SECTION 4.4 Recursive Algorithms

2. First, we use the recursive step to write $6! = 6 \cdot 5!$. We then use the recursive step repeatedly to write $5! = 5 \cdot 4!$, $4! = 4 \cdot 3!$, $3! = 3 \cdot 2!$, $2! = 2 \cdot 1!$, and $1! = 1 \cdot 0!$. Inserting the value of $0! = 1$, and working back through the steps, we see that $1! = 1 \cdot 1 = 1$, $2! = 2 \cdot 1! = 2 \cdot 1 = 2$, $3! = 3 \cdot 2! = 3 \cdot 2 = 6$, $4! = 4 \cdot 3! = 4 \cdot 6 = 24$, $5! = 5 \cdot 4! = 5 \cdot 24 = 120$, and $6! = 6 \cdot 5! = 6 \cdot 120 = 720$.

4. First, because $n = 10$ is even, we use the **else if** clause to see that

$$mpower(2, 10, 7) = mpower(2, 5, 7)^2 \textbf{ mod } 7.$$

We next use the **else** clause to see that

$$mpower(2, 5, 7) = (mpower(2, 2, 7)^2 \textbf{ mod } 7 \cdot 2 \textbf{ mod } 7) \textbf{ mod } 7.$$

Then we use the **else if** clause again to see that

$$mpower(2, 2, 7) = mpower(2, 1, 7)^2 \textbf{ mod } 7.$$

Using the **else** clause again, we have

$$mpower(2, 1, 7) = (mpower(2, 0, 7)^2 \textbf{ mod } 7 \cdot 2 \textbf{ mod } 7) \textbf{ mod } 7.$$

Finally, using the **if** clause, we see that $mpower(2, 0, 7) = 1$. Now we work backward: $mpower(2, 1, 7) = (1^2 \textbf{ mod } 7 \cdot 2 \textbf{ mod } 7) \textbf{ mod } 7 = 2$, $mpower(2, 2, 7) = 2^2 \textbf{ mod } 7 = 4$, $mpower(2, 5, 7) = (4^2 \textbf{ mod } 7 \cdot 2 \textbf{ mod } 7) \textbf{ mod } 7 = 4$, and finally $mpower(2, 10, 7) = 4^2 \textbf{ mod } 7 = 2$. We conclude that $2^{10} \textbf{ mod } 7 = 2$.

6. With this input, the algorithm uses the **else** clause to find that $\gcd(12, 17) = \gcd(17 \bmod 12, 12) = \gcd(5, 12)$. It uses this clause again to find that $\gcd(5, 12) = \gcd(12 \bmod 5, 5) = \gcd(2, 5)$, then to get $\gcd(2, 5) = \gcd(5 \bmod 2, 2) = \gcd(1, 2)$, and once more to get $\gcd(1, 2) = \gcd(2 \bmod 1, 1) = \gcd(0, 1)$. Finally, to find $\gcd(0, 1)$ it uses the first step with $a = 0$ to find that $\gcd(0, 1) = 1$. Consequently, the algorithm finds that $\gcd(12, 17) = 1$.

8. The sum of the first n positive integers is the sum of the first $n - 1$ positive integers plus n. This trivial observation leads to the recursive algorithm shown here.

procedure *sum of first*(n : positive integer)
if $n = 1$ **then** *sum of first*(n) := 1
else *sum of first*(n) := *sum of first*($n - 1$) + n

10. The recursive algorithm works by comparing the last element with the maximum of all but the last. We assume that the input is given as a sequence.

procedure $max(a_1, a_2, \ldots, a_n$: integers)
if $n = 1$ **then** $max(a_1, a_2, \ldots, a_n) := a_1$
else
begin
    $m := max(a_1, a_2, \ldots, a_{n-1})$
    if $m > a_n$ **then** $max(a_1, a_2, \ldots, a_n) := m$
    else $max(a_1, a_2, \ldots, a_n) := a_n$
end

