CHAPTER 3 Algorithms

SECTION 3.1 Algorithms

Many of the exercises here are actually miniature programming assignments. Since this is not a book on programming, we have glossed over some of the finer points. For example, there are (at least) two ways to pass variables to procedures—by value and by reference. In the former case the original values of the arguments are not changed. In the latter case they are. In most cases we will assume that arguments are passed by reference. None of these exercises are tricky; they just give the reader a chance to become familiar with algorithms written in pseudocode. The reader should refer to Appendix 3 for more details of the pseudocode being used here.

1. Initially *max* is set equal to the first element of the list, namely 1. The **for** loop then begins, with i set equal to 2. Immediately i (namely 2) is compared to n, which equals 10 for this sequence (the entire input is known to the computer, including the value of n). Since $2 < 10$, the statement in the loop is executed. This is an **if**... **then** statement, so first the comparison in the **if** part is made: *max* (which equals 1) is compared to $a_i = a_2 = 8$. Since the condition is true, namely $1 < 8$, the **then** part of the statement is executed, so *max* is assigned the value 8.

The only statement in the **for** loop has now been executed, so the loop variable i is incremented (from 2 to 3), and we repeat the process. First we check again to verify that i is still less than n (namely $3 < 10$), and then we execute the **if**... **then** statement in the body of the loop. This time, too, the condition is satisfied, since $max = 8$ is less than $a_3 = 12$. Therefore the assignment statement $max := a_i$ is executed, and *max* receives the value 12.

Next the loop variable is incremented again, so that now $i = 4$. After a comparison to determine that $4 < 10$, the **if**... **then** statement is executed. This time the condition fails, since $max = 12$ is not less than $a_4 = 9$. Therefore the **then** part of the statement is not executed. Having finished with this pass through the loop, we increment i again, to 5. This pass through the loop, as well as the next pass through, behave exactly as the previous pass, since the condition $max < a_i$ continues to fail. On the sixth pass through the loop, however, with $i = 7$, we find again that $max < a_i$, namely $12 < 14$. Therefore *max* is assigned the value 14.

After three more uneventful passes through the loop (with $i = 8$, 9, and 10), we finally increment i to 11. At this point, when the comparison of i with n is made, we find that i is no longer less than or equal to n, so no further passes through the loop are made. Instead, control passes beyond the loop. In this case there are no statements beyond the loop, so execution halts. Note that when execution halts, *max* has the value 14 (which is the correct maximum of the list), and i has the value 11. (Actually in many programming languages, the value of i after the loop has terminated in this way is undefined.)

3. We will call the procedure *AddEmUp*. Its input is a list of integers, just as was the case for Algorithm 1. Indeed, we can just mimic the structure of Algorithm 1. We assume that the list is not empty (an assumption made in Algorithm 1 as well).

```
procedure $AddEmUp(a_1, a_2, \ldots, a_n$ : integers)
$sum := a_1$
for $i := 2$ to $n$
    $sum := sum + a_i$
return $sum$
{ $sum$ is the sum of all the elements in the list}
```

5. We need to go through the list and find cases when one element is equal to the following element. However, in order to avoid listing the values that occur more than once more than once, we need to skip over repeated duplicates after we have found one duplicate. The following algorithm will do it. (If we wanted to "**return**" the answer, we would form c_1, c_2, ..., c_k into a list and **return** that list.)

```
procedure $duplicates(a_1, a_2, \ldots, a_n$ : integers in nondecreasing order)
$k := 0$ {this counts the duplicates}
$j := 2$
while $j \le n$
    if $a_j = a_{j-1}$ then
        $k := k + 1$
        $c_k := a_j$
        while $j \le n$ and $a_j = c_k$
            $j := j + 1$
    $j := j + 1$
{ $c_1, c_2, \ldots, c_k$ is the desired list}
```

7. We need to go through the list and record the index of the last even integer seen.

```
procedure $last\ even\ location(a_1, a_2, \ldots, a_n$ : integers)
$k := 0$
for $i := 1$ to $n$
    if $a_i$ is even then $k := i$
return $k$ { $k$ is the desired location (or 0 if there are no evens)}
```

9. We just need to look at the list forward and backward simultaneously, going about half-way through it.

```
procedure $palindrome\ check(a_1 a_2 \ldots a_n$ : string)
$answer :=$ true
for $i := 1$ to $\lfloor n/2 \rfloor$
    if $a_i \ne a_{n+1-i}$ then $answer :=$ false
return $answer$ { $answer$ is true if and only if string is a palindrome}
```

11. We cannot simply write $x := y$ followed by $y := x$, because then the two variables will have the same value, and the original value of x will be lost. Thus there is no way to accomplish this task with just two assignment statements. Three are necessary, and sufficient, as the following code shows. The idea is that we need to save temporarily the original value of x.

```
$temp := x$
$x := y$
$y := temp$
```

13. We will not give these answers in quite the detail we used in Exercise 1.

a) Note that $n = 8$ and $x = 9$. Initially i is set equal to 1. The **while** loop is executed as long as $i \le 8$ and the i^{th} element of the list is not equal to 9. Thus on the first pass we check that $1 \le 8$ and that $9 \ne 1$ (since $a_1 = 1$), and therefore perform the statement $i := i + 1$. At this point $i = 2$. We check that $2 \le 8$ and $9 \ne 3$, and therefore again increment i, this time to 3. This process continues until $i = 7$. At that point the condition "$i \le 8$ and $9 \ne a_i$" is false, since $a_7 = 9$. Therefore the body of the loop is not executed (so i is still equal to 7), and control passes beyond the loop.

The next statement is the **if**... **then** statement. The condition is satisfied, since $7 \le 8$, so the statement *location* $:= i$ is executed, and *location* receives the value 7. The **else** clause is not executed. This completes the procedure, so *location* has the correct value, namely 7, which indicates the location of the element x (namely 9) in the list: 9 is the seventh element.

b) Initially i is set equal to 1 and j is set equal to 8. Since $i < j$ at this point, the steps of the **while** loop are executed. First m is set equal to $\lfloor (1+8)/2 \rfloor = 4$. Then since x (which equals 9) is greater than a_4 (which equals 5), the statement $i := m+1$ is executed, so i now has the value 5. At this point the first iteration through the loop is finished, and the search has been narrowed to the sequence $a_5, \ldots, a_8$.

In the next pass through the loop (there is another pass since $i < j$ is still true), m becomes $\lfloor (5+8)/2 \rfloor = 6$. Since again $x > a_m$, we reset i to be $m+1$, which is 7. The loop is now repeated with $i = 7$ and $j = 8$. This time m becomes 7, so the test $x > a_m$ (i.e., $9 > 9$) fails; thus $j := m$ is executed, so now $j = 7$.

At this point $i \not< j$, so there are no more iterations of the loop. Instead control passes to the statement beyond the loop. Since the condition $x = a_i$ is true, *location* is set to 7, as it should be, and the algorithm is finished.

15. We need to find where x goes, then slide the rest of the list down to make room for x, then put x into the space created. In the procedure that follows, we employ the trick of temporarily tacking $x+1$ onto the end of the list, so that the **while** loop will always terminate. Also note that the indexing in the **for** loop is slightly tricky since we need to work from the end of the list toward the front.

```
procedure insert($x, a_1, a_2, \ldots, a_n$ : integers)
{the list is in order: $a_1 \le a_2 \le \cdots \le a_n$}
$a_{n+1} := x + 1$
$i := 1$
while $x > a_i$
        $i := i + 1$ {the loop ends when $i$ is the index for $x$}
for $j := 0$ to $n - i$ {shove the rest of the list to the right}
        $a_{n-j+1} := a_{n-j}$
$a_i := x$
{$x$ has been inserted into the correct spot in the list, now of length $n + 1$}
```

17. This algorithm is similar to *max*, except that we need to keep track of the location of the maximum value, as well as the maximum value itself. Note that we need a strict inequality in the test $max < a_i$, since we do not want to change *location* if we find another occurrence of the maximum value. As usual we assume that the list is not empty.

```
procedure first largest($a_1, a_2, \ldots, a_n$ : integers)
$max := a_1$
$location := 1$
for $i := 2$ to $n$
        if $max < a_i$ then
                $max := a_i$
                $location := i$
return $location$
{$location$ is the location of the first occurrence of the largest element in the list}
```

19. We need to handle the six possible orderings in which the three integers might occur. (Actually there are more than six possibilities, because some of the numbers might be equal to each other—we get around this problem by using $\le$ rather than $<$ for our comparisons.) We will use the **if**... **then**... **else if**... **then**... **else if**... construction. A condition such as $a \le b \le c$ is really the conjunction of two conditions: $a \le b$ and $b \le c$. (Alternately, we could have handled the cases in a nested fashion.) Note that the mean is computed first, independent of the ordering.

```
procedure statistics($a, b, c$ : integers)
mean := $(a + b + c)/3$
if $a \le b \le c$ then
        min := $a$
        median := $b$
        max := $c$
else if $a \le c \le b$ then
        min := $a$
        median := $c$
        max := $b$
else if $b \le a \le c$ then
        min := $b$
        median := $a$
        max := $c$
else if $b \le c \le a$ then
        min := $b$
        median := $c$
        max := $a$
else if $c \le a \le b$ then
        min := $c$
        median := $a$
        max := $b$
else if $c \le b \le a$ then
        min := $c$
        median := $b$
        max := $a$
{the correct values of mean, median, max, and min have been assigned}
```

21. We must assume that the sequence has at least three terms. This is a special case of a sorting algorithm. Our approach is to interchange numbers in the list when they are out of order. It is not hard to see that this needs to be done only three times in order to guarantee that the elements are finally in correct order: test and interchange (if necessary) the first two elements, then test and interchange (if necessary) the second and third elements (insuring that the largest of the three is now third), then test and interchange (if necessary) the first and second elements again (insuring that the smallest is now first).

```
procedure first three($a_1, a_2, \ldots, a_n$ : integers)
if $a_1 > a_2$ then interchange $a_1$ and $a_2$
if $a_2 > a_3$ then interchange $a_2$ and $a_3$
if $a_1 > a_2$ then interchange $a_1$ and $a_2$
{the first three elements are now in nondecreasing order}
```

23. For notation, assume that $f : A \to B$, where A is the set consisting of the distinct integers a_1, a_2, ..., a_n, and B is the set consisting of the distinct integers b_1, b_2, ..., b_m. All $n+m+1$ of these entities (the elements of A, the elements of B, and the function f) are the input to the algorithm. We set up an array called *hit* (indexed by the integers) to keep track of which elements of B are the images of elements of A; thus $hit(b_i)$ equals 0 until we find an a_j such that $f(a_j) = b_i$, at which time we set $hit(b_i)$ equal to 1. Simultaneously we keep track of how many hits we have made (i.e., how many times we changed some $hit(b_i)$ from 0 to 1). If at the end we have made m hits, then f is onto; otherwise it is not. Note that we record the output as a logical value assigned to the variable that has the name of the procedure. This is a common practice in some programming languages.

```
procedure onto($f$ : function, $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m$ : integers)
for $i := 1$ to $m$
        $hit(b_i) := 0$ {no one has been hit yet}
$count := 0$ {there have been no hits yet}
for $j := 1$ to $n$
        if $hit(f(a_j)) = 0$ then {a new hit!}
                $hit(f(a_j)) := 1$
                $count := count + 1$
if $count = m$ then return true else return false
{ $f$ is onto if and only if there have been $m$ hits}
```

