CHAPTER 3
The Fundamentals: Algorithms, the Integers, and Matrices

SECTION 3.1 Algorithms

2. a) This procedure is not finite, since execution of the **while** loop continues forever.

b) This procedure is not effective, because the step $m := 1/n$ cannot be performed when $n = 0$, which will eventually be the case.

c) This procedure lacks definiteness, since the value of i is never set.

d) This procedure lacks definiteness, since the statement does not tell whether x is to be set equal to a or to b.

4. Set the answer to be $-\infty$. For i going from 1 through $n-1$, compute the value of the $(i+1)^{\text{st}}$ element in the list minus the i^{th} element in the list. If this is larger than the answer, reset the answer to be this value.

6. We need to go through the list and count the negative entries.

```
procedure negatives($a_1, a_2, \ldots, a_n$ : integers)
$k := 0$
for $i := 1$ to $n$
    if $a_i < 0$ then $k := k + 1$
end { $k$ is the number of negative integers in the list }
```

8. This is similar to Exercise 7, modified to keep track of the largest even integer we encounter.

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

10. We assume that if the input $x = 0$, then $n > 0$, since otherwise x^n is not defined. In our procedure, we let $m = |n|$ and compute x^m in the obvious way. Then if n is negative, we replace the answer by its reciprocal.

```
procedure power($x$ : real number, $n$ : integer)
$m := |n|$
$power := 1$
for $i := 1$ to $m$
    $power := power \cdot x$
if $n < 0$ then $power := 1/power$
{ $power = x^n$ }
```

12. Four assignment statements are needed, one for each of the variables and a temporary assignment to get started so that we do not lose one of the original values.

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

14. a) With linear search we start at the beginning of the list, and compare 7 successively with 1, 3, 4, 5, 6, 8, 9, and 11. When we come to the end of the list and still have not found 7, we conclude that it is not in the list.

b) We begin the search on the entire list, with $i = 1$ and $j = n = 8$. We set $m := 4$ and compare 7 to the fourth element of the list. Since $7 > 5$, we next restrict the search to the second half of the list, with $i = 5$ and $j = 8$. This time we set $m := 6$ and compare 7 to the sixth element of the list. Since $7 \not> 8$, we next restrict ourselves to the first half of the second half of the list, with $i = 5$ and $j = 6$. This time we set $m := 5$, and compare 7 to the fifth element. Since $7 > 6$, we now restrict ourselves to the portion of the list between $i = 6$ and $j = 6$. Since at this point $i \not< j$, we exit the loop. Since the sixth element of the list is not equal to 7, we conclude that 7 is not in the list.

16. We let *min* be the smallest element found so far. At the end, it is the smallest element, since we update it as necessary as we scan through the list.

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

18. This is similar to Exercise 17.

```
procedure last smallest($a_1, a_2, \ldots, a_n$ : integers)
$min := a_1$
$location := 1$
for $i := 2$ to $n$
        if $min \geq a_i$ then
        begin
                $min := a_i$
                $location := i$
        end
{ $location$ is the location of the last occurrence of the smallest element in the list}
```

20. We just combine procedures for finding the largest and smallest elements.

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

22. We assume that the input is a sequence of symbols, a_1, a_2, ..., a_n, each of which is either a letter or a blank. We build up the longest word in *word*; its length is *length*. We denote the empty word by λ.

```
procedure longest word($a_1, a_2, \ldots, a_n$ : symbols)
maxlength := 0
maxword := $\lambda$
$i := 1$
while $i \le n$
begin
        word := $\lambda$
        length := 0
        while $a_i \ne$ blank and $i \le n$
        begin
                length := length + 1
                word := concatenation of word and $a_i$
                $i := i + 1$
        end
        if length > max then
        begin
                maxlength := length
                maxword := word
        end
        $i := i + 1$
end
```

