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

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 *sum*. 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).

29. This product telescopes. The n or in the fraction for k cancels the denominator in the fraction for $k+1$. So all that remains of the p ct is the numerator for $k = 100$ and the denominator for $k = 1$, namely $101/1 = 101$.

31. There is no good way to determine a nice rule for this kind of problem. One just has to look at the sequence and see what seems to be happening. In this sequence, we notice that $10 = 2 \cdot 5$, $39 = 3 \cdot 13$, $172 = 4 \cdot 43$, and $885 = 5 \cdot 177$. We then also notice that $3 = 1 \cdot 3$ for the second and third terms. So each odd-indexed term (assuming that we call the first term a_1) comes from the term before it, by multiplying by successively larger integers. In symbols, this says that $a_{2n+1} = n \cdot a_{2n}$ for all $n > 0$. Then we notice that the even-indexed terms are obtained in a similar way by adding: $a_{2n} = n + a_{2n-1}$ for all $n > 0$. So the next four terms are $a_{13} = 6 \cdot 891 = 5346$, $a_{14} = 7 + 5346 = 5353$, $a_{15} = 7 \cdot 5353 = 37471$, and $a_{14} = 8 + 37471 = 37479$.

WRITING PROJECTS FOR CHAPTER 2

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. A classic source here is [Wil]. It gives a very readable account of many philosophical issues in the foundations of mathematics, including the topic for this essay.

2. Our list of references mentions several history of mathematics books, such as [Bo4] and [Ev3]. You should also browse the shelves in your library, around QA 21.

3. Go to the Encyclopedia's website, `http://www.research.att.com/~njas/sequences/`.

4. A Web search should turn up some useful references here, including an article in *Science News Online*. It gets its name from the fact that a graph describing it looks like the output of an electrocardiogram.

5. A Web search for this phrase will turn up much information.

6. A classic source here is [Wil]. It gives a very readable account of many philosophical issues in the foundations of mathematics, including the topic for this essay. Of course a Web search will turn up lots of useful material, as well.

procedure *sum*($a_1, a_2, \ldots, a_n$: integers)
$sum := a_1$
for $i := 2$ **to** n
    $sum := sum + a_i$
{ *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.

procedure *duplicates*($a_1, a_2, \ldots, a_n$: integers in nondecreasing order)
$k := 0$ {this counts the duplicates}
$j := 2$
while $j \leq n$
begin
    if $a_j = a_{j-1}$ **then**
    begin
        $k := k + 1$
        $c_k := a_j$
        while ($j \leq n$ and $a_j = c_k$)
            $j := j + 1$
    end
$j := j + 1$
end { $c_1, c_2, \ldots, c_k$ 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** $k := i$
end { 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 at least 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 $a_i \neq a_{n+1-i}$ **then** *answer* := **false**
end { *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 $i \leq 8$ and the i^{th} element of the list is not equal to 9. Thus on the first pass we check that $1 \leq 8$ and that $9 \neq 1$ (since $a_1 = 1$), and therefore perform the statement $i := i + 1$. At this point $i = 2$. We check that $2 \leq 8$ and $9 \neq 3$, and therefore again increment i, this time to 3. This process continues until $i = 7$. At that point the condition "$i \leq 8$ and $9 \neq a_i$" 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 $7 \leq 8$, 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 eqnal 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: $a_1 \leq a_2 \leq \cdots \leq a_n$ }
$a_{n+1} := x + 1$
$i := 1$
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**
    begin
        $max := a_i$
        $location := i$
    end
{ *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 nnmbers might be equal to each other—we get around this problem by using $\leq$ rather than $<$ for our comparisons.) We will use the **if**... **then**... **else if**... **then**... **else if**... construction. A condition such as $a \leq b \leq c$ is really the conjunction of two conditions: $a \leq b$ and

