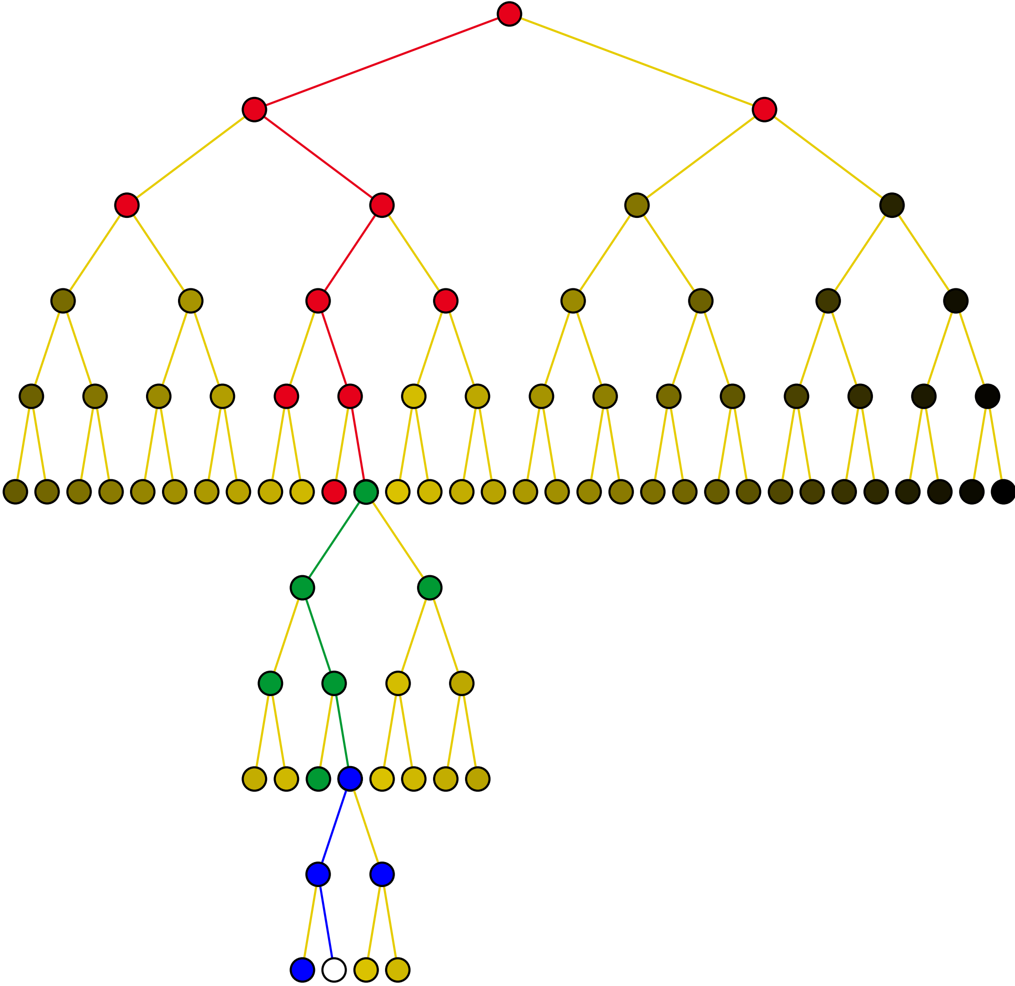
Computer Algorithms, Third Edition,
Solutions to Selected Exercises

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`

©Copyright 2000 Sara Baase and Allen Van Gelder. All rights reserved.

Permission is granted for college and university instructors to make a reasonable number of copies, free of charge, as needed to plan and administer their courses. Instructors are expected to exercise reasonable precautions against further, unauthorized copies, whether on paper, electronic, or other media.

Permission is also granted for Addison-Wesley-Longman editorial, marketing, and sales staff to provide copies free of charge to instructors and prospective instructors, and to make copies for their own use.

Other copies, whether paper, electronic, or other media, are prohibited without prior written consent of the authors.

