read();
            // c is either -1 or in the range of the type char.  */
            if (c == -1)
                T.eof = true;
                newpos = 0;
                break;

            if (length(T.q) == maxLength(T.q))
                // there's no room, delete one first
                deq(T.q);
            enq(T.q, (char)c);
            newpos ++;
        T.position = newpos;

    /** Precondition:  backup + getPosition(/T/) < length(T.q).  */
    /** Postcondition:  getPosition(T) = getPosition(/T/) + backup.  */
    void backupText(Text T, int backup)
        T.position += backup;

    char getTextChar(Text T)
        return getElmt(T.q, T.position));
```

Section 11.3: *The Knuth-Morris-Pratt Algorithm*

11.6

a)

A	*A*	*A*	*B*
0	1	2	3

b)

A	*A*	*B*	*A*	*A*	*C*	*A*	*A*	*B*	*A*	*B*	*A*
0	1	2	1	2	3	1	2	3	4	5	1

c)

A	*B*	*R*	*A*	*C*	*A*	*D*	*A*	*B*	*R*	*A*
0	1	1	1	2	1	2	1	2	3	4

d)

A	S	T	R	A	C	A	S	T	R	A
0	1	1	1	1	2	1	2	3	4	5

11.8

At most one character comparison is done each time the body of the loop is executed, in the last of the three **`if`** statements. Each time this **`if`** statement is executed, either j is incremented (along with k) or k is decremented. So we count how many times j is incremented overall and how many times k is decremented. Since j begins at 1, is incremented by exactly 1 whenever it is incremented, and never decreases, and the loop terminates when $j > n$ (where n is the length of T), j can increase at most n times. Since k is incremented the same number of times as j (note that in the second **`if`** statement, if $k = 0$, k is assigned 1), k is incremented at most n times. Since k starts at 1 and is never negative, k can decrease at most n times. Thus the last **`if`** statement, where the character comparison is done, is executed at most $2n$ times.

11.10

a) The fail links are

A	A	A	$\cdot$	$\cdot$	$\cdot$	A	B
0	1	2	$\cdot$	$\cdot$	$\cdot$	$m-2$	$m-1$

The procedure **`kmpSetup`** does $m-2$ character comparisons to compute them.

b) $2n-(m-1)$ (for $m \geq 2$).

c) Let Q consist of $m-2$ A's followed by two B's. Then **`kmpSetup`** will do $2m-5$ character comparisons (for $m > 2$).

11.12

Add the following loop at the end of the procedure **`kmpSetup`** (after the end of both the **`while`** loop and the **`for`** loop in the procedure).

```
for (k = 2; k ≤ m; k ++)
    if (p_k == p_fail[k])
        fail[k] = fail[fail[k]];
```

Section 11.4: *The Boyer-Moore Algorithm*

11.15

a)
```
charJump[A] = 0
charJump[B] = 2
charJump[C] = 6
charJump[D] = 4
charJump[R] = 1
charJump[x] = 11   otherwise.
```

b)
```
charJump[A] = 0
charJump[C] = 5
charJump[R] = 1
charJump[S] = 3
charJump[T] = 2
charJump[x] = 11   otherwise.
```

11.17

It is possible that **`charJump[`**t_j**`]`** is so small that $j +$ **`charJump`**$[t_j]$ is smaller than the index of the text character that is currently lined up with the end of the pattern. To see that this can happen, suppose t_j is the same as p_m, the last pattern character. Then **`charJump`**$[t_j] = 0$. The pattern would shift left instead of right; that should never happen.

The distance from t_j to the text character that is currently lined up with the end of the pattern is $m-k$, so the correct way to increment j to force the pattern to move at least one place to the right is

```
j += max(charJump[t_j], m-k+1);
```

11.19

Assuming that **matchJump[k]** $\geq$ **charJump[**t_j**]**, the number of new characters of text read is

$$\textbf{slide}[k] = \textbf{matchJump}[k] - (m-k) = k - m + \textbf{matchJump}[k].$$

This doesn't count text characters already read while matching right-to-left from p_m to the current p_k.

If **charJump[**t_j**]** $>$ **matchJump[k]**, then the number of new text characters needed is greater than the above amount.

Another exercise: Determine in exactly which cases **charJump[**t_j**]** $>$ **matchJump[k]** when a mismatch occurs. Consider cases in which t_j does or does not occur in P, combined with cases in which $p_{k+1} \cdots p_m$ matches an earlier substring of P or some suffix of $p_{k+1} \cdots p_m$ matches a prefix of P or neither (i.e., the longest suffix and prefix that "match" are length 0). Do you need to split these cases still further?

***Section 11.5*:** *Approximate String Matching*

11.21

Use an additional 2-dimensional array **length** such that **length[i][j]** is the length of the text segment ending at t_j that matches the pattern segment $p_1 \cdots p_i$. In the algorithm, the number of differences between the two strings at this point, that is $D[i][j]$, is computed by taking the minimum of the following values:

1. $D[i-1][j-1]$ (if $p_i = t_j$)
2. $D[i-1][j-1]+1$ (if $p_i \neq t_j$)
3. $D[i-1][j]+1$ (the case where p_i is missing from T);
4. $D[i][j-1]+1$ (the case where t_j is missing from P).

The entry in the **length** table is updated as follows, according to which case above gave the minimum value.

1. `length[i][j] = length [i-1][j-1] +1;`
2. `length[i][j] = length [i-1][j-1] +1;`
3. `length[i][j] = length [i-1][j];`
4. `length[i][j] = length [i][j-1] +1.`

Now suppose $D[m][j] \leq k$ for some j, where m is the pattern length and k is the maximum number of differences allowed. Then **length[m][j]** is the length of the text segment that matches the pattern.

Note that more than one of the cases may occur (more than one of the values considered when finding the minimum may have the minimum value). Thus there may be more than one correct value for length of the matched segment. If one wants to bias the algorithm in favor of a particular kind of mismatch, one can do so.

Additional Problems

11.23

The following modification works for both the straightforward algorithm and Knuth-Morris-Pratt (Algorithms 11.1 and 11.3). It outputs a list (possibly empty) of indexes in T where a copy of P begins.

```
Initialization lines unchanged;
while (endText(T, j) == false) // New outer while loop
     Unchanged while loop from Alg.  5.1 or 5.3.
          ⋮
     if (k > m)
          Output (j-m).
          j = j-m+1;
          // This will find overlapping copies; to find only
          // non-overlapping copies, leave j unchanged.
          k = 1;
     // Continue main while loop.
```

For the Boyer-Moore algorithm (Algorithm 11.6), an outer **while** loop is added as above. The new **if** statement that follows the original (now inner) **while** loop would be

```
if (k == 0)
     Output (j+1);
     j = j+m+1;  // j += 2*m for non-overlapping instances
     k = m;
```

11.25

Use the Knuth-Morris-Pratt algorithm to determine if there is a copy of X in YY (that is, two copies of Y concatenated together). Setting up the fail links takes $\Theta(n)$ time and the scan takes $\Theta(2n) = \Theta(n)$ time.

Chapter 12
Polynomials and Matrices

Section 12.2: *Evaluating Polynomial Functions*

12.2

a) Observe that $p(x) = x^n + \cdots + 1 = \dfrac{x^{n+1}-1}{x-1}$. Let $k = \lfloor \lg(n+1) \rfloor$.

To get x^{n+1}, start with $t = x$ and multiply t by itself k times, saving the intermediate results, x, x^2, x^4, $x^8, \ldots, x^{2^k}$. Now, $n+1$ is the sum of powers of 2 among $2^0, 2^1, \ldots, 2^k$, so x^{n+1} can be computed by multiplying together the appropriate terms, using at most k more multiplications. Then subtract 1 and divide by $x-1$, for a total of at most $2k+3 = 2\lfloor \lg(n+1) \rfloor + 3$ arithmetic operations on the data.

b) Observe that $p(x) = (x+1)^n$. This can be computed with at most by $2\lfloor \lg n \rfloor + 1$ arithmetic operations on the data by the same technique used in part (a).

12.4

In a straight-line program, a step of the form

s **=** $q/0$

would be illegal. In the proof of Lemma 12.1, 0 is substituted for a_n wherever it appears. If there were any steps of the form

s **=** q/a_n

this transformation would not produce a valid program.

12.6

Since $n = 2^k - 1$, we need to show that $A(k) = 3 \cdot 2^{k-1} - 2$.

Solution 1. Since we were given the answer, the simplest solution is to verify it by induction. $3 \cdot 2^0 - 2 = 1$, as required for $k = 1$. For $k > 1$, $A(k) = 2A(k-1) + 2 = 2 \cdot 3 \cdot 2^{k-2} - 2$, the latter by the inductive hypothesis. The last expression simplifies to $3 \cdot 2^{k-1} - 2$.

Solution 2. Let's suppose we weren't given the answer and want to discover it. Start by expanding the recurrence to see the pattern of the terms.

$$\begin{aligned} A(k) &= 2 + 2A(k-1) = 2 + 2[2 + 2A(k-2)] = 2 + 4 + 4A(k-2) \\ &= 2 + 4 + 4[2 + 2A(k-3)] = 2 + 4 + 8 + 8A(k-3) \\ &= 2^1 + 2^2 + \cdots + 2^j + 2^j A(k-j) \qquad \text{for } 1 \le j < k \end{aligned}$$

So let $j = k-1$ to get a sum without any recursive A terms; that is, we use the fact that $A(1) = 1$.

$$A(k) = \sum_{i=1}^{k-1} 2^i + 2^{k-1} \cdot 1 = 2^k - 2 + 2^{k-1} = 3 \cdot 2^{k-1} - 2.$$

Section 12.3: *Vector and Matrix Multiplication*

12.8

No */'s are required because each term $u_i v_i$ in the dot product can be computed by adding u_i copies of v_i.

12.10

a) n.

b) $n+1$.

Section 12.4: *The Fast Fourier Transform and Convolution*

12.12

a) If $n = 1$, the only nth root of 1 is 1 itself, so the roots don't sum to zero. If n divides c, then $\omega^c = 1$, so $\sum_{j=0}^{n-1} (\omega^c)^j = n$.

b) For Property 1: ω^2 is an $(n/2)$-th root of 1 because $(\omega^2)^{n/2} = \omega^n = 1$. Suppose $1, \omega^2, (\omega^2)^2, \ldots, (\omega^2)^{n/2-1}$ are not distinct. Let ω^k and ω^m be two of these terms that are equal. But k and m are both less than $n-1$, contradicting the fact that ω is a primitive nth root of 1. Thus the terms must be distinct, and ω^2 is a primitive $(n/2)$-th root of 1.

For Property 2: The square roots of 1 are 1 and –1. $(\omega^{n/2})^2 = 1$, but $\omega^{n/2} \neq 1$ because ω is a primitive nth root of 1, so $\omega^{n/2}$ must be –1.

An alternative argument for Property 2: In polar coordinates, $\omega = (1, 2\pi/n)$, so $\omega^{n/2} = (1^{n/2}, n/2 \cdot 2\pi/n) = (1, \pi) = -1$.

12.14

Let a, b, c, and d be the given data, representing the complex numbers $a+bi$ and $c+di$. The product of these complex numbers is $(ac-bd)+(ad+bc)i$, so we need to compute $ac-bd$ and $ad+bc$ using only three multiplications. Here's the algorithm.

