**Chapter 2**

**Algorithm Discovery and Design**

**A Guide to this Instructor’s Manual:**

We have designed this Instructor’s Manual to supplement and enhance your teaching experience through classroom activities and a cohesive chapter summary.

This document is organized chronologically, using the same headings that you see in the textbook. Under the headings you will find: lecture notes that summarize the section, Teaching Tips, Class Discussion Topics, and Additional Projects and Resources. Pay special attention to teaching tips and activities geared towards quizzing your students and enhancing their critical thinking skills.

In addition to this Instructor’s Manual, our Instructor’s Resources also contain PowerPoint Presentations, Test Banks, and other supplements to aid in your teaching experience.

|  |
| --- |
| **At a Glance** |

**Instructor’s Manual Table of Contents**

* Overview
* Learning Objectives
* Teaching Tips and Quick Quizzes
* Class Discussion Topics
* Additional Projects
* Additional Resources
* Key Terms
* Solutions to End-of-Chapter Exercises

|  |
| --- |
| Lecture Notes |

**Overview**

Chapter 2 examines algorithmic problem solving as it is seen in computer science. It defines pseudocode, which uses statements in English and math notation, along with structure similar to a programming language. The chapter explains why pseudocode is better for algorithm design than a natural language like English or a programming language: structure is good for clarity, but excessive focus on syntax is bad. The constructs needed for pseudocode are sequential (computations, input, and output), conditional, and iterative statements. The chapter ends with a series of case studies, where the process of developing an algorithmic solution to a problem is carefully illustrated. The problems are multiplication using repeated addition, linear search, finding the maximum element of a list, and pattern matching.

# **Learning Objectives**

* Explain the benefits of pseudocode over natural language or a programming language
* Represent algorithms using pseudocode
* Identify algorithm statements as sequential, conditional, or iterative
* Define abstraction and top-down design, and explain their use in breaking down complex problems
* Illustrate the operation of algorithms for:
  + Multiplication by repeated addition
  + Sequential search of a collection of values
  + Finding the maximum element in a collection
  + Finding a pattern string in a larger piece of text

# **Teaching Tips**

**2.1 Introduction**

Begin by discussing the fact that the algorithm examples in Chapter 1 are drawn from everyday life and are not of interest to computer scientists due to their reflection of everyday life. At this time, a computer cannot wash your hair. This chapter will provide examples of problems that are of interest to computer scientists. Note that these examples are straightforward and basic, chosen to make these concepts easily understandable.

**2.2 Representing Algorithms**

1. The important concept in this section is why we choose to describe algorithms in pseudocode and what pseudocode is. Explain how a natural language and a programming language are two ends of a spectrum, and both are poor choices for abstract design. Figures 2.1 and 2.2 give examples of the addition algorithm from figure 1.2 in the two extreme forms.
2. The flaws in natural languages include a lack of structure and a tendency toward ambiguity.
3. The flaws in a programming language for algorithm design are too many cryptic forms and fussy grammatical details—issues that are unnecessary when first designing or describing an algorithm.
4. Introduce the term **pseudocode** and discuss its form: A set of English-language constructs designed to more or less resemble statements in a programming language but that do not actually run on a computer.
5. Explain to students that pseudocode is not a precise set of notational rules to be memorized and rigidly followed. It is a flexible notation that can be adjusted to fit your own view about how best to express ideas and algorithms.
6. Introduce the term **variable**.Explain that it is simply a named storage location that can hold a data value.
7. Introduce the terms **computation**: the act of a computing agent solving an algorithm; **input**: algorithmic data from outside being put into the computing agent;and **output**: the result of the computation of the computing agent. These comprise the basic sequential operations.
8. Introduce the term **sequential algorithm** and explain that it can only execute from beginning to end. It cannot select among alternative operations or perform a block of instructions more than once; Figure 2.3 contains a sample sequential algorithm.
9. Introduce the terms **control operations** and **conditional statements**. Use examples from Chapter 1 to illustrate conditional statements students have already seen. Figure 2.4 gives a diagrammatic view of a conditional statement, and Figure 2.5 shows a sample algorithm containing a conditional statement.
10. Introduce the terms **iteration**, using examples from the previous chapter to start. Go over the parts of a while statement with care: the **continuation condition**, and **loop body**. Figure 2.6 gives a diagrammatic view of a while loop, and Figure 2.7 shows a sample algorithm with iteration. Figure 2.8 gives a diagrammatic view of the do-while alternative form. Consider rewriting the sample algorithm using do-while.