24. This is similar to Exercise 23. We let the array *hit* keep track of which elements of the codomain B have already been found to be images of elements of the domain A. When we find an element that has already been hit being hit again, we conclude that the function is not one-to-one.

```
procedure one_one($f$ : function, $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m$ : integers)
for $i := 1$ to $m$
        $hit(b_i) := 0$
one_one := true
for $j := 1$ to $n$
        if $hit(f(a_j)) = 0$ then $hit(f(a_j)) := 1$
        else one_one := false
```

26. There are two changes. First, we need to test $x = a_m$ (right after the computation of m) and take appropriate action if equality holds (what we do is set i and j both to be m). Second, if $x \not> a_m$, then instead of setting j equal to m, we can set j equal to $m - 1$. The advantages are that this allows the size of the "half" of the list being looked at to shrink slightly faster, and it allows us to stop essentially as soon as we have found the element we are looking for.

28. This could be thought of as just doing two iterations of binary search at once. We compare the sought-after element to the middle element in the still-active portion of the list, and then to the middle element of either the top half or the bottom half. This will restrict the subsequent search to one of four sublists, each about one-quarter the size of the previous list. We need to stop when the list has length three or less and make explicit checks. Here is the pseudocode.

```
procedure tetrary search($x$ : integer, $a_1, a_2, \ldots, a_n$ : increasing integers)
$i := 1$
$j := n$
while $i < j - 2$
begin
        $l := \lfloor (i + j)/4 \rfloor$
        $m := \lfloor (i + j)/2 \rfloor$
        $u := \lfloor 3(i + j)/4 \rfloor$
```

```
          if $x > a_m$ then if $x \le a_u$ then
                begin
                      $i := m + 1$
                      $j := u$
                end
                else $i := u + 1$
          else if $x > a_l$ then
                begin
                      $i := l + 1$
                      $j := m$
                end
                else $j := l$
    end
    if $x = a_i$ then location := $i$
    else if $x = a_j$ then location := $j$
    else if $x = a_{\lfloor (i+j)/2 \rfloor}$ then location := $\lfloor (i + j)/2 \rfloor$
    else location := 0
    { location is the subscript of the term equal to $x$ (0 if not found)}
```

30. The following algorithm will find all modes in the sequence and put them into a list L. At each point in the execution of this algorithm, *modecount* is the number of occurrences of the elements found to occur most often so far (the elements in L). Whenever a more frequently occurring element is found (the main inner loop), *modecount* and L are updated; whenever an element is found with this same count, it is added to L.

```
procedure find all modes($a_1, a_2, \ldots, a_n$ : nondecreasing integers)
modecount := 0
$i := 1$
while $i \le n$
begin
      value := $a_i$
      count := 1
      while $i \le n$ and $a_i$ = value
      begin
            count := count + 1
            $i := i + 1$
      end
      if count > modecount then
      begin
            modecount := count
            set $L$ to consist just of value
      end
      else if count = modecount then add value to $L$
end
{ $L$ is a list of all the values occurring most often, namely modecount times}
```

32. The following algorithm will find all terms of a finite sequence of integers that are greater than the sum of all the previous terms. We put them into a list L, but one could just as easily have them printed out, if that were desired. It might be more useful to put the *indices* of these terms into L, rather than the terms themselves (i.e., their values), but we take the former approach for variety. As usual, the empty list is considered to have sum 0, so the first term in the sequence is included in L if and only if it positive.

```
procedure find all biggies($a_1, a_2, \ldots, a_n$ : integers)
set $L$ to be the empty list
sum := 0
$i := 1$
```

```
while $i \le n$
begin
        if $a_i > sum$ then append $a_i$ to $L$
        $sum := sum + a_i$
        $i := i + 1$
end
{ $L$ is a list of all the values that exceed the sum of all the previous terms in the sequence}
```

34. There are five passes through the list. After one pass the list reads $2,3,1,5,4,6$, since the 6 is compared and moved at each stage. During the next pass, the 2 and the 3 are not interchanged, but the 3 and the 1 are, as are the 5 and the 4, yielding $2,1,3,4,5,6$. On the third pass, the 2 and the 1 are interchanged, yielding $1,2,3,4,5,6$. There are two more passes, but no further interchanges are made, since the list is now in order.