```
z = (a+b) * (c+d);  // z = ac + ad + bc + bd
u = a * c;
v = b * d;
realPart = u - v;  // realPart = ac - bd
imaginaryPart = z - u - v;  // imaginaryPart = ad + bc
```

12.16

a) The ith component of F_nV is $\sum_{j=0}^{n-1} \omega^{ij} v_j$. Permuting the columns of F_n and elements of V in the same way will affect the order of the terms in the summation, but not the sum itself.

b) $G_1[i][j] = \omega^{2ij}$. The corresponding element in G_3 is $\omega^{2(i+n/2)j} = \omega^{2ij+nj} = \omega^{2ij}$ since $\omega^{nj} = 1$.

c) $G_2[i][j] = \omega^{i(2j+1)}$. The corresponding element in G_4 is $\omega^{(i+n/2)(2j+1)} = \omega^{i(2j+1)+nj+n/2} = -\omega^{i(2j+1)}$, since $\omega^{nj} = 1$ and $\omega^{n/2} = -1$.

d) The element in row i and column j of DG_1 is $\omega^i G_1[i][j] = \omega^i \omega^{2ij} = \omega^{i(2j+1)} = G_2[i][j]$.

e) The entries of G_1 are ω^{2ij}, and ω^2 is a primitive $(n/2)$-th root of 1. The formula for $\widetilde{F}_nV$ follows from (a), (b), (c), and (d).

f) The recurrence equation for each operation is

$$A(k) = 2A(k-1) + n/2 \quad \text{for } k \geq 1, \qquad A(0) = 0.$$

This method does roughly the same number of arithmetic operations on the data as **`recursiveFFT`**. (See the recurrence that follows Algorithm 12.4.)

Additional Problems

12.17

Use the technique of repeated squaring given in the solution for Exercise 12.2(a) to compute A^{n-1}. This requires about $2\lg(n)$ matrix multiplications of 2×2 matrices, each involving eight multiplications and four additions. There is also bookkeeping proportional to $\lg(n)$ to find the 1-bits in the binary form of n, etc. So the total is about $16\lg(n)$ *'s and $8\lg(n)$ +'s, plus bookkeeping.

The recursive computation of F_n requires about F_{n-1} +'s, which is about 1.618^{n-1}. The iterative computation of F_n requires about n +'s. So this matrix-power method with repeated squaring is "exponentially faster."

Chapter 13
NP-Complete Problems

***Section 13.2*:** *P and NP*

13.2
The graph has no edges.

13.4
Note: Part (e) of this exercise was updated by a correction that was too late for the second printing. Be sure that the current errata (available via Internet) are applied if this error is present in any of the texts being used.

The formats for the proposed solutions are certainly not unique; these are some of several reasonable approaches. In all cases, part of the checking is to check for correct syntax, so we do not state that each time. Also, spaces may appear for readability although they are not mentioned. We would expect different levels of sophistication and completeness in solutions in undergraduate courses and graduate courses. For example, undergraduates would probably omit the discussion of the length of the proposed solution that is emphasized in parts (a) and (c) below.

a) The input is an integer k and a sequence of rational numbers, $s_1, \ldots, s_n$. Let L be the length of the input, which can be arbitrarily larger than n.

Proposed solution: a sequence of indexes with commas separating items to go in the same bin and semi-colons marking the end of the sequence of indexes for each bin. Example:

2, 7; 1, 3, 4; 6, 5;

The properties to be checked to verify the solution are:

- The proposed solution contains at most $2n$ symbols, counting each integer, comma, or semi-colon as one symbol; also each integer is in the range $1, \ldots, n$.
- The indexes in the sequence include all of the integers $1, \ldots, n$.
- There are at most k semicolons (at most k bins).
- For each set of indexes I denoting a set of elements to go in a single bin, $\sum_{i \in I} s_i \leq 1$.

The input to the bin packing problem has length L larger than n since each of the n sizes may be many digits long. We will show that the amount of time needed to check a proposed solution is bounded by a polynomial in L.

The length of a valid solution is in $O(n \log n)$ since writing each index may use up to $\Theta(\log n)$ digits. An invalid solution may be arbitrarily long but the checking algorithm returns *false* as soon as it detects an integer greater than n or more than $2n$ symbols. A simple marking scheme can be used to check that the indexes listed include all integers in $1, \ldots, n$. Counting the semicolons is easy. Assuming that the sizes s_i are in an array, the sums can be computed and checked in one pass over the proposed solution string. The only remaining question is how long that pass takes. Since the s_i are arbitrary precision, we cannot assume they can be added in one step.

If each s_i is represented as a terminating decimal, the time to add two of them digit by digit is in $O(L)$. An intermediate sum will never exceed L digits, so adding s_i to an intermediate sum is also in $O(L)$. In this case the total time for checking sums is certainly in $O(Ln^2)$, or $O(L^3)$.

The issue is less clear if s_i are represented in the "pure" form of rational numbers, as (n_i/d_i), where n_i and d_i are positive integers. Now forming the sum involves computing a common denominator. First note that the total number of digits in all d_i together is less than L. So multiplying them all together yields a product with at most L digits. Similarly, any numerator needed to represent an s_i with the common denominator is at most L digits. Multiplying two L-digit numbers takes time in $O(L^2)$. Multiplying n L-digit numbers, where the product is at most L digits, takes time in $O(nL^2)$. We need to compute one common denominator and n new numerators, and that can be done in $O(n^2L^2)$, which is in $O(L^4)$. Adding up the new numerators is in $O(Ln^2)$, or $O(L^3)$.

Note: The issue of arithmetic with rational numbers also arises in Exercises 13.8 and 13.59.

An advanced study question: Suppose the s_i are written as (n_i/d_i) but n_i and d_i may be written in factored form. That is, let's use "$2**100$" to denote 2^{100} and use $\times$ to denote multiplication. Then a very large integer can be written with just a few characters, such as $2**100 \times 7**400$, which is 13 characters in this form, but would be 369 digits if written out in decimal form. Is bin-packing still in NP? There is no problem about multiplying such numbers; the problem is adding them in polynomial time.

b) The input is an integer n and a sequence of edges ij, where i and j are between 1 and n. (An alternative is an integer n and an $n \times n$ bit matrix.)

Proposed solution: a sequence of indexes, $v_1, v_2, \ldots, v_k$. Interpretation: This is the order in which to visit the vertices in a Hamiltonian cycle.

The properties to be checked to verify the solution are:

- The proposed solution contains exactly n indexes (i.e., $k = n$) and each index is in the range $1, \ldots, n$.
- No index in the proposed solution is repeated.
- There is an edge $v_i v_{i+1}$ for $1 \leq i \leq n-1$ and an edge $v_n v_1$.

The work done to check these properties depends on the representation of the graph, but is clearly straightforward and can be done in $O(n^2)$ time. A proposed solution that is extremely long or contains extremely large numbers can be rejected as soon as we detect an integer that is not in the correct range or detect more than n integers.

c) The input is a CNF formula in which the propositional variables are represented by integers in the range 1, ..., n. The overall input length is L. Note that n might be much larger than L.

Proposed solution: a sequence of integers in the range 1, ..., n. Interpretation: if i is present, then the truth value to be assigned to variable i is *true*; otherwise $\bar{i}$ is *true*.

The properties to be checked to verify the solution are:

- The proposed solution is no longer than the input, i.e., at most L.
- Each clause in the CNF expression evaluates to *true* with these truth values. A clause consisting of the literals $\{x_j\}$ evaluates to *true* if at least one of the x_j evaluates to *true*. If $x_j = i$ and the i appears in the proposed solution, then x_j evaluates to *true*. If $x_j = \bar{i}$ and i does *not* appear in the proposed solution, then x_j evaluates to *true*.

The input formula can be checked clause by clause in time $O(L^2)$, since each literal in a clause can be checked in $O(L)$ by scanning the proposed solution. (Since n may be very large in comparison to L, we do not want to assume we can use an array with n entries.)

d) The input is an integer k and a graph with vertices $1, \ldots, n$ and m edges.

Proposed solution: a sequence of integers, $v_1, \ldots v_q$. Interpretation: These are the vertices in a vertex cover.

The properties to be checked to verify the solution are:

- The proposed solution contains at most k integers (or if $k > m$, at most m integers), and each of these integers is incident upon *some* edge in the input graph.
- For each edge (i, j) in the graph, at least one of i and j is in the sequence.

The work to check these properties is clearly straightforward and can be done in $O(m^3)$ time. A proposed solution that is extremely long or contains extremely large numbers can be rejected as soon as we detect an integer that is not incident upon any input edge or detect more than m integers.

e) *Note*: *Due to an error, the first two printings of the text ask for a proposed solution to the question,* "Is n prime?", *but the intended question is* "Is n nonprime?". The first question is indeed also in NP, but it would be a very advanced graduate-student exercise, too hard even for *two* stars in the context of the text; see Vaughan Pratt, "Every prime has a succinct certificate," *SIAM Journal on Computing*, vol. 4, no. 3, pp. 214–220, 1975.

The input is an L-bit integer n.

Proposed solution: Two positive integers, x and y. Interpretation: $xy = n$.

The properties to be checked to verify the solution are:

- The proposed solution is a pair of integers of at most L bits each.
- Each integer is greater than 1.
- $xy = n$.

The product xy can be computed by the manual long-multiplication technique in time proportional to L^2.

Section 13.3: *NP-Complete Problems*

13.6

Let the problems be P, R, and Q, where $P \leq_P R$ and $R \leq_P Q$. There is a polynomial-time transformation T that takes input x for P and produces $T(x)$, an input for R, such that the answer for R with this input is the correct answer for P with input x. There is a polynomial-time transformation U that takes input y for R and produces $U(y)$, an input for Q, such that the answer for Q with this input is the correct answer for R with input y.

Now, let UT be the composite function that first applies T to its argument, then applies U to the result, i.e., $UT(x) = U(T(x))$. There are two things we must verify, that UT preserves correct answers and that it can be computed in polynomial time. The first is easy to see. UT takes input x for P and produces $U(T(x))$, an input for Q. The answer for Q with this input is the correct answer for P with input x.