|  |  |
| --- | --- |
| ***Teaching Tip*** | The figures in the book gradually elaborate an algorithm for computing gas mileage, starting with a sequential algorithm and ending with one that contains iteration and conditionals. Pick a different example, such as computing sales tax, number of tables to seat guests, and so on., and have your students create algorithms themselves: first a simple sequential algorithm, then one with a conditional, and so on. |

1. Figure 2.9 summarizes the primitive operations for the book’s pseudocode. Discuss the meaning of the term **primitive operation**.

**Quick Quiz 1**

1. (True or false) A conditional operation causes its sequential operations to repeat over and over again.   
   Answer: False
2. A variable is a named storage location that can hold a data \_\_\_\_\_\_\_\_\_\_.

Answer: value

1. (True or false) The real power of a computer comes not from doing a calculation once but from doing it many, many times.

Answer: True

1. List the two parts of a while loop.

Answer: Continuation condition and loop body

1. List one way in which pseudocode is similar to English, and one way in which it is similar to a programming language.

Answer: Similar to English: It uses English sentences, grammar, and so on. Similar to a programming language: It is structured, has a set of known operations, and uses constructs similar to programming language constructs.

1. A posttest loop performs the true/false test at the beginning of the loop body.

Answer: False

**2.3 Examples of Algorithmic Problem Solving**

1. Explain that the problem for the first example, “Go Forth and Multiply,” is to multiply two numbers without using multiplication. Instead, multiply two values by using repeated addition. This problem comes from Chapter 1, Exercise 12. Be prepared to justify why we might want to do such a thing. Possible justifications include: if we had a small, simple computer that could only do addition, we might need such an algorithm, or to show that the minimal set of operations does not need to include multiplication.
2. There are two approaches to take here. One works through the pseudocode line by line from start to end. The other starts with a vaguely worded description and refines it, piece by piece, to create the final pseudocode. Pick an approach, and work through the algorithm. Figure 2.10 shows the final pseudocode for the “Go Forth and Multiply” problem.

|  |  |
| --- | --- |
| ***Teaching Tip*** | Many students at this stage have trouble remembering the difference between sequential, conditional, and iterative statements. Take the time to label the steps of the examples or, better yet, ask your students to label them. Perhaps pair them up and have them label a completed algorithm, and then collect the labels as a whole class. |

1. Introduce the term **algorithm discovery**.
2. Introduce the sequential search problem: finding a piece of data from a list of data. As a class exercise, ask students to solve the problem of locating a particular person’s name in a telephone book. Provide them with a sample telephone book that has the names in no particular order.
3. Work through the three versions of the sequential search algorithm. Figure 2.11 shows a sequential algorithm solution for finding a name in a telephone book. Discuss with students the drawbacks to this algorithm (length, work done, and repetition). Figure 2.12 shows an algorithm that properly uses a loop and conditionals to avoid repetition and work. Figure 2.13 shows a version that works correctly even when the name does not appear in the telephone book.
4. Point out that the structure of data matters a great deal: if the numbers were in numerical order, we could take advantage of that fact and find phone numbers much more quickly.
5. Introduce the “find largest” or “find extreme” algorithm, demonstrating that finding the largest or smallest values are nearly identical. Give some useful applications to motivate the problem: finding the highest bid in an online auction, finding the lowest price on a set of items.
6. Introduce the term **library**, where useful algorithms can be collected and kept for future reuse.
7. As a class exercise, have the students work in teams through the pictorial version of the algorithm as given in the book: given a pile of cards with numbers on them, find the largest. Then ask them to develop the algorithm in pseudocode. Compare their work to the book’s—from Figure 2.14.
8. Describe the “pattern matching” problem, and emphasize its importance across a wide range of applications: MRI analysis, human and animal genomes, web searching, and so on.
9. The pattern-matching algorithm here is composed of two parts. The first part of the algorithm aligns the pattern with a particular position in the text and slides the pattern ahead one character when it fails to match. Figure 2.15 shows this part of the algorithm, with an abstract step filling in for the second part. The second part compares the pattern with the characters at a given position of the text and determines if there is a match. Figure 2.16 shows the completed algorithm with the second part elaborated.

