SECTION 4.4 Recursive Algorithms

Recursive algorithms are important theoretical entities, but they often cause a lot of grief on first encounter. Sometimes it is helpful to "play computer" very carefully to see how a recursive algorithm works. Ironically, however, it is good to avoid doing that after you get the idea. Instead, convince yourself that if the recursive algorithm handles the base case correctly, and handles the recursive step correctly (gives the correct answer assuming that the correct answer was obtained on the recursive call), then the algorithm works. Let the computer worry about actually "recursing all the way down to the base case"!

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

3. First, because $n = 11$ is odd, we use the **else** clause to see that

$$mpower(3, 11, 5) = (mpower(3, 5, 5)^2 \textbf{ mod } 5 \cdot 3 \textbf{ mod } 5) \textbf{ mod } 5 .$$

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

$$mpower(3, 5, 5) = (mpower(3, 2, 5)^2 \textbf{ mod } 5 \cdot 3 \textbf{ mod } 5) \textbf{ mod } 5 .$$

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

$$mpower(3, 2, 5) = mpower(3, 1, 5)^2 \textbf{ mod } 5 .$$

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

$$mpower(3, 1, 5) = (mpower(3, 0, 5)^2 \textbf{ mod } 5 \cdot 3 \textbf{ mod } 5) \textbf{ mod } 5 .$$

Finally, using the **if** clause, we see that $mpower(3, 0, 5) = 1$. Now we work backward: $mpower(3, 1, 5) = (1^2 \textbf{ mod } 5 \cdot 3 \textbf{ mod } 5) \textbf{ mod } 5 = 3$, $mpower(3, 2, 5) = 3^2 \textbf{ mod } 5 = 4$, $mpower(3, 5, 5) = (4^2 \textbf{ mod } 5 \cdot 3 \textbf{ mod } 5) \textbf{ mod } 5 = 3$, and finally that $mpower(3, 11, 5) = (3^2 \textbf{ mod } 5 \cdot 3 \textbf{ mod } 5) \textbf{ mod } 5 = 2$. We conclude that $3^{11} \textbf{ mod } 5 = 2$.

5. With this input, the algorithm uses the **else** clause to find that $\gcd(8, 13) = \gcd(13 \textbf{ mod } 8, 8) = \gcd(5, 8)$. It uses this clause again to find that $\gcd(5, 8) = \gcd(8 \textbf{ mod } 5, 5) = \gcd(3, 5)$, then to get $\gcd(3, 5) = \gcd(5 \textbf{ mod } 3, 3) = \gcd(2, 3)$, then $\gcd(2, 3) = \gcd(3 \textbf{ mod } 2, 2) = \gcd(1, 2)$, and once more to get $\gcd(1, 2) = \gcd(2 \textbf{ mod } 1, 1) = \gcd(0, 1)$. 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 $\gcd(8, 13) = 1$.

7. The key idea for this recursive procedure is that $nx = (n - 1)x + x$. Thus we compute nx by calling the procedure recursively, with n replaced by $n - 1$, and adding x. The base case is $1 \cdot x = x$.

procedure *product*(n : positive integer, x : integer)
if $n = 1$ **then** $product(n, x) := x$
else $product(n, x) := product(n - 1, x) + x$

9. If we have already found the sum of the first $n - 1$ odd positive integers, then we can find the sum of the first n positive integers simply by adding on the value of the n^{th} odd positive integer. We need to realize that the n^{th} odd positive integer is $2n - 1$, and we need to note that the base case (in which $n = 1$) gives us a sum of 1. The algorithm is then straightforward.

procedure *sum of odds*(n : positive integer)
if $n = 1$ **then** *sum of odds*(n) := 1
else *sum of odds*(n) := *sum of odds*($n - 1$) + $2n - 1$

11. We recurse on the size of the list. If there is only one element, then it is the smallest. Otherwise, we find the smallest element in the list consisting of all but the last element of our original list, and compare it with the last element of the original list. Whichever is smaller is the answer. (We assume that there is already a function min, defined for two arguments, which returns the smaller.)

procedure *smallest*($a_1, a_2, \ldots, a_n$: integers)
if $n = 1$ **then** *smallest*($a_1, a_2, \ldots, a_n$) := a_1
else *smallest*($a_1, a_2, \ldots, a_n$) := min(*smallest*($a_1, a_2, \ldots, a_{n-1}$), a_n)

13. We basically just take the recursive algorithm for finding $n!$ and apply the **mod** operation at each step. Note that this enables us to calculate $n!$ **mod** m without using excessively large numbers, even if $n!$ is very large.