25. This algorithm is straightforward.

```
procedure count ones($a_1 a_2 \ldots a_n$ : bit string)
$count := 0$ {no 1's yet}
for $i := 1$ to $n$
        if $a_i = 1$ then $count := count + 1$
return $count$ { $count$ contains the number of 1's}
```

27. We start with the pseudocode for binary search given in the text and modify it. In particular, we need to compute two middle subscripts (one third of the way through the list and two thirds of the way through) and compare x with two elements in the list. Furthermore, we need special handling of the case when there are two elements left to be considered. The following pseudocode is reasonably straightforward.

```
procedure ternary search($x$ : integer, $a_1, a_2, \ldots, a_n$ : increasing integers)
$i := 1$
$j := n$
while $i < j - 1$
        $l := \lfloor (i + j)/3 \rfloor$
        $u := \lfloor 2(i + j)/3 \rfloor$
        if $x > a_u$ then $i := u + 1$
        else if $x > a_l$ then
                $i := l + 1$
                $j := u$
        else $j := l$
if $x = a_i$ then $location := i$
else if $x = a_j$ then $location := j$
else $location := 0$
return $location$
{ $location$ is the subscript of the term equal to $x$ (0 if not found)}
```

29. The following algorithm will find the first mode in the sequence. At each point in the execution of this algorithm, *modecount* is the number of occurrences of the element found to occur most often so far (which is called *mode*). Whenever a more frequently occurring element is found (the main inner loop), *modecount* and *mode* are updated.

```
procedure find a mode($a_1, a_2, \ldots, a_n$ : nondecreasing integers)
modecount := 0
$i := 1$
while $i \le n$
        value := $a_i$
        count := 1
        while $i \le n$ and $a_i$ = value
                count := count + 1
                $i := i + 1$
        if count > modecount then
                modecount := count
                mode := value
return mode
{ mode is the first value occurring most often, namely modecount times}
```

31. The following algorithm goes through the terms of the sequence one by one, and, for each term, compares it to all previous terms. If it finds a match, then it stores the subscript of that term in *location* and terminates the search. If no match is ever found, then *location* is set to 0.

```
procedure find duplicate($a_1, a_2, \ldots, a_n$ : integers)
location := 0 {no match found yet}
$i := 2$
while $i \le n$ and location = 0
        $j := 1$
        while $j < i$ and location = 0
                if $a_i = a_j$ then location := $i$
                else $j := j + 1$
        $i := i + 1$
return location
{ location is the subscript of the first value that repeats a previous value in the sequence
  and is 0 if there is no such value}
```

33. The following algorithm goes through the terms of the sequence one by one, and, for each term, checks whether it is less than the immediately preceding term. If it finds such a term, then it stores the subscript of that term in *location* and terminates the search. If no term satisfies this condition, then *location* is set to 0.

```
procedure find decrease($a_1, a_2, \ldots, a_n$ : positive integers)
location := 0 {no match found yet}
$i := 2$
while $i \le n$ and location = 0
        if $a_i < a_{i-1}$ then location := $i$
        else $i := i + 1$
return location
{ location is the subscript of the first value that is less than the immediately preceding one
  and is 0 if there is no such value}
```

35. There are four passes through the list. On the first pass, the 3 and the 1 are interchanged first, then the next two comparisons produce no interchanges, and finally the last comparison results in the interchange of the 7 and the 4. Thus after one pass the list reads $1, 3, 5, 4, 7$. During the next pass, the 5 and the 4 are interchanged, yielding $1, 3, 4, 5, 7$. There are two more passes, but no further interchanges are made, since the list is now in order.

37. We need to add a Boolean variable to indicate whether any interchanges were made during a pass. Initially this variable, which we will call *still_interchanging*, is set to **true**. If no interchanges were made, then we can quit. To do this neatly, we turn the outermost loop into a **while** loop that is executed as long as $i < n$ and *still_interchanging* is true. Thus our pseudocode is as follows.

```
procedure betterbubblesort($a_1, \ldots, a_n$)
i := 1
still_interchanging := true
while $i < n$ and still_interchanging
        still_interchanging := false
        for $j := 1$ to $n - i$
                if $a_j > a_{j+1}$ then
                        still_interchanging := true
                        interchange $a_j$ and $a_{j+1}$
        $i := i + 1$
{ $a_1, \ldots, a_n$ is in nondecreasing order}
```

39. We start with $3, 1, 5, 7, 4$. The first step inserts 1 correctly into the sorted list 3, producing $1, 3, 5, 7, 4$. Next 5 is inserted into $1, 3$, and the list still reads $1, 3, 5, 7, 4$, as it does after the 7 is inserted into $1, 3, 5$. Finally, the 4 is inserted, and we obtain the sorted list $1, 3, 4, 5, 7$. At each insertion, the element to be inserted is compared with the elements already sorted, starting from the beginning, until its correct spot is found, and then the previously sorted elements beyond that spot are each moved one position toward the back of the list.

41. We assume that when the least element is found at each stage, it is interchanged with the element in the position it wants to occupy.

a) The smallest element is 1, so it is interchanged with the 3 at the beginning of the list, yielding $1, 5, 4, 3, 2$. Next, the smallest element among the remaining elements in the list (the second through fifth positions) is 2, so it is interchanged with the 5 in position 2, yielding $1, 2, 4, 3, 5$. One more pass gives us $1, 2, 3, 4, 5$. At this point we find the fourth smallest element among the fourth and fifth positions, namely 4, and interchange it with itself, again yielding $1, 2, 3, 4, 5$. This completes the sort.

b) The process is similar to part **(a)**. We just show the status at the end of each of the four passes: $1, 4, 3, 2, 5$; $1, 2, 3, 4, 5$; $1, 2, 3, 4, 5$; $1, 2, 3, 4, 5$.

c) Again there are four passes, but all interchanges result in the list remaining as it is.

43. We carry out the linear search algorithm given as Algorithm 2 in this section, except that we replace $x \neq a_i$ by $x < a_i$, and we replace the **else** clause with **else** *location* $:= n + 1$. The cursor skips past all elements in the list less than x, the new element we are trying to insert, and ends up in the correct position for the new element.

45. We are counting just the comparisons of the numbers in the list, not any comparisons needed for the bookkeeping in the **for** loop. The second element in the list must be compared with the first and compared with itself (in other words, when $j = 2$ in Algorithm 5, i takes the values 1 and 2 before we drop out of the **while** loop). The third element must be compared with the first two and itself, since it exceeds them both. We continue in this way, until finally the n^{th} element must be compared with all the elements. So the total number of comparisons is $2 + 3 + 4 + \cdots + n$, which can be written as $(n^2 + n - 2)/2$. This is the worst case for insertion sort in terms of number of comparisons; see Example 6 in Section 3.3. On the other hand, no movements of elements are required, since each new element is already in its correct position.

47. There are two kinds of steps—the searching and the inserting. We assume the answer to Exercise 44, which is to use Algorithm 3 but replace the final check with **if** $x < a_i$ **then** *location* $:= i$ **else** *location* $:= i + 1$. So the first step is to find the location for 2 in the list 3, and we insert it in front of the 3, so the list now reads $2, 3, 4, 5, 1, 6$. This took one comparison. Next we use binary search to find the location for the 4, and we see, after comparing it to the 2 and then the 3, that it comes after the 3, so we insert it there, leaving still $2, 3, 4, 5, 1, 6$. Next we use binary search to find the location for the 5, and we see, after comparing it to the 3 and then the 4, that it comes after the 4, so we insert it there, leaving still $2, 3, 4, 5, 1, 6$. Next we use binary search to find the location for the 1, and we see, after comparing it to the 3 and then the 2 and then the 2 again, that it comes before the 2, so we insert it there, leaving $1, 2, 3, 4, 5, 6$. Finally we use binary search to find the location for the 6, and we see, after comparing it to the 3 and then the 4 and then the 5, that it comes after the 5, so we insert it there, giving the final answer $1, 2, 3, 4, 5, 6$. Note that this took 11 comparisons in all.

49. We combine the search technique of Algorithm 3, as modified in Exercises 44 and 47, with the insertion part of Algorithm 5.

```
procedure binary insertion sort($a_1, a_2, \ldots, a_n$ : real numbers with $n \geq 2$)
for $j := 2$ to $n$
        {binary search for insertion location $i$}
        $left := 1$
        $right := j - 1$
        while $left < right$
                $middle := \lfloor (left + right)/2 \rfloor$
                if $a_j > a_{middle}$ then $left := middle + 1$
                else $right := middle$
        if $a_j < a_{left}$ then $i := left$ else $i := left + 1$
        {insert $a_j$ in location $i$ by moving $a_i$ through $a_{j-1}$ toward back of list}
        $m := a_j$
        for $k := 0$ to $j - i - 1$
                $a_{j-k} := a_{j-k-1}$
        $a_i := m$
{$a_1, a_2, \ldots, a_n$ are sorted}
```

51. If the elements are in close to the correct order, then we would usually find the correct spot for the next item to be inserted near the upper end of the list of already-sorted elements. Hence the variation from Exercise 50, which starts comparing at that end, would be best.

53. In each case we use as many quarters as we can, then as many dimes to achieve the remaining amount, then as many nickels, then as many pennies.

a) The algorithm uses the maximum number of quarters, two, leaving 1 cent. It then uses the maximum number of dimes (none) and nickels (none), before using one penny.

b) two quarters, leaving 19 cents, then one dime, leaving 9 cents, then one nickel, leaving 4 cents, then four pennies

c) three quarters, leaving 1 cent, then one penny

d) two quarters, leaving 10 cents, then one dime

55. In each case we uses as many quarters as we can, then as many dimes to achieve the remaining amount, then as many pennies.

a) The algorithm uses the maximum number of quarters, two, leaving 1 cent. It then uses the maximum number of dimes (none), before using one penny. Since the answer to Exercise 53a used no nickels anyway, the greedy algorithm here certainly used the fewest coins possible.

b) The algorithm uses two quarters, leaving 19 cents, then one dime, leaving 9 cents, then nine pennies. The greedy algorithm thus uses 12 coins. Since there are no nickels available, we must either use nine pennies or else use only one quarter and four pennies, along with four dimes to reach the needed total of 69 cents. This uses only nine coins, so the greedy algorithm here did not achieve the optimum.

c) The algorithm uses three quarters, leaving 1 cent, then one penny. Since the answer to Exercise 53c used no nickels anyway, the greedy algorithm here certainly used the fewest coins possible.

d) The algorithm uses two quarters, leaving 10 cents, then one dime. Since the answer to Exercise 53c used no nickels anyway, the greedy algorithm here certainly used the fewest coins possible.