The second point, the timing for UT, may seem clear too, but there is a (small) difficulty in the proof that should not be overlooked. Let p and q be polynomial time bounds for algorithms to compute T and U, respectively. Let n be the size of x, an input for P. Students are likely to say: Since T and U are computed with input x, the total time is $(p+q)(n)$. The hitch is that U is applied to $y = T(x)$, which may have size much larger than n. Let c be the (constant) limit on the number of units of output (perhaps characters) an algorithm may produce in one step. Then we can conclude that the size of y is at most $cp(n)$, so the time for the transformation UT is at most $p(n) + q(cp(n))$, which is polynomial. (A similar argument appears in the proof of Theorem 13.3.) Hence, $P \leq_P Q$.

13.8

Let SS_I be the subset sum problem for integers and let SS_R be the subset sum problem for rational numbers. Since the integers are a subset of the rational numbers, any algorithm for SS_R solves SS_I, so $SS_I \leq_P SS_R$.

For the other direction, let $s_1, \ldots, s_n$, and C be an input for SS_R. Let d be the product of the denominators of $s_1, \ldots, s_n$, and C. The input is transformed to $ds_1, \ldots, ds_n$, and dC (all of which are integers). As discussed in Exercise 13.4, this can be done in polynomial time. For any subset J of $\{1, \ldots, n\}$, $\sum_{j \in J} s_j = C$ if and only if $\sum_{j \in J} ds_j = dC$. So the input for SS_R has a *yes* answer if and only if the transformed input for SS_I does, and $SS_R \leq_P SS_I$.

13.10

Transform an undirected graph G into a directed graph H by replacing each edge vw of G by edges in each direction, vw and wv.

Then if there is a Hamiltonian cycle in G, there is one in H. For each edge in the cycle in G, the cycle in H uses the edge from a vw and wv pair that corresponds to the direction the edge was traversed in G.

If there is a Hamiltonian cycle in H, then there is one in G; simply replace either edge vw or wv in the cycle in H by the corresponding undirected edge in G.

13.12

Let the transformation T take an undirected graph $G = (V, E)$ as input and produce H and k as output, where H is a *complete weighted* undirected graph and k is an integer. H and k are calculated from G as follows. $H = (V, F)$, where the weight of an edge $(i, j) \in F$ is 1 if $(i, j) \in E$ and is 2 otherwise. The value of k is $|V|$. Clearly $T(G)$ can be computed in polynomial time.

If G has a Hamiltonian cycle, then H has a tour of length k by taking the edges of the Hamiltonian cycle.

If H has a tour of length k, it must consist entirely of edges of weight 1 because k cities need to be visited. Therefore the edges in the tour all occur in G, so they provide a Hamiltonian cycle for G.

13.14

$T(G)$ is bipartite, but there are some graphs G that have Hamiltonian cycles where $T(G)$ does not. A Hamiltonian cycle does not in general traverse every edge in the graph. Thus a cycle in $T(G)$ that corresponds to a cycle in G will miss those new vertices in the edges that aren't in G's cycle. There is an example in the figure below; G has a Hamiltonian cycle, but $T(G)$ does not.

G (Has a Hamiltonian cycle)

T(G) (Has no Hamiltonian cycle)

13.16

There is a well-known way to encode graph colorability instances as CNF-satisfiability instances. We will use this encoding as our transformation to prove that graph coloring is reducible to SAT. Suppose a graph G has n vertices and m edges, and k colors are available. G is encoded as a CNF formula that is satisfiable if and only if G is k-colorable. (The exercise asked for $k = 3$, but the encoding is general.) The set of propositional variables is $\{x_{vc} \mid 1 \le v \le n,\ 1 \le c \le k\}$. The "meaning" of x_{vc} is that vertex v is colored c.

No two adjacent vertices may have the same color, so for every edge $(i, j) \in G$ and every color $1 \le c \le k$ we create the clause $(\neg x_{ic} \vee \neg x_{jc})$. This amounts to mk clauses, with two literals each.

Every vertex must have *some* color, so for every vertex $1 \le v \le n$ we create the clause $(x_{v1} \vee x_{v2} \vee \cdots \vee x_{vk})$. This amounts to n clauses of k literals each.

So the total number of literals in the formula is $2mk + nk$. The problem is trivial if $k \ge n$, so we never need to encode such problems. Therefore the formula's length is in $O(n^3)$ and is easily computable in polynomial time. For this exercise, k was specified as 3, so the length is in $O(n+m)$. Also the formula is in 3-CNF in this case. It remains to show that this transformation preserves both yes and no answers for the graph colorability problem.

If G is k-colorable, then the CNF formula is satisfiable by setting all the propositional variables according to their "meaning," as given above.

If the formula is satisfiable, then we claim that the propositional variables that are true in a satisfying assignment define a k-coloring for G. For each v at least one of x_{vc} is true, for some $1 \le c \le k$. (If *more than* one x_{vc} is true for a given v, choose any one of them.) And for each edge (i, j), if x_{ic} is true, then x_{jc} must be false, so i and j have different colors.

13.18

To see that the clique problem is in NP, let a proposed solution be a sequence of vertices. We may immediately reject any input with $k > n$, so in the following discussion, we assume $k \le n$. It is easy to check in polynomial time that there are k vertices in the sequence, that each is actually a vertex in the graph, and that they are mutually adjacent. (There are $\Theta(k^2)$, hence $O(n^2)$, edges to be checked.)

Now, we show that $SAT \le_P Clique$. First, suppose a CNF formula E is satisfiable and fix some satisfying truth assignment for the variables; we will show that the graph constructed from E (as described in the exercise) has a k-clique.

For each clause C_i, let t_i be an index of a true literal in that clause. (If more than one literal in a clause is true, choose any one.) So l_{i,t_i}, for $1 \le i \le p$, are true. Consider the vertices $V' = \{(i, t_i) \mid 1 \le i \le p\}$. Edges are omitted from the graph only if $l_{i,t_i} = \overline{l_{j,t_j}}$, but that can't happen here because all these literals are true. So each pair of vertices in V' is adjacent, and V' is a k-clique. (Recall $k = p$.)

Now suppose that the graph constructed for E has a k-clique; we show that E is satisfiable. The vertices in the k-clique must differ in the first component, because there are no edges between vertices that correspond to literals in the same clause. Thus we may conclude that the vertices in the k-clique are (i, m_i), for $1 \le i \le k$. Assign the value *true* to each literal l_{i,m_i} for $1 \le i \le k$. There can be no conflicts in this assignment because $l_{i,m_i} \ne \overline{l_{jm_j}}$. One literal in each clause will be true, so E is satisfiable.

Finally, we note that the transformation can be done in quadratic time.

Another exercise: Show that *Independent Set* is NP-complete by reducing *SAT* to *Independent Set*. A related construction can be used.

13.20

This solution uses the notion of the *complement* of a graph. For an undirected graph G, the *complement* of G, denoted $\overline{G}$, has the same vertices as G and has exactly those undirected edges that are *not* in G.

Let the input for *Clique* be a graph G and an integer k. For purposes of the transformation, we use only vertices incident upon some edge of G; let n be the number of such vertices. (We explain this restriction later.)

We transform G and k to the graph $H = \overline{G}$, where $\overline{G}$ is the complement of G, and the integer $c = n - k$. The question for the transformed input is, "Does H have a vertex cover of size c (or less)?"

If G has a clique of size k, let it be the set of vertices K. In H, there are no edges between any pair of vertices in K. Therefore it is not necessary for any vertex in K to be in a vertex cover of H. Choose the remaining $c = n - k$ vertices (those *not* in K) as the vertex cover of H.

If H has a vertex cover of size c, let it be the set of vertices C. Let i and j be any two vertices not in C. Then ij cannot be an edge in H, so it must be an edge in G. Therefore all the vertices not in C comprise a clique in G, and there are $k = n - c$ of them.

The transformation can be computed in time $O(n^3)$.

Why did we define n to exclude isolated vertices? The problem has to do with the size of H; that is, whether the size of H is bounded by a polynomial function of the size of the input graph G. Suppose G actually has N vertices. Thus H may have $Theta(N^2)$ edges. There is no problem if G is presented using N adjacency lists or an $N \times N$ adjacency matrix; either has size in $\Omega(N)$. However, G could be presented as the integer N, specifying the number of vertices without enumerating them, then a sequence of edges. If the number of edges is tiny (e.g., logarithmic) compared to N, the roughly N^2 edges of H cannot be constructed in polynomial time.

Other variations: Show that any of *Clique*, *Vertex Cover*, and *Independent Set* can be reduced to any of the others. All the transformations have a similar flavor.

13.21

Let G (an undirected graph) and k be an input for the vertex cover problem. Transform G to a directed graph H by replacing each edge vw of G by edges in each direction, vw and wv. The input for the feedback vertex set problem will be H and k.

Suppose G has a vertex cover using at most k vertices. We claim that the vertex cover for G is a feedback vertex set for H. Consider any directed cycle in H, and select any edge in the cycle. This edge came from an edge vw in G, so at least one vertex is in the vertex cover.

On the other hand, suppose H has a feedback vertex set. We claim it is a vertex cover for G. Let vw be an edge of G. In H there is a cycle consisting of only the two directed edges vw and wv. Thus at least one of the vertices v or w must be in the feedback vertex set.

The transformation can be done in linear time.

13.23

If each vertex has degree at most two, then each connected component of the graph is either an isolated vertex, a simple chain of vertices, or a simple cycle. (The number of edges is in $O(n)$.) The chromatic number of the graph will be three if there is a cycle of odd length; otherwise it will be two (if there are any edges at all) or one (if there are no edges).

These conditions can be detected using a straightforward depth-first search.

13.26

Check each subset of four vertices. There are $\binom{n}{4} = n(n-1)(n-2)(n-3)/24$ such sets. For each set of four vertices, the existence (or nonexistence) of six edges must be checked. Construct the adjacency matrix representation of the graph in $\Theta(n^2)$ time if it is not given in that form. Then any edge can be checked in $O(1)$, so the algorithm's time is in $\Theta(n^4)$.

13.28

Suppose we have a polynomial time Boolean subroutine `canPack(S, n, k)` which returns *true* if and only if the n items whose sizes are in the array S can be packed in k bins. We use the function as follows.