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

15. We need to worry about which of our arguments is the larger. Since we are given $a < b$ as input, we need to make sure always to call the algorithm with the first argument less than the second. There need to be two stopping conditions: when $a = 0$ (in which case the answer is b), and when the two arguments have become equal (in which case the answer is their common value). Otherwise we use the recursive condition that $\gcd(a, b) = \gcd(a, b - a)$, taking care to reverse the arguments if necessary.

procedure *gcd*(a, b : nonnegative integers with $a < b$)
if $a = 0$ **then** *gcd*(a, b) := b
else if $a = b - a$ **then** *gcd*(a, b) := a
else if $a < b - a$ **then** *gcd*(a, b) := *gcd*($a, b - a$)
else *gcd*(a, b) := *gcd*($b - a, a$)

17. We build the recursive steps into the algorithm.

procedure *multiply*(x, y : nonnegative integers)
if $y = 0$ **then** *multiply*(x, y) := 0
else if y is even **then** *multiply*(x, y) := $2 \cdot$ *multiply*($x, y/2$)
else *multiply*(x, y) := $2 \cdot$ *multiply*($x, (y - 1)/2$) + x

19. We use strong induction on a, starting at $a = 0$. If $a = 0$, we know that $\gcd(0, b) = b$ for all $b > 0$, so the **if** clause handles this basis case correctly. Now fix $k > 0$ and assume the inductive hypothesis—that the algorithm works correctly for all values of its first argument less than k. Consider what happens with input (k, b), where $k < b$. Since $k > 0$, the **else** clause is executed, and the answer is whatever the algorithm gives as output for inputs (b **mod** k, k). Note that b **mod** $k < k$, so the input pair is valid. By our inductive hypothesis, this output is in fact $\gcd(b$ **mod** $k, k)$. By Lemma 1 in Section 3.6 (when the Euclidean algorithm was introduced), we know that $\gcd(k, b) = \gcd(b$ **mod** $k, k)$, and our proof is complete.

21. For the basis step, if $n = 1$, then $nx = x$, and the algorithm correctly returns x. For the inductive step, assume that the algorithm correctly computes kx, and consider what it does to compute $(k + 1)x$. The recursive clause applies, and it recursively computes the product of $k + 1 - 1 = k$ and x, and then adds x. By the inductive hypothesis, it computes that product correctly, so the answer returned is $kx + x = (k + 1)x$, which is the correct answer.

23. As usual with recursive algorithms, the algorithm practically writes itself.

procedure *square*(n : nonnegative integer)
if $n = 0$ **then** *square*(n) := 0
else *square*(n) := *square*($n - 1$) + $2(n - 1) + 1$

The proof of correctness, by mathematical induction, practically writes itself as well. Let $P(n)$ be the statement that this algorithm correctly computes n^2. Since $0^2 = 0$, the algorithm works correctly (using the **if** clause) if the input is 0. Assume that the algorithm works correctly for input k. Then for input $k + 1$ it gives as output (because of the **else** clause) its output when the input is k, plus $2(k + 1 - 1) + 1$. By the inductive hypothesis, its output at k is k^2, so its output at $k+1$ is $k^2+2(k+1-1)+1 = k^2+2k+1 = (k+1)^2$, exactly what it should be.

25. Algorithm 2 uses 2^n multiplications by a, one for each factor of a in the product a^{2^n}. The algorithm in Exercise 24, based on squaring, uses only n multiplications (each of which is a multiplication of a number by itself). For instance, to compute $a^{2^4} = a^{16}$, this algorithm will compute $a \cdot a = a^2$ (one multiplication), then $a^2 \cdot a^2 = a^4$ (a second multiplication), then $a^4 \cdot a^4 = a^8$ (a third), and finally $a^8 \cdot a^8 = a^{16}$ (a fourth multiplication).

27. Algorithm 2 uses n multiplications by a, one for each factor of a in the product a^n. The algorithm in Exercise 26 will use $O(\log n)$ multiplications as it computes squares. Furthermore, in addition to squaring, sometimes a multiplication by a is needed; this will add at most another $O(\log n)$ multiplications. Thus a total of $O(\log n)$ multiplications are used altogether.

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

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

31. The iterative version is much more efficient. The analysis is exactly the same as that for the Fibonacci sequence given in this section. Indeed, the n^{th} term in this sequence is actually just 2^{f_n}, as is easily shown by induction.