|  |  |
| --- | --- |
| ***Teaching Tip*** | Refer students to the link below for an excellent discussion and animation of a broad range of string-matching algorithms.  <http://igm.univ-mlv.fr/~lecroq/string/index.html> |

1. Introduce the terms **abstraction** and **top-down design**. Emphasize the importance of abstraction to computer science, including metaphors like desktop, folder, and trash can.

**Quick Quiz 2**

1. A library is a collection of \_\_\_\_\_\_\_\_\_ algorithms.  
   Answer: predefined
2. Why did we add a conditional statement to the second version of the “multiplication by repeated addition” algorithm?
3. To check if the answer value had become too large
4. To avoid wasted computations
5. To check if the input values were numbers
6. To check if the first number was larger than the second

Answer: (b)

1. (True or false) Sequential search requires that the data be sorted in order.

Answer: False

1. \_\_\_\_\_\_\_\_\_\_\_\_\_ is the separation of the high-level view of an entity or operation from the low-level details of its implementation.

Answer: Abstraction

1. Name two real uses of pattern-matching algorithms.

Answer: Searching in documents, finding documents online, medical imaging, and human genome searching

1. (True or false) Top-down design refers to a method where you view an operation at a high level of abstraction and flesh out the details of its implementation at a later time.

Answer: True

# **Class Discussion Topics**

1. Explain the purpose of pseudocode. In doing so, explain how and why it differs from natural language and formal programming languages.
2. From everyday life, provide and discuss examples of sequential, conditional, and iterative operations. How do the ways we state these operations in everyday life differ from the ways we state them in pseudocode? Why might there be a difference?
3. Explain why it is necessary for the pattern-matching algorithm to have a “loop inside of a loop.”

# **Additional Projects**

1. Design an algorithm in pseudocode that will take three numbers, and determine which of the three is the middle value (less than or equal to one number and greater than or equal to the other).
2. Working in groups of two or three, take an algorithm written in pseudocode. It may be one from the book or one provided by your instructor. Label each line of the algorithm with one of the following labels: computation, input, output, conditional, or iteration.
3. Starting with the definition of find maximum from the book, modify it so that it finds both maximum and minimum at the same time. Could you modify it so that it finds the three largest values?

# **Additional Resources**

1. Another view of pseudocode: <http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html>
2. Sequential search: <http://video.franklin.edu/Franklin/Math/170/common/mod01/linearSearchAlg.html>
3. Human Genome Project: <http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml>
4. MIT Pseudocode 101PDF :

https://courses.edx.org/c4x/MITx/6.00.1x/asset/files\_ps04\_files\_WhyPseudocode.pdf

**Key Terms**

* **Abstraction:** The separation of the high-level view of an entity from the low-level details of its implementation.
* **Algorithm discovery:** The process of finding an algorithmic solution to a given problem.
* **Computation:** An algorithmic operation that carries out a single numeric computation and stores the result.
* **Conditional statements:** Operations that ask a question and select the next instruction to carry out based on the answer to that question.
* **Continuation condition:** The true/false condition that will determine when the iteration has been completed.
* **Control operations:** Operations that alter the normal sequential flow of control within an algorithm.
* **Input:** An operation that causes data values from the outside world to be brought into the algorithm.
* **Iteration:** The repetitive execution of a block of operations.
* **Library:** A collection of useful prewritten algorithms that can be used during problem solving.
* **Loop body:** The block of statements that are to be repetitively executed.
* **Output:** An operation that causes computed values to be sent to the outside world for viewing or saving.
* **Primitive operations:** Operations that can be directly understood by the computing agent executing the algorithm and which do not have to be further clarified or explained.
* **Pseudocode:** A notation used to design algorithms. It uses English constructs, mathematical notation, and an informal algorithmic structure designed to look like a high-level programming language.
* **Sequential algorithm:** An algorithm that executes its operations in a straight line, from top to bottom, without any branching.
* **Top-down design:** A problem solving strategy in which you begin at the highest-level view of the problem and, in steps, address the lower-level details of how to accomplish each operation.
* **Variable:** A named storage location that can hold a data value.