57. We first sort the talks by finishing times and get the following list: 9:00–9:45, 9:30–10:00, 9:50–10:15, 10:10–10:25, 10:00–10:30, 10:15–10:45, 10:30–10:55, 10:30–11:00, 11:00–11:15, 10:55–11:25, 10:45–11:30. We then go through the list in order and schedule every talk that is compatible with the talks already scheduled. So first we schedule the 9:00–9:45 talk, then the first one on our list that starts after that talk is finished, namely the 9:50–10:15 talk. Next comes the 10:15–10:45 talk, and then the 11:00–11:15 talk. There are no more talks that start after this one ends, so we are done, having scheduled four talks.

59. **a)** We propose the following greedy algorithm. Order the talks by starting time. Number the lecture halls 1, 2, 3, and so on. For each talk, assign it to lowest numbered lecture hall that is currently available. So, for example, using the talks from Exercise 57, we would assign the 9:00–9:45 talk to lecture hall 1, the 9:30–10:00 talk to lecture hall 2, the 9:50–10:15 talk to lecture hall 1, the 10:00–10:30 talk to lecture hall 2, the 10:10–10:25 talk to lecture hall 3, the 10:15–10:45 talk to lecture hall 1, the 10:30–10:55 talk to lecture hall 2, the 10:30–11:00 talk to lecture hall 3, the 10:45–11:30 talk to lecture hall 1, the 10:55–11:25 talk to lecture hall 2, and the 11:00–11:15 talk to lecture hall 3. Therefore three halls were sufficient.

b) This algorithm is optimal, because if it uses n lecture halls, then at the point the n^{th} hall was first assigned, it *had* to be used (otherwise a lower-numbered hall would have been assigned), which means that n talks were going on simultaneously (this talk just assigned and the $n-1$ talks currently in halls 1 through $n-1$).

61. In the algorithm presented here, the input consists, for each man, of a list of all women in his preference order, and for each woman, of a list of all men in her preference order. At the risk of being sexist, we will let the men be the suitors and the women the suitees (although obviously we could reverse these roles). The procedure needs to have data structures (lists) to keep track, for each man, of his status (rejected or not) and the list of women who have rejected him, and, for each woman, of the men currently on her proposal list.

```
procedure stable($M_1, M_2, \ldots, M_s, W_1, W_2, \ldots, W_s$, : preference lists)
for $i := 1$ to $s$
        mark man $i$ as rejected
for $i := 1$ to $s$
        set man $i$'s rejection list to be empty
for $j := 1$ to $s$
        set woman $j$'s proposal list to be empty
while rejected men remain
        for $i := 1$ to $s$
                if man $i$ is marked rejected then add $i$ to the proposal list
                for the woman $j$ who ranks highest on his preference list
                but does not appear on his rejection list, and mark $i$ as not rejected
        for $j := 1$ to $s$
                if woman $j$'s proposal list is nonempty then remove from
                $j$'s proposal list all men $i$ except the man $i_0$ who ranks highest
                on her preference list, and for each such man $i$ mark him
                as rejected and add $j$ to his rejection list
for $j := 1$ to $s$
        match $j$ with the one man on $j$'s proposal list
{This matching is stable.}
```

63. Suppose the assignment is not stable. Then there is a man m and a woman w such that m prefers w to the woman (call her w') with whom he is matched, and w prefers m to the man with whom she is matched. But

m must have proposed to w before he proposed to w', since he prefers the former. And since m did not end up matched with w, she must have rejected him. Since women reject a suitor only when they get a better proposal, and they eventually get matched with a pending suitor, the woman with whom w is matched must be better in her eyes than m, contradicting our original assumption. Therefore the matching is stable.

65. The algorithm is simply to run the two programs on their inputs concurrently and wait for one to halt. This algorithm will terminate by the conditions of the problem, and we'll have the desired answer.

SECTION 3.2 The Growth of Functions

The big-O notation is used extensively in computer science and other areas. Think of it as a crude ruler for measuring functions in terms of how fast they grow. The idea is to treat all functions that are more or less the same as one function—one mark on this ruler. Thus, for example, all linear functions are simply thought of as $O(n)$. Although technically the big-O notation gives an upper bound on the growth of a function, in practice we choose the smallest big-O estimate that applies. (This is made more rigorous with the big-Θ notation, also discussed in this section.) In essence, one finds best big-O estimates by discarding lower order terms and multiplicative constants. Furthermore, one usually chooses the simplest possible representative of the big-O (or big-Θ) class (for example, writing $O(n^2)$ rather than $O(3n^2+5)$). A related concept, used in combinatorics and applied mathematics, is the **little-*o* notation**, *dealt with in Exercises 61–69.*

1. Note that the choices of witnesses C and k are not unique.

a) Yes, since $|10| \le |x|$ for all $x > 10$. The witnesses are $C = 1$ and $k = 10$.

b) Yes, since $|3x+7| \le |4x| = 4|x|$ for all $x > 7$. The witnesses are $C = 4$ and $k = 7$.

c) No. There is no *constant* C such that $|x^2+x+1| \le C|x|$ for all sufficiently large x. To see this, suppose this inequality held for all sufficiently large positive values of x. Then we would have $x^2 \le Cx$, which would imply that $x \le C$ for *all* sufficiently large x, an obvious impossibility.

d) Yes. This follows from the fact that $\log x < x$ for all $x > 1$ (which in turn follows from the fact that $x < 2^x$, which can be formally proved by mathematical induction—see Section 5.1). Therefore $|5\log x| \le 5|x|$ for all $x > 1$. The witnesses are $C = 5$ and $k = 1$.

e) Yes. This follows from the fact that $\lfloor x \rfloor \le x$. Thus $\big|\lfloor x \rfloor\big| \le |x|$ for all $x > 0$. The witnesses are $C = 1$ and $k = 0$.

f) Yes. This follows from the fact that $\lceil x/2 \rceil \le (x/2)+1$. Thus $\big|\lceil x/2 \rceil\big| \le |(x/2)+1| \le |x|$ for all $x > 2$. The witnesses are $C = 1$ and $k = 2$.

3. We need to put some bounds on the lower order terms. If $x > 9$ then we have $x^4 + 9x^3 + 4x + 7 \le x^4 + x^4 + x^4 + x^4 = 4x^4$. Therefore $x^4 + 9x^3 + 4x + 7$ is $O(x^4)$, taking witnesses $C = 4$ and $k = 9$.

5. We use long division to rewrite this function:

$$\frac{x^2+1}{x+1} = \frac{x^2-1+2}{x+1} = \frac{x^2-1}{x+1} + \frac{2}{x+1} = x - 1 + \frac{2}{x+1}$$

Now this is certainly less than x as long as $x > 1$, so our function is $O(x)$. The witnesses are $C = 1$ and $k = 1$.

7. **a)** Since $\log x$ grows more slowly than x, $x^2 \log x$ grows more slowly than x^3, so the first term dominates. Therefore this function is $O(x^3)$ but not $O(x^n)$ for any $n < 3$. More precisely, $2x^3+x^2\log x \le 2x^3+x^3 = 3x^3$ for all x, so we have witnesses $C=3$ and $k=0$.

b) We know that $\log x$ grows so much more slowly than x that *every* power of $\log x$ grows more slowly than x (see Exercise 58). Thus the first term dominates, and the best estimate is $O(x^3)$. More precisely, $(\log x)^4 < x^3$ for all $x > 1$, so $3x^3 + (\log x)^4 \le 3x^3 + x^3 = 4x^3$ for all x, so we have witnesses $C = 4$ and $k=1$.

c) By long division, we see that $f(x) = x +$ lower order terms. Therefore this function is $O(x)$, so $n=1$. In fact, $f(x) = x + \frac{1}{x+1} \le 2x$ for all $x > 1$, so the witnesses can be taken to be $C=2$ and $k=1$.

d) Again by long division, this quotient has the form $f(x) = 1 +$ lower order terms. Therefore this function is $O(1)$. In other words, $n=0$. Since $5\log x < x^4$ for $x > 1$, we have $f(x) \le 2x^4/x^4 = 2$, so we can take as witnesses $C=2$ and $k=1$.

9. On the one hand we have $x^2+4x+17 \le x^2+x^2+x^2 = 3x^2 \le 3x^3$ for all $x > 17$, so $x^2+4x+17$ is $O(x^3)$, with witnesses $C=3$ and $k=17$. On the other hand, if x^3 were $O(x^2+4x+17)$, then we would have $x^3 \le C(x^2+4x+17) \le 3Cx^2$ for all sufficiently large x. But this says that $x \le 3C$, clearly impossible for the constant C to satisfy for all large x. Therefore x^3 is not $O(x^2+4x+17)$.

11. For the first part we have $3x^4+1 \le 4x^4 = 8|x^4/2|$ for all $x > 1$; we have witnesses $C=8$, $k=1$. For the second part we have $x^4/2 \le 3x^4 \le 1\cdot|3x^4+1|$ for all x; witnesses are $C=1$, $k=0$.

13. To show that 2^n is $O(3^n)$ it is enough to note that $2^n \le 3^n$ for all $n > 0$. In terms of witnesses we have $C=1$ and $k=0$. On the other hand, if 3^n were $O(2^n)$, then we would have $3^n \le C\cdot 2^n$ for all sufficiently large n. This is equivalent to $C \ge \left(\frac{3}{2}\right)^n$, which is clearly impossible, since $\left(\frac{3}{2}\right)^n$ grows without bound as n increases.

15. A function f is $O(1)$ if $|f(x)| \le C$ for all sufficiently large x. In other words, f is $O(1)$ if its absolute value is **bounded** for all $x > k$ (where k is some constant).

17. Let C_1, C_2, k_1, and k_2 be numbers such that $|f(x)| \le C_1|g(x)|$ for all $x > k_1$ and $|g(x)| \le C_2|h(x)|$ for all $x > k_2$. Let $C = C_1C_2$ and let k be the larger of k_1 and k_2. Then for all $x > k$ we have $|f(x)| \le C_1|g(x)| \le C_1C_2|h(x)| = C|h(x)|$, which is precisely what we needed to show.

19. Because $2^{n+1} = 2\cdot 2^n$, clearly it is $O(2^n)$ (take $C=2$). To see that 2^{2n} is not $O(2^n)$, look at the ratio: $2^{2n}/2^n = 2^n$. Because this is unbounded, there is no constant C such that $2^{2n} \le C\cdot 2^n$ for all sufficiently large n.

21. The order is $1000\log n$, $\sqrt{n}$, $n\log n$, $n^2/1000000$, 2^n, 3^n, $2n!$. That each is big-O of the next is clear.

