Pointers for Structuring a Solution

Pointers for Structuring a Solution

The computer should be a tool to help people find solutions to their problems and to increase their productivity. The solutions must be developed according to methods that accomplish these goals. You can develop efficient computer solutions to problems if you heed the following pointers:

1. Use modules—break the whole into parts, with each part having a particular function.
2. Use the three logic structures to ensure that the solution flows smoothly from one instruction to the next, rather than jumping from one point in the solution to another.
   a. The **sequential structure** executes instructions one after another in a sequence. See Figure 4.1.
   b. The **decision structure** branches to execute one of two possible sets of instructions. See Figure 4.2.
   c. The **loop structure** executes a set of instructions many times. See Figure 4.3.
3. Eliminate the rewriting of identical processes by using modules.
4. Use techniques to improve readability, including the four logic structures, proper naming of variables, internal documentation, and proper indentation.

sequential logic structure

decision logic structure

loop logic structure

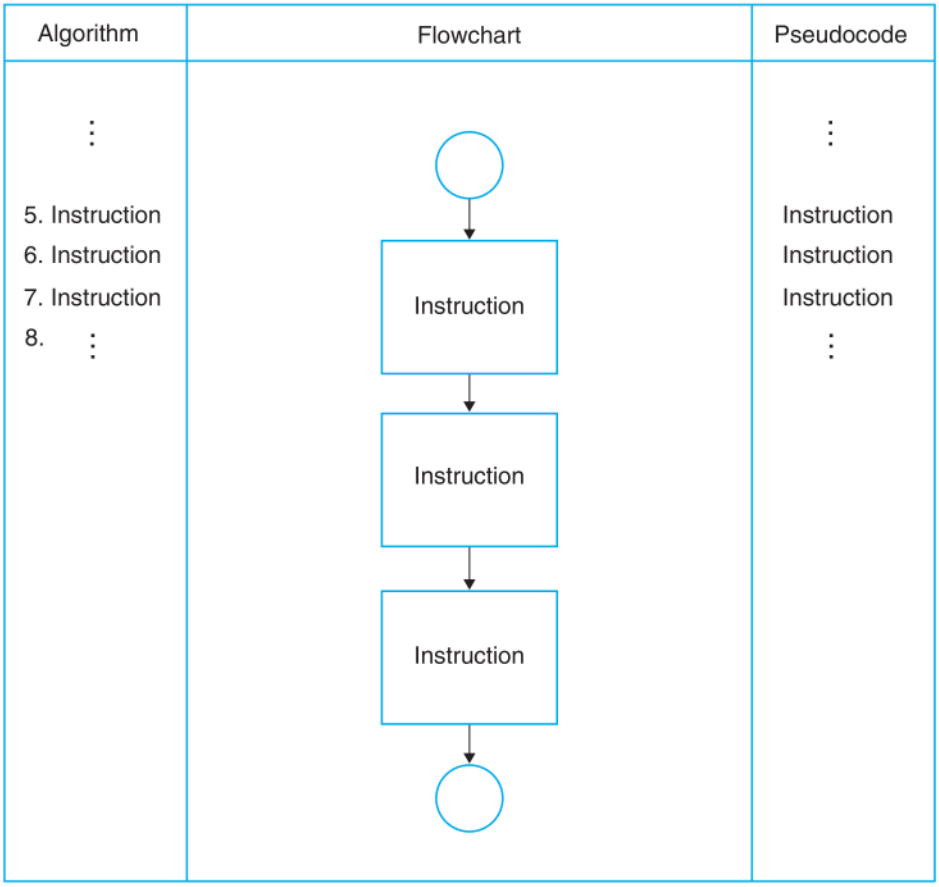


Figure 4.1 Sequential Logic Structure

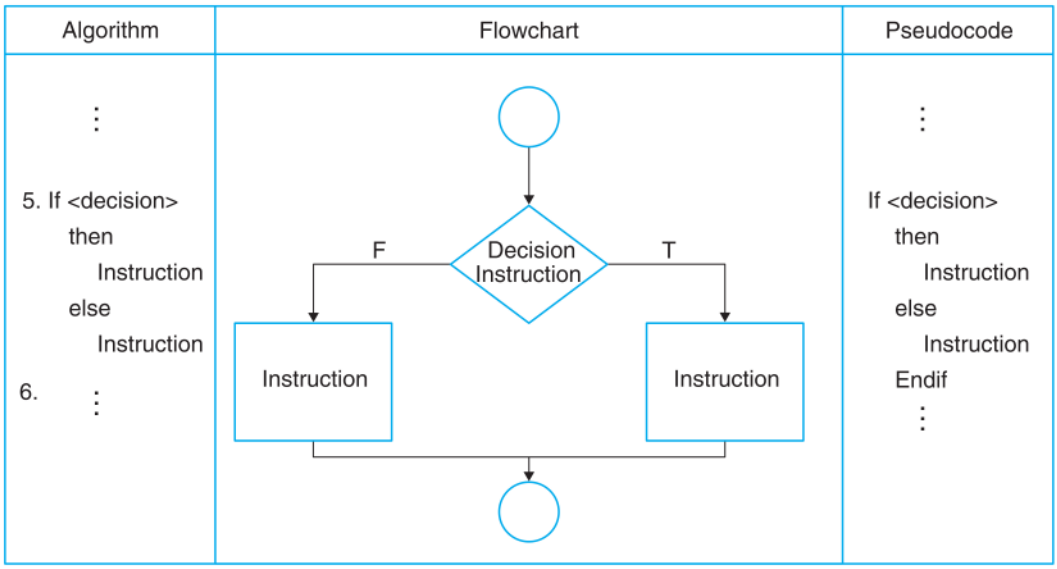


Figure 4.2 Decision Logic Structure

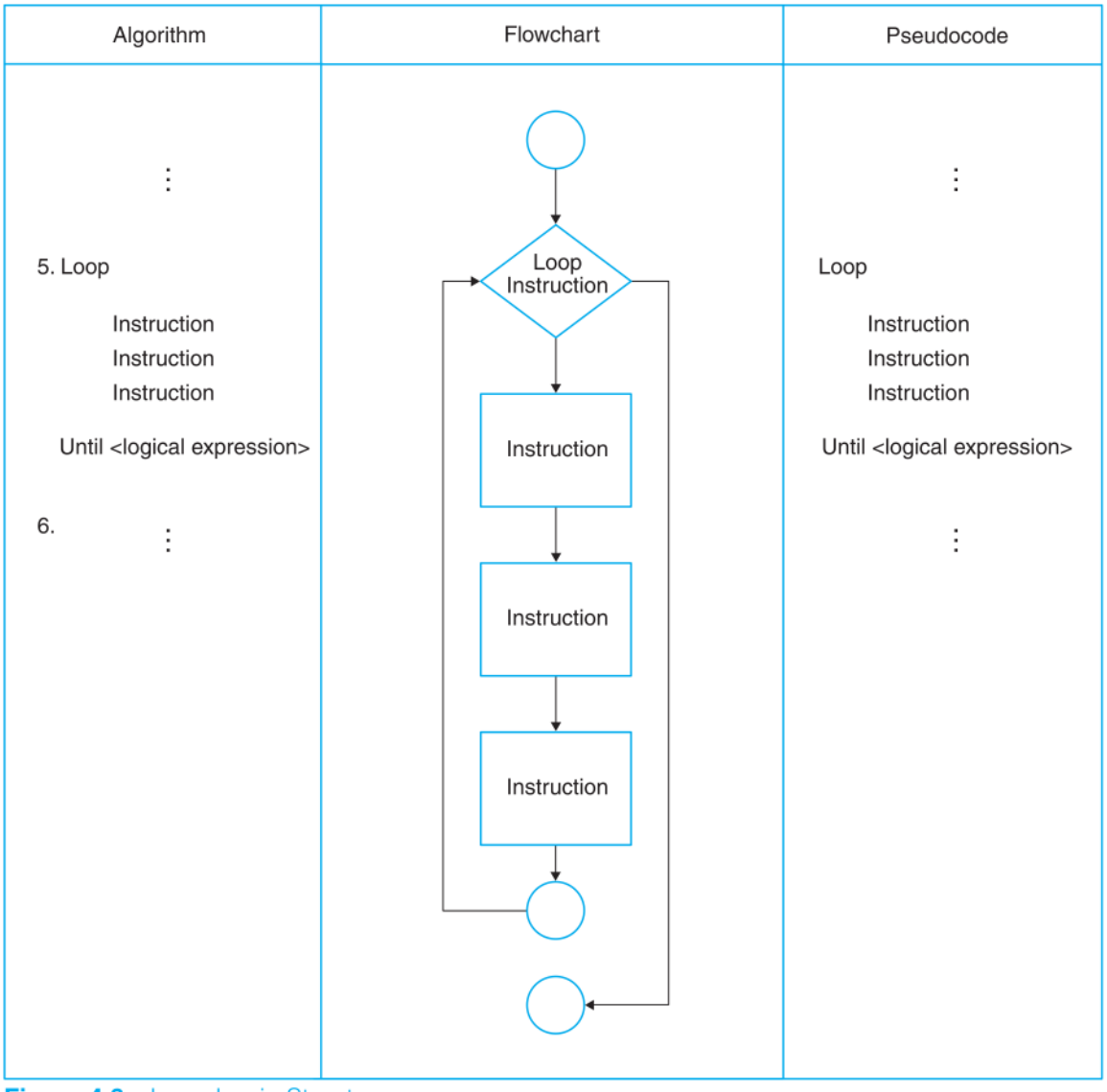


Figure 4.3 Loop Logic Structure

The *End, Exit,* or *Return* instruction specifies the completion of a module. *End* is used to end the *Control* module and indicates that the processing of the solution is complete. *Exit* is used to end a subordinate module if there is no return value, and indicates that the processing will continue in another module, the module where the *Process* instruction originated.