# **Solutions to End-of-Chapter Exercises**

1. (a) Set the value of *area* to ½ (*b* × *h*)

(b) Set the value of *interest* to *I* × *B*

Set the value of *FinalBalance* to (1 + *I*) × *B*

(c) Set the value of *FlyingTime* to *M/AvgSpeed*

1. Algorithm:

Step 1: Get values for *B, I,* and *S*

Step 2: Set the value of *FinalBalance* to (1 + *I*/12)12*B*

Step 3: Set the value of *Interest* to *FinalBalance – B*

Step 4: Set the value of *FinalBalance* to *FinalBalance – S*

Step 5: Print the message 'Interest Earned: '

Step 6: Print the value of *Interest*

Step 7: Print the message 'Final Balance: '

Step 8: Print the value of *FinalBalance*

1. Algorithm:

Step 1: Get values for *E1, E2, E3 and F*

Step 2: Set the value of *Ave* to (*E1 + E2 + E3 + 2F*)/5

Step 3: Print the value of *Ave*

1. Algorithm:

Step 1: Get values for *P* and *Q*

Step 2: Set the value of *Subtotal* to *P* × *Q*

Step 3: Set the value of *TotalCost* to (1.06) × *Subtotal*

Step 4: Print the value of *TotalCost*

1. (a) If *y* ≠ 0 then   
    Print the value of (*x/y*)

Else   
Print the message 'Unable to perform the division.'

(b) If *r* > 1.0, then  
 Set the value of Area to π × *r*2  
 Set the value of *Circum* to 2 × π × *r*

Else  
 Set the value of *Circum* to 2 × π × *r*

1. Algorithm:

Step 1: Get a value for *B, I,* and *S*

Step 2: Set the value of *FinalBalance* to (1 + *I*/12)12*B*

Step 3: Set the value of *Interest* to *FinalBalance* – *B*

Step 4: If *B* < 1000 then Set the value of *FinalBalance* to *FinalBalance* – *S*

Step 5: Print the message 'Interest Earned: '

Step 6: Print the value of *Interest*

Step 7: Print the message 'Final Balance: '

Step 8: Print the value of *FinalBalance*

1. Algorithm:

Step 1: Set the value of *i* to 1

Step 2: Set the values of *Won, Lost,* and *Tied* all to 0

Step 3: While *i* < 10 do

Step 4: Get the value of *CSUi* and *OPPi*

Step 5: If *CSUi* > *OPPi*  then

Step 6: Set the value of *Won* to *Won* + 1

Step 7: Else if *CSUi* < *OPPi* then

Step 8: Set the value of *Lost* to *Lost* + 1

Step 9: Else   
  
Step 10: Set the value of *Tied* to *Tied* + 1

Step 11: Set the value of *i* to *i* + 1

End of the While loop

Step 12: Print the values of *Won*, *Lost*, and *Tied*

Step 13: If *Won =* 10, then

Step 14: Print the message, 'Congratulations on your undefeated season.'