$b \leq c$. (Alternately, we could have handled the cases in a nested fashion.) Note that the mean is computed first, independent of the ordering.

```
procedure mean median max min(a, b, c : integers)
mean := (a + b + c)/3
if a ≤ b ≤ c then
begin
        min := a
        median := b
        max := c
end
else if a ≤ c ≤ b then
begin
        min := a
        median := c
        max := b
end
else if b ≤ a ≤ c then
begin
        min := b
        median := a
        max := c
end
else if b ≤ c ≤ a then
begin
        min := b
        median := c
        max := a
end
else if c ≤ a ≤ b then
begin
        min := c
        median := a
        max := b
end
else if c ≤ b ≤ a then
begin
        min := c
        median := b
        max := a
end
{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, ..., 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 $a_1, a_2, \ldots, a_n$, and B is the set consisting of the distinct integers $b_1, b_2, \ldots, b_m$. 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 np 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, ..., a_n, b_1, b_2, ..., 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!}
        begin
                hit(f(a_j)) := 1
                count := count + 1
        end
if count = m then onto := true
else onto := 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 ... a_n : bit string)
count := 0 {no 1's yet}
for i := 1 to n
        if a_i = 1 then count := count + 1
{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, ..., a_n : increasing integers)
i := 1
j := n
while i < j - 1
begin
        l := ⌊(i + j)/3⌋
        u := ⌊2(i + j)/3⌋
        if x > a_u then i := u + 1
        else if x > a_l then
        begin
                i := l + 1
                j := u
        end
        else j := l
end
if x = a_i then location := i
else if x = a_j then location := j
else location := 0
{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 $i \leq n$
begin
    value := a_i
    count := 1
    while $i \leq n$ and a_i = *value*
    begin
        count := *count* + 1
        $i := i + 1$
    end
    if *count* > *modecount* **then**
    begin
        modecount := *count*
        mode := *value*
    end
end
{ *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 $i \leq n$ and *location* = 0
begin
    $j := 1$
    while $j < i$ and *location* = 0
        if $a_i = a_j$ **then** *location* := i
        else $j := j + 1$
    $i := i + 1$
end
{ *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 $i \leq n$ and *location* = 0
    if $a_i < a_{i-1}$ **then** *location* := i
    else $i := i + 1$
{ *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, ..., a_n)
i := 1
still_interchanging := true
while i < n and still_interchanging
begin
    still_interchanging := false
    for j := 1 to n - i
        if a_j > a_{j+1} then
        begin
            still_interchanging := true
            interchange a_j and a_{j+1}
        end
    i := i + 1
end {a_1, ..., 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 $\mathbf{1}, 3, 5, 7, 4$. 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
begin
  {binary search for insertion location i}
  left := 1
  right := $j-1$
  while *left* < *right*
  begin
    middle := $\lfloor (left + right)/2 \rfloor$
    if $a_j > a_{middle}$ **then** *left* := *middle* + 1
    else *right* := *middle*
  end
  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** $j-i-1$
    $a_{j-k} := a_{j-k-1}$
  $a_i := m$
end {$a_1, a_2, \ldots, a_n$ 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. **a)** We keep track of a variable f giving the finishing time of the talk last selected, starting out with f equal to the time the hall becomes available. (We ignore any time that might be needed to clear the hall between talks.) We order the talks in increasing order of the ending times, and start at the top of the list. At each stage of the algorithm, we go down the list of talks from where we left off, and find the first one whose starting time is not less than f. We schedule that talk and update f to record its finishing time. (See also Example 11 in Section 4.1.)

b) We schedule the 9:00 talk and set f to 9:45. The talk with the earliest finishing time among those that do not start before 9:45 is the one that starts at 9:50, so we schedule it and set f to 10:15. The talk with the earliest finishing time among those that do not start before 10:15 is the one that starts at 10:15, so we schedule it and set f to 10:45. The talk with the earliest finishing time among those that do not start before 10:45 is the one that starts at 11:00, so we schedule it and set f to 11:15. There are no more talks that start after this, so we are done, having scheduled four talks.

59. **a)** 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
begin
    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

end
for $j := 1$ **to** s
match j with the one man on j's proposal list
{This matching is stable.}

b) The algorithm will terminate if at some point at the conclusion of the **while** loop, no man is rejected. If this happens, then that must mean that each man has one and only one proposal pending with some woman, because he proposed to only one in that round, and since he was not rejected, his proposal is the only one pending with that woman. It follows that at that point there are s pending proposals, one from each man, so each woman will be matched with a unique man. Finally, we argue that there are at most s^2 iterations of the **while** loop, so the algorithm must terminate. Indeed, if at the conclusion of the **while** loop rejected men remain, then some man must have been rejected, because no man is marked as rejected at the conclusion of the proposal phase (first **for** loop inside the **while** loop). If a man is rejected, then his rejection list grows. Thus each pass through the **while** loop, at least one more of the s^2 possible rejections will have been recorded, unless the loop is about to terminate. (Actually there will be fewer than s^2 iterations, because no man is rejected by the woman with whom he is eventually matched.) There is one more subtlety we need to address. Is it possible that at the end of some round, some man has been rejected by *every* woman and therefore the algorithm cannot continue? We claim not. If at the end of some round some man has been rejected by every woman, then every woman has one pending proposal at the completion of that round (from someone she likes better otherwise she never would have rejected that poor man), and of course these proposals are all from different men because a man proposes only once in each round. That means s men have pending proposals, so in fact our poor universally-rejected man does not exist.

c) 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.

61. 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.