33. This is essentially just Algorithm 8, with a different operation and with different (and more) initial conditions.

```
procedure iterative(n : nonnegative integer)
if n = 0 then z := 1
else if n = 1 then z := 2
else
begin
        x := 1
        y := 2
        z := 3
        for i := 1 to n - 2
        begin
                w := x + y + z
                x := y
                y := z
                z := w
        end
end
{ z is the n^th term of the sequence }
```

35. These algorithms are very similar to the procedures for computing the Fibonacci numbers. Note that for the recursive version, we can combine the three base cases (stopping rules) into one.

procedure *recursive*(n : nonnegative integer)
if $n < 3$ **then** $recursive(n) := 2n + 1$
else $recursive(n) := recursive(n-1) \cdot (recursive(n-2))^2 \cdot (recursive(n-3))^3$

procedure *iterative*(n : nonnegative integer)
if $n = 0$ **then** $z := 1$
else if $n = 1$ **then** $z := 3$
else
begin
    $x := 1$
    $y := 3$
    $z := 5$
    for $i := 1$ **to** $n - 2$
    begin
        $w := z \cdot y^2 \cdot x^3$
        $x := y$
        $y := z$
        $z := w$
    end
end
{ z is the n^{th} term of the sequence}

The recursive version is much easier to write, but the iterative version is much more efficient. In doing the computation for the iterative version, we just need to go through the loop $n-2$ times in order to compute a_n, so it requires $O(n)$ steps. In doing the computation for the recursive version, we are constantly recalculating previous values that we've already calculated, just as was the case with the recursive version of the algorithm to calculate the Fibonacci numbers.

37. We use the recursive definition of the reversal of a string given in Exercise 35 of Section 4.3, namely that $(vy)^R = y(v^R)$, where y is the last symbol in the string and v is the substring consisting of all but the last symbol. The right-hand side of the last statement in this procedure means that we concatenate b_n with the output of the recursive call.

procednre *reverse*($b_1 b_2 \ldots b_n$: bit string)
if $n = 0$ **then** $reverse(b_1 b_2 \ldots b_n) := \lambda$
else $reverse(b_1 b_2 \ldots b_n) := b_n\, reverse(b_1 b_2 \ldots b_{n-1})$

39. The procedure correctly gives the reversal of λ as λ (the basis step of our proof by mathematical induction on n), and because the reversal of a string consists of its last character followed by the reversal of its first $n-1$ characters (see Exercise 35 in Section 4.3), the algorithm behaves correctly when $n > 0$ by the inductive hypothesis.

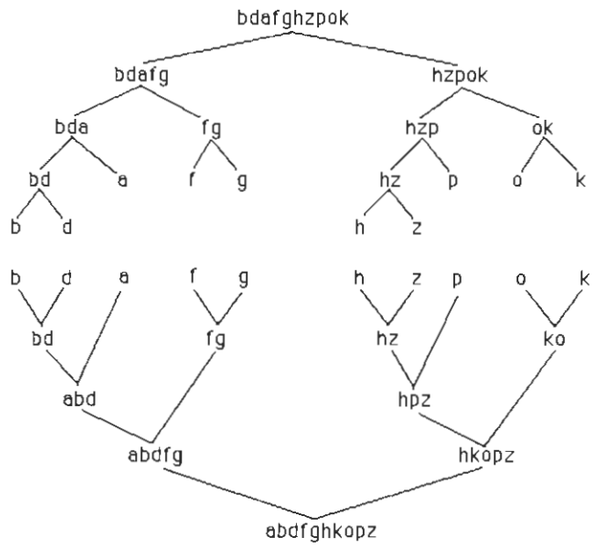
41. The algorithm merely implements the idea of Example 13 in Section 4.1. If $n = 1$ (the basis step here), we simply place the one right triomino so that its armpit corresponds to the hole in the 2×2 board. If $n > 1$, then we divide the board into four boards, each of size $2^{n-1} \times 2^{n-1}$, notice which quarter the hole occurs in, position one right triomino at the center of the board with its armpit in the quarter where the missing square is (see Figure 8 in Section 4.1), and invoke the algorithm recursively four times—once on each of the $2^{n-1} \times 2^{n-1}$ boards, each of which has one square missing (either because it was missing to begin with, or because it is covered by the central triomino).