12. This is the inefficient method.

procedure *power*(x, n, m : positive integers)
if $n = 1$ **then** *power*(x, n, m) := x **mod** m
else *power*(x, n, m) := $(x \cdot power(x, n - 1, m))$ **mod** m

14. This is actually quite subtle. The recursive algorithm will need to keep track not only of what the mode actually is, but also of how often the mode appears. We will describe this algorithm in words, rather than in pseudocode. The input is a list $a_1, a_2, \ldots, a_n$ of integers. Call this list L. If $n = 1$ (the base case), then the output is that the mode is a_1 and it appears 1 time. For the recursive case ($n > 1$), form a new list L' by deleting from L the term a_n and all terms in L equal to a_n. Let k be the number of terms deleted. If $k = n$ (in other words, if L' is the empty list), then the output is that the mode is a_n and it appears n times. Otherwise, apply the algorithm recursively to L', obtaining a mode m, which appears t times. Now if $t \geq k$, then the output is that the mode is m and it appears t times; otherwise the output is that the mode is a_n and it appears k times.

16. The sum of the first one positive integer is 1, and that is the answer the recursive algorithm gives when $n = 1$, so the basis step is correct. Now assume that the algorithm works correctly for $n = k$. If $n = k + 1$, then the **else** clause of the algorithm is executed, and $k + 1$ is added to the (assumed correct) sum of the first k positive integers. Thus the algorithm correctly finds the sum of the first $k + 1$ positive integers.

18. We use mathematical induction on n. If $n = 0$, we know that $0! = 1$ by definition, so the **if** clause handles this basis step correctly. Now fix $k \geq 0$ and assume the inductive hypothesis—that the algorithm correctly computes $k!$. Consider what happens with input $k + 1$. Since $k + 1 > 0$, the **else** clause is executed, and the answer is whatever the algorithm gives as output for input k, which by inductive hypothesis is $k!$, multiplied by $k + 1$. But by definition, $k! \cdot (k + 1) = (k + 1)!$, so the algorithm works correctly on input $k + 1$.

20. Our induction is on the value of y. When $y = 0$, the product $xy = 0$, and the algorithm correctly returns that value. Assume that the algorithm works correctly for smaller values of y, and consider its performance on y. If y is even (and necessarily at least 2), then the algorithm computes 2 times the product of x and $y/2$. Since it does the product correctly (by the inductive hypothesis), this equals $2(x \cdot y/2)$, which equals xy by the commutativity and associativity of multiplication. Similarly, when y is odd, the algorithm computes 2 times the product of x and $(y-1)/2$ and then adds x. Since it does the product correctly (by the inductive hypothesis), this equals $2(x \cdot (y-1)/2) + x$, which equals $xy - x + x = xy$, again by the rules of algebra.

22. The largest in a list of one integer is that one integer, and that is the answer the recursive algorithm gives when $n = 1$, so the basis step is correct. Now assume that the algorithm works correctly for $n = k$. If $n = k+1$, then the **else** clause of the algorithm is executed. First, by the inductive hypothesis, the algorithm correctly sets m to be the largest among the first k integers in the list. Next it returns as the answer either that value or the $(k+1)$st element, whichever is larger. This is clearly the largest element in the entire list. Thus the algorithm correctly finds the maximum of a given list of integers.