The *Return* (variable) is used to place a value in the name of the module. The *Return* is used when the module is to be processed within an expression. The only way to preserve the value of the function name is either to use it in an expression or to place the value in a variable through the use of the assignment statement. The processing continues in the calling module, the module in which the instruction that contained the expression originated. The programmer writes each instruction in its entirety within the appropriate flowchart symbol. The list of instructions in Table 5.1 will be expanded in future chapters.

The Sequential Logic Structure

Sequential Logic Structure

The most commonly used and the simplest logic structure is the sequential structure. All problems use the sequential structure, and most problems use it in conjunction with one or more of the other logic structures. To illustrate how to use this structure, this chapter develops a sample problem solution demonstrating each of the computer problem-solving steps explained in Chapter 3.

A programmer who uses the sequential logic structure is asking the computer to process a set of instructions in sequence from the top to the bottom of an algorithm. In general, the form of the algorithm looks like this:

ModuleName (list of parameters)
1. *Instruction*
2. *Instruction*
3. . . .
. . .
XX.End, Exit, or *Return* (variable)

The algorithm begins with the *Start* instruction represented by the name, the list of parameters, and number of the module. The numbered instructions following the *Start* might be any of the instructions other than *Start* and *End/Exit/Return* depending on the problem. The form of the flowchart corresponds to the form of the algorithm (see Figure 5.1). The first block is the *Start* block. Other instruction blocks follow. These blocks can be of any type other than the flattened ellipse, which indicates *Start* or *End/Exit/Return.* The final block is the *End, Exit,* or *Return* block. See Figure 5.2 for an example of the algorithm and the flowchart for a solution that enters a name and an age into the computer and prints them to the screen.

The name of the module is *NameAge.* There are no parameters. The flowchart uses the flattened ellipse to indicate that this is the start of this module. The first instruction to be executed is the *Enter* instruction. The instruction is in the form of the word *Enter* followed by the list of data items whose values will be entered from the keyboard, a data block. The flowchart uses a parallelogram to designate that this is an input instruction. The second instruction to be executed is the *Print* instruction. This instruction is in the form of the word *Print* followed by the list of data items whose values will be printed to the screen. The flowchart uses the parallelogram to indicate that this is an output

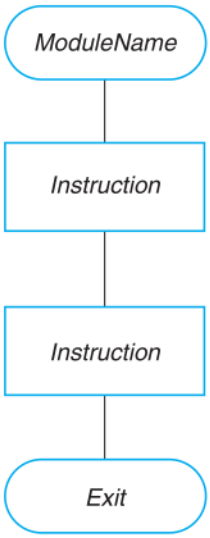


Figure 5.1 Flowchart Diagram for the Sequential Structure

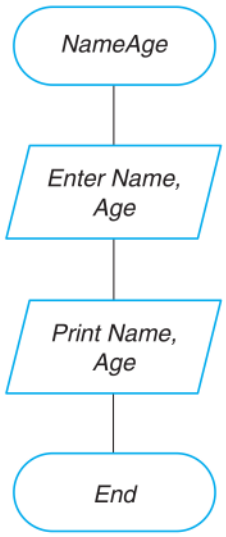
Algorithm	Flowchart	Pseudocode
NameAge 1. *Enter Name, Age* 2. *Print Name, Age* 3. *End*		*Enter Name, Age* *Print Name, Age* *End*

Figure 5.2 Algorithm, Flowchart, and Pseudocode to Enter and Print Two Variables

instruction. The input and output instructions both use the parallelogram in the flowchart. The last instruction is the *End* instruction. This signals the end of the list of instructions and to stop the execution of the program. The end is placed in a flattened ellipse in the flowchart. All of the start and ending instructions use the flattened ellipse in the flowchart. Notice that the algorithm instructions are numbered starting with the first instruction after the name of the module.

Solution Development

Solution Development

In Chapter 3, the tools for organizing problems were introduced. Now you will use them to solve problems from the business world. These organizing tools are used in the first six steps of developing a solution.

1. The problem analysis chart—helps you define and understand the problem, develop ideas for the solution, and select the best solution.
2. The interactivity chart—breaks the solution to the problem into parts.
3. The IPO chart—helps define the input, the output, and the processing steps.
4. The coupling diagram and the data dictionary designate the data flow between modules. The data dictionary records information on the items.
5. The algorithms define the steps of the solution.
6. The flowcharts are a graphic form of the algorithms.
7. The pseudocode presents a generic language representation of the algorithm.

The seventh step in solving the problem is to test the solution to catch any errors in logic or calculation. Remember, the charts are aids to the efficient development of a well-written program. They will not be perfect the first time around and will often need several revisions.

As you follow the solution to the sample sequential structure problem that follows, take the time to think through each step. Photocopy the forms in Appendix D to complete the seven steps. The solution to the following problem will be developed. This may seem like a lot of work for this simple of a problem; however, it is easier to learn on a simple short problem than on a long difficult one. If you can understand this problem and the use of the tools involved in the solution of this problem, you will be able to solve more difficult problems as they come up.

The Problem

Problem: Mary Smith is looking for the bank that will give the most return on her money over the next five years. She has $2,000 to put into a savings account. The standard equation to calculate principal plus interest at the end of a period of time is

$$Amount = P * (1 + I/M)^{\wedge}(N * M)$$

where P = *Principal* (amount of money to invest, in this case $2,000)
I = *Interest* (percentage rate the bank pays to the investor)
N = *Number of Years* (time for which the principal is invested)
M = *Compound Interval* (the number of times per year the interest is calculated and added to the principal)

(Refer to Table 2.5 to review the meanings of the operators used here.)

Problem Analysis

Problem Analysis

The first step in analyzing the problem is to understand what is needed and what is given and to separate them from all of the nonessential information in the problem. Write down the input in the Given Data section and the output in the Required Results section. Record the processing that the problem demands in the section of the problem analysis chart headed Required Processing. Finally, write possible solutions for the problem and any ideas related to the solution in the Solution Alternatives section. Generate solution ideas through brainstorming and creative thinking. When you are first learning problem-solving skills, brainstorming is best accomplished in groups of two to three people. The more ideas, the easier it is to develop a good solution.

Chapter 6

Problem Solving with Decisions

Overview

Objectives

When you have finished this chapter, you should be able to:

1. Develop problems using the decision logic structure in conjunction with the sequential logic structure.
2. Use problem-solving tools when developing a solution using the decision logic structure.
3. Use nested decision instructions to develop a problem solution.
4. Distinguish the different uses of straight-through, positive, and negative nested decision logic structures.

5. Convert a positive decision logic structure to a negative decision logic structure.
6. Develop decision tables given a set of policies.
7. Develop a decision logic structure from a decision table.
8. Develop problems using the case logic structure.
9. Use problem-solving tools when developing a solution using the case logic structure.

The logic structures will almost always be used in some combination with each other in a program. In this chapter you will learn how to use a second logic structure, the decision structure, as a way to design instructions. The decision structure is one of the most powerful because it is the only way that the computer can choose between two or more sets of actions. Without decisions, the computer would be nothing more than a fast calculator.

The decision logic structure is relatively easy to understand since it works much like the way humans think. Because of the complexity of some of the decisions a programmer has to design, however, it is often one of the most difficult structures to implement. The difficulty goes back to the fact that people make decisions without understanding the reasoning involved. Also, a decision can be written in many different ways, which can add to the confusion.

The Decision Logic Structure

If/Then/Else instruction

The decision logic structure uses the ***If/Then/Else*instruction.** It tells the computer that *If* a condition is true, *Then* execute a set of instructions, or *Else* execute another set of instructions. The *Else* part is optional, as there is not always a set of instructions if the conditions are false. When there are no instructions for true, a *Continue* statement must be used. The following algorithm instruction for the decision structure (the decision instruction) should be added to the list in Table 5.1. Notice the brackets and the indentation in the illustration of the algorithm of the decision instruction. Programmers use these features when they are writing instructions in the decision structure because they improve readability.

```
If <condition(s)>
Then
    True      <True instruction(s)>
Else
    False     <False instruction(s)>
```

The *True* instructions are processed when the resultant of the condition is *True,* and the *False* instructions are processed when the resultant of the condition is *False.* A condition can be one of four things: a logical expression, that is, an expression that uses logical operators (AND, OR, and NOT); an expression using relational operators (greater than, less than, greater than or equal to, less than or equal to, equal to, and not equal to); a variable of the logical data type (*True* or *False*); or a combination of logical, relational, and mathematical operators.

Some examples of conditional expressions are as follows:

1. $A < B$ (A and B are the same data type—either numeric, character, or string)
2. $X + 5 >= Z$ (X and Z are numeric data)
3. $E < 5$ or $F > 10$ (E and F are numeric data)
4. *DataOk* (*DataOk* is a logical datum)

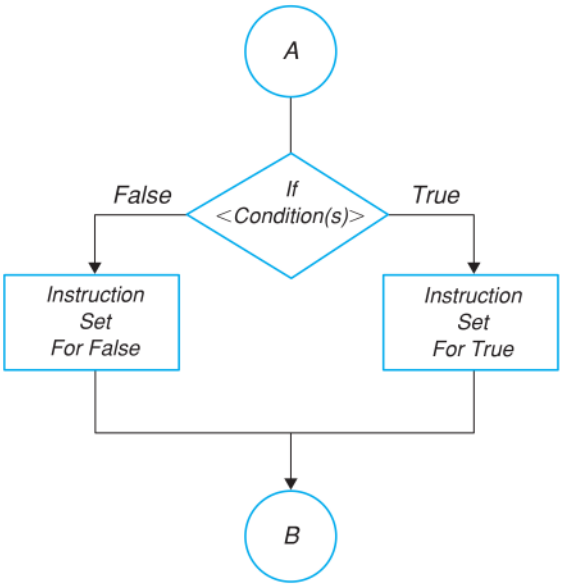


Figure 6.1 Flowchart Diagram of the Decision Structure

Logical operators are used to link more than one condition together. For example, consider a store policy requiring that to cash a check, a customer must have a driver's license, AND the check must be under $50. Two conditions are linked by the logical operator AND. (See Tables 2.6, 2.7 and 2.8 for the definitions and the hierarchy of all the operators.)