1. Algorithm:

Step 1: Set the value of *i* to 1

Step 2: Set the value of *Total* to 0

Step 3: While *i* < 14 do

Step 4: Get the value of *Ei*

Step 5: Set the value of *Total* to *Total* + *Ei*

Step 6: Set the value of *i* to *i* + 1

End of While loop

Step 7: Get the value of *F*

Step 8: Set the value of *Total* to *Total* + 2*F*

Step 9: Set the value of *Ave* to *Total* / 16

Step 10: Print the value of *Ave*

1. Algorithm:

Step 1: Set the value of *TotalCost* to 0

Step 2: Do

Step 3: Get values for *P* and *Q*

Step 4: Set the value of *Subtotal* to *P* × *Q*

Step 5: Set the value of *TotalCost* to *TotalCost* + (1.06) × *Subtotal*

While (*TotalCost* < 1000)

Step 6: Print the value of *TotalCost*

1. The tricky part is in steps 6 through 9. If you use no more than 1000 kilowatt hours in the month then you get charged $.06 for each. If you use more than 1000, then you get charged $.06 for the first 1000 hours and $.08 for each of the remaining hours. There are *Mi* – 1000 remaining hours, since *Mi* is the number of hours in the *i*th month. Also, remember that *KWBegini* and *KWEndi* are meter readings, so we can determine the total kilowatt-hours used for the whole year by subtracting the first meter reading (*KWBegin*1) from the last (*KWEnd*12).

Step 1: Set the value of *i* to 1

Step 2: Set the value of *AnnualCharge* to 0

Step 3: While *i* < 12 do

Step 4: Get the value of *KWBegini* and *KWEndi*

Step 5: Set the value of *Mi* to *KWEndi* – *KWBegini*

Step 6: If *Mi* < 1000 then

Step 7: Set *AnnualCharge* to *AnnualCharge* + (.06*Mi*)

Step 8: Else

Step 9: Set *AnnualCharge* to *AnnualCharge* + (.06)1000

+ (.08)(*Mi* – 1000)

Step 10: Set the value of *i* to *i* + 1

End of While loop

Step 11: Print the value of *AnnualCharge*

Step 12: If (*KWEnd*12 – *KWBegin*1) < 500, then

Step 13: Print the message 'Thank you for conserving electricity.'

1. Algorithm:

Step 1: Do

Step 2: Get the values of *HoursWorked* and *PayRate*

Step 3: If *HoursWorked* > 54 then

Step 4: *DT* = *HoursWorked* – 54

Step 5: *TH* = 14

Step 6: *Reg* = 40

Step 7: Else if *HoursWorked* > 40 then

Step 8: *DT* = 0

Step 9: *TH* = *HoursWorked* – 40

Step 10: *Reg* = 40

Step 11: Else (*HoursWorked* < 40)

Step 12: *DT* = 0

Step 13: *TH* = 0

Step 14: *Reg* = *HoursWorked*

Step 15: *GrossPay* = *PayRate* × *Reg*

*+* 1.5 × *PayRate* × *TH +* 2 × *PayRate* × *DT*

Step 16: Print the value of *GrossPay*

Step 17: Print the message 'Do you wish to do another computation?'

Step 18: Get the value of *Again*

While(*Again* =yes)

1. Steps 1, 2, 5, 6, 7, and 9 are sequential operations and steps 4 and 8 are conditional operations. After their completion, the algorithm moves on to the step below it, so none of these could cause an infinite loop. Step 3, however, is a while loop, and it could possibly cause an infinite loop. The true/false looping condition is “*Found* = NO and *i* ≤ 10,000.” If NUMBER is ever found in the loop then *Found* gets set to YES, the loop stops, and the algorithm ends after executing steps 8 and 9. If NUMBER is never found, then 1 is added to *i* at each iteration of the loop. Since step 2 initializes *i* to 1, *i* will become 10,001 after the 10,000th iteration of the loop. At this point the loop will halt, steps 8 and 9 will be executed, and the algorithm will end.
2. Algorithm:

Step 1: Get values for *NUMBER, T1, . . . . T*10000, and *N*1*, . . . ., N*10000

Step 2: Set the value of *i* to 1 and set the value of *NumberFound* to 0

Step 3: While (*i* ≤ 10,000) do steps 4 through 7

Step 4: If *NUMBER* equals *Ti* then

Step 5: Print the name of the corresponding person, *Ni*

Step 6: Set the value of *NumberFound* to *NumberFound* + 1

Step 7: Add 1 to the value of *i*

Step 8: Print the message *NUMBER* ' was found ' *NumberFound* 'times'

Step 9: Stop

1. Let’s assume that FindLargest is now a primitive to us, and use it to repeatedly remove the largest element from the list until we reach the median.