```
int minBins(S, n)
    for (k = 1; k ≤ n; k ++)
        cando = canPack(S, n, k);
        if (cando)
            break;
    return k;
```

We know there will be at most n iterations of the **for** loop, so if **canPack** runs in polynomial time, so does **minBins**.

Section 13.4: *Approximation Algorithms*

13.30

$FS(F)$ is the set of all subsets of F that are covers of A. For any such cover x, $val(F,x)$ is the number of sets in x.

Section 13.5: *Bin Packing*

13.32

The objects are specified in four groups by decreasing sizes. The idea is that the optimum packing fills $k-1$ bins *exactly* by using one object each from Groups 1, 3, and 4, and puts all of Group 2 in one bin. However, since FFD handles larger sizes first, it is "tricked" into distributing Group 2 among the first $k-1$ bins after it distributes Group 1 into those bins. Nothing else will fit into the first $k-1$ bins, creating some waste.

This is the usual technique for fooling a greedy algorithm: make sure the optimal solution requires more look-ahead than the greedy strategy has. If a person were trying to fill the bins, he or she would probably notice that one Group-1 object and one Group-2 object left a lot of waste space, but one each of Groups 1, 3, and 4 was a perfect fit.

So it is easy to make FFD use an extra bin. The technical difficulty is forcing it to put $k-1$ objects into that bin. We give two schemes for this.

Solution 1. The following construction works for $k > 4$. We'll define $\varepsilon(k) = 1/k^2$.

$$\begin{array}{lll} \text{Group 1:} & k-1 \text{ objects} & \text{size } 1-\dfrac{2}{k} \\ \text{Group 2:} & k-1 \text{ objects} & \text{size } \dfrac{1}{(k-1)} \\ \text{Group 3:} & k-1 \text{ objects} & \text{size } \dfrac{1}{k}+\varepsilon(k) = \dfrac{(k+1)}{k^2} \\ \text{Group 4:} & k-1 \text{ objects} & \text{size } \dfrac{1}{k}-\varepsilon(k) = \dfrac{(k-1)}{k^2} \end{array}$$

In an optimal packing one object from each of the groups 1, 3, and 4 fill a bin completely, so $k-1$ bins are filled with all the objects in these groups. All the objects in Group 2 fit in one bin, so the optimal number of bins is k.

We establish the behavior of *First Fit Decreasing* on this input by verifying a series of claims. We won't include all the details; we will simply state the claim and the condition it requires on k, if any.

1. Claim: The sizes are listed in nonincreasing order.
   - Verify: $1-\dfrac{2}{k} > \dfrac{1}{k-1}$. (True if $k \geq 4$.)
   - Verify: $\dfrac{1}{k-1} > \dfrac{k+1}{k^2}$.
2. Claim: *FFD* puts one object from Group 1 and one object from Group 2 together in each of the first $k-1$ bins.
   - Verify: Two objects from Group 1 won't fit. (True if $k > 4$.)
   - Verify: An object from Group 1 and Group 2 do fit; i.e., $1-\dfrac{2}{k}+\dfrac{1}{k-1} \leq 1$. (True if $k \geq 2$.)
   - Verify: No other objects fit with these two; i.e., $1-\dfrac{2}{k}+\dfrac{1}{k-1}+\dfrac{k-1}{k^2} > 1$.
3. Claim: *FFD* puts all the Group 3 objects, and no others, in the kth bin.

- Verify: All Group 3 objects fit in one bin; i.e., $(k-1)\dfrac{k+1}{k^2} < 1$.
- Verify: No Group 4 object fits in this bin; i.e., $(k-1)\dfrac{k+1}{k^2} + \dfrac{k-1}{k^2} > 1$. (True if $k > 2$.)

Thus *FFD* has $k-1$ remaining objects that it puts in extra bins. (In fact, they will all go in one bin.) It is informative to work out an example with $k = 10$.

Solution 2. The following construction works for $k \geq 2$, and so covers all cases of Lemma 13.10. It is a refinement of the idea in solution 1. We'll define $D(k)$, the common denominator, to be

$$D(k) = (k+2)(k+3)(k+4) = k^3 + 9k^2 + 26k + 24.$$

$$\begin{array}{lll} \text{Group 1:} & k-1 \text{ objects} & \text{size } 1 - \dfrac{2k^2+13k+19}{D(k)} \\ \text{Group 2:} & k-1 \text{ objects} & \text{size } \dfrac{k^2+7k+12}{D(k)} \\ \text{Group 3:} & k-1 \text{ objects} & \text{size } \dfrac{k^2+7k+10}{D(k)} \\ \text{Group 4:} & k+2 \text{ objects} & \text{size } \dfrac{k^2+6k+9}{D(k)} \end{array}$$

In an optimal packing one object from each of the groups 1, 3, and 4 fill a bin completely, so $k-1$ bins are filled with the objects in these groups, leaving Group 2 and three objects from Group 4. All the remaining objects fit in one bin, since

$$(k-1)(k^2+7k+12) + 3(k^2+6k+9) < D(k).$$

so the optimal number of bins is k. The behavior of FFD is shown with claims similar to solution 1.

1. Claim: The sizes are listed in nonincreasing order.
   - Verify: $D(k) - (2k^2+13k+19) > (k^2+7k+12)$. (True if $k \geq 2$.)
2. Claim: *FFD* puts one object from Group 1 and one object from Group 2 together in each of the first $k-1$ bins.
   - Verify: Two objects from Group 1 won't fit. (True if $k \geq 2$.)
   - Verify: An object from Group 1 and Group 2 do fit.
   - Verify: No other objects fit with these two; i.e., $(k^2+7k+12) + (k^2+6k+9) > (2k^2+13k+19)$.
3. Claim: *FFD* puts all the Group-3 objects and exactly three Group-4 objects in the kth bin.
   - Verify: These objects fit; i.e., $(k-1)(k^2+7k+10) + 3(k^2+6k+9) \leq D(k)$.
   - Verify: A fourth Group-4 object does not fit; i.e., $(k-1)(k^2+7k+10) + 4(k^2+6k+9) > D(k)$. (True if $k \geq 2$.)

Thus *FFD* has $k-1$ remaining Group-4 objects that it puts in an extra bin.

13.34

Best Fit Decreasing follows the procedure of *First Fit Decreasing* (Algorithm 13.1) closely, except that all bins are considered before choosing one.