The programmer can set up the conditions for a decision through the use of operands and various operators. These conditions can be used alone or linked with other conditions for use in the instruction. The programmer derives the information needed to set up the condition(s) for a decision from the problem itself during problem analysis.

Figure 6.1 shows a simple flowchart illustrating the decision structure. In the figure, the resultant of the condition(s) can be *True* or *False* depending on the data. From the decision block (the diamond), there will be a flowline, a branch for each set of instructions: one branch for the instructions to be executed if the resultant is *True,* and another branch for the instructions to be executed if the resultant is *False.* There can never be more than two flowlines coming from a decision block since there are only two possibilities for the resultant of the condition(s). The *True* branch and the *False* branch can come from any of the lower three points of the diamond. It is best to be as consistent as possible in choosing the points of the diamond on which these branches are placed. The convention is to draw the *True* branch on the right and the *False* branch on the left or bottom, although it depends on the decisions to be made. This book follows that convention whenever possible. The processing flow from both the *True* branch and the *False* branch must connect to the rest of the processing somewhere before exiting the module. There can be no dangling flowchart blocks—blocks with no flowlines leading to another block. The processing must flow somewhere. Each module can have only one entrance and one exit. The only way to get to another module is through a *Process* instruction, that is, by telling the computer to process the instructions in another module and then return to continue processing in the current module.

Single Condition

A simple decision with only one condition and one action or set of actions (one instruction or set of instructions) for *True* and another for *False* is relatively simple. For example, assume you are calculating pay at an hourly rate and overtime pay (over 40 hours) at 1.5 times the hourly rate. The decision to calculate pay would be stated in the following way: *If* the hours are greater than 40, *Then* the pay is calculated for overtime, or *Else* the pay is calculated in the usual way (see Figure 6.2).

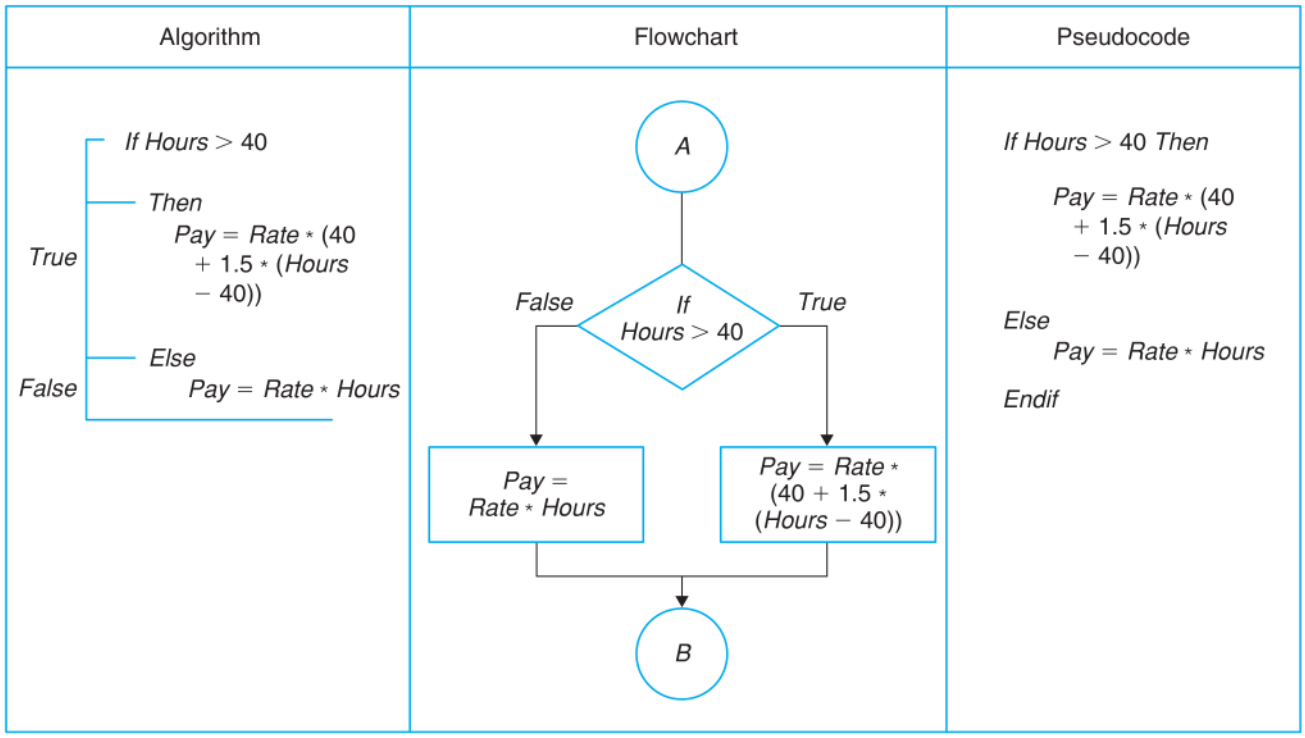


Figure 6.2 Single Condition—Two Possible Actions or Sets of Actions

Multiple Condition

Decisions in which you have *multiple* conditions that lead to one action or set of actions for *True* and another action or set of actions for *False* are slightly more complicated than those with single conditions. In these decisions you will use logical operators to connect the conditions. As with decisions based on a single condition, the resultant will determine whether the *True* or the *False* actions are taken.

The decision structure becomes more complicated as the number of conditions and/or the number of actions for a *True* or *False* resultant increases.

Multiple *If/Then/Else* Instructions

Multiple *If/Then/Else* Instructions

There are three types of decision logic you will use to write algorithms for solutions consisting of more than one decision. These types of decision logic include straight-through logic, positive logic, and negative logic.

Straight-through logic

Straight-through logic means that all of the decisions are processed sequentially, one after the other. There is no *Else* part of the instructions; the *False* branch always goes to the next decision, and the *True* branch goes to the next decision after the instructions for the *True* branch have been processed.

Positive logic

On the other hand, with positive logic not all of the instructions are processed. **Positive logic** allows the flow of the processing to continue through the module instead of processing succeeding decisions, once the resultant of a decision is *True.* Whenever the resultant is *False* (the *Else* part of the decision), another decision in the sequence is processed until the resultant is *True,* or there are no more decisions to process. At that time, the *False* branch processes the remaining instructions.

negative logic

Negative logic is similar to positive logic except that the flow of the processing continues through the module when the resultant of a decision is *False.* Whenever the

resultant is *True,* another decision is processed until the resultant is *False,* or there are no more decisions to process. At that time, the *True* branch processes the remaining instructions. Negative logic is the hardest to use and understand.

Some series of decisions will require a combination of two or more of these logic types. The use of each decision logic type will be further explained and illustrated later in this chapter.

nested If/Then/Else instructions

In algorithms containing multiple decisions, you may have to write **nested If/Then/Else instructions.** Decisions using positive and negative logic use nested *If/Then/Else* instructions. Decisions using straight-through logic do not. Nested *If/Then/Else* instructions are sets of instructions in which each level of a decision is embedded in a level before it; like nesting baskets, one goes inside the next. You use the nested *If/Then/Else* structure only when one of several possible actions or sets of actions are to be processed (see Figure 6.3). In composing an *If/Then/Else* instruction, draw a large bracket around each *If/Then/Else* in the algorithm to ensure that each instruction is in the proper place. The top part of the bracket indicates the condition. The middle section of the bracket indicates the *True* branch, or the *Then* part of the decision. The bottom part of the bracket indicates the *False* branch, or the *Else* part of the decision.

In Figure 6.3, the decision is *If PayType* = "Hourly" and says, Is the value of the variable *PayType* equal to the string constant "Hourly"? If the two are equal, then the resultant is *True* and the instruction set for the *True* is processed. If the resultant is *False,* the instruction set for the *False* branch is processed. The *True* branch uses a nested *If/Then/Else* instruction; the decision is embedded in the *True* branch of the first decision. This decision says, Are the hours worked greater than 40? If the answer is "yes," the resultant is *True,* and the pay is processed at the overtime rate. When the resultant is