Step 1: Get the values *L*1, *L*2, . . ., *LN* of the numbers in the list

Step 2: If *N* is even then

Let *M* = *N* / 2

Else

Let *M* = (*N* + 1) / 2

Step 3: While (N ≥ M) do steps 4 through 9

Step 4: Use FindLargest to find the *location* of the largest number

in the list *L*1*, L*2, . . ., *LN*

Step 5: Exchange *Llocation* and *LN* as follows

Step 6: *Temp* = *LN*

Step 7: *LN* = *Llocation*

Step 8: *Llocation* = *Temp*

Step 9: Set *N* to *N* – 1 and effectively shorten the list

Step 10: Print the message 'The median is: '

Step 11: Print the value of *LM*

Step 12: Stop

1. This algorithm will find the first occurrence of the largest element in the collection. This element will become *LargestSoFar*, and from then on *A*­i will be tested to see if it is greater than *LargestSoFar*. Some of the other elements are equal to *LargestSoFar* but none are greater than it.
2. (a) If *n* < 2, then the test would be true, so the loop would be executed. In fact, the test would never become false. Thus, the algorithm would either loop forever, or generate an error when referring to an invalid *Ai* value. If *n* > 2, then the test would be false the first time through, so the loop would be skipped and *A*1would be reported as the largest value.
   1. The algorithm would find the largest of the first *n* – 1 elements and would not look at the last element, as the loop would exit when *i* = *n*.

(c) For *n* = 2 the loop would execute once, comparing the *A*1and *A*2 values. Then the loop would quit on the next pass, returning the larger of the first two values. For any other value of *n*, the loop would be skipped, reporting *A*1 as the largest value.

1. (a) The algorithm would still find the largest element in the list, but if the largest were not unique then the algorithm would find the last occurrence of the largest element in the list.

(b) The algorithm would find the smallest element in the list.

The relational operations are very important, and care must be taken to choose the correct one, for mixing them up can drastically change the output of the algorithm.

1. (a) The algorithm will find the three occurrences of "and". First in the word band, second in the word and, and third in the word handle.

(b) We could search for “and ”. That is, the word itself surrounded by spaces. Note that the word "and" is special in that it is almost always surrounded by spaces in a sentence. Other words may start or end sentences and be followed by punctuation.

1. It would go into an infinite loop, because *k* will stay at 1, and we will never leave the outside while loop. We will keep checking the 1 position over and over again.

**20.**  Step 1: Get the value for *N*

Step 2: Set the value of *i* to 2

Step 3: Set the value of *R* to 1;

Step 4: While (*i* < *N* and *R* ≠ 0) do Steps 5-6

Step 5: Set *R* to the remainder upon computing *N*/*i*

Step 6: Set the value of *i* to *i* + 1

Step 7: If *R* = 0 then

Print the message 'not prime'

Else

Print the message 'prime'

(This algorithm could be improved upon because it is enough to look for divisors of N less than or equal to.)

**21.** Here we assume that we can perform "arithmetic" on characters, so that m + 3 = p, for example. Step 4 is the difficult part that must handle the "wraparound" from the end of the alphabet back to the beginning.