36. The procedure is the same as that given in the solution to Exercise 35. We will exhibit the lists obtained after each step, with all the lists obtained during one pass on the same line.

$dfkmab$, $dfkmab$, $dfkmab$, $dfkamb$, $dfkabm$
$dfkabm$, $dfkabm$, $dfakbm$, $dfabkm$
$dfabkm$, $dafbkm$, $dabfkm$
$adbfkm$, $abdfkm$
$abdfkm$

38. We start with $6,2,3,1,5,4$. The first step inserts 2 correctly into the sorted list 6, producing $2,6,3,1,5,4$. Next 3 is inserted into $2,6$, and the list reads $2,3,6,1,5,4$. Next 1 is inserted into $2,3,6$, and the list reads $1,2,3,6,5,4$. Next 5 is inserted into $1,2,3,6$, and the list reads $1,2,3,5,6,4$. Finally 4 is inserted into $1,2,3,5,6$, and the list reads $1,2,3,4,5,6$. 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.

40. We start with d,f,k,m,a,b. The first step inserts f correctly into the sorted list d, producing no change. Similarly, no change results when k and m are inserted into the sorted lists d,f and d,f,k, respectively. Next a is inserted into d,f,k,m, and the list reads a,d,f,k,m,b. Finally b is inserted into a,d,f,k,m, and the list reads a,b,d,f,k,m. 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.

42. We let *minspot* be the place at which the minimum remaining element is found. After we find it on the i^{th} pass, we just have to interchange the elements in location *minspot* and location i.

```
procedure selection$(a_1, a_2, \ldots, a_n)$
for $i := 1$ to $n - 1$
begin
        $minspot := i$
        for $j := i + 1$ to $n$
                if $a_j < a_{minspot}$ then $minspot := j$
        interchange $a_{minspot}$ and $a_i$
end {the list is now in order}
```

44. We carry out the binary search algorithm given as Algorithm 3 in this section, except that we replace the final check with **if** $x < a_i$ **then** *location* $:= i$ **else** *location* $:= i + 1$.

46. 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 only with the first (in other words, when $j = 2$ in Algorithm 5, i takes the values 1 before we drop out of the **while** loop). Similarly, the third element must be compared only with the first. We continue in this way, until finally the n^{th} element must be compared only with the first. So the total number of comparisons is $n - 1$. This is the best case for insertion sort in terms of the number of comparisons, but moving the elements to do the insertions requires much more effort.

48. For the insertion sort, one comparison is needed to find the correct location of the 4, one for the 3, four for the 8, one for the 1, four for the 5, and two for the 2. This is a total of 13 comparisons. For the binary insertion sort, one comparison is needed to find the correct location of the 4, two for the 3, two for the 8, three for the 1, three for the 5, and four for the 2. This is a total of 15 comparisons. If the list were long (and not almost in decreasing order to begin with), we would use many fewer comparisons using binary insertion sort. The reason that the answer came out "wrong" here is that the list is so short that the binary search was not efficient.

50. a) This is essentially the same as Algorithm 5, but working from the other end. However, we can do the moving while we do the searching for the correct insertion spot, so the pseudocode has only one section.

```
procedure backward insertion sort($a_1, a_2, \ldots, a_n$ : real numbers with $n \geq 2$)
for $j := 2$ to $n$
begin
        $m := a_j$
        $i := j - 1$
        while ($m < a_i$ and $i > 0$)
        begin
                $a_{i+1} := a_i$
                $i := i - 1$
        end
        $a_{i+1} := m$
end {$a_1, a_2, \ldots, a_n$ are sorted}
```

b) On the first pass the 2 is compared to the 3 and found to be less, so the 3 moves to the right. We have reached the beginning of the list, so the loop terminates ($i = 0$), and the 2 is inserted, yielding $2, 3, 4, 5, 1, 6$. On the second pass the 4 is compared to the 3, and since $4 > 3$, the **while** loop terminates and nothing changes. Similarly, no changes are made as the 5 is inserted. One the fourth pass, the 1 is compared all the way to the front of the list, with each element moving toward the back of the list as the comparisons go on, and finally the 1 is inserted in its correct position, yielding $1, 2, 3, 4, 5, 6$. The final pass produces no change.