43. Essentially all we do is write down the definition as a procedure.

procedure *ackermann*$(m, n$: nonnegative integers)
if $m = 0$ **then** *ackermann*$(m, n) := 2 \cdot n$
else if $n = 0$ **then** *ackermann*$(m, n) := 0$
else if $n = 1$ **then** *ackermann*$(m, n) := 2$
else *ackermann*$(m, n) :=$ *ackermann*$(m - 1,$ *ackermann*$(m, n - 1))$

45. We assume that sorting is to be done into alphabetical order. First the list is split into the two lists b, d, a, f, g and h, z, p, o, k, and each of these is sorted by merge sort. Let us assume for a moment that this has been done, so the two lists are a, b, d, f, g and h, k, o, p, z. Then these two lists are merged into one sorted list, as follows. We compare a with h and find that a is smaller; thus a comes first in the merged list, and we pass on to b. Comparing b with h, we find that b is smaller, so b comes next in the merged list, and we pass on to d. We repeat this process (using Algorithm 10) until the lists are merged into one sorted list, a, b, d, f, g, h, k, o, p, z. (It was just a coincidence that every element in the first of these two lists came before every element in the second.)

Let us return to the question of how each of the 5-element lists was sorted. For the list b, d, a, f, g, we divide it into the sublists b, d, a and f, g. Again we sort each piece by the same algorithm, producing a, b, d and f, g, and we merge them into the sorted list a, b, d, f, g. Going one level deeper into the recursion, we see that sorting b, d, a was accomplished by splitting it into b, d and a, and sorting each piece by the same algorithm. The first of these required further splitting into b and d. One element lists are already sorted, of course. Similarly, the other 5-element list was sorted by a similar recursive process. A tree diagram for this problem is displayed below. The top half of the picture is a tree showing the splitting part of the algorithm. The bottom half shows the merging part as an upside-down tree.



47. All we have to do is to make sure that one of the lists is exhausted only when the other list has only one element left in it. In this case, a comparison is needed to place every element of the merged list into place, except for the last element. Clearly this condition is met if and only if the largest element in the combined list is in one of the initial lists and the second largest element is in the other. One such pair of lists is $\{1, 2, \ldots, m - 1, m + n\}$ and $\{m, m + 1, \ldots, m + n - 1\}$.

49. We use strong induction on n, showing that the algorithm works correctly if $n = 1$, and that if it works correctly for $n = 1$ through $n = k$, then it also works correctly for $n = k + 1$. If $n = 1$, then the algorithm does nothing, which is correct, since a list with one element is already sorted. If $n = k + 1$, then the list is

split into two lists, L_1 and L_2. By the inductive hypothesis, *mergesort* correctly sorts each of these sublists, and so it remains only to show that *merge* correctly merges two sorted lists into one. This is clear, from the code given in Algorithm 10, since with each comparison, the smallest element in $L_1 \cup L_2$ not yet put into L is put there.

51. We need to compare every other element with a_1. Thus at least $n-1$ comparisons are needed (we will assume for Exercises 53 and 55 that the answer is exactly $n-1$). The actual number of comparisons depends on the actual encoding of the algorithm. With any reasonable encoding, it should be $O(n)$.

53. In our analysis we assume that a_1 is considered to be put between the two sublists, not as the last element of the first sublist (which would require an extra pass in some cases). In the worst case, the original list splits into lists of length 3 and 0 (with a_1 between them); by Exercise 51, this requires $4-1=3$ comparisons. No comparisons are needed to sort the second of these lists (since it is empty). To sort the first, we argue in the same way: the worst case is for a splitting into lists of length 2 and 0, requiring $3-1=2$ comparisons. Similarly, $2-1=1$ comparison is needed to split the list of length 2 into lists of length 1 and 0. In all, then, $3+2+1=6$ comparisons are needed in this worst case. (One can prove that this discussion really does deal with the worst case by looking at what happens in the various other cases.)

55. In our analysis we assume that a_1 is considered to be put between the two sublists, not as the last element of the first sublist (which would require an extra pass in some cases). We claim that the worst case complexity is $n(n-1)/2$ comparisons, and we prove this by induction on n. This is certainly true for $n=1$, since no comparisons are needed. Otherwise, suppose that the initial split is into lists of size k and $n-k-1$. By the inductive hypothesis, it will require $\big(k(k-1)/2\big)+\big((n-k-1)(n-k-2)/2\big)$ comparisons in the worst case to finish the algorithm. This quadratic function of k attains its maximum value if $k=0$ (or $k=n-1$), namely the value $(n-1)(n-2)/2$. Also, it took $n-1$ comparisons to perform the first splitting. If we add these two quantities ($(n-1)(n-2)/2$ and $n-1$) and do the algebra, then we obtain $n(n-1)/2$, as desired. Thus in the worst case the complexity is $O(n^2)$.