```
binpackBFD(S, n, bin)
    float[] used = new float[n+1];
    // used[j] is the amount of space in bin j already used up.
    int binsUsed;
    int i, j, bestJ;
    Initialize all used entries to 0.0.
    Sort S into descending (nonincreasing) order, giving the sequence
    s_1 ≥ s_2 ≥ ··· ≥ s_n.
    // Indexes for the array bin correspond to indexes in the sorted sequence;
    // bin[i] will be the number of the bin to which s_i is assigned.

    binsUsed = 0;
    for (i = 1; i ≤ n; i ++)
        // Look for a bin in which s[i] fits best.
        bestJ = binsUsed + 1;
        for (j = 1; j ≤ binsUsed; j ++)
            if (used[j] + s_i ≤ 1.0 && used[j] > used[bestJ])
                bestJ = j;
        bin[i] = bestJ;
        used[bestJ] += s_i;
        if (bestJ > binsUsed)
            binsUsed = bestJ;
        // Continue for (i)
```

The worst case time occurs when n bins are used. In such cases, at the beginning of the ith iteration of the outer **for** loop, **binsUsed** $= i-1$, so the inner **for** loop executes $\Theta(n^2)$ times altogether.

13.35

a) 0.6, 0.3, 0.3, 0.2, 0.2, 0.2, 0.2. BFD uses three bins, but two is optimal.

b) 0.7, 0.4, 0.35, 0.25, 0.1, 0.1, 0.1. BFD uses two bins, but FFD uses three.

***Section 13.6*:** *The Knapsack and Subset Sum Problems*

13.37

Consider the input where $s_1 = 1$ and $s_2 = C$. The greedy algorithm described in the exercise puts s_1 in the knapsack and quits, although s_2 would fill the knapsack completely. The ratio of the optimal to the algorithm's result is C, which can be chosen arbitrarily large.

13.39

The main departure from Algorithm 13.2 is that *profit density* is used. This is defined as p_i/s_i, where object i has profit p_i and has size s_i. In the simplified version in the text, we assumed that $p_i = s_i$, so that all objects had equal profit density.

As in Algorithm 13.2, we assume the class **IndexSet** is available to represent subsets of N and the needed set operations are implemented efficiently in this class. The output parameter **take** is in this class. Lines that are new or changed, relative to Algorithm 13.2, are numbered.

```
knapsack_k(C, S, P, take)
     int maxSum, sum;
  0. float maxProfit, profit;
     IndexSet T = new IndexSet;
     int j;
  1. Sort S and P into descending (nonincreasing) order of profit density, giving
     the sequences s_1, s_2, ..., s_n and p_1, p_2, ..., p_n such that p_1/s_1 ≥ p_2/s_2 ≥ ··· ≥ p_n/s_n.

  2. take = ∅; maxSum = 0; maxProfit = 0;
     For each subset T ⊆ N with at most k elements:
          sum = Σ_{i∈T} s_i;
  3.      profit = Σ_{i∈T} p_i;
          if (sum ≤ C)
               // Consider remaining objects.
               For each j not in T:
                    if (sum + s_j ≤ C)
                         sum += s_j;
  4.                    profit += p_j;
                        T = T∪{j};
               // See if T fills the knapsack best so far.
  5.           if (maxProfit < profit)
                    maxSum = sum;
  6.                maxProfit = profit;
                    Copy fields of T into take.
          // Continue with next subset of at most k indexes.
  7. return maxProfit;
```

Theorem 13.12 must be modified to include the time for the sorting. The only case where this affects the result is when $k = 0$, since for $k > 0$, the worst-case time for A_k is in $\Omega(n^2)$. The worst-case time for A_0 was linear before, now it is in $\Theta(n \log n)$. The proof of Theorem 13.13 becomes much more complicated.

Theorem. Let A_k be the sequence of algorithms given above for the general knapsack problem. For $k > 0$, $R_{A_k}(m)$ and $S_{A_k}(n)$, the worst-case ratios of the optimal solution to the value found by A_k, are at most $1 + 1/k$ for all m and n.

Proof: (Based on S. Sahni, "Approximate algorithms for the 0/1 knapsack problem," *JACM*, Jan., 1975).

Fix k and a particular input I with $opt(I) = m$. Let T^* be the set of objects in an optimal solution. To keep the notation simple, we will denote these objects by indexes $1, 2, \ldots, t$ rather their actual indexes. Thus this indexing does not correspond exactly to the indexes used in the algorithm and does not imply that the optimal solution consists of the t objects with greatest profit density.

If $t \leq k$, then T^* is explicitly considered by A_k so $val(A_k, I)) = m$ and $r_{A_k}(I) = 1$.

Suppose $t > k$. We will assume that the objects in the optimal solution are indexed in a particular way, described next. Let $p_1 \geq p_2 \geq \ldots \geq p_k$ be the k objects from T^* with the largest profits. This subset will be considered explicitly as T by A_k. We will examine the action of A_k when it works with this particular T. (If the algorithm's output uses a different T, that set must give at least as good a profit sum, so proving the theorem for T^* will be sufficient.) A_k considers the remaining objects in profit density order, adding those that fit. Let $p_{k+1}, \ldots, p_t$ be indexed in decreasing (or nonincreasing) profit density order. Let q be the first index in the optimal solution that is *not* added to T by A_k. (If there is no such q, then A_k gives an optimal solution.) Thus the optimal solution is indexed as follows, and it will be useful to think of the objects divided into the three groups indicated.

$$\underbrace{p_1 \geq \cdots \geq p_k}_{k \text{ largest profits}}, \quad \underbrace{p_{k+1}/s_{k+1} \geq \cdots \geq p_{q-1}/s_{q-1}}_{\text{Middle group}} \quad \underbrace{\geq p_q/s_q \geq \cdots \geq p_t/s_t}_{\text{Small group}}$$

Observe that p_q is not one of the k largest profits in the optimal solution, which has total profit m, so

$$p_q \leq m/(k+1). \tag{13.1}$$

We will use this bound later.

The sum of the profits in the optimal solution can be described as:

$$m = kLargest + middles + smalls \tag{13.2}$$

At the point when A_k considers and rejects object q, its **profit** is

$$\textbf{profit} = kLargest + middles + extras, \tag{13.3}$$

where the *extras* all have profit density at least p_q/s_q (because they were considered before the qth object).

Now we aim to get bounds on the profits of some of the groups that make up the optimal solution and the algorithm's solution. We will use $||group||$ to mean the total size of the objects in a *group*. The first bound is:

$$extras - \left(\frac{p_q}{s_q}\right) ||extras|| \geq 0. \tag{13.4}$$

Proof:

$$extras = \sum p_j = \sum \left(\frac{p_j}{s_j}\right) s_j \geq \sum \left(\frac{p_q}{s_q}\right) s_j = \left(\frac{p_q}{s_q}\right) ||extras||,$$

where the j in the summations indexes the extra objects added by A_k that are not in the optimal solution.

The next bound we need involves C, the capacity of the knapsack:

$$smalls \leq \left(\frac{p_q}{s_q}\right) (C - ||kLargest|| - ||middles||). \tag{13.5}$$

Proof:

$$\begin{aligned} smalls &= \sum_{i=q}^{t} p_i \leq \sum_{i=q}^{t} \left(\frac{p_q}{s_q}\right) s_i \leq \left(\frac{p_q}{s_q}\right) ||smalls|| \\ &\leq \left(\frac{p_q}{s_q}\right) (C - ||kLargest|| - ||middles||). \end{aligned}$$

The last inequality follows from the fact that the small group must fit in the knapsack along with the largest and middle groups.

So now using Equations (13.2), (13.4), and (13.5), we can conclude that

$$\begin{aligned} m &= kLargest + middles + smalls \\ &\leq kLargest + middles + \left(extras - \left(\frac{p_q}{s_q}\right) ||extras||\right) + \left(\frac{p_q}{s_q}\right) (C - ||kLargest|| - ||middles||) \\ &= (kLargest + middles + extras) + \left(\frac{p_q}{s_q}\right) (C - ||kLargest|| - ||middles|| - ||extras||) \\ &< (kLargest + middles + extras) + p_q \end{aligned}$$

because $C - ||kLargest|| - ||middles|| - ||extras|| < s_q$; otherwise A_k would have put the qth object in the knapsack. So, using the last result and Equations (13.1) and (13.3), we now have

$$\begin{aligned} m &< (kLargest + middles + extras) + p_q < \textbf{profit} + \frac{m}{k+1} \\ &\leq val(A_k, I) + \frac{m}{k+1}. \end{aligned}$$

and it is easy to rearrange terms to get

$$\frac{m}{val(A_k, I)} < 1 + \frac{1}{k}.$$

Thus $R_{A_k}(m) \leq 1 + \frac{1}{k}$ and $S_{A_k}(n) \leq 1 + \frac{1}{k}$.

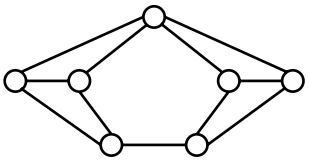
Section 13.7: *Graph Coloring*

13.42

For each vertex v, Sequential Coloring tries each color in turn, beginning with color 1, until it finds one it can use. A color is rejected only if there is a vertex adjacent to v that has already been assigned that color, so the maximum number of colors that can be rejected is the degree of v. The color assigned to v is at most $degree(v) + 1$. Hence the maximum number of colors used is at most $\Delta(G) + 1$.

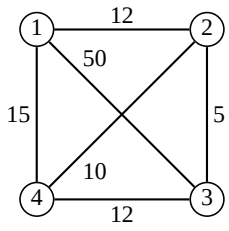
13.44
$G_1[G_2]$ has $n_1 n_2$ vertices and $n_1 m_2 + m_1 n_2^2$ edges.

13.47
In the graph at the right, the neighborhood of every vertex is 2-colorable, but the graph is not 3-colorable. To verify the latter, note that any triangle must use three colors; the possibilities are quickly exhausted.



Section 13.8: *The Traveling Salesperson Problem*

13.49
Consider the graph at the right. Start at vertex 1. The nearest neighbor algorithm chooses the tour 1,2,3,4,1, which has weight 44. The shortest link algorithm chooses edge (2,3), then (2,4), and is forced to choose (1,3) with weight 50.



13.51

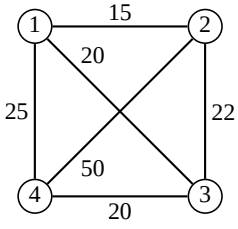
a)