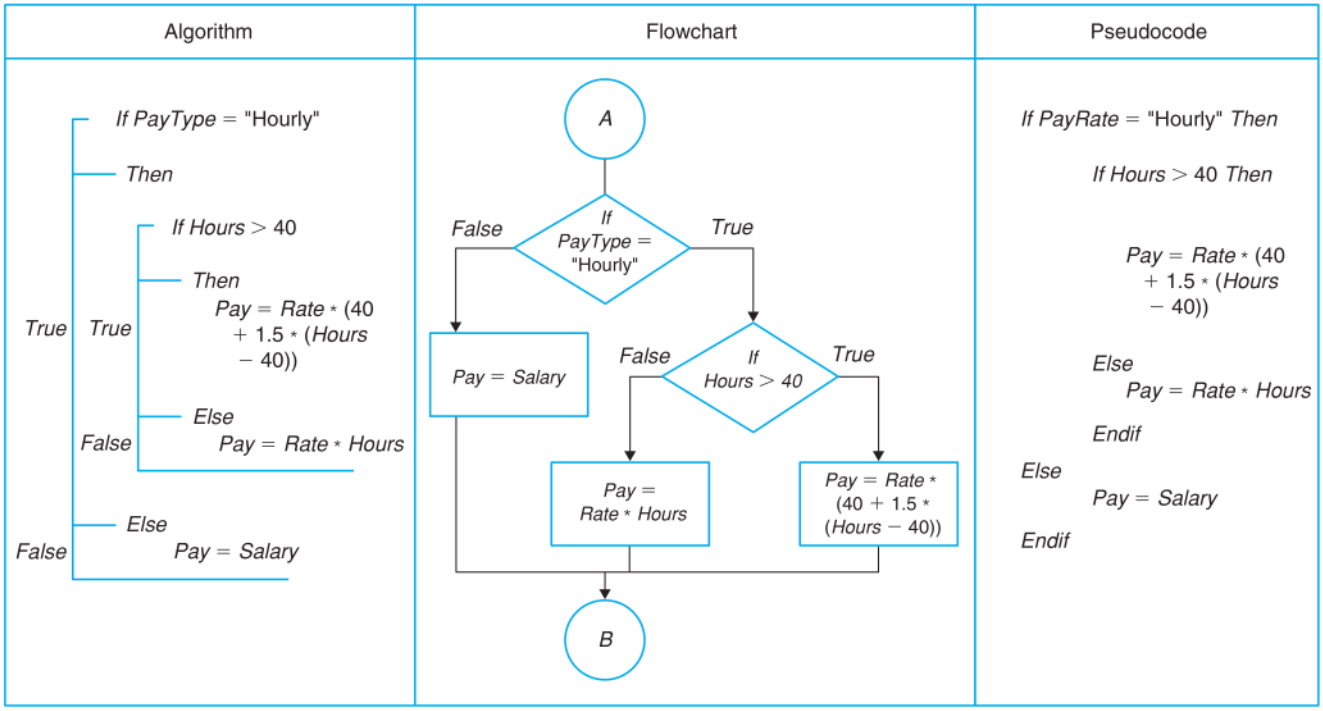


Figure 6.3 Nested *If/Then/Else* Instructions

False, that is, when the hours are not greater than 40, the pay is processed at the normal rate. When the resultant of the first decision is *False,* that is, when the *PayType* is not equal to "Hourly," then the pay is processed at the salary rate. (Notice that when the *PayType* is not equal to "Hourly," the embedded decision on the *True* branch cannot be processed—an example of negative logic.) A good programmer thoroughly tests algorithms that use decision logic with many sets of test data. Every path in every algorithm is checked, and a separate set of data is used for each path. It is important to test each path so errors will be eliminated.

Using Straight-Through Logic

Straight-Through Logic

With decisions following straight-through logic, all conditions are tested. To test a condition means to process a condition to get a *True* or *False* resultant. Straight-through logic is the least efficient of all types of decision logic because all the decisions must be processed. However, you must use it to solve certain problems—those that require two or more unrelated decisions and those in which all decisions must be processed. It is often used in conjunction with the other two logic types and in data validation (checking data to make sure it is correct). It is also used with languages that have limited features in their decision instruction.

The problem illustrated in Figure 6.4 is to find the amount to charge people of varying ages for a concert ticket. When the person is under 16, the charge is $7; when the person is 65 or over, the charge is $5; all others are charged $10. The conditions are the following:

Age	Charge
$Age < 16$	7
$Age >= 16$ and $Age < 65$	10
$Age >= 65$	5

Notice in Figure 6.4 that even when the age is less than 16, the other decisions must be processed.

In the algorithm and flowchart in Figure 6.4, notice that there is no *Else* in straight-through logic. If the resultant of the condition is *False,* then the processing flow drops to the next instruction. There is no need for the *Else* since there are no instructions for the *False* branch. If the resultant of the condition is *True,* then the set of instructions for the *True* part is processed, and the processing flow then drops to the next instruction. The set of conditions in Figure 6.4 is more efficiently executed by using either of the other types of logic since only one condition has to be met in order to execute the correct set of instructions.

The problem found in Figure 6.5 is to change the value of X to 0 when X becomes greater than 100, and to change the value of Y to 0 when Y becomes greater than 250. Each decision is independent of the other. The conditions are

$X > 100$	X becomes 0
$Y < 250$	Y becomes 0

Straight-through logic is required when all the decisions have to be processed, and when they are independent of each other. Based on these criteria, the only appropriate logic type for the decision in Figure 6.5 would be straight-through logic.

The following example is a similar problem, but with two sets of actions rather than one. When the age of a person is greater than 18, the person can vote; when the age

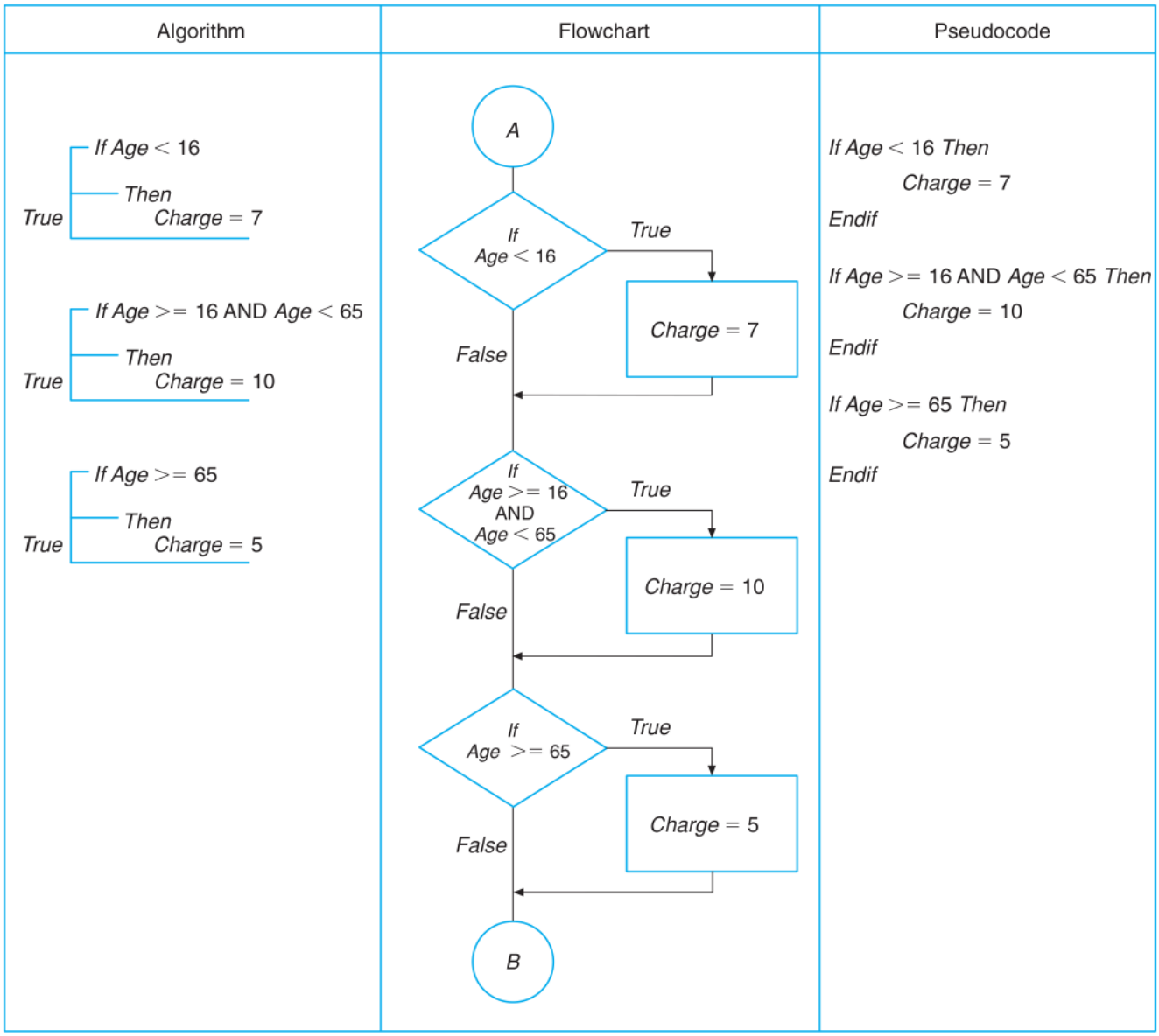


Figure 6.4 Straight-Through Logic—Example 1

is greater than 21, the person can have alcoholic beverages. Both decisions are contingent on the age of the person; however, the two actions take place at different ages. At the age of 20, a person can vote, but cannot be served alcoholic beverages. In this case, straight-through logic is required because all the decisions have to be processed and because they are independent of each other.

Using Positive Logic

Positive Logic

Positive logic is the easiest type for most people to use and understand because it is the way we think. Positive logic always uses nested *If/Then/Else* instructions. In general, when you use positive logic, you are telling the computer to follow a set of instructions