c) Only one comparison is used during each pass, since the condition $m < a_i$ is immediately false. Therefore a total of $n - 1$ comparisons are used.

d) The j^{th} pass requires $j - 1$ comparisons of elements, so the total number of comparisons is $1 + 2 + \cdots + (n-1) = n(n-1)/2$.

52. 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, three, leaving 12 cents. It then uses the maximum number of dimes (one) and nickels (none), before using two pennies.

b) one quarter, leaving 24 cents, then two dimes, leaving 4 cents, then four pennies

c) three quarters, leaving 24 cents, then two dimes, leaving 4 cents, then four pennies

d) one quarter, leaving 8 cents, then one nickel and three pennies

54. a) The algorithm uses the maximum number of quarters, three, leaving 12 cents. It then uses the maximum number of dimes (one), and then two pennies. The greedy algorithm worked, since we got the same answer as in Exercise 52.

b) one quarter, leaving 24 cents, then two dimes, leaving 4 cents, then four pennies (the greedy algorithm worked, since we got the same answer as in Exercise 52)

c) three quarters, leaving 24 cents, then two dimes, leaving 4 cents, then four pennies (the greedy algorithm worked, since we got the same answer as in Exercise 52)

d) The greedy algorithm would have us use one quarter, leaving 8 cents, then eight pennies, a total of nine coins. However, we could have used three dimes and three pennies, a total of six coins. Thus the greedy algorithm is not correct for this set of coins.

56. One approach is to come up with an example in which using the 12-cent coin before using dimes or nickels would be inefficient. A dime and a nickel together are worth 15 cents, but the greedy algorithm would have us use four coins (a 12-cent coin and three pennies) rather than two. An alternative example would be 29 cents, in which case the greedy algorithm would use a quarter and four pennies, but we could have done better using two 12-cent coins and a nickel.

58. If all the men get their first choices, then the matching will be stable, because no man will be part of an unstable pair, preferring another woman to his assigned partner. Thus the pairing (m_1w_3, m_2w_1, m_3w_2) is stable. Similarly, if all the women get their first choices, then the matching will be stable, because no woman will be part of an unstable pair, preferring another man to her assigned partner. Thus the matching (m_1w_1, m_2w_2, m_3w_3) is stable. Two of the other four matchings pair m_1 with w_2, and this cannot be stable, because m_1 prefers w_1 to w_2, his assigned partner, and w_1 prefers m_1 to her assigned partner, whoever it is, because m_1 is her favorite. In a similar way, the matching (m_1w_3, m_2w_2, m_3w_1) is unstable because of the unhappy unmatched pair m_3w_3 (each preferring the other to his or her assigned partner). Finally, the matching (m_1w_1, m_2w_3, m_3w_2) is stable, because each couple has a reason not to break up: w_1 got her favorite and so is content, m_3 got his favorite and so is content, and w_3 only prefers m_3 to her assigned partner but he doesn't prefer her to his assigned partner.

60. Suppose we had a program S that could tell whether a program with its given input ever prints the digit 1. Here is an algorithm for solving the halting problem: Given a program P and its input I, construct a program P', which is just like P but never prints anything (even if P did print something) except that if and when it is about to halt, it prints a 1 and halts. Then P halts on an input if and only if P' ever prints a 1 on that same input. Feed P' and I to S, and that will tell us whether or not P halts on input I. Since we know that the halting problem is in fact not solvable, we have a contradiction. Therefore no such program S exists.

62. The decision problem has no input. The answer is either always yes or always no, depending on whether or not the specific program with its specific input halts or not. In the former case, the decision procedure is "say yes," and in the latter case it is "say no."