Section 4.4 Recreational Algorithms

(a) If the function \( f(X) \) is defined as a sum of \( n \) terms, where each term is of the form \( a_iX^{b_i} \), then the number of operations required is \( O(n) \).

(b) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(c) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(d) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(e) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(f) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(g) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(h) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(i) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(j) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(k) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(l) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(m) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(n) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(o) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(p) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(q) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(r) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(s) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(t) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(u) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(v) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(w) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(x) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(y) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).

(z) The number of operations required to compute \( f(X) \) for a given \( X \) is \( O(n) \).
Chapter 4  Induction and Recursion

6. With this last, the algorithm uses the else clause to find that \( \text{gcd}(12, 17) = \text{gcd}(17 \mod 12, 12) = \text{gcd}(5, 12) \).

8. The sum of the first \( n \) positive integers is the sum of the first \( n \) positive integers plus \( n \). This trivial observation leads to the recursive algorithm shown here.

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

12. This is the inefficient method:

14. This is actually quite subtle. The recursive algorithm will need to keep track not only of what the new value \( s \) is but also of how often the code appears. We will describe this algorithm in words, rather than in pseudocode. The input is \( n \in \{a_1, a_2, \ldots, a_n, h\} \) of integers. Call this list \( L \). If \( n = 1 \) (the base case), the result is \( s \). Proceed as follows:

16. The sum of the first \( n \) positive integers is 1, and that is the answer the recursive algorithm gives when \( n = 1 \).

18. We use mathematical induction on \( n \). If \( n = 0 \), we know that \( 0! = 1 \) by definition, so the if clause handles this base step correctly. Now assume that the algorithm works correctly for \( n = k \) and \( n = k + 1 \). Since \( k \geq 0 \), the else clause is executed, and the computer \( k! \). Consider what happens with input \( k + 1 \). Since \( k \geq 0 \), the else clause is executed, and the // the answer is 0, which by inductive hypothesis is \( k! \), incorrect by \( k + 1 \). But by definition, \( k! \cdot (k + 1) = (k + 1)! \), so the algorithm works correctly on input \( k + 1 \).

Section 4.4  Recursive Algorithms

20. Our induction is on the value of \( y \). When \( y = 0 \), the product \( x \cdot 0 = 0 \), 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 \( 2 \cdot (x \cdot (y/2)) \) with the product correctly by the inductive hypothesis, which equals \( 2 \cdot (x \cdot (y/2)) \), which equals \( x \cdot (y-1)/2 \) to find that \( \text{gcd}(1, 8) = 1 \). Consequently, the algorithm finds that \( \text{gcd}(3, 17) = 1 \).

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 base step is correct. Now assume that the algorithm works correctly for \( n = k \). If \( n = k + 1 \), then the value of the algorithm is 1, sorted. First, by the inductive hypothesis, the algorithm correctly sorts the list of \( n \) positive integers. The result it returns is the answer either that value or the \((k+1)\)th 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:

26. We use this idea in Exercise 25.6, together with the fact that \( a^n = (a^{n/2})^2 \) if \( n \) is even, and \( a^n = a \cdot (a^{n-1})^2 \) if \( n \) is odd, to obtain the following recursive algorithm. In essence we are using the binary expansion of \( a \)

28. To compute \( \sqrt{a} \), Algorithm 7 requires \( n^2 - 2n + 1 \) additions, and Algorithm 8 requires \( 7 - 1 = 6 \) additions.

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

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.
44. The iterative algorithm is much more efficient here. If we compute with the recursive algorithm, we end up computing the same thing (very few times) in the sequence over and over and over again (try it for $n = 6$).

36. We obtain the answer by computing $P_0(n, n)$, where $P(n, m)$ is the following procedure, which we obtain simply by counting the recursive definition in Exercise 27 in Section 4.2 into an algorithm.

```plaintext
procedure $P(n, m)$:
  if $m > n$ then $P(n, m) = 1$
  else if $n < m$ then $P(n, m) = P(n, m - 1)$
  else $P(n, m) = P(n, n) + P(n, m - 1)$

38. The following algorithm practically writes itself.

```plaintext
procedure $power(x, i)$:
  if $i = 0$ then $power(x, i) = 1$
  else $power(x, i) = x$ concatenated with $power(x, i - 1)$
```

40. If $i = 0$, then by definition $x^i$ is a copy of $x$, so it is correct to output the empty string. Inductively, if the algorithm correctly returns the $i$th power of $x_i$ then it correctly returns the $(i + 1)^{th}$ power of $x_i$ by concatenating one more copy of $x_i$.

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 vertices than the original polygon. 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 6. 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 removed from both lists.

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

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

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

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 3 possible outcomes (the element of the first list can be greater than 0, 1, 2, or 3 elements of the second list). Therefore in decision tree theory (see Section 10.2), at least $\log_7 3 = 3$ 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 list, and then at most two comparisons are needed to find the correct place for this element.

b) Algorithm 10 merges the lists with $3$ comparisons. We must show that $3$ are needed in the worst case. Notice applying decision tree theory does not help, since $\log_7 3 > 2$, so there are at least 2 outcomes possible requiring $\log_7 3 = 4$ comparisons. If $k = 1$ or $a_1 > b_1$, then there are at least 3 outcomes, again requiring 4 more comparisons.

c) There are $7 + 3 + 1 = 11$ outcomes, so at least $\log_7 11 = 4$ comparisons are needed. On the other hand Algorithm 10 uses only 4 comparisons.

d) There are $7 + 4 + 1 = 12$ outcomes, so at least $\log_7 12 = 4$ 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, 9, 11, 13, 15 (in this 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 lists. The numbers in parentheses are the numbers that are correctly placed by the algorithm on the current level of recursion, and the brackets are these elements that were correctly placed previously. Five levels of recursion are required: [12][578][40][0].

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 $a_i$, with similar notations for the other lists. Also, rather than putting $a_1$ at the end of the first list, we put it between the two sublists and do not have to deal with it in other sublists.

```plaintext
procedure $split(a_1, a_2, \ldots, a_n)$:
  $L$ := the empty list
  $C$ := the empty list
  for $i := 0$ to $n - 1$
    if $a_i < a_1$ then $a_i$ to the end of list $L$
    else $a_i$ to the end of list $C$
    (notation: $m = \text{length}(L)$ and $n = \text{length}(C)$)

    if $m > 0$ then $\text{split}(a_{m+1}, \ldots, a_n)$(i)
    if $k \neq 0$ then $\text{split}(a_1, \ldots, a_k)$(ii)

    for $i := 1$ to $k$
      $C[i]$ := $a_i$
    end for

    for $i := 1$ to $k$
      $L[i]$ := $a_i$
    end for

    (the list is now sorted)
```