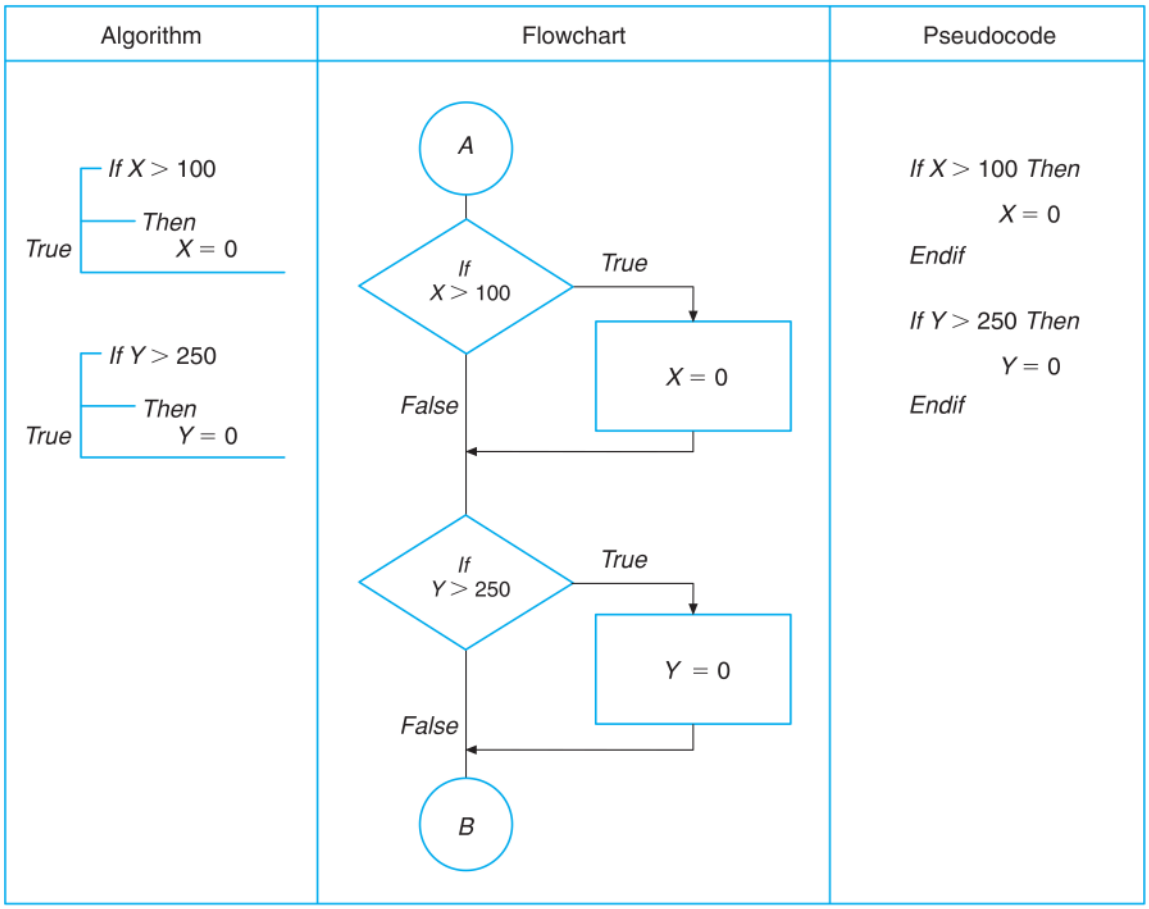


Figure 6.5 Another Example of Straight-Through Logic—Example 2

and continue processing if the condition is *True;* if the condition is not *True,* then the computer processes another decision. When you use this logic, no more decisions are processed once the resultant of a condition is *True.* Taking the same problem, and therefore, the same set of conditions as in Figure 6.4, Figure 6.6 shows how the algorithm would be structured using positive logic. Notice that there are fewer decisions processed than in Figure 6.4—two compared to three—even if all the decisions are processed. There is no need to process the third decision because the resultant of the third condition statement has to be *True* once you have processed the second statement and found it *False.* That is, when the age is NOT less than 16, it has to be greater than or equal to 16. Therefore, this test can be eliminated if nested *If/Then/Else* instructions are used. Likewise, if the age is NOT less than 65, it must be greater than or equal to 65. Therefore, the third decision in the set of conditions is not necessary.

Figure 6.7 illustrates another set of conditions using positive logic and a nested *If/Then/Else* structure. The problem illustrated in this figure is to calculate the commission rate for a salesperson, given the amount of sales. When the salesperson has sold less than or equal to $2,000 worth of goods, the commission is 2%. When the sales total is more than $2,000 and less than or equal to $4,000, the commission is 4%. When the sales total is more than $4,000 and less than or equal to $6,000, the commission is 7%.

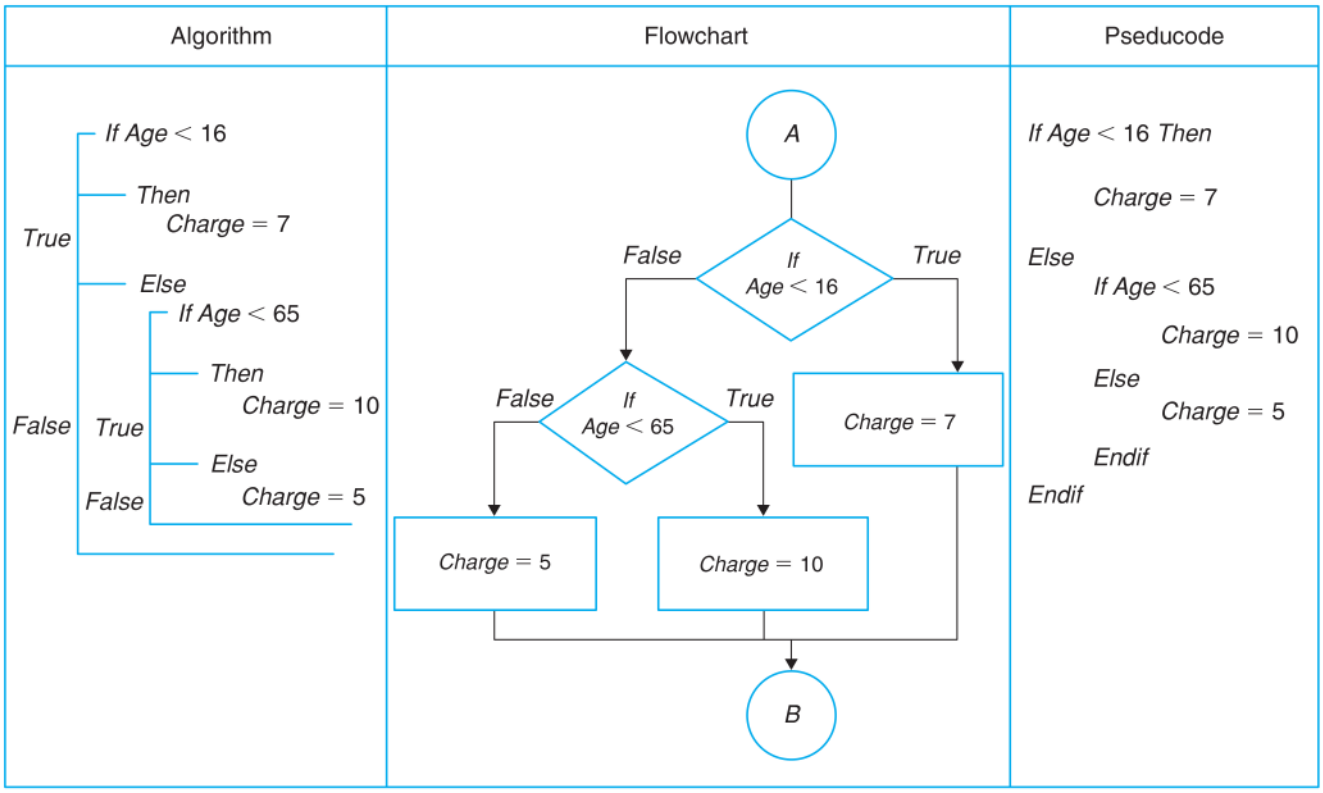


Figure 6.6 Positive Logic—Example 1

When the person has sold more than $6,000, the commission is 10%. The conditions are the following:

Sales	Commission
< = 2000	.02
2001–4000	.04
4001–6000	.07
> 6000	.10

Notice that in Figure 6.7 there are only three decisions. The fourth decision is not necessary; when the sales are NOT < = 6000 they have to be > 6000. Remember, when setting up an expression, you must have an operand on each side of the operator. *Sales* = 2001–4000 does not mean this is true for any sales amount between 2001 and 4000. The computer will subtract 4000 from 2001 first, then it will compare *Sales* to the resultant, –1999. The proper expression for this condition is *Sales* > 2000 and *Sales* < = 4000. Also, remember that if this is the second decision in a nested decision and the first decision is *Sales* < = 2000, the expression in the second decision will be *Sales* < = 4000 since *Sales* > 2000 will always be true. Any *Sales* > = 2000 will have dropped out of the nested *Ifs* during the first decision.

Figure 6.7 is easy to read and understand because of the structure of the algorithm: Brackets and indentation are used with positive logic to show the nested levels of the decision structure. These features are important to readability.

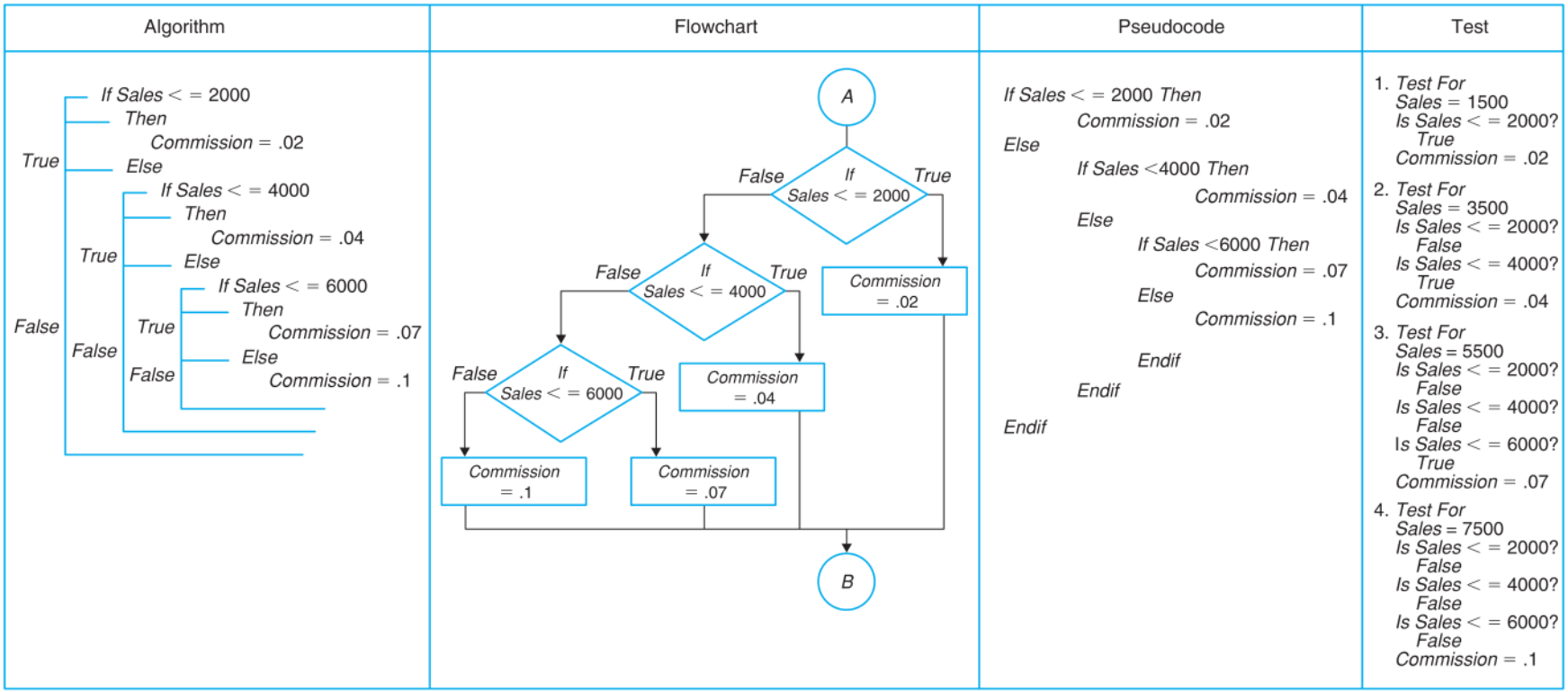


Figure 6.7 Positive Logic—Example 2

Chapter 7

Problem Solving with Loops

Overview

The Loop Logic Structure
Incrementing
Accumulating
While/WhileEnd
Putting It All Together
Repeat/Until
Putting It All Together
Automatic-Counter Loop
Putting It All Together
Nested Loops
Indicators
Algorithm Instructions and Flowchart Symbols
Recursion