List of Solved Exercises

1 Analyzing Algorithms and Problems: Principles and Examples **1**

1.1 1
1.2 2
1.4 2
1.6 2
1.8 3
1.10 3
1.12 3
1.13 3
1.15 4
1.18 4
1.20 4
1.22 4
1.23 4
1.25 5
1.28 5
1.31 6
1.33 6
1.35 6
1.37 6
1.39 6
1.42 7
1.44 7
1.46 7
1.47 7
1.48 7
1.50 8

2 Data Abstraction and Basic Data Structures **9**

2.2 9
2.4 9
2.6 9
2.8 9
2.10 11
2.12 11
2.14 12
2.16 13
2.18 14

3 Recursion and Induction **17**

3.2 17
3.4 17
3.6 17
3.8 18
3.10 18
3.12 18

4 Sorting **19**

4.2 19
4.4 19
4.6 19
4.9 19
4.11 19
4.13 20
4.15 20
4.17 20
4.19 21
4.21 21
4.23 21
4.25 22
4.26 22
4.27 23
4.29 23
4.31 23
4.34 24
4.35 24
4.37 24
4.40 24
4.42 24
4.44 25
4.45 25
4.46 25
4.48 25
4.49 25
4.51 26
4.53 26
4.55 27
4.57 27
4.59 28
4.61 28
4.63 29
4.65 29

5 Selection and Adversary Arguments **31**

5.2 31
5.4 32
5.6 32
5.8 33
5.10 34
5.12 34
5.14 34
5.16 34
5.19 35
5.21 35
5.22 36
5.24 37

6 Dynamic Sets and Searching **39**

6.1 39
6.2 39
6.4 40
6.6 40
6.8 41
6.10 41
6.12 41
6.14 43
6.16 45
6.18 45
6.20 45
6.22 46
6.24 47
6.26 47
6.28 47
6.30 47
6.32 48
6.34 49
6.36 49
6.37 49
6.40 50

7 Graphs and Graph Traversals 51

7.1 51
7.3 51
7.4 51
7.6 51
7.8 51
7.10 52
7.12 52
7.14 53
7.16 53
7.18 53
7.20 54
7.22 54
7.24 54
7.27 57
7.28 57
7.30 57
7.32 57
7.34 57
7.35 58
7.37 59
7.39 59
7.40 59
7.41 59
7.43 59
7.45 60
7.47 60
7.49 61

8 Graph Optimization Problems and Greedy Algorithms 63

8.1 63
8.3 63
8.5 63
8.7 64
8.8 64
8.10 64
8.12 64
8.14 64
8.16 65
8.18 65
8.20 65
8.22 67
8.24 67
8.26 67
8.27 67

9 Transitive Closure, All-Pairs Shortest Paths 69

9.2 69
9.4 70
9.6 71
9.7 71
9.8 71
9.10 71
9.12 72
9.14 72
9.16 72
9.18 72

10 Dynamic Programming 73

10.2 73
10.4 73
10.5 73
10.7 73
10.9 73
10.10 74
10.12 75
10.14 75
10.16 75
10.18 76
10.19 77
10.21 78
10.23 78
10.26 79

11 String Matching 81

11.1 81
11.2 81
11.4 81
11.6 83
11.8 84
11.10 84
11.12 84
11.15 84
11.17 84
11.19 85
11.21 85
11.23 85
11.25 86

12 Polynomials and Matrices 87

12.2 87
12.4 87
12.6 87
12.8 87
12.10 87
12.12 88
12.14 88
12.16 88
12.17 88

13 NP-Complete Problems 89

13.2 89
13.4 89
13.6 91
13.8 91
13.10 91
13.12 91
13.14 92
13.16 92
13.18 92
13.20 93
13.21 93
13.23 93
13.26 93
13.28 93
13.30 94
13.32 94
13.34 96
13.35 96
13.37 96
13.39 96
13.42 98
13.44 99
13.47 99
13.49 99

13.51 99
13.53 100
13.54 100
13.55 100
13.57 100
13.59 101
13.61 101

14 Parallel Algorithms 103

14.2 103
14.4 103
14.5 103
14.7 104
14.8 104
14.10 104
14.11 104
14.13 104
14.14 105
14.16 105
14.18 105
14.19 106
14.20 106
14.22 106
14.24 106
14.25 106
14.27 107
14.29 107
14.30 108
14.32 108

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 $0 < n \le k$ 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 $2^{x-1} < n+1 \le 2^x$.

```
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.

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

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

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

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

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,

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

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) {
```