24. We use the hint.

procedure *twopower*(n : positive integer, a : real number)
if $n = 1$ **then** *twopower*$(n, a) := a^2$
else *twopower*$(n, a) :=$ *twopower*$(n-1, a)^2$

26. We use the idea in Exercise 24, together with the fact that $a^n = (a^{n/2})^2$ if n is even, and $a^n = a \cdot (a^{(n-1)/2})^2$ if n is odd, to obtain the following recursive algorithm. In essence we are using the binary expansion of n implicitly.

procedure *fastpower*(n : positive integer, a : real number)
if $n = 1$ **then** *fastpower*$(n, a) := a$
else if n is even **then** *fastpower*$(n, a) :=$ *fastpower*$(n/2, a)^2$
else *fastpower*$(n, a) := a \cdot$ *fastpower*$((n-1)/2, a)^2$

28. To compute f_7, Algorithm 7 requires $f_8 - 1 = 20$ additions, and Algorithm 8 requires $7 - 1 = 6$ additions.

30. This is essentially just Algorithm 8, with a different operation and different initial conditions.

```
procedure iterative(n : nonnegative integer)
if n = 0 then y := 1
else
begin
        x := 1
        y := 2
        for i := 1 to n - 1
        begin
                z := x · y
                x := y
                y := z
        end
end
{y is the n^th term of the sequence}
```

32. This is very similar to the recursive procedure for computing the Fibonacci numbers. Note that we can combine the three base cases (stopping rules) into one.

procedure *sequence*(n : nonnegative integer)
if $n < 3$ **then** *sequence*(n) := $n + 1$
else *sequence*(n) := *sequence*($n - 1$) + *sequence*($n - 2$) + *sequence*($n - 3$)

34. The iterative algorithm is much more efficient here. If we compute with the recursive algorithm, we end up computing the small values (early terms in the sequence) over and over and over again (try it for $n = 5$).

36. We obtain the answer by computing $P(m, m)$, where P is the following procedure, which we obtain simply by copying the recursive definition from Exercise 47 in Section 4.3 into an algorithm.

procedure $P(m, n$: positive integers)
if $m = 1$ **then** $P(m, n) := 1$
else if $n = 1$ **then** $P(m, n) := 1$
else if $m < n$ **then** $P(m, n) := P(m, m)$
else if $m = n$ **then** $P(m, n) := 1 + P(m, m - 1)$
else $P(m, n) := P(m, n - 1) + P(m - n, n)$

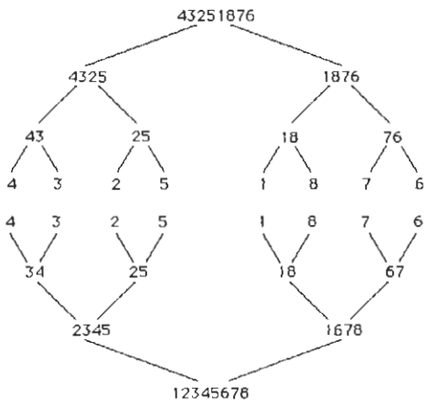
38. The following algorithm practically writes itself.

procedure *power*(w : bit string, i : nonnegative integer)
if $i = 0$ **then** *power*(w, i) := λ
else *power*(w, i) := w concatenated with *power*($w, i - 1$)

40. If $i = 0$, then by definition w^i is no copies of w, so it is correct to output the empty string. Inductively, if the algorithm correctly returns the i^{th} power of w, then it correctly returns the $(i + 1)^{\text{st}}$ power of w by concatenating one more copy of w.

42. If $n = 3$, then the polygon is already triangulated. Otherwise, by Lemma 1 in Section 4.2, the polygon has a diagonal; draw it. This diagonal splits the polygon into two polygons, each of which has fewer than n vertices. Recursively apply this algorithm to triangulate each of these polygons. The result is a triangulation of the original polygon.

44. The procedure is the same as that given in the solution to Example 9. We will show the tree and inverted tree that indicate how the sequence is taken apart and put back together.



46. From the analysis given before the statement of Lemma 1, it follows that the number of comparisons is $m + n - r$, where the lists have m and n elements, respectively, and r is the number of elements remaining in one list at the point the other list is exhausted. In this exercise $m = n = 5$, so the answer is always $10 - r$.

a) The answer is $10 - 1 = 9$, since the second list has only 1 element when the first list has been emptied.

b) The answer is $10 - 5 = 5$, since the second list has 5 elements when the first list has been emptied.

c) The answer is $10 - 2 = 8$, since the second list has 2 elements when the first list has been emptied.