Objectives

When you have finished this chapter, you should be able to:

1. Develop problems using the loop logic structure in conjunction with the decision and sequential logic structures.
2. Use the problem-solving tools when developing a solution using the loop logic structure.
3. Use counters and accumulators in a problem solution.

4. Use nested loop instructions to develop a problem solution.
5. Distinguish the different uses of three types of loop logic structures.
6. Use recursion in a simple problem.

The Loop Logic Structure

loop logic structure

A third logic structure for designing decisions is the loop structure. The loop logic structure is the repeating structure. Most problems in business involve doing the same task over and over for different sets of data, so this structure is extremely important. Through the loop structure, a program can process many payroll records or inventory items, or put a mailing list in alphabetical or zip code order. This structure is harder to understand than the decision structure, but easier to use. The main difficulty in using it as part of a solution is identifying what instructions should be repeated. Besides being used to repeat instructions in a solution, the loop structure is also used to return the processing to an earlier point in the solution. When it is used in this way, it is a replacement for the *GoTo* statement. (The *GoTo* is an instruction that tells the computer to transfer to another instruction in the solution instead of processing the next instruction in sequence.) Replacing the *GoTo* statement in this way increases the readability of the program.

There are three types of loop structures, and each of them can be written in another way as a set of instructions using the *If/Then/Else* structure and the *GoTo* instruction. However, the *If/Then/Else* is a decision logic structure and generally should not be used to facilitate a loop. The loop structure is easier to read and to maintain than the *If/Then/Else* and *GoTo* combination. Since you already understand the decision logic structure, the decision equivalence to each loop structure will be illustrated to aid your understanding of loops.

The first type of loop structure is the *While/WhileEnd* loop, which repeats instructions while a condition is *True* and stops repeating when a condition arises that is not *True.* The second type is the *Repeat/Until* loop, which repeats instructions while a condition is *False* or until a condition is *True.* The third is the automatic-counter loop in which a variable is set equal to a given number and increases in equal given increments until it is greater than an ending number. Each of the three loop structures has a specific use according to the language and/or the problem.

The algorithm and the flowchart differ with each type of loop structure. It is extremely important to indent the instructions in algorithms using the loop structure—as it is with the decision structure—to improve readability. Also, nested loops are common, and in these solutions indentation with bracketing allows you to easily identify each loop.

Several standard types of tasks are accomplished through the use of the loop structure. Two of these are counting (also called incrementing and decrementing) and accumulating (also called calculating a sum or a total). In both tasks a number is added or subtracted from a variable and the result is stored back into the same variable. The basic difference between counting and accumulating is in the value that is added or subtracted. When counting, the value is a constant; when accumulating, the value is a variable. In each case the resultant variable is assigned the value of zero before starting the loop. This is called *initializing* the variable.

An updated table of the Algorithm instructions and Flowchart symbols is found in Table 7.1 on pages 167–168. These have either been introduced in the previous chapters or will be introduced in this chapter. Please use this table as a reference to the instructions you will be using in this chapter.

Incrementing

incrementing

The task of **incrementing,** or counting, as we said, is done by adding a constant, such as 1 or 2, to the value of a variable. To write the instruction to increment a variable, you use an assignment statement. For example,

$$Counter = Counter + 1 \text{ or } C = C + 1$$

when counting by ones. An assignment statement allows the variable on the left side of the equal sign to be assigned the value of, or to be replaced by, the resultant of the expression on the right side of the equal sign. The counter or incrementor instruction is a special version of the assignment instruction. This instruction enables the programmer to count the number of items, people, temperatures, and so on, as part of the solution to a problem.

Notice the structure of the assignment instruction for incrementing. The same variable name is on both sides of the equal sign; the amount the variable is to be incremented follows the plus sign. This instruction takes the value of the counter, adds one to it, and then replaces the old value of the counter with the new. The increment can be one, two, or any other constant, including negative numbers if you want to decrement rather than increment. In this example *Counter* or *C* must be initialized to zero before starting the loop.

Accumulating

accumulating

Another task that a program must often perform is **accumulating,** or summing, a group of numbers. The process of accumulating is similar to incrementing, except a variable instead of a constant is added to another variable, which holds the value of the sum or total. The instruction for accumulating is the following:

$$Sum = Sum + Variable \text{ or } S = S + V$$

For example, the instruction to find total sales would be

$$TotalSales = TotalSales + Sales$$

Like the instruction to increment, the instruction to accumulate is a special version of the assignment instruction. As in incrementing, the same variable name appears on both sides of the equal sign (*TotalSales*) but in accumulating, a variable rather than a constant is on the right-hand side of the plus sign (*Sales* in this case). In other words, with an accumulator you are adding a variable (the item you are accumulating) to the value of another variable, which holds the sum or total. In these examples *Sum* and *TotalSales* must be initialized to zero before starting the loop.

Products

Calculating the product of a series of numbers is similar to finding the sum of a series of numbers with two exceptions. First, the plus sign is replaced by the multiplication sign (*). Second, the product variable must be initialized to 1 instead of 0. For example

$$Product = 1$$
$$Product = Product * Number$$

Finding the product of a series of numbers is used very little. However, it will be used in the discussion on recursion at the end of this chapter.

While/WhileEnd

while/whileend

The first of the three types of loop structures is the **While/WhileEnd** structure. This type of loop tells the computer that while the condition is *True,* repeat all instructions between the *While* and the *WhileEnd.* The form of the algorithm is the following:

```
While <Condition(s)>
   Instruction
   Instruction
   .
   .
WhileEnd
```

When you design *While/WhileEnd* loops, use brackets and indentation to improve the readability of the algorithms. In the illustration, notice how the bracket connects the beginning (*While*) and the end (*WhileEnd*) of the loop, and how the instructions are indented within the brackets to make the algorithm easily readable. The general form of the *While/WhileEnd* flowchart is shown in Figure 7.1.

The symbol used for the *While* part of the instruction is the diamond, the symbol for a decision. Since this flowchart shows only part of an algorithm, on-page connectors are used. The completed algorithm would have one part that would connect to *A* and another part that would connect to *B.*

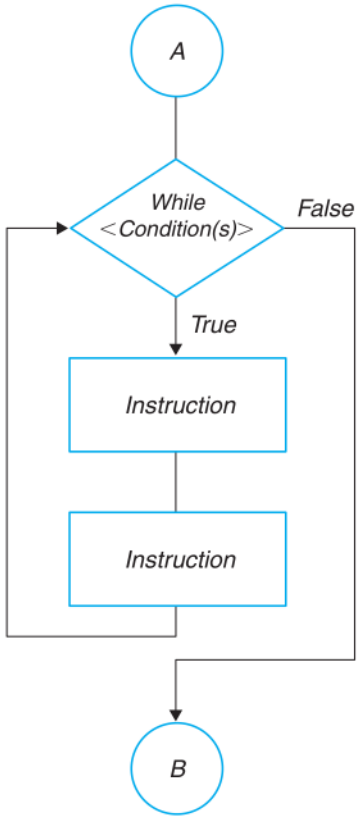


Figure 7.1 Flowchart Diagram of *While/WhileEnd*

At the beginning of the *While/WhileEnd* loop, the program processes the condition in order to decide whether to execute the loop instructions, as illustrated in the diamond at the top of the flowchart. When the resultant of the condition is *False* at the start, the instructions within the loop will not be executed. When the resultant of the condition is *True,* the complete set of instructions will be executed, and then the computer will go back to the beginning of the loop and process the condition again. The loop will repeat until the resultant of the condition is *False,* at which time the processing will continue with the instruction following the *WhileEnd.*

Figure 7.2 shows the decision-structure equivalent to the *While/WhileEnd* loop structure. Notice that the decision structure requires a separate *GoTo* instruction to return to the top of the loop. Also notice that when the condition is *False,* the line of execution transfers to the instruction after the loop instructions.