```
nearest2ends(V, E, W)
     Select an arbitrary vertex s to start the cycle C.
     u = s; // One end of the chain.
     v = s; // The other end.
     While there are vertices not yet in C:
          Select an edge uw or vw of minimum weight, where w is not in C.
          Add the selected edge to C;
          If the selected edge is uw
               u = w;
          else
               v = w;
     Add the edge vu to C.
     return C;
```

b) It finds the same tour.

Another exercise: Construct a graph for which **`nearest2ends`** does better than the nearest-neighbor strategy.

c) It might seem that having more choices at each step gives a better chance of finding a minimum tour. However, since being greedy doesn't work for the TSP problem, more chances to be greedy can produce even worse tours. In the graph at the right, **`nearest2ends`** produces a worse tour than the ordinary nearest-neighbor strategy, starting at any vertex. Starting at 1, **`nearest2ends`** selects the edge (1, 2), then (3, 1), then (4, 3), and finally it completes the cycle 4, 3, 1, 2, 4 for a cost of 105. The nearest-neighbor strategy produces the cylce 1, 2, 3, 4, 1 for a cost of 82. The results are similar starting elsewhere.



13.53

a) $\Theta(n^2)$ is the best possible because the graph has about $n^2/2$ edges (so we can't do better) and Prim's algorithm runs in $\Theta(n^2)$. Prim's algorithm produces an in-tree representation of the minimum spanning tree, but an out-tree is more convenient for preorder traversal. However, Exercise 2.18 showed that an in-tree can be converted to an out-tree in linear time.

b) Rather than give a formal proof, we will present an intuitive argument. The triangle inequality implies that a direct route between two cities is never more than a round-about route. Although it only is explicit about round-about routes of two edges, it is easy to see that it extends to round-about routes of $k \geq 2$ edges.

Consider an optimum tour, $v_1, v_2, \ldots, v_n, v_1$ with total weight T. Remove the minimum-weight edge in this tour. What remains is a spanning tree, so it has a weight at least as great as the minimum spanning tree. So it suffices to show that the tour generated by the MST-based algorithm costs at most twice the weight of the MST.

Now think of the MST-based tour as using only edges in the MST, but using each edge twice, once forward and once backward, in the order they are traversed by depth-first search. Viewing DFS as a journey around the graph (i.e., the MST) is helpful here. The cost of this "pseudo-tour" is twice the weight of the MST. The vertices are visited in the same order as in the MST-based tour, if we take preorder time as being when they are visited. So this pseudo-tour that stays in the MST edges costs twice the MST weight, and the actual MST-based tour costs at most the same, because any direct routes it uses are less in weight than the round-about route used by the pseudo-tour.

***Section 13.9*:** *Computing with DNA*

13.54

At the beginning of Step 2c, we have single DNA strands representing paths of length n, beginning at v_{start} and ending at v_{end}. Choose any two consecutive edges in such a path. For simplicity of numbering, suppose they are (1,2) and (2,3). Let R_1, R_2, and R_3 be the strings for vertices 1, 2, and 3. Then, some strands include the sequence $S_{1\to2}S_{2\to3}$, i.e.,

$$d_{1,11}\cdots d_{1,20}\, d_{2,1}\cdots d_{2,10}\, d_{2,11}\cdots d_{2,20}\, d_{3,1}\cdots d_{3,10}.$$

If R_4 happens to be $d_{1,16}\cdots d_{1,20}\, d_{2,1}\cdots d_{2,15}$, then $\overline{R_4}$ could attach to such a strand.

Additional Problems

13.55

a) True. The traveling salesperson problem is NP-complete.

b) False. $P \subseteq NP$.

c) True. Any algorithm for satisfiability solves 2-SAT (2-satisfiability, sometimes called 2-CNF-SAT), so the identity transformation shows that 2-SAT $\leq_P$ satisfiability. But this does not imply that 2-SAT is NP-hard. (In fact, 2-SAT is in P, so using a general satisfiability algorithm would be overkill.)

d) Unknown. If $P = NP$, then exact graph coloring would be in P. All we know now is that if $P \neq NP$, then there is no such approximation algorithm.

13.57

A simple greedy algorithm could work as follows.

1. Sort the job time requirements, t_i, into decreasing (nonincreasing) order.
2. Always assign the next job to the processor that finishes soonest. (In terms of bins, add the next t_i to a bin with the smallest current sum.)

This clearly can be implemented in $O(n^2)$ time. The example with $p = 2$ and time requirements 7, 6, 3, 2, 2 shows that this strategy does not always produce optimal solutions.

13.59

a) The TSP is commonly defined with integer edge weights. If rational weights are allowed, we will assume that the weights have been scaled to integers as a preprocessing step. As discussed in the solution of Exercise 13.4, this can be done in polynomial time by putting all the rational numbers over a common denominator, then multiplying all the weights by that common denominator. This blows up the input length by a quadratic factor at most. (The method described in this solution does not work if the weights are floating point numbers.)

Let **`decideTSP(G, k)`** be a Boolean function that decides the traveling salesperson decision problem for a complete weighted graph $G = (V,E,W)$ and tour bound k. Let W_{max} be the largest edge weight in G. Let $T_{max} = W_{max}n$. The weight of an optimal tour is at most T_{max}. Note that the *values* of W_{max} and T_{max} are not polynomially bounded in the size of G, so we don't time time to check all the integers in the range 0 through T_{max}. The idea of the algorithm is to use Binary Search on the integers $0, \ldots, T_{max}$ to find the optimal tour weight. (This is similar to the solution of Exercise 1.47.)

```
int minTSP(G)
     W_max = max_{e∈E} W(e); T_max = W_max * n;
     low = 0; high = T_max;
     while (low + 1 ≤ high)
          mid = (low + high) / 2;
          if (decideTSP(G, mid) == true)
               high = mid;
          else
               low = mid + 1;
          // Continue loop.
     return high;
```

The number of iterations of the loop is about $\lg T_{max} = \lg(W_{max}) + \lg(n)$, which is polynomially bounded in the size of the input. If **`decideTSP`** runs in polynomial time, then so does **`minTSP`**.

b) Use the algorithm **`minTSP`** in part (a) to find T, the weight of an optimal tour for the graph $G = (V,E,W)$. The next algorithm constructs an optimal tour by testing each edge in G and discarding those that are not required. We consider an edge discarded if its final weight is greater than T; certain weights are increased in the **for** loop. Clearly, an edge with weight greater than T cannot be in a tour of total weight T.

```
findTour(G, T)
     For each edge e in G:
          Change W(e) to T+1.
          if (decideTSP(G, T) == false)
               // e is needed for a minimum tour.
               Change W(e) back to its original value.
     Output edges with W(e) ≤ T.
```

`decideTSP` is called $|E|$ times, so if **`decideTSP`** runs in polynomial time, then **`findTour`** does.

13.61

Let **`polySatDecision(E)`** be a polynomial-time Boolean function that determines whether or not the CNF formula E is satisfiable. Let n be the number of variables in E, and let the variables be $x_1, \ldots, x_n$. The idea is to choose a truth value for each variable in turn, substitute it into the formula, and then use **`polySatDecision`** to check whether there is a truth value assignment for the remaining variables that will make the simpler formula true. (The substitution process may create empty clauses. Although the text does not go into detail on empty clauses, the definitions given imply that an empty clause cannot be made true by any assignment, and so such a clause causes the CNF formula to be unsatisfiable.) If not, choose the opposite value for the variable. A **`boolean`** array, **`assign`**, is supplied to store the assignment.