23. Because $n\log n$ is $O(n^{3/2})$ but $n^{3/2}$ is not $O(n\log n)$, the first algorithm uses fewer operations for large n. In fact, if we solve $n\log n < n^{3/2}$, we get $n > 4$. In other words, the first algorithm uses fewer operations for all $n > 4$.

25. **a)** The significant terms here are the n^2 being multiplied by the n; thus this function is $O(n^3)$.

b) Since $\log n$ is smaller than n, the significant term in the first factor is n^2. Therefore the entire function is $O(n^5)$.

c) For the first factor we note that $2^n < n!$ for $n \ge 4$, so the significant term is $n!$. For the second factor, the significant term is n^3. Therefore this function is $O(n^3 n!)$.

27. **a)** First we note that $\log(n^2+1)$ and $\log n$ are in the same big-O class, since $\log n^2 = 2\log n$. Therefore the second term here dominates the first, and the simplest good answer would be just $O(n^2 \log n)$.
b) The first term is in the same big-O class as $O(n^2(\log n)^2)$, while the second is in a slightly smaller class, $O(n^2 \log n)$. (In each case, we can throw away the smaller order terms, since they are dominated by the terms we are keeping—this is the essence of doing big-O estimates.) Therefore the answer is $O(n^2(\log n)^2)$.
c) The only issue here is whether 2^n or n^2 is the faster-growing, and clearly it is the former. Therefore the best big-O estimate we can give is $O(n^{2^n})$.

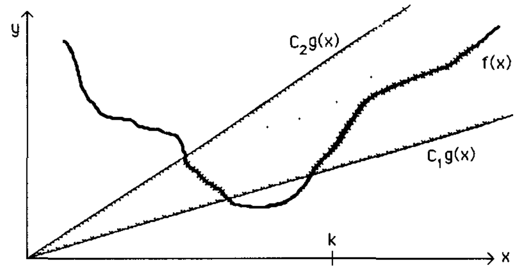
29. We can use the following rule of thumb to determine what simple big-Θ function to use: throw away all the lower order terms (those that don't grow as fast as other terms) and all constant coefficients.

a) This function is $\Theta(x)$, so it is not $\Theta(x^2)$, since x^2 grows faster than x. To be precise, x^2 is not $O(17x+11)$. For the same reason, this function is not $\Omega(x^2)$.
b) This function is $\Theta(x^2)$; we can ignore the " $+1000$" since it is a lower order term. Of course, since $f(x)$ is $\Theta(x^2)$, it is also $\Omega(x^2)$.
c) This function grows more slowly than x^2, since $\log x$ grows more slowly than x. Therefore $f(x)$ is not $\Theta(x^2)$ or $\Omega(x^2)$.
d) This function grows faster than x^2. Therefore $f(x)$ is not $\Theta(x^2)$, but it is $\Omega(x^2)$.
e) Exponential functions (with base larger than 1) grow faster than all polynomials, so this function is not $O(x^2)$ and therefore not $\Theta(x^2)$. But it is $\Omega(x^2)$.
f) For large values of x, this is quite close to x^2, since both factors are quite close to x. Certainly $\lfloor x \rfloor \cdot \lceil x \rceil$ is always between $x^2/2$ and $2x^2$, for $x > 2$. Therefore this function is $\Theta(x^2)$ and hence also $\Omega(x^2)$.

31. If $f(x)$ is $\Theta(g(x))$, then $|f(x)| \le C_2|g(x)|$ and $|g(x)| \le C_1^{-1}|f(x)|$ for all $x > k$. Thus $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$. Conversely, suppose that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$. Then (with appropriate choice of variable names) we may assume that $|f(x)| \le C_2|g(x)|$ and $|g(x)| \le C|f(x)|$ for all $x > k$. (The k here will be the larger of the two k's involved in the hypotheses.) If $C > 0$ then we can take $C_1 = C^{-1}$ to obtain the desired inequalities in "$f(x)$ is $\Theta(g(x))$." If $C \le 0$, then $g(x) = 0$ for all $x > k$, and hence by the first inequality $f(x) = 0$ for all $x > k$; thus we have $f(x) = g(x)$ for all $x > k$, and we can take $C_1 = C_2 = 1$.

33. The definition of "$f(x)$ is $\Theta(g(x))$" is that $f(x)$ is both $O(g(x))$ and $\Omega(g(x))$. That means that there are positive constants C_1, k_1, C_2, and k_2 such that $|f(x)| \le C_2|g(x)|$ for all $x > k_2$ and $|f(x)| \ge C_1|g(x)|$ for all $x > k_1$. That is practically the same as the statement in this exercise. We need only note that we can take k to be the larger of k_1 and k_2 if we want to prove the "only if" direction, and we can take $k_1 = k_2 = k$ if we want to prove the "if" direction.

35. In the following picture, the wavy line is the graph of the function f. For simplicity we assume that the graph of g is a straight line through the origin. Then the graphs of $C_1 g$ and $C_2 g$ are also straight lines through the origin, as drawn here. The fact that $f(x)$ is $\Theta(g(x))$ is shown by the fact that for $x > k$ the graph of f is confined to the shaded wedge-shaped space between these latter two lines (see Exercise 33). (We assume that $g(x)$ is positive for positive x, so that $|g(x)|$ is the same as $g(x)$.)



37. Looking at the definition tells us that if $f(x)$ is $\Theta(1)$ then $|f(x)|$ has to be bounded between two positive constants. In other words, $f(x)$ can't get too large (either positive or negative), and it can't get too close to 0.

39. We are given that $|f(x)| \leq C|g(x)|$ for all $x > k$. Hence $|f^n(x)| = |f(x)|^n \leq C^n|g^n(x)|$ for all $x > k$, so $f^n(x)$ is $O(g^n(x))$ (take the constants in the definition to be C^n and k).

41. Since the functions are given to be increasing and unbounded, we may assume that they both take on values greater than 1 for all sufficiently large x. The hypothesis can then be written as $f(x) \leq Cg(x)$ for all $x > k$. If we take the logarithm of both sides, then we obtain $\log f(x) \leq \log C + \log g(x)$. Finally, this latter expression is less than $2 \log g(x)$ for large enough x, since $\log g(x)$ is growing without bound. Note that the "converse" implication does not hold; even though $\log(3^n)$ is $O(\log(2^n))$ (both functions are linear), it is not true that 3^n is $O(2^n)$.

43. By definition there are positive constants C_1, C_1', C_2, C_2', k_1, k_1', k_2, and k_2' such that $f_1(x) \geq C_1|g(x)|$ for all $x > k_1$, $f_1(x) \leq C_1'|g(x)|$ for all $x > k_1'$, $f_2(x) \geq C_2|g(x)|$ for all $x > k_2$, and $f_2(x) \leq C_2'|g(x)|$ for all $x > k_2'$. We are able to omit the absolute value signs on the $f(x)$'s since we are told that they are positive; we are also told here that the $g(x)$'s are positive, but we do not need that. Adding the first and third inequalities we obtain $f_1(x) + f_2(x) \geq (C_1 + C_2)|g(x)|$ for all $x > \max(k_1, k_2)$; and similarly with the second and fourth inequalities we know $f_1(x) + f_2(x) \leq (C_1' + C_2')|g(x)|$ for all $x > \max(k_1', k_2')$. Thus $f_1(x) + f_2(x)$ meets the definition of being $\Theta(g(x))$.

If the f's can take on negative values, then this is no longer true. For example, let $f_1(x) = x^2 + x$, let $f_2(x) = -x^2 + x$, and let $g(x) = x^2$. Then each $f_i(x)$ is $\Theta(g(x))$, but the sum is $2x$, which is not $\Theta(g(x))$.

45. This is not true. It is similar to Exercise 43, and essentially the same counterexample suffices. Let $f_1(x) = x^2 + 2x$, $f_2(x) = x^2 + x$, and $g(x) = x^2$. Then clearly $f_1(x)$ and $f_2(x)$ are both $\Theta(g(x))$, but $(f_1 - f_2)(x) = x$ is not.

47. The key here is that if a function is to be big-O of another, then the appropriate inequality has to hold for *all* large inputs. Suppose we let $f(x) = x^2$ for even x and x for odd x. Similarly, we let $g(x) = x^2$ for odd x and x for even x. Then clearly neither inequality $|f(x)| \leq C|g(x)|$ nor $|g(x)| \leq C|f(x)|$ holds for all x, since for even x the first function is much bigger than the second, while for odd x the second is much bigger than the first.

49. We are given that there are positive constants C_1, C_1', C_2, C_2', k_1, k_1', k_2, and k_2' such that $|f_1(x)| \geq C_1|g_1(x)|$ for all $x > k_1$, $|f_1(x)| \leq C_1'|g_1(x)|$ for all $x > k_1'$, $|f_2(x)| \geq C_2|g_2(x)|$ for all $x > k_2$, and $|f_2(x)| \leq C_2'|g_2(x)|$ for all $x > k_2'$. Since f_2 and g_2 never take on the value 0, we can rewrite the last two of these inequalities as $|1/f_2(x)| \leq (1/C_2)|1/g_2(x)|$ and $|1/f_2(x)| \geq (1/C_2')|1/g_2(x)|$. Now we multiply the first inequality and the rewritten fourth inequality to obtain $|f_1(x)/f_2(x)| \geq (C_1/C_2')|g_1(x)/g_2(x)|$ for all $x > \max(k_1, k_2')$. Working with the other two inequalities gives us $|f_1(x)/f_2(x)| \leq (C_1'/C_2)|g_1(x)/g_2(x)|$ for all $x > \max(k_1', k_2)$. Together these tell us that f_1/f_2 is big-Θ of g_1/g_2.

51. We just make the analogous change in the definition of big-Θ that was made in the definition of big-O: there exist positive constants C_1, C_2, k_1, k_2, k_1', k_2' such that $|f(x,y)| \le C_1|g(x,y)|$ for all $x > k_1$ and $y > k_2$, and $|f(x,y)| \ge C_2|g(x,y)|$ for all $x > k_1'$ and $y > k_2'$.

53. For all values of x and y greater than 1, each term of the expression inside parentheses is less than x^2y, so the entire expression inside parentheses is less than $3x^2y$. Therefore our function is less than $27x^6y^3$ for all $x > 1$ and $y > 1$. By definition this shows that it is big-O of x^6y^3. Specifically, we take $C = 27$ and $k_1 = k_2 = 1$ in the definition.

55. For all positive values of x and y, we know that $\lfloor xy \rfloor \le xy$ by definition (since the floor function value cannot exceed the argument). Thus $\lfloor xy \rfloor$ is $O(xy)$ from the definition, taking $C = 1$ and $k_1 = k_2 = 0$. In fact, $\lfloor xy \rfloor$ is also $\Omega(xy)$ (and therefore $\Theta(xy)$); this is easy to see since $\lfloor xy \rfloor \ge (x-1)(y-1) \ge (\frac{1}{2}x)(\frac{1}{2}y) = \frac{1}{4}xy$ for all x and y greater than 2.