Step 1:Get the values for *nextChar* and *k*

Step 2: While (*nextChar* ≠ $) do steps 3 through 5

Step 3: Set the value of *outChar* to *nextChar* + *k*

Step 4: If *outChar* > z then

Set the value of *outChar* to (*outChar* -26)

Step 5: Print *outChar*

**22.**  Step 1: Get the values for *k* and *N*1, *N*2, …, *N*k

Step 2: Set the value of *front* to 1

Step 3: Set the value of *back* to *k*

Step 4: While (*front* ≤ *back*) do steps 5 through 9

Step 5: Set the value of *Temp* to *N*back

Step 6: Set the value of *Nback* to *Nfront*

Step 7: Set the value of *Nfront* to *Temp*

Step 8: Set *front* = *front* + 1

Step 9 Set *back* = *back* – 1

**23.** Step 1: Get the values for *N*1, *N*2, …, *N*k, and *SUM*

Step 2: Set the value of *i* to 1

Step 3: Set the value of *j* to 2

Step 4: While (*i* < *k*) do steps 5 through 11

Step 5: While (*j* ≤ *k*) do steps 6 through 9

Step 6: If *N*i + *N*j = *SUM* then

Step 7: Print (*N*i, *N*j)

Step 8: Stop

Else

Step 9: Set the value of *j* to *j* + 1

Step 10: Set the value of *i* to *i* + 1

Step 11: Set the value of *j* to *i* + 1

Step 12: Print the message 'Sorry, there is no such pair of values.'

**24.** Set count to 0

Set sum to 0

Get a value for V

While V ≠ -1

Set sum to sum + V

Set count to count + 1

Get the next value for V

End of the loop

*Let’s make sure that we had at least one value so we don’t divide by 0*

If (count > 0)

Set average to sum / count

Print the value of average

Else

Print the message ‘I was given no input data’

Stop

**25.** Set adjacent to NO

Get values for V1 and V2 *We can do this since we know there are at least 2 values*

While (V2 ≠ -1) AND (adjacent = NO)

If V1 = V2

Set adjacent to YES

Else

Set V1 = V2

Get a new value for V2

End of loop

Print the value of adjacent

Stop

**26.** We only need to make one simple change. Instead of writing

Print the value of adjacent

we change that to read:

If (adjacent = YES)

Print the message 'Yes, the numbers ' V1 ' and ' V1 ' are adjacent.'

Else

Print just the value of adjacent

**Discussion of Challenge Work**

1. The general algorithm is fairly clear, in English, in the text.

Step 1: Read values for *start, step,* and *accuracy*

Step 2: While |*step*| > *accuracy* do steps 3 through 9

Step 3: If *f(start)* > 0 then set *FirstSign* to +

Step 4: Else set *FirstSign* to –

Step 5: Do steps 6 through 8

Step 6: Set the value of *start* to *start* + *step*

Step 7: If *f(start)* > 0 then set the value of *Sign* to +

Step 8: Else set the value of *Sign* to –

while (*Sign* = *FirstSign)*

Step 9: If |*step|* > *accuracy* then set the value of *step* to (-0.1) *step*

Step 10: Set the value of *root* to *start* – *step*/2

Step 11: Print the value of *root*.

1. Many excellent simulations of sorting algorithms are available on the Web, suggest students examine them if they have questions about this algorithm.

The Find Largest algorithm given in the book always searches the whole list. First, we should create a variation that takes, in addition to the list of values, two indices which bound the range of the list that should be searched. Also, it is easiest to return the location of the largest value, for use in the sort algorithm. Below is a sketch of how it should change:

FindLargest(A, *start, end*)

Step 1: Set the value of *loc* to *start*

Step 2: Set the value of *i* to *start* + 1

Step 3: While (*i* < *end*) do

Step 4: If *Ai* > *Aloc* then

Step 5: Set *loc* to *i*

Step 6: Add 1 to the value of *i*

Step 7: End of the loop

Step 8: Return the value *loc*

The Selection Sort algorithm is quite simple, once we have a suitable form for the Find Largest portion of it.

Step 1: SelectionSort(*A, n*)

Step 2: Set *lastpos* to *n*

Step 3: While (*lastpos* > 1) do

Step 4: Set *biggestpos* to FindLargest(*A,* 1, *lastpos*)

Step 5: Swap *Alastpos* and *Abiggestpos*

Step 6: Subtract 1 from *lastpos*

Step 7: End of loop

1. Students should be provided with concrete leads to reference materials about non-European mathematicians, including references to online resources.