48. In each case we need to show that a certain number of comparisons is necessary in the worst case, and then we need to give an algorithm that does the merging with this many comparisons.

a) There are 5 possible outcomes (the element of the first list can be greater than 0, 1, 2, 3, or 4 elements of the second list). Therefore by decision tree theory (see Section 10.2), at least $\lceil \log 5 \rceil = 3$ comparisons are needed. We can achieve this with a binary search: first compare the element of the first list to the second element of the second, and then at most two comparisons are needed to find the correct place for this element.

b) Algorithm 10 merges the lists with 5 comparisons. We must show that 5 are needed in the worst case. Naively applying decision tree theory does not help, since $\lceil \log 15 \rceil = 4$ (there are $C(5+2-1,2) = 15$ ways to choose the places among the second list for the elements of the first list to go). Instead, suppose that the lists are a_1, a_2 and b_1, b_2, b_3, b_4, in order. Then without loss of generality assume that the first comparison is a_1 against b_i. If $i \geq 2$ and $a_1 < b_i$, then there are at least 9 outcomes still possible, requiring $\lceil \log 9 \rceil = 4$ more comparisons. If $i = 1$ and $a_1 > b_1$, then there are 10 outcomes, again requiring 4 more comparisons.

c) There are $C(5+3-1,3) = 35$ outcomes, so at least $\lceil \log 35 \rceil = 6$ comparisons are needed. On the other hand Algorithm 10 uses only 6 comparisons.

d) There are $C(5+4-1,4) = 70$ outcomes, so at least $\lceil \log 70 \rceil = 7$ comparisons are needed. On the other hand Algorithm 10 uses only 7 comparisons.

50. On the first pass, we separate the list into two lists, the first being all the elements less than 3 (namely 1 and 2), and the second being all the elements greater than 3, namely $5, 7, 8, 9, 4, 6$ (in that order). As soon as each of these two lists is sorted (recursively) by quick sort, we are done. We show the entire process in the following sequence of list. The numbers in parentheses are the numbers that are correctly placed by the algorithm on the current level of recursion, and the brackets are those elements that were correctly placed previously. Five levels of recursion are required. 12(3)578946, (1)2[3]4(5)7896, [1](2)[3](4)[5]6(7)89, [1][2][3][4][5](6)[7](8)9, [1][2][3][4][5][6][7][8](9)

52. In practice, this algorithm is coded differently from what we show here, requiring more comparisons but being more efficient because the data structures are simpler (and the sorting is done in place). We denote the list $a_1, a_2, \ldots, a_n$ by a, with similar notations for the other lists. Also, rather than putting a_1 at the end of the first sublist, we put it between the two sublists and do not have to deal with it in either sublist.

procedure *quick*$(a_1, a_2, \ldots, a_n)$
$b :=$ the empty list
$c :=$ the empty list
$temp := a_1$
for $i := 2$ **to** n
    if $a_i < a_1$ **then** adjoin a_i to the end of list b
    else adjoin a_i to the end of list c
{notation: $m =$ length(b) and $k =$ length(c)}
if $m \neq 0$ **then** *quick*$(b_1, b_2, \ldots, b_m)$
if $k \neq 0$ **then** *quick*$(c_1, c_2, \ldots, c_k)$
{now put the sorted lists back into a}
for $i := 1$ **to** m
    $a_i := b_i$
$a_{m+1} := temp$
for $i := 1$ **to** k
    $a_{m+i+1} := c_i$
{the list a is now sorted}

54. In the best case, the initial split will require 3 comparisons and result in sublists of length 1 and 2 still to be sorted. These require 0 and 1 comparisons, respectively, and the list has been sorted. Therefore the answer is $3 + 0 + 1 = 4$.