57. Because $d < c$, clearly $n^d < n^c$ for all $n \ge 2$; therefore n^d is $O(n^c)$. To see that n^d is not $O(n^c)$, look at the ratio: $n^d/n^c = n^{d-c}$. Because this is unbounded (the exponent is positive), there is no constant C such that $n^d \le Cn^c$ for all sufficiently large n.

59. If f and g are positive-valued functions for which we can compute the limit of the ratio $f(x)/g(x)$ as $x \to \infty$, then the value of that limit will tell us about the asymptotic behavior of those functions relative to each other. Indeed, if the limit is a finite number C, then for large enough x, we have $f(x) < (C+1)g(x)$, so $f(n)$ is $O(g(n))$. If the limit is ∞, then clearly $f(n)$ is not $O(g(n))$.

In the case at hand, $\lim_{x\to\infty} x^d/b^x = 0$ (apply L'Hôpital's rule $\lceil d \rceil$ times), and similarly $\lim_{x\to\infty} b^x/x^d = \infty$. The desired conclusions follow.

61. All that we need to do is determine whether the ratio of the two functions approaches 0 as x approaches infinity.

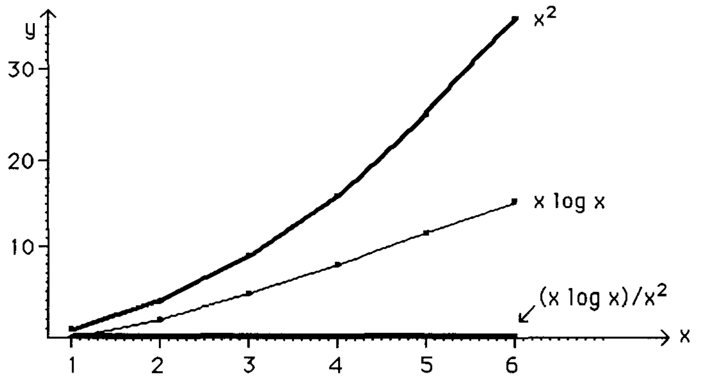
a) $\displaystyle\lim_{x\to\infty} \frac{x^2}{x^3} = \lim_{x\to\infty} \frac{1}{x} = 0$

b) $\displaystyle\lim_{x\to\infty} \frac{x\log x}{x^2} = \lim_{x\to\infty} \frac{\log x}{x} = \lim_{x\to\infty} \frac{1}{x\ln 2} = 0$ (using L'Hôpital's rule for the second equality)

c) $\displaystyle\lim_{x\to\infty} \frac{x^2}{2^x} = \lim_{x\to\infty} \frac{2x}{2^x\ln 2} = \lim_{x\to\infty} \frac{2}{2^x(\ln 2)^2} = 0$ (with two applications of L'Hôpital's rule)

d) $\displaystyle\lim_{x\to\infty} \frac{x^2+x+1}{x^2} = \lim_{x\to\infty} (1 + \frac{1}{x} + \frac{1}{x^2}) = 1 \ne 0$

63. The picture shows the graph of $y = x^2$ increasing quite rapidly and $y = x\log x$ increasing less rapidly. The ratio is hard to see on the picture; it rises to about $y = 0.53$ at about $x = 2.7$ and then slowly decreases toward 0. The limit as $x \to \infty$ of $(x\log x)/x^2$ is in fact 0.



65. No. As one example, take $f(x) = x^{-2}$ and $g(x) = x^{-1}$. Then $f(x)$ is $o(g(x))$, since $\lim_{x\to\infty} x^{-2}/x^{-1} = \lim_{x\to\infty} 1/x = 0$. On the other hand $\lim_{x\to\infty}(2^{x^{-2}}/2^{x^{-1}}) = \lim_{x\to\infty} 2^{x^{-2}-x^{-1}} = 2^0 = 1 \neq 0$.

67. **a)** Since the limit of $f(x)/g(x)$ is 0 (as $x \to \infty$), so too is the limit of $|f(x)|/|g(x)|$. In particular, for x large enough, this ratio is certainly less than 1. In other words $|f(x)| \leq |g(x)|$ for sufficiently large x, which meets the definition of "$f(x)$ is $O(g(x))$."

b) We can simply let $f(x) = g(x)$ be any function with positive values. Then the limit of their ratio is 1, not 0, so $f(x)$ is not $o(g(x))$, but certainly $f(x)$ is $O(g(x))$.

69. This follows immediately from Exercise 67a (whereby we can conclude that $f_2(x)$ is $O(g(x))$) and Corollary 1 to Theorem 2.

71. What we want to show is equivalent to the statement that $\log(n^n)$ is at most a constant times $\log(n!)$, which in turn is equivalent to the statement that n^n is at most a constant power of $n!$ (because of the fact that $C \log A = \log(A^C)$—see Appendix 2). We will show that in fact $n^n \leq (n!)^2$ for all $n > 1$. To do this, let us write $(n!)^2$ as $(n\cdot 1)\cdot((n-1)\cdot 2)\cdot((n-2)\cdot 3)\cdots(2\cdot(n-1))\cdot(1\cdot n)$. Now clearly each product pair $(i+1)\cdot(n-i)$ is at least as big as n (indeed, the ones near the middle are significantly bigger than n). Therefore the entire product is at least as big as n^n, as desired.

73. For $n = 5$ we compute that $\log 5! \approx 6.9$ and $(5 \log 5)/4 \approx 2.9$, so the inequality holds (it actually holds for all $n > 1$). Therefore we can assume that $n \geq 6$. Since $n!$ is the product of all the integers from n down to 1, we certainly have $n! > n(n-1)(n-2)\cdots\lceil n/2\rceil$ (since at least the term 2 is missing). Note that there are more than $n/2$ terms in this product, and each term is at least as big as $n/2$. Therefore the product is greater than $(n/2)^{(n/2)}$. Taking the log of both sides of the inequality, we have

$$\log n! > \log\left(\frac{n}{2}\right)^{n/2} = \frac{n}{2}\log\frac{n}{2} = \frac{n}{2}(\log n - 1) > (n \log n)/4\,,$$

since $n > 4$ implies $\log n - 1 > (\log n)/2$.

75. In each case we need to evaluate the limit of $f(x)/g(x)$ as $x \to \infty$. If it equals 1, then f and g are asymptotic; otherwise (including the case in which the limit does not exist) they are not. Most of these are straightforward applications of algebra, elementary notions about limits, or L'Hôpital's rule.

a) $f(x) = \log(x^2+1) \geq \log(x^2) = 2\log x$. Therefore $f(x)/g(x) \geq 2$ for all x. Thus the limit is not 1 (in fact, of course, it's 2), so f and g are not asymptotic.

b) By the algebraic rules for exponents, $f(x)/g(x) = 2^{-4} = 1/16$. Therefore the limit of the ratio is $1/16$, not 1, so f and g are not asymptotic.

c) $\lim_{x\to\infty} \dfrac{2^{2^x}}{2^{x^2}} = \lim_{x\to\infty} 2^{2^x - x^2}$. As x gets large, the exponent grows without bound, so this limit is ∞. Thus f and g are not asymptotic.

d) $\lim_{x\to\infty} \dfrac{2^{x^2+x+1}}{2^{x^2+2x}} = \lim_{x\to\infty} 2^{1-x}$. As x gets large, the exponent grows in the negative direction without bound, so this limit is 0. Thus f and g are not asymptotic.

SECTION 3.3 Complexity of Algorithms

Some of these exercises involve analysis of algorithms, as was done in the examples in this section. These are a matter of carefully counting the operations of interest, usually in the worst case. Some of the others are algebra exercises that display the results of the analysis in real terms—the number of years of computer time, for example, required to solve a large problem. **Horner's method** *for evaluating a polynomial, given in Exercise 14, is a nice trick to know. It is extremely handy for polynomial evaluation on a pocket calculator (especially if the calculator is so cheap that it does not use the usual precedence rules).*

1. The statement $t := t + ij$ is executed just 12 times, so the number of operations is $O(1)$. (Specifically, there are just 24 additions or multiplications.)

3. The nesting of the loops implies that the assignment statement is executed roughly $n^2/2$ times. Therefore the number of operations is $O(n^2)$.

5. Assuming that the algorithm given to find the smallest element of a list is identical to Algorithm 1 in Section 3.1, except that the inequality is reversed (and the name *max* replaced by the name *min*), the analysis will be identical to the analysis given in Example 1 in the current section. In particular, there will be $2n - 1$ comparisons needed, counting the bookkeeping for the loop.

7. The linear search would find this element after at most 9 comparisons (4 to determine that we have not yet finished with the **while** loop, 4 more to determine if we have located the desired element yet, and 1 to set the value of *location*). Binary search, according to Example 3, will take $2\log 32 + 2 = 2\cdot 5 + 2 = 12$ comparisons. Since $9 < 12$, the linear search will be faster, in terms of comparisons.

9. The algorithm simply scans the bits one at a time. Thus clearly $O(n)$ comparisons are required (perhaps one for bookkeeping and one for looking at the i^{th} bit, for each i from 1 to n).

11. **a)** We can express the suggested algorithm in pseudocode as follows. Notice that the Boolean variable *disjoint* is set to **true** at the beginning of the comparison of sets S_i and S_j, and becomes **false** if and when we find an element common to those two sets. If *disjoint* is never set to **false**, then we have found a disjoint pair, and *answer* is set to **true**. This process is repeated for each pair of sets (controlled by the outer two loops).

```
procedure disjointpair($S_1, S_2, \ldots, S_n$ : subsets of $\{1, 2, \ldots, n\}$)
answer := false
for $i := 1$ to $n$
        for $j := i + 1$ to $n$
                disjoint := true
                for $k := 1$ to $n$
                        if $k \in S_i$ and $k \in S_j$ then disjoint := false
                if disjoint then answer := true
return answer
```

b) The three nested loops imply that the elementhood test needs to be applied $O(n^3)$ times.

13. **a)** Here we have $n = 2$, $a_0 = 1$, $a_1 = 1$, $a_2 = 3$, and $c = 2$. Initially, we set *power* equal to 1 and y equal to 1. The first time through the **for** loop (with $i = 1$), *power* becomes 2 and so y becomes $1 + 1\cdot 2 = 3$. The second and final time through the loop, *power* becomes $2\cdot 2 = 4$ and y becomes $3 + 3\cdot 4 = 15$. Thus the value of the polynomial at $x = 2$ is 15.

b) Each pass through the loop requires two multiplications and one addition. Therefore there are a total of $2n$ multiplications and n additions in all.