Use the *While/WhileEnd* structure when you do not know the number of times the instructions will be repeated, or if there are cases when the instructions in the loop should not be processed. In those cases, the resultant of the condition is *False* at the start of the loop, so the loop is not processed. Examples of such cases are when you are entering information on clients and you don't know the number of clients; when you are finding

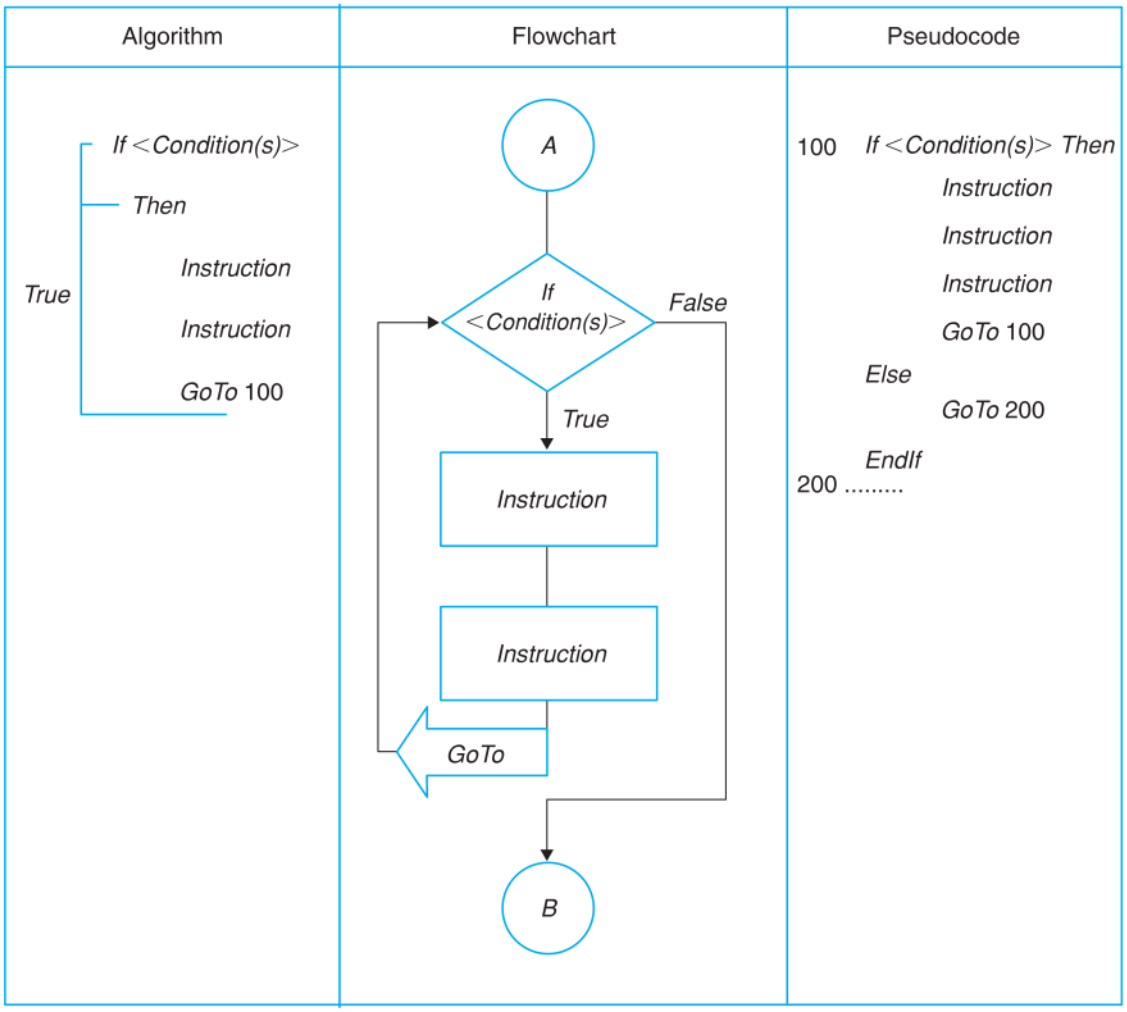


Figure 7.2 Decision Equivalent to *While/WhileEnd*

the total amount of sales for the day and you don't know the number of sales; or when you are calculating test averages for each student in a class. In the case of calculating test averages, the loop instructions should not be executed for students who for some reason have no test grades. To calculate the average you would accumulate and count the test scores and then divide the total score by the number of tests. Since the number of grades would be zero for the student who didn't take any tests, the computer would try to divide by zero, and an error would result. The *While/WhileEnd* loop would enable the processing to pass over the student and go on to the next student.

Putting It All Together

Example

Problem: Create the algorithm and the flowchart to find the average age of all the students in a class.

Figure 7.3 demonstrates the use of a *While/WhileEnd* structure to calculate the average age of a group of people. The first instruction initializes the value of the *Sum* (the total of all ages) to zero. The second instruction initializes the value of the counter variable to zero. Next the first age is entered. An age must be entered before the loop so that the condition has data to use to get a resultant. This is called a ***primer Read*** because it gives the *While/WhileEnd* loop a valid value for the variable *Age* in order for the conditions to be true the first time through the loop. If there is no primer *Read,* the value of *Age* is unknown. The next read is at the end of the loop for two reasons. First, the first value of *Age* must be included in the calculations. Second, the test for the trip value (the value of age that makes the condition false and signals the end of the list of students) must be just before the condition, so the trip value will not be calculated as part of the average age. The *While* begins the loop and processes the condition $Age <> 0$. When the resultant is *True,* the instructions in the loop are processed; when the resultant is *False,* the processing continues with the calculation of the average. The first loop instruction accumulates the ages by adding each age to the total, followed by an instruction that counts the people by adding one to the counter variable. The final instruction in the loop enters the next age. The processing then returns to process another condition. The last age should be zero. This zero is called a *trip value,* or a *flag.* It allows the value of *Age* to control when to stop the looping process and continue with the rest of the solution. When the looping process stops, the processing continues at the calculation of the average age, and finally, the printing of the average age.

primer read

The data for this problem can be entered from the keyboard; the user enters a 0 when there are no more ages to enter.

Repeat/Until

repeat/Until

The second type of loop structure is the ***Repeat/Until*** structure. This type of loop tells the computer to repeat the set of instructions between the *Repeat* and the *Until,* until a condition is *True.* There are two major differences between this instruction and the *While/WhileEnd.* First, in the *While/WhileEnd* loop, the program continues to loop as long as the resultant of the condition is *True;* in the *Repeat/Until* loop, the program stops the loop process when the resultant of the condition is *True.* Second, in the *While/WhileEnd* loop, the condition is processed at the beginning; in the *Repeat/Until* loop, the condition is processed at the end. When the condition is processed at the end,

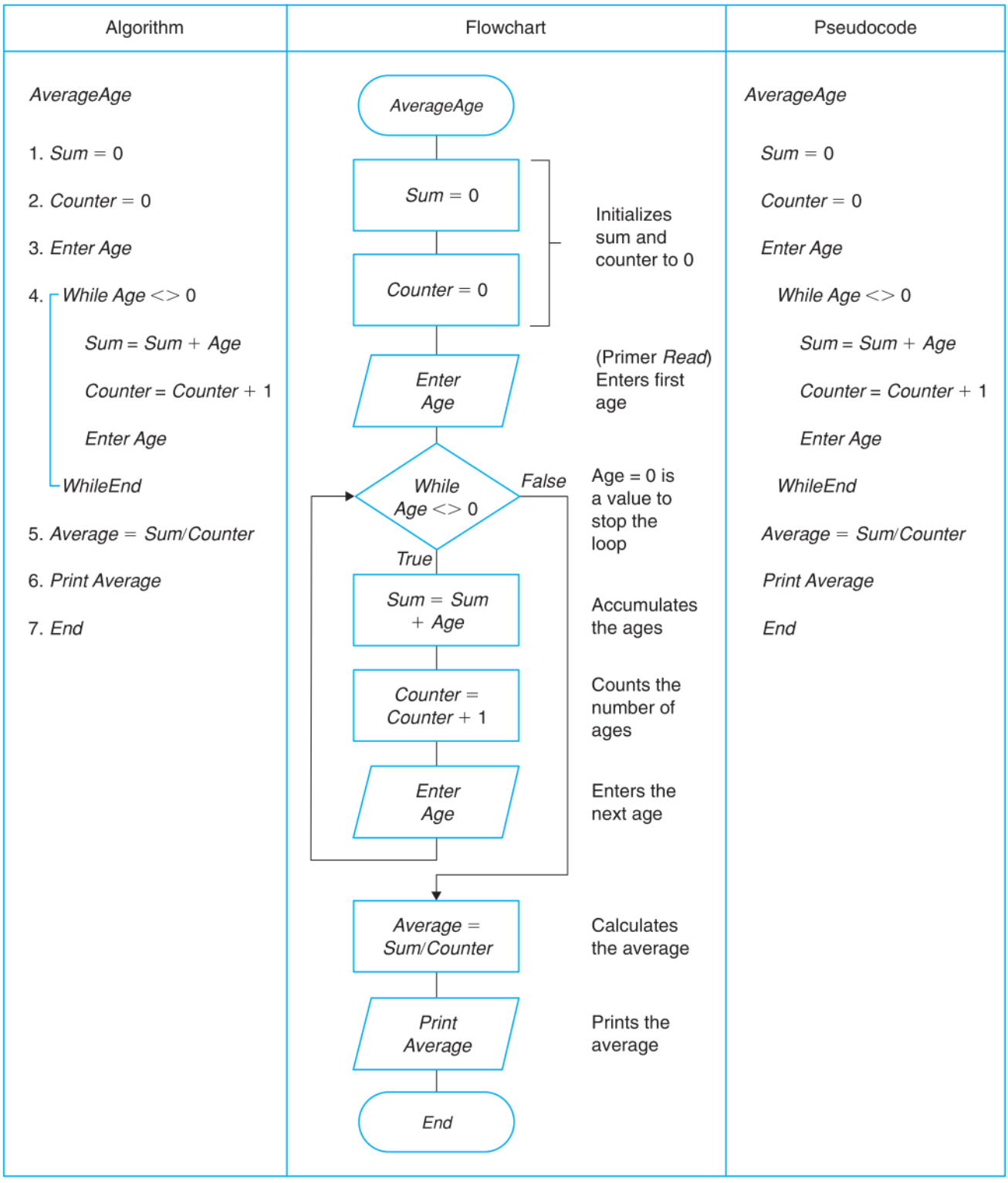


Figure 7.3 Average Age of a Class—*While/WhileEnd*

the instructions in the loop are processed entirely at least once, regardless of whether it is necessary.