```
satAssign(E, assign)
     if (polySatDecision(E) == false)
          Output "not satisfiable".
          return;

     for (i = 1; i ≤ n; i ++)
          // Try assigning true to x_i.
          Etrue = the formula constructed from E by deleting all clauses
          containing x_i and deleting the literal ¬x_i from all other clauses.
          Efalse = the formula constructed from E by deleting all clauses
          containing ¬x_i and deleting the literal x_i from all other clauses.
          if (polySatDecision(Etrue) == true)
               assign[i] = true;
               E = Etrue;
          else
               assign[i] = false;
               E = Efalse;
          // Continue loop.
     Output truth assignment in assign array.
```

Since there are n variables, the size of the formula is at least n, so the number of iterations of the loop is polynomial in the size. The modifications needed to produce the new formula can be made in one pass over the current version of the formula, and the formula never gets larger. Thus, if **`polySatDecision`** runs in polynomial time, then **`setAssign`** does also.

Chapter 14
Parallel Algorithms

Section 14.3: *Some Simple PRAM Algorithms*

14.2

Each processor's index is **pid**, in the range $0, \ldots, n-1$. The array **idx[0]**, ..., **idx[2*n-1]** is used and the final result is left in **idx[0]** instead of **M[0]**.

```
parTournMaxIndex(M, idx, n)
     int incr;
     Write −∞ (some very small value) into M[n].
     Write pid into idx[pid] and write n into idx[pid+n].
     incr = 1;
     while (incr < n)
          int  idx0, idx1, idxBig;
          Key  key0, key1;
          Read idx[pid] into idx0 and read idx[pid+incr] into idx1.
          Read M[idx0] into key0 and read M[idx1] into key1.
          if (key1 > key0) idxBig = idx1; else idxBig = idx0;
          Write idxBig into idx[pid].
          incr = 2 * incr;
```

Section 14.4: *Handling Write Conflicts*

14.4

Use n^3 processors, indexed $P_{i,j,k}$ for $0 \leq i, j, k < n$, to parallelize the nested **for** loops in the original algorithm. An element **newR[i][j][k]** in a three-dimensional work array is set to 1 in each iteration if and only if a path from i to j through k has been found, and no path from i to j was already known. Then use binary fan-in to combine the results for each (i, j) and update r_{ij}. If r_{ij} is already 1, $P_{i,j,k}$, for $0 \leq k < n$, do no more work.

```
parTransitiveClosure(A, R, n)
     int incr;
     P_{i,j,0} copies a_{ij} into r_{ij}.
     P_{i,i,0} (those on the diagonal) set r_{ii} to 1.
     incr = 1;
     while (incr < n)
          if (r_{ij} == 0)
               Each P_{i,j,k} does:  newR[i][j][k] = r_{ik} ∨ r_{kj}.
               Each group of processors with k as their third index does:
                    Compute the Boolean OR of newR[i][j][0], ..., newR[i][j][n-1],
                    using binary fan-in, leaving the result in newR[i][j][0].
               Each P_{i,j,0} does:  if(newR[i][j][0] == 1) write 1 into r_{ij}.
          incr = 2 * incr;
```

14.5

This is a simple modification of Algorithm 14.1. The integers to be added are in memory cells **M[0]**, **M[1]**, ..., **M[n-1]**; the sum will be left in **M[0]**. Each processor carries out the algorithm using its own index for **pid**.

```
parSum(M, n)
     int incr, sum0, sum1;
     Write 0 into M[n+pid].
     incr = 1;
     while (incr < n)
          Read M[pid] into sum0 and read M[pid+incr] into sum1;
          Write (sum0 + sum1) into M[pid];
          incr = 2 * incr;
```

14.7
Let the matrices be A and B, and let C be the product;

$$c_{ij} = \bigvee_{k=0}^{n-1} a_{ik} b_{kj}.$$

We use n^3 processors, one group of n processors $P_{i,j,0}, \ldots, P_{i,j,n-1}$ to compute each entry c_{ij} of C.

```
parMultBooleanMatrix(A, B, C, n)
    Step 1:
        Each P_{i,j,0} writes 0 into c_{ij}.
    Steps 2 and 3:
        Each P_{i,j,k} computes a_{ik} ∧ b_{kj} and stores it in cTerm[i][j][k].
    Step 4:
        P_{i,j,0}, ... , P_{i,j,n-1} compute the Boolean OR
        of cTerm[i][j][0], ..., cTerm[i][j][n-1] as in Algorithm 14.2,
        writing the result in c_{ij} instead of M[0].
```

In Algorithm 14.2 more than one processor may write a 1 in the same cell at one time, so this algorithm works on Common-Write or stronger PRAM models.

14.8
The solution is similar to Exercise 14.4 (see above), except that **newR** is a local variable in each processor and if $P_{i,j,k}$'s value of **newR** is 1, it writes a 1 into r_{ij}. Thus, if multiple processors write into the same cell, they all write a 1. Now the body of the **while** loop executes in $\Theta(1)$ steps and there are $\Theta(\log n)$ iterations.

14.10
Suppose that there were an algorithm that can multiply two $n \times n$ Boolean matrices in $o(\log n)$ steps on a CREW PRAM. Let $x_0, x_1, \ldots, x_{n-1}$ be n bits. We will show that the matrix multiplication algorithm could be used to compute the Boolean *or* of these bits in $o(\log n)$ steps, contradicting the lower bound for Boolean *or*.

Let A be an $n \times n$ Boolean matrix whose first row contains the x_i, and let B be an $n \times n$ Boolean matrix whose first column contains all 1's. The first (upper left corner) element in the matrix product is $x_0 \vee x_1 \vee \cdots \vee x_{n-1}$.

14.11
Algorithm 14.3 could fail if each processor chooses the loser arbitrarily when the keys it compares are equal. Suppose there are three keys, all equal. P_{01} might write 1 in loser[0], P_{02} might write 1 in loser[2], and P_{12} might write 1 in loser[1], so there will be no entry in the loser array that is zero.

***Section 14.5*:** *Merging and Sorting*

14.13
There are two natural solutions. One solution overlaps the insertion of keys; that is, it begins a new iteration of the outer loop (the **for** loop) in Algorithm 4.1 before earlier iterations are completed. The solution given here is a little easier to follow; it does each insertion (the work of the **while** loop) in parallel in a constant number of steps. It uses $n-1$ processors, indexed $P_0, \ldots, P_{n-2}$. A local variable **pid** is the processor's index.

```
parInsertionSort(M, n)
    Key  key0, key1, newKey;
    for (i = 1; i < n; i ++)
        // Insert the key in M[i] into its proper place among the sorted
        // keys in M[0], ..., M[i-1].
        Each processor does:
            Read M[i] into newKey;
            Read M[pid] into key1;
            If (pid > 0) read M[pid-1] into key0; else key0 = −∞.
            if (newKey < key1) write key1 into M[pid+1].
            // This moves all keys larger than M[i] to the ''right.''
            If (key0 ≤ newKey < key1) write newKey into M[pid].
            // Only one processor writes newKey (the original M[i]), and
            // it writes into the correct position.
```

Since there are no concurrent writes, this is a CREW PRAM algorithm.

14.14

Assume that there is an X element K_x at position m_x and a Y element K_y at position m_y. Let the cross-rank of K_x be r_x and the cross-rank of K_y be r_y.

If $K_x \leq K_y$, then $r_y > m_x$ and $r_x \leq m_y$. So $m_x + r_x < m_y + r_y$, and K_x and K_y cannot be written to the same output location.

Similarly, if $K_x > K_y$ then $r_y \leq m_x$ and $r_x > m_y$. So $m_x + r_x > m_y + r_y$, and K_x and K_y cannot be written to the same output location.

The exercise asks only about an X element and a Y element. It is also easy to show that two elements in X, K_1 and K_2, cannot be written to the same output location. The argument for two elements in Y is the same. Suppose K_1 is at position m_1 and K_2 is at position m_2, where $m_1 < m_2$. Since $K_1 \leq K_2$, the cross-rank of K_1 is less than or equal that of K_2. So the output location of K_1 is less than that of K_2.

14.16

The keys are in **M[0]**, ..., **M[n-1]**. Use n^2 processors, $P_{i,j}$ for $0 \leq i, j < n$. The algorithm computes elements of a matrix **isBefore** such that **isBefore[i][j]** is 1 if **M[i]** should be somewhere before **M[j]** in the sorted order and is 0 otherwise. Then the algorithm sums each column of the matrix to find $rank(j)$, the number of keys that should precede **M[j]** in the sorted order. Finally, the original **M[j]** is placed in **M[**$rank(j)$**]**.