15. This is an exercise in algebra, numerical analysis (for some of the parts), and using a calculator. Since each bit operation requires 10^{-9} seconds, we want to know for what value of n there will be at most 10^9 bit operations required. Thus we need to set the expression equal to 10^9, solve for n, and round down if necessary.

a) Solving $\log n = 10^9$, we get (recalling that "log" means logarithm base 2) $n = 2^{10^9}$. By taking $\log_{10}$ of both sides, we find that this number is approximately equal to $10^{300,000,000}$. Obviously we do not want to write out the answer explicitly!

b) Clearly $n = 10^9$.

c) Solving $n \log n = 10^9$ is not trivial. There is no good formula for solving such **transcendental** equations. An algorithm that works well with a calculator is to rewrite the equation as $n = 10^9/\log n$, enter a random starting value, say $n = 2$, and repeatedly calculate a new value of n. Thus we would obtain, in succession, $n = 10^9/\log 2 = 10^9$, $n = 10^9/\log(10^9) \approx 33{,}447{,}777.3$, $n = 10^9/\log(33{,}447{,}777.3) \approx 40{,}007{,}350.14$, $n = 10^9/\log(40{,}007{,}350.14) \approx 39{,}598{,}061.08$, and so on. After a few more iterations, the numbers stabilize at approximately $39{,}620{,}077.73$, so the answer is $39{,}620{,}077$.

d) Solving $n^2 = 10^9$ gives $n = 10^{4.5}$, which is $31{,}622$ when rounded down.

e) Solving $2^n = 10^9$ gives $n = \log(10^9) \approx 29.9$. Rounding down gives the answer, 29.

f) The quickest way to find the largest value of n such that $n! \le 10^9$ is simply to try a few values of n. We find that $12! \approx 4.8 \times 10^8$ while $13! \approx 6.2 \times 10^9$, so the answer is 12.

17. If each bit operation takes 10^{-12} second, then we can carry out 10^{12} bit operations per second, and therefore $60 \cdot 10^{12}$ bit operations per minute. Therefore in each case we want to solve the equation $f(n) = 60 \cdot 10^{12}$ for n and round down to an integer. Obviously a calculator will come in handy here.

a) If $\log\log n = 60 \cdot 10^{12}$, then $n = 2^{2^{60\ 10^{12}}}$, which is an unfathomably huge number.

b) If $\log n = 60 \cdot 10^{12}$, then $n = 2^{60\ 10^{12}}$, which is still an unfathomably huge number.

c) If $(\log n)^2 = 60 \cdot 10^{12}$, then $\log n = \sqrt{60} \cdot 10^6$, so $n = \lfloor 2^{\sqrt{60}\cdot 10^6} \rfloor$, which is still an extremely large number (it has over 2 million digits).

d) If $1{,}000{,}000n = 60 \cdot 10^{12}$, then $n = 60 \cdot 10^6 = 60{,}000{,}000$.

e) If $n^2 = 60 \cdot 10^{12}$, then $n = \lfloor \sqrt{60} \cdot 10^6 \rfloor = 7{,}745{,}966$.

f) If $2^n = 60 \cdot 10^{12}$, then $n = \lfloor \log(60 \cdot 10^{12}) \rfloor = \lfloor \log 60 + 12 \log 10 \rfloor = 45$. (Remember, we are taking log to the base 2.)

g) If $2^{n^2} = 60 \cdot 10^{12}$, then $n = \lfloor \sqrt{\log(60 \cdot 10^{12})} \rfloor = \lfloor \sqrt{\log 60 + 12 \log 10} \rfloor = 6$. (Remember, we are taking log to the base 2.)

19. In each case, we just multiply the number of seconds per operation by the number of operations (namely 2^{50}). To convert seconds to minutes, we divide by 60; to convert minutes to hours, we divide by 60 again. To convert hours to days, we divide by 24; to convert days to years, we divide by $365\frac{1}{4}$.

a) $2^{50} \times 10^{-6} = 1{,}125{,}899{,}907$ seconds ≈ 36 years

b) $2^{50} \times 10^{-9} = 1{,}125{,}899.907$ seconds ≈ 13 days

c) $2^{50} \times 10^{-12} = 1{,}125.899907$ seconds ≈ 19 minutes

21. In each case we want to compare the function evaluated at $n + 1$ to the function evaluated at n. The most desirable form of the comparison (subtraction or division) will vary.

a) Notice that $\log(n+1) - \log n = \log \frac{n+1}{n}$. If n is at all large, the fraction in this expression is approximately equal to 1, and therefore the expression is approximately equal to 0. In other words, hardly any extra time is required. For example, in going from $n = 10$ to $n = 11$, the number of extra milliseconds is $\log 11/10 \approx 0.14$.

b) $100(n+1) - 100n = 100$. One hundred extra milliseconds are required, independent of n.

c) Because $(n+1)^2 - n^2 = 2n + 1$, we conclude that $2n + 1$ additional milliseconds are needed for the larger problem.

d) Because $(n+1)^3 - n^3 = 3n^2 + 3n + 1$, we conclude that $3n^2 + 3n + 1$ additional milliseconds are needed for the larger problem.

e) This time it makes more sense to use a ratio comparison, rather than a difference comparison. Because $2^{n+1}/2^n = 2$, we see that twice as much time is required for the larger problem.

f) Because $2^{(n+1)^2}/2^{n^2} = 2^{2n+1}$, we see that 2^{2n+1} times as many milliseconds are required for the larger problem.

g) Because $(n+1)!/n! = n+1$, we see that $n+1$ times as many milliseconds are required for the larger problem. For example, a problem with $n = 10$ takes ten times as long as a problem with $n = 9$.

23. If the element is not in the list, then $2n+2$ comparisons are needed: two for each pass through the loop, one more to get out of the loop, and one more for the statement just after the loop. If the element is in the list, say as the i^{th} element, then we need to enter the loop i times, each time costing two comparisons, and use one comparison for the final assignment of *location*. Thus $2i+1$ comparisons are needed. We need to average the numbers $2i+1$ (for i from 1 to n), to find the average number of comparisons needed for a successful search. This is $(3+5+\cdots+(2n+1))/n = (n+(2+4+\cdots+2n))/n = (n+2(1+2+\cdots+n))/n = (n+2(n(n+1)/2))/n = n+2$. Finally, we average the $2n+2$ comparisons for the unsuccessful search with this average $n+2$ comparisons for a successful search to obtain a grand average of $(2n+2+n+2)/2 = (3n+4)/2$ comparisons.

25. We will count comparisons of elements in the list to x. (This ignores comparisons of subscripts, but since we are only interested in a big-O analysis, no harm is done.) Furthermore, we will assume that the number of elements in the list is a power of 3, say $n = 3^k$. Just as in the case of binary search, we need to determine the maximum number of times the **while** loop is iterated. Each pass through the loop cuts the number of elements still being considered (those whose subscripts are from i to j) by a factor of 3. Therefore after k iterations, the active portion of the list will have length 1; that is, we will have $i = j$. The loop terminates at this point. Now each iteration of the loop requires two comparisons in the worst case (one with a_u and one with a_l). Two more comparisons are needed at the end. Therefore the number of comparisons is $2k+2$, which is $O(k)$. But $k = \log_3 n$, which is $O(\log n)$ since logarithms to different bases differ only by multiplicative constants, so the time complexity of this algorithm (in all cases, not just the worst case) is $O(\log n)$.

27. The algorithm we gave for finding a mode essentially just goes through the list once, doing a little bookkeeping at each step. In particular, between any two successive executions of the statement $i := i + 1$ there are at most about six operations (such as comparing *count* with *modecount*, or reinitializing *value*). Therefore at most about $6n$ steps are done in all, so the time complexity in all cases is $O(n)$.

29. The worst case is that in which we do not find any term equal to some previous term. In that case, we need to go through all the terms a_2 through a_n, and for each of those, we need to go through all the terms occurring prior to that term. Thus the inner loop of our algorithm is executed once for $i = 2$ (namely for $j = 1$), twice for $i = 3$ (namely for $j = 1$ and $j = 2$), three times for $i = 4$, and so on, up to $n-1$ times for $i = n$. Thus the number of comparisons that need to be made in the inner loop is $1+2+3+\cdots+(n-1)$. As was mentioned in this section (and will be shown in Section 5.1), that sum is $(n-1)(n-1+1)/2$, which is clearly $O(n^2)$ but no better. Bookkeeping details do not increase this estimate.

31. We needed to go through the sequence only once, making one comparison of terms (and two bookkeeping comparisons) until we found the desired term (or had exhausted the list). Thus this algorithm's time complexity is clearly $O(n)$.

33. We had to read the string simultaneously from the front and the back and compare characters to make sure they were equal. Thus we will need at least $\lfloor n/2 \rfloor$ comparisons in the worst case, which is $O(n)$.

35. Since we are doing binary search to find the correct location of the j^{th} element among the first $j-1$ elements, which are already sorted, we need $O(\log n)$ comparisons for each element. Therefore we use $O(n \log n)$ comparisons in all. The cost of swapping of items to make room for the insertions is $O(n^2)$, however (the originally first item may need to move $n-1$ places, the second item $n-2$ places, and so on).

37. There are 2^n subsets of the n talks. For each one, we need to compare the times to see whether they overlap. To compare a pair of talks for overlap takes $O(1)$ time (there is no overlap if and only if the first talk's start time exceeds the second talk's end time, or vice versa, so just two comparisons are needed). There are $O(n^2)$ pairs of talks to compare in a typical subset. (We also need to compare the size of this subset with the size of the best subset found so far, but the time for that is negligible.) Therefore the total complexity is $O(n^2 2^n)$.

39. **a)** The linear search algorithm uses about n comparisons for a list of length n, and $2n$ comparisons for a list of length $2n$. Therefore the number of comparisons, like the size of the list, doubles.

b) The binary search algorithm uses about $\log n$ comparisons for a list of length n, and $\log(2n) = \log 2 + \log n = 1 + \log n$ comparisons for a list of length $2n$. Therefore the number of comparisons increases by about 1.

41. We just use Algorithm 1, where **A** and **B** are now $n \times n$ upper triangular matrices, by replacing m by n in line 1, and having q iterate only from i to j, rather than from 1 to k.

43. See the solution to Exercise 42. Looking at the nested loops, we see that the number of multiplications is given by the following expression: $[1+2+\cdots+n]+[1+2+\cdots(n-1)]+\cdots+[1]$. To simplify this, we need the fact (from Table 2 in Section 2.4) that the sum of the first k positive integers is $k(k+1)/2$, as well as the result from Exercise 15 in Section 5.1, which says that the sum $1\cdot 2+2\cdot 3+3\cdot 4+\cdots+k(k+1)$ equals $k(k+1)(k+1)/3$. Doing the algebra, we obtain $n(n+1)(n+2)/6$ as the total number of multiplications.

45. There are five different ways to perform this multiplication:

$$(\mathbf{AB})(\mathbf{CD}),\quad ((\mathbf{AB})\mathbf{C})\mathbf{D},\quad \mathbf{A}(\mathbf{B}(\mathbf{CD})),\quad (\mathbf{A}(\mathbf{BC}))\mathbf{D},\quad \mathbf{A}((\mathbf{BC})\mathbf{D}).$$

We can use the result stated in the preamble to find the numbers of multiplications needed in these five cases. For example, in the first case we need $30\cdot 10\cdot 40 = 12{,}000$ multiplications to compute the 30×40 matrix **AB**, $40\cdot 50\cdot 30 = 60{,}000$ multiplications to compute the 40×30 matrix **CD**, and then $30\cdot 40\cdot 30 = 36{,}000$ multiplications to multiply these two matrices together to obtain the final answer. This gives a total of $12{,}000 + 60{,}000 + 36{,}000 = 108{,}000$ multiplications. Similar calculations for the other four cases yield $30\cdot 10\cdot 40 + 30\cdot 40\cdot 50 + 30\cdot 50\cdot 30 = 117{,}00$, $40\cdot 50\cdot 30 + 10\cdot 40\cdot 30 + 30\cdot 10\cdot 30 = 81{,}000$, $10\cdot 40\cdot 50 + 30\cdot 10\cdot 50 + 30\cdot 50\cdot 30 = 80{,}000$, and $10\cdot 40\cdot 50 + 10\cdot 50\cdot 30 + 30\cdot 10\cdot 30 = 44{,}000$, respectively. The winner is therefore $\mathbf{A}((\mathbf{BC})\mathbf{D})$, requiring 44,000 multiplications. Note that the worst arrangement requires 117,00 multiplications; it will take more than twice as long.

GUIDE TO REVIEW QUESTIONS FOR CHAPTER 3

1. **a)** See p. 191.

b) in English, in a computer language, in pseudocode

c) An algorithm is more of an abstract idea—a method in theory that will solve a problem. A computer program is the implementation of that idea into a specific syntactically correct set of instructions that a real computer can use to solve the problem. It is rather like the difference between a dollar (a certain amount of money, capable in theory of purchasing a certain quantity of goods and services) and a dollar bill.

2. **a)** See first three lines of text following Algorithm 1 on p. 193. **b)** See Algorithm 1 in Section 3.1.
c) See Example 1 in Section 3.3.

3. **a)** See p. 205. **b)** $n^2 + 18n + 107 \leq n^3 + n^3 + n^3 = 3n^3$ for all $n > 107$.

c) n^3 is not less than a constant times $n^2 + 18n + 107$, since their ratio exceeds $n^3/(3n^2) = n/3$ for all $n > 107$.

4. $(\log n)^3$, $\sqrt{n}$, $100n + 101$, $n^3/1000000$, $2^n n^2$, 3^n, $n!$

5. **a)** For the sum, take the largest term; for the product multiply the factors together.
b) $g(n) = n^{n-2}2^n = (2n)^n/n^2$

6. **a)** the largest, average, and smallest number of comparisons used by the algorithm before it stops, among all lists of n integers
b) all are $n - 1$

7. **a)** See pp. 194–196. **b)** See p. 220.

c) No—it depends on the lists involved. (However, the worst case complexity for binary search is always better than that for linear search for lists of any given size except for very short lists.)

8. **a)** See pp. 196–197.

b) On the first pass, the 5 bubbles down to the end, producing 2 4 1 3 5. On the next pass, the 4 bubbles down to the end, producing 2 1 3 4 5. On the next pass, the 1 and the 2 are swapped. No further changes are made on the fourth pass.

c) $O(n^2)$; see Example 5 in Section 3.3

9. **a)** See pp. 197–198.

b) On the first pass, the 5 is inserted into its correct position relative to the 2, producing 2 5 1 4 3. On the next pass, the 1 is inserted into its correct position relative to 2 5, producing 1 2 5 4 3. On the next pass, the 4 is inserted into its correct position relative to 1 2 5, producing 1 2 4 5 3. On the final pass, the 3 is inserted, producing the sorted list.

c) $O(n^2)$; see Example 6 in Section 3.

10. **a)** See p. 198. **b)** See Example 6 and Theorem 1 in Section 3.1.

c) See Exercise 55b in Section 3.1.

11. See p. 226 for "tractable." A solvable problem is simply one that can be solved by an algorithm. The halting problem is proved on pp. 201–202 to be unsolvable.

SUPPLEMENTARY EXERCISES FOR CHAPTER 3

1. **a)** This algorithm will be identical to the algorithm *first largest* for Exercise 17 of Section 3.1, except that we want to change the value of *location* each time we find another element in the list that is equal to the current value of *max*. Therefore we simply change the strict less than ($<$) in the comparison $max < a_i$ to a less than or equal to ($\leq$), rendering the fifth line of that procedure "**if** $max \leq a_i$ **then**."

b) The number of comparisons used by this algorithm can be computed as follows. There are $n-1$ passes through the **for** loop, each one requiring a comparison of *max* with a_i. In addition, n comparisons are needed for bookkeeping for the loop (comparison of i with n, as i assumes the values $2, 3, \ldots, n+1$). Therefore $2n-1$ comparisons are needed altogether, which is $O(n)$.

3. **a)** We will try to write an algorithm sophisticated enough to avoid unnecessary checking. The answer—**true** or **false**—will be placed in a variable called *answer*.

```
procedure zeros($a_1 a_2 \ldots a_n$ : bit string)
$i := 1$
$answer :=$ false {no pair of zeros found yet}
while $i < n$ and $\neg answer$
        if $a_i = 1$ then $i := i + 1$
        else if $a_{i+1} = 1$ then $i := i + 2$
        else $answer :=$ true
return $answer$
{ $answer$ was set to true if and only if there were a pair of consecutive zeros}
```

b) The number of comparisons depends on whether a pair of 0's is found and also depends on the pattern of increments of the looping variable i. Without getting into the intricate details of exactly which is the worst case, we note that at worst there are approximately n passes through the loop, each requiring one comparison of a_i with 1 (there may be two comparisons on some passes, but then there will be fewer passes). In addition, n bookkeeping comparisons of i with n are needed (we are ignoring the testing of the logical variable *answer*). Thus a total of approximately $2n$ comparisons are used, which is $O(n)$.

5. **a)** and **b)**. We have a variable *min* to keep track of the minimum as well as a variable *max* to keep track of the maximum.

```
procedure smallest and largest($a_1, a_2, \ldots, a_n$ : integers)
$min := a_1$
$max := a_1$
for $i := 2$ to $n$
        if $a_i < min$ then $min := a_i$
        if $a_i > max$ then $max := a_i$
{ $min$ is the smallest integer among the input, and $max$ is the largest}
```

c) There are two comparisons for each iteration of the loop, and there are $n-1$ iterations, so there are $2n-2$ comparisons in all.

7. We think of ourselves as observers as some algorithm for solving this problem is executed. We do not care what the algorithm's strategy is, but we view it along the following lines, in effect taking notes as to what is happening and what *we* know as it proceeds. Before any comparisons are done, there is a possibility that each element could be the maximum and a possibility that it could be the minimum. This means that there are $2n$ different possibilities, and $2n-2$ of them have to be eliminated through comparisons of elements, since we need to find the unique maximum and the unique minimum. We classify comparisons of two elements as "nonvirgin" or "virgin," depending on whether or not both elements being compared have been in any previous comparison. A virgin comparison, between two elements that have not yet been involved in any comparison, eliminates the possibility that the larger one is the minimum and that the smaller one is the maximum; thus each virgin comparison eliminates two possibilities, but it clearly cannot do more. A nonvirgin comparison, one involving at least one element that has been compared before, must be between two elements that are still in the running to be the maximum or two elements that are still in the running to be the minimum, and at least one of these elements must *not* be in the running for the other category. For example, we might be comparing x and y, where all we know is that x has been eliminated as the minimum. If we find that $x > y$ in this case, then only one possibility has been ruled out—we now know that y is not the maximum. Thus in the worst case, a nonvirgin comparison eliminates only one possibility. (The cases of other nonvirgin comparisons are similar.) Now there are at most $\lfloor n/2 \rfloor$ comparisons of elements that have never been compared before, each removing two possibilities; they remove $2\lfloor n/2 \rfloor$ possibilities altogether. Therefore we need $2n - 2 - 2\lfloor n/2 \rfloor$ more comparisons that, as we have argued, can remove only one possibility each, in order to find the answers in the worst case, since $2n-2$ possibilities have to be eliminated. This gives us a total of $2n-2-2\lfloor n/2 \rfloor + \lfloor n/2 \rfloor$ comparisons in all. But $2n-2-2\lfloor n/2 \rfloor+\lfloor n/2 \rfloor = 2n-2-\lfloor n/2 \rfloor = 2n-2+\lceil -n/2 \rceil = \lceil 2n-n/2 \rceil -2 = \lceil 3n/2 \rceil -2$, as desired.

Note that this gives us a lower bound on the number of comparisons used in an algorithm to find the minimum and the maximum. On the other hand, Exercise 6 gave us an upper bound of the same size. Thus the algorithm in Exercise 6 is the most efficient algorithm possible for solving this problem.

9. The following uses the brute-force method. The sum of two terms of the sequence $a_1, a_2, \ldots, a_n$ is given by $a_i + a_j$, where $i < j$. If we loop through all pairs of such sums and check for equality, we can output two pairs whenever we find equal sums. A pseudocode implementation for this process follows. Because of the nested loops, the complexity is $O(n^4)$.

```
procedure equal sums($a_1, a_2, \ldots, a_n$)
for $i := 1$ to $n$
    for $j := i+1$ to $n$ {since we want $i < j$}
        for $k := 1$ to $n$
            for $l := k+1$ to $n$ {since we want $k < l$}
                if $a_i + a_j = a_k + a_l$ and $(i,j) \neq (k,l)$ then output these pairs
```

11. After the comparison and possible exchange of adjacent elements on the first pass, from front to back, the list is $3, 1, 4, 5, 2, 6$, where the 6 is known to be in its correct position. After the comparison and possible exchange of adjacent elements on the second pass, from back to front, the list is $1, 3, 2, 4, 5, 6$, where the 6 and the 1 are known to be in their correct positions. After the next pass, the result is $1, 2, 3, 4, 5, 6$. One more pass finds nothing to exchange, and the algorithm terminates.

13. There are possibly as many as $n-1$ passes through the list (or parts of it—it depends on the particular way it is implemented), and each pass uses $O(n)$ comparisons. Thus there are $O(n^2)$ comparisons in all.

15. Since $\log n < n$, we have $(n \log n + n^2)^3 \leq (n^2 + n^2)^3 \leq (2n^2)^3 = 8n^6$ for all $n > 0$. This proves that $(n \log n + n^2)^3$ is $O(n^6)$, with witnesses $C = 8$ and $k = 0$.

17. In the first factor the x^2 term dominates the other term, since $(\log x)^3$ is $O(x)$. Therefore by Theorem 2 in Section 3.2, this term is $O(x^2)$. Similarly, in the second factor, the 2^x term dominates. Thus by Theorem 3 of Section 3.2, the product is $O(x^2 2^x)$.