```
parCountingSort(M, n)
    Each P_{i,j} does:
        Read M[i] and M[j].
        if (M[i] < M[j] || (M[i] == M[j] && i < j))
            Write 1 into isBefore[i][j];
        else
            Write 0 into isBefore[i][j];
    Each group of processors with j as their second index does:
        Sum column j of isBefore using binary fan-in, leaving
        the result in isBefore[0][j].  Use the isBefore matrix for work space.
    Each P_{0,j} does:
        Read isBefore[0][j] into rankJ.  // M[j] was read earlier.
        Write the saved value of M[j] into M[rankJ].
```

The binary fan-in takes $\Theta(\log n)$ steps and the other parts take $\Theta(1)$ steps.

Section 14.6: *Finding Connected Components*

14.18

First, perform Transitive Closure by Shortcuts by the method from Exercise 14.4 or 14.8, depending on the PRAM model being used. Call the resulting matrix R.

Next, for each row v of R, determine the minimum column index among elements in that row that contain a 1. This index in row v is **leader[v]**. A simple variant of Algorithm 14.1 can be used. The processors work in n groups of n

for each row in parallel. (For the Common-Write model, a variant of Algorithm 14.3 can be used, but it won't change the asymptotic order. The processors work in n groups of n^2 for each row in parallel.)

a) For the Common-Write model, the Transitive Closure by Shortcuts can be done in $\Theta(\log n)$ steps by the method from Exercise 14.8. The leaders can be computed in $\Theta(1)$ steps with a variant of Algorithm 14.3. So the total is $\Theta(\log n)$.

b) For the CREW model, the Transitive Closure by Shortcuts can be done in $\Theta(\log^2 n)$ steps by the method from Exercise 14.4. The leaders can be computed in $\Theta(\log n)$ steps with a variant of Algorithm 14.1. So the total time is $\Theta(\log^2 n)$.

14.19

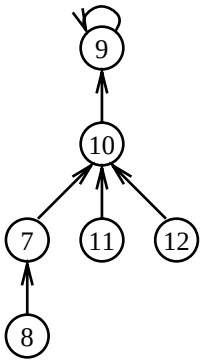
The idea is to count the roots. Let **`root`** be an n-element array in memory. Use n processors. Each P_i does:

```
if (parent[i] == i) root[i] = 1; else root[i] = 0;
```

Then compute the sum of the entries in the **`root`** array in $O(\log n)$ steps (using the solution to Exercise 14.5).

14.20

The result is the same tree in both cases because 7 is its own parent and 7 is 8's parent.



14.22

If there is any undirected edge ij in G such that $i > j$ then the *conditional singleton hooking* step of **`initParCC`** hooks i to something, so node i is not a root for the next step. No cycles are formed by conditional singleton hooking because all directed edges (parent pointers) in the data structure point from a higher numbered vertex to a lower numbered one.

Now consider the *unconditional singleton hooking* step. If node i is a singleton, then any undirected edge ij in G must satisfy $i < j$, so j cannot be a root at this point. Therefore, no node in j's tree is a singleton, so no node in j's tree is attached to any new node in this step. Attaching the singleton i to the parent of j cannot produce a cycle.

In either case, node i must get attached to something, so all nodes are in in-trees of two of more nodes at the end of this step, except for isolated nodes.

14.24

```
Do in parallel for all vertices v:
    singleton[v] = true;
    if (parent[v] ≠ v)
        singleton[v] = false;
        singleton[parent[v]] = false;
```

14.25

Let the undirected graph G_n consist of edges $(1,i)$ and $(i-1,i)$ for $2 \le i \le n$. In the *conditional singleton hooking* step of **`initParCC`** it is possible that every node $i \ge 2$ is attached to node 1, immediately making one star containing all

the nodes. It is also possible that each node $i \geq 2$ is attached to node $i-1$, making an in-tree of height $n-1$. Nothing is hooked in the *unconditional singleton hooking* step in either case.

In the first case, the algorithm terminates after one iteration of the **`while`** loop because nothing changes. In the second case, the depth of node n starts at $n-1$ and shrinks by a factor of approximately 2 in each iteration (see the proof of Theorem 14.2), so $\Theta(\log n)$ iterations are needed.

14.27

Based on Exercise 14.26, it is only necessary to record the edges ij that were associated with *successful* hooking operations. After a write operation is performed, it may be impossible to determine which processor actually succeeded in writing. (Two processors may have tried to write the same value in the same memory location, but for different hook operations, i.e., different edges.) We introduce a new array **`reserve`** and modify the hooking operation given in Definition 14.4 of the text so that processors first try to reserve the right to write in **`parent[v]`** by writing their **`pid`** in **`reserve[v]`**.

```
hook(i, j)
     P_{i,j} does:
     Read parent[j] into parJ.
     Read parent[i] into parI.
     Write pid into reserve[parI].
     Read reserve[parI] into winningPid.
     if (winningPid == pid)
          Write parJ into parent[parI].
          Write 1 into stc[i][j].
```

Additional Problems

14.29

Processors are indexed by $0, \ldots, n-1$.

a) Each processor P_i reads k from $M[0]$. If $k > i$ then write 1 into $M[i]$, else write 0.

b) **Solution 1.**

```
P_i does:
Read M[i] into bit.
if (bit == 0)
     Write i into M[0].
else if (i == n-1)
     Write n into M[0].
```

The result may be written into a location different from **`M[0]`**, if desired.

Solution 2. Each processor P_i reads $M[n-1-i]$. If it is a 1, write $n-i$ to $M[0]$. Suppose there are 1's in $M[0]$, $\ldots$, $M[k-1]$, where $k > 0$. Processors P_{n-k}, P_{n-k+1}, $\ldots$, P_{n-1} will all be trying to write results k, $k-1$, $\ldots$, 1, respectively, into $M[0]$. Because low numbered processors have priority, only processor P_{n-k} will succeed in writing k to $M[0]$.

If there are no 1's in $M[0]$, $\ldots$, $M[n-1]$, then no processor will attempt to write into $M[0]$, but $M[0]$ already contains a 0 so the result is correct. However, if the problem were changed slightly to require the output in a different location, then P_{n-1}, the lowest-priority processor, reads $M[0]$ and writes a 0 if $M[0] = 0$ or writes a 1 if $M[0] = 1$.

c) Each processor P_i reads $M[i]$ into **`cur`**. Each processor P_i reads $M[i+1]$ into **`next`**, except that P_{n-1} just assigns 0 to **`next`**. Each processor P_i does: If **`cur`** is 1 and **`next`** is 0, then write $i+1$ to $M[0]$. This works because only one processor, the one that reads the final 1, will write a result to $M[0]$ and so there can be no write conflict.

As in part (b), Solution 2, if no processor writes, then $M[0] = 0$, which is correct. But if the output is required to be in a different location, then P_0 decides whether to write a 0 (if $M[0] = 0$) or a 1 (if $M[0] = 1$ and $M[1] = 0$) or not write (both are 1's). This decision is made during the computation phase, which precedes the write phase, so P_0 still finishes in the second step.

14.30

This solution is a simplified version of one given by Marc Snir, "On Parallel Searching," *SIAM Journal on Computing*, vol. 14, no. 3, 1985. The idea is to divide the keys into $p+1$ roughly equal intervals and use the p processors to "sample" the endpoints of the first p intervals. From the results of these $2p$ comparisons, the processors can determine which interval can contain K (x, in the statement of the exercise) if it is in the array at all. The search then proceeds in the same manner on an interval roughly $1/(p+1)$ times the size of the previously searched interval.

The keys reside in indexes $0,\ldots,n-1$ of M. We can think of **M[n]** as containing ∞. The variables **first** and **last** (the boundaries of the interval being searched) are in the shared memory. Each processor has local variables **lower** and **upper** that indicate which elements it is to examine, as well as **pid**, its processor index.

```
parSearch(M, n, p, K)
    P0 and Pp-1 do in parallel:
        Write 0 into first; write n into last;
        if (K < M[0]) Output K " is not in the array"; halt.
        if (K > M[n-1]) Output K " is not in the array"; halt.

    Each processor reads first and last.
    Each processor executes the following loop.
    while (first < last - 1)
        // Precondition:  M[first] ≤ K < M[last].
        lower = ((p - pid + 1) * first + (pid) * last) / (p+1);
        upper = ((p - pid) * first + (pid + 1) * last) / (p+1);
        if (M[lower] ≤ K < M[upper])
            Write lower into first.
            Write upper into last.
        else if (pid == n-1 && K ≥ M[upper])
            Write upper into first.
            Write last into last.
        // Only one processor writes into first and last.
        Read first and read last.
        // Continue while loop.
    P0 does:
        if (K == M[first])
            Output first.
        else
            Output K " is not in the array".
```

The work done in one iteration of the loop is constant. The size of the interval being searched is roughly $1/(p+1)$ times the size of the interval at the preceding iteration, so the number of iterations is roughly $\lg(n+1)/\lg(p+1)$. Thus the number of steps is in $\Theta(\log(n+1)/\log(p+1))$.

14.32

All the following problems are in NC:

- Finding the maximum, the sum, and the Boolean *or* of n inputs (Algorithm 14.1).
- Matrix multiplication (Section 14.3.2).
- Merging sorted sequences (Algorithm 14.4).
- Sorting (Algorithm 14.5).
- Finding connected components (Algorithm 14.6).
- Finding the transitive closure of a relation (Exercise 14.4).

Almost all the algorithms described in or to be written for the exercises solve problems that are in NC. (The wording of the exercise is a little loose; *problems*, not *algorithms*, are or are not in NC. The more precise question would be "Are there any algorithms in this chapter that do not run in poly-log time with a polynomially bounded number of processors?") The $O(n)$ implementation of Insertion Sort (Exercise 14.13) does not run in poly-log time. A straightforward parallelization of some dynamic programming algorithms might not run in poly-log time.