19. Let us look at the ratio

$$\frac{n!}{2^n} = \frac{n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1}{2 \cdot 2 \cdot 2 \cdots 2 \cdot 2 \cdot 2} = \frac{n}{2} \cdot \frac{n-1}{2} \cdot \frac{n-2}{2} \cdots \frac{3}{2} \cdot \frac{2}{2} \cdot \frac{1}{2}.$$

Each of the fractions in the final expression is greater than 1 except the last one, so the entire expression is at least $(n/2)/2 = n/4$. Since $n!/2^n$ increases without bound as n increases, $n!$ cannot be bounded by a constant times 2^n, which tells us that $n!$ is not $O(2^n)$.

21. Each of these functions is of the same order as n^2, so all pairs are of the same order. One way to see this is to think about "throwing away lower order terms." Notice in particular that $\log 2^n = n$, and that the n^3 terms cancel in the fourth function.

23. We know that exponential functions grow faster than polynomial functions, so such a value of n must exist. If we take logs of both sides, then we obtain the equivalent inequality $2^{100}\log(n) < n$, or $n/\log n > 2^{100}$. This tells us that n has to be very large. In fact, $n = 2^{100}$ is not large enough, because for that value, the left-hand side is smaller by a factor of 100. If we go a little bigger, however, then the inequality will be satisfied. For example, if $n = 2^{107}$, then we have

$$\frac{n}{\log n} = \frac{2^{107}}{107} = \frac{128}{107}\cdot 2^{100} > 2^{100}\,.$$

25. Clearly n^n is growing the fastest, so it belongs at the end of our list. Next, 1.0001^n is an exponential function, so it is bigger (in big-O terms) than any of the others not yet considered. Next, note that $\sqrt{\log_2 n} < \log_2 n$ for large n, so taking 2 to the power of these two expressions shows that $2^{\sqrt{\log_2 n}}$ is $O(n)$. Therefore, the competition for third and fourth places from the right in our list comes down to $n^{1.0001}$ and $n(\log n)^{1001}$, and since all positive powers of n grow faster than any power of $\log n$ (see Exercise 58 in Section 3.2), the former wins third place. Finally, to see that $(\log n)^2$ is the slowest-growing of these functions, compare it to $2^{\sqrt{\log_2 n}}$ by taking the base-2 log of both and noting that $2\log\log n$ grows more slowly than $\sqrt{\log n}$. So the required list is $(\log n)^2$, $2^{\sqrt{\log_2 n}}$, $n(\log n)^{1001}$, $n^{1.0001}$, 1.0001^n, n^n.

27. We want the functions to play leap-frog, with first one much bigger, then the other. Something like this will do the trick: Let $f(n) = n^{2\lfloor n/2\rfloor+1}$. Thus, $f(1) = 1^1$, $f(2) = 2^3$, $f(3) = 3^3$, $f(4) = 4^5$, $f(5) = 5^5$, $f(6) = 6^7$, and so on. Similarly, let $g(n) = n^{2\lceil n/2\rceil}$. Thus, $g(1) = 1^2$, $g(2) = 2^2$, $g(3) = 3^4$, $g(4) = 4^4$, $g(5) = 5^6$, $g(6) = 6^6$, and so on. Then for even n we have $f(n)/g(n) = n$, and for odd n we have $g(n)/f(n) = n$. Because both ratios are unbounded, neither function is big-O of the other.

29. **a)** We loop through all pairs (i, j) with $i < j$ and check whether $a_i + a_j = a_k$ for some k (by looping through all values of k).

```
procedure brute($a_1, a_2, \ldots, a_n$ : integers)
for $i := 1$ to $n - 1$
    for $j := i + 1$ to $n$
        for $k := 1$ to $n$
            if $a_i + a_j = a_k$ then return true else return false
```

b) Because of the loop within a loop within a loop, clearly the time complexity is $O(n^3)$.

31. There are six possible matchings. We need to determine which ones are stable. The matching $[(m_1, w_1), (m_2, w_2), (m_3, w_3)]$ is not stable, because m_3 and w_2 prefer each other over their current mate. The matching $[(m_1, w_1), (m_2, w_3), (m_3, w_2)]$ is stable; although m_1 would prefer w_3, she ranks him lowest, and men m_2 and m_3 got their first picks. The matching $[(m_1, w_2), (m_2, w_1), (m_3, w_3)]$ is stable; although m_1 would prefer w_1 or w_3, they both rank him lowest, and m_2 and m_3 will not break their respective matches because their potential girlfriends got their first choices. The matching $[(m_1, w_2), (m_2, w_3), (m_3, w_1)]$ is not stable, because m_3 and w_3 prefer each other over their current mate. The matching $[(m_1, w_3), (m_2, w_1), (m_3, w_2)]$ is not stable, because m_2 and w_3 will run off together. Finally, $[(m_1, w_3), (m_2, w_2), (m_3, w_1)]$ is not stable, again because m_3 and w_3 will run off together. To summarize, the stable matchings are $[(m_1, w_1), (m_2, w_3), (m_3, w_2)]$ and $[(m_1, w_2), (m_2, w_1), (m_3, w_3)]$. Reading off this list, we see that the valid partners are as follows: for m_1: w_1 and w_2; for m_2: w_1 and w_3; for m_3: w_2 and w_3; for w_1: m_1 and m_2; for w_2: m_1 and m_3; and for w_3: m_2 and m_3.

33. We just take the definition of male optimal and female pessimal and interchange the sexes. Thus, a matching in which each woman is assigned her valid partner ranking highest on her preference list is called female optimal, and a matching in which each man is assigned his valid partner ranking lowest on his preference list is called male pessimal.

35. a) We just rewrite the preamble to Exercise 60 in Section 3.1 with the appropriate modifications: Suppose we have s men $m_1, m_2, \ldots, m_s$ and t women $w_1, w_2, \ldots, w_t$. We wish to match each person with a person of the opposite gender, to the extent possible (namely, $\min(s,t)$ marriages). Furthermore suppose that each person ranks, in order of preference, with no ties, the people of the opposite gender. We say that a matching of people of opposite genders to form couples is **stable** if we cannot find a man m and a woman w who are not assigned to each other such that m prefers w over his assigned partner (or lack of a partner) and w prefers m to her assigned partner (or lack of a partner). (We assume that each person prefers any mate to being unmatched.)

b) The simplest way to do this is to create $|s-t|$ fictitious people (men or women, whichever is in shorter supply) so that the number of men and the number of women become the same, and to put these fictitious people at the bottom of the preference lists of the people of opposite gender. The preference lists for the fictitious people can be specified arbitrarily. We then just run the algorithm as before.

c) Because being unmatched is least desirable, it follows immediately from the proof that the original algorithm produces a stable matching (Exercise 63 in Section 3.1) that the modified algorithm produces a stable matching.

37. If we schedule the jobs in the order shown, then Job 3 will finish at time 20, Job 1 will finish at time 45, Job 4 will finish at time 50, Job 2 will finish at time 65, and Job 5 will finish at time 75. Comparing these to the deadlines, we see that only Job 2 is late, and it is late by 5 minutes (we'll call the time units minutes for convenience). A similar calculation for the other order shows that Job 1 is 10 minutes late, and Job 2 is 15 minutes late, so the maximum lateness is 15.

39. Consider the first situation in Exercise 37. We saw there that it is possible to achieve a maximum lateness of 5. If we schedule the jobs in order of increasing time required, then Job 1 will be scheduled last and finish at time 75. This will give it a lateness of 25, which gives a maximum lateness worse than the previous schedule. (Using the algorithm from Exercise 40 gives a maximum lateness of only 5.)

41. a) We want to maximize the mass of the contents of the knapsack. So we can try each possible subset of the available items to see which ones can fit into the knapsack (i.e., have total mass not exceeding W) and thereby find a subset having the maximum mass. In detail, then, examine each of the 2^n subsets S of $\{1, 2, \ldots, n\}$, and for each subset compute the total mass of the corresponding items (the sum of w_j for all $j \in S$). Keep track of the subset giving the largest such sum that is less than or equal to W, and return that subset as the output of the algorithm. Note that this is quite inefficient, because the number of subsets to examine is exponential. In fact, there is no known efficient algorithm for solving this problem.

b) Essentially we find the solution here by inspection. Among the 32 subsets of items we try (from the empty set to the set of all five items), we stumble upon the subset consisting of the food pack and the portable stove, which has total mass equal to the capacity, 18.

43. a) The makespan is always at least as large as the load on the processor assigned to do the lengthiest job, which must obviously be at least $\max_{j=1,2,\ldots,n} t_j$. Therefore the minimum makespan L^* satisfies this inequality.

b) The total amount of time the processors need to spend working on the jobs (the total load) is $\sum_{j=1}^{n} t_j$. Therefore the average load per processor is $\frac{1}{p}\sum_{j=1}^{n} t_j$. The maximum load cannot be any smaller than the average, so the makespan is always at least this large. It follows that the minimum makespan L^* is at least this large, as we were asked to prove.

45. The algorithm will assign job 1 to processor 1 (one of the processors with smallest load at that point, namely 0), assign job 2 to processor 2 (for the same reason), and assign job 3 to processor 3. The loads are 3, 5, and 4 at this point. Then job 4 gets assigned to processor 1 because it has the lightest load; the loads are now 10, 5, 4. Finally, job 5 gets assigned to processor 3, giving it load 12, so the makespan is 12. Notice that we can do better by assigning job 1 and job 4 to processor 1 (load 10), job 2 and job 3 to processor 2 (load 9), and job 5 to processor 3 (load 8), for a makespan of 10. Notice, too, that this latter solution is best possible, because to achieve a makespan of 9, all three processors would have to have a load of 9, and this clearly cannot be achieved.

WRITING PROJECTS FOR CHAPTER 3

Books and articles indicated by bracketed symbols below are listed near the end of this manual. You should also read the general comments and advice you will find there about researching and writing these essays.

1. Algorithms are as much a subject of computer science as they are a subject of mathematics. Thus sources on the history of computer science (or perhaps even verbose introductory programming or comprehensive computer science textbooks) might be a good place to look. See also [Ha2].

2. The original is [Ba2]. Good history of mathematics books would be a place to follow up.

3. See [Me1], or do a Web search.

4. Knuth's volume 3 [Kn] is the bible for sorting algorithms.

5. Look up Moore's law.

6. A modern book on algorithm design, such as [CoLe] is the obvious best source.

7. The Association for Computing Machinery (ACM) is the organization that bestows this prestigious award.

8. This is a vast and extremely important subject in computer science today. One basic book on the subject is [Gi]; see also an essay in [De2].

9. See the comments for Writing Project 8, above. One basic algorithm to think about doing in parallel is sorting. Imagine a group of school-children asked to arrange themselves in a row by height, and any child can determine whether another child is shorter or taller than him/herself.

10. Many websites discuss NP-complete problems, including `www.claymath.org/millennium/P_vs_NP`, which discusses the $1,000,000 prize the Clay Institute has offered for finding an efficient algorithm for such problems or showing that none exists. The classic book reference is [GaJo].

11. See [GaJo] or a website about NP-complete problems.