It is important that you be able to distinguish between the loop structures so you can use them appropriately. Both the setup and the uses for the two loop structures are different. When you are using the *While/WhileEnd,* you must initialize the data so that the resultant of the condition is *True* the first time through the loop. Otherwise, the loop instructions will never be processed. With the *Repeat/Until,* you can set the operands of the conditions anywhere within the loop since the condition is processed at the end. If there is any reason that the loop instructions should not be processed the first time, then the *While/WhileEnd* must be used because the *Repeat/Until* always processes the instructions at least once.

The format of the *Repeat/Until* algorithm is the following:

```
Repeat
  Instruction
  Instruction
  .
  .
Until<Condition(s)>
```

Here again, use brackets and indentation to improve readability. Figure 7.4 shows the form of the flowchart. When you are designing a *Repeat/Until* flowchart, write the word *Repeat* above the first instruction of the loop.

Figure 7.5 shows the equivalent decision structure. Unlike the *While/WhileEnd* decision equivalent, in which the processing of the condition is always at the beginning of the instructions to be repeated, with the *Repeat/Until* decision equivalence, the processing of

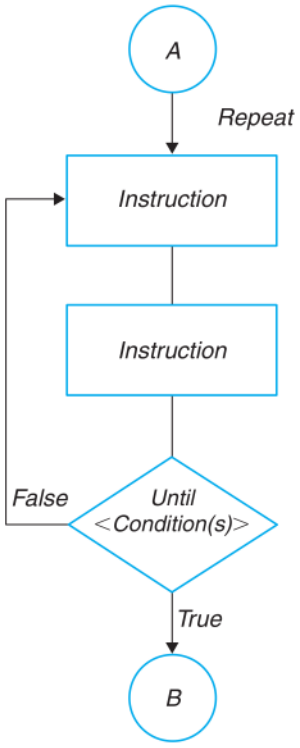


Figure 7.4 Flowchart Diagram of *Repeat/Until*

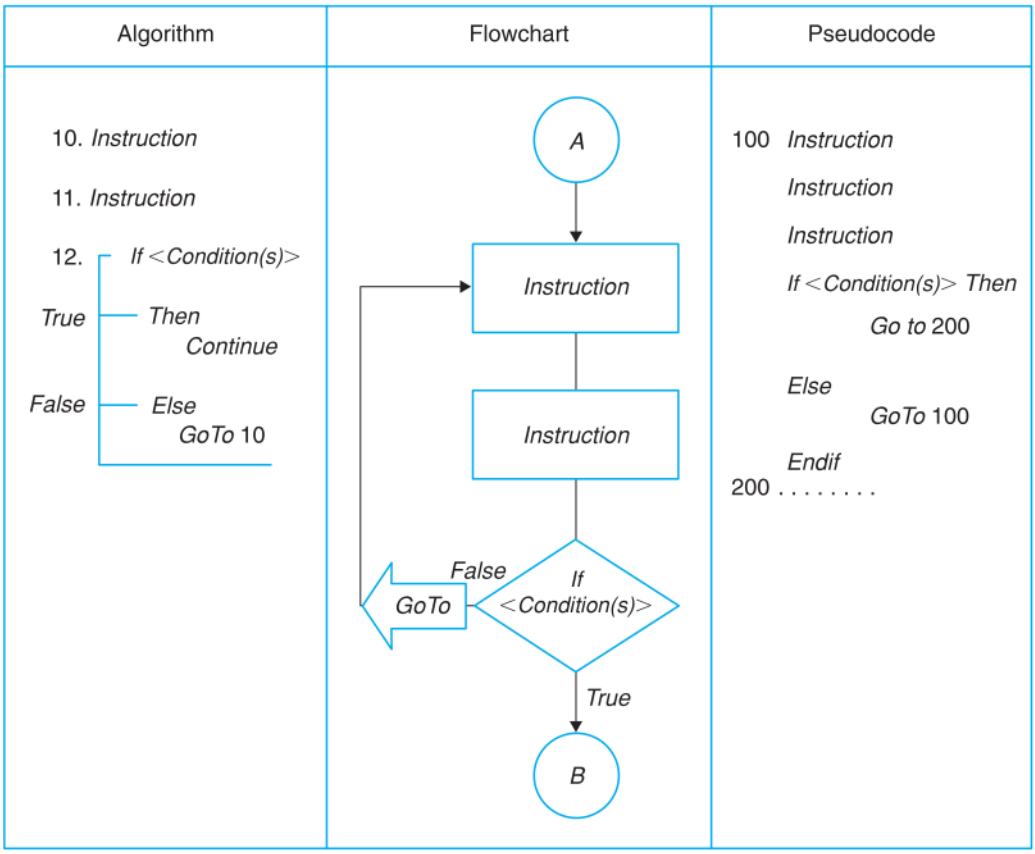


Figure 7.5 Decision Equivalent to *Repeat/Until*

the condition always takes place at the end of the instructions to be repeated. Also notice that there is no evidence of a loop until the condition is reached. The loop structure provides a solution that is easier to read and understand.

Use the *Repeat/Until* structure when you do not know the number of times the instructions will be executed, when you know they will be executed at least once, or when you do not know the condition for repeating until after the instructions in the loop are processed. Usually, the problem will dictate which loop to use.

Putting It All Together

Using the *Repeat/Until* loop structure, this section illustrates the same problem as in the Putting It All Together section for the *While/WhileEnd* loop.

Example

Figure 7.6 demonstrates the use of the *Repeat/Until* structure to calculate the average age of a group of people. The first instruction initializes the value of the *Sum* (the total of all ages) to zero. The second instruction initializes the value of the counter variable to zero. Next the first age is entered. An age must be entered so that there is a value for *Age* to use in the calculations the first time through. The data must be entered just before processing the condition, or the trip values will be used in the last calculations in the solution. This is the primer *Read.* However, unlike the *While/WhileEnd* loop structure, when a zero is entered at this place in the processing, the instructions within the loop will be executed at least once, depending on what is entered at the second *Enter*

Algorithm	Flowchart	Pseudocode

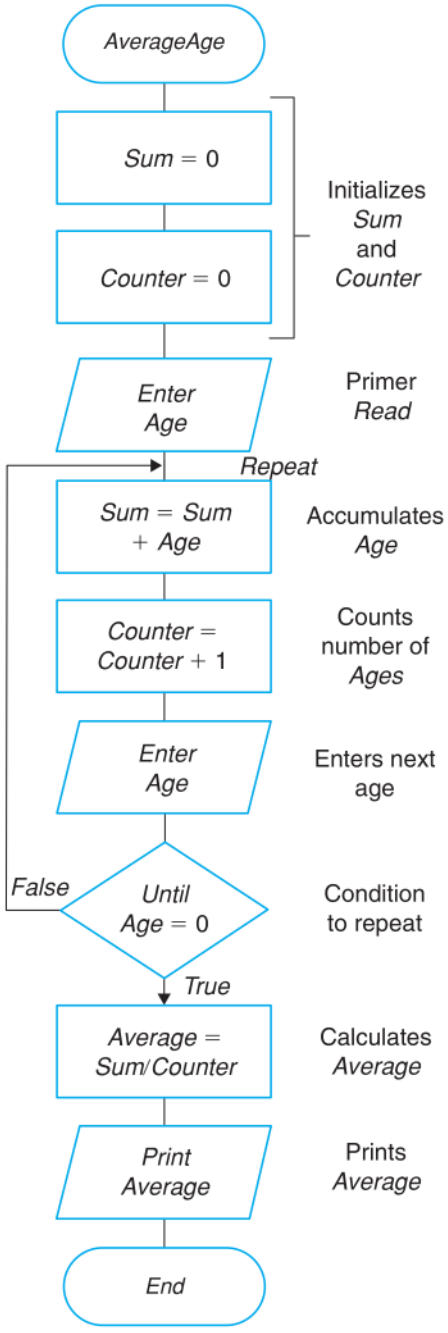
Algorithm

AverageAge

1. *Sum* = 0
2. *Counter* = 0
3. *Enter Age*
4. *Repeat*
   - *Sum* = *Sum* + *Age*
   - *Counter* = *Counter* +1
   - *Enter Age*

   Until Age = 0
5. *Average* = *Sum*/*Counter*
6. *Print Average*
7. *End*

Flowchart



Pseudocode

AverageAge

Sum = 0
Counter = 0
Enter Age
Repeat
    Sum = *Sum* + *Age*
    Counter = *Counter* +1
    Enter Age
Until Age = 0
Average = *Sum*/*Counter*
Print Average
End

Figure 7.6 Average Age of a Class—*Repeat/Until*

instruction. For this reason, the *While/WhileEnd* loop structure is often the preferred choice. The *Repeat* begins the loop. The first instruction in the loop accumulates the ages by adding each age to the total, followed by an instruction that counts the people by adding 1 to the counter variable. The last instruction in the loop enters the next age. The next instruction is the loop-end instruction. This instruction processes the condition *Age* = 0. When the resultant is *False* the instructions in the loop are processed; when the resultant is *True* the processing continues with the calculation of the average. The last age is a zero, the trip value or dummy value. It allows the value of *Age* to control when to stop the loop process and continue with the rest of the solution. When the looping process stops, the processing continues at the calculation of the average age, and finally, the printing of the average age.

The programmer or the language dictates which loop structure to use. Sometimes the *While/WhileEnd* and the *Repeat/Until* could be used with equal efficiency. The choice is then a matter of the programmer's preference.

Automatic-Counter Loop

automatic-counter loop

The third type of loop structure is the **automatic-counter loop.** This type of loop increments or decrements a variable each time the loop is repeated. To design an automatic-counter loop, the programmer uses a variable as a counter that starts counting at a specified number and increments the variable each time the loop is processed. The amount to be incremented is specified by the instruction. The set of instructions within the loop repeats until the counter is greater than an ending number. The beginning value, the ending value, and the increment value may be constants, variables, or expressions (calculations). They should not be changed during the processing of the instructions in the loop. They may be changed after the loop is completed. The test for whether to process the loop instructions in an automatic-counter loop will be found at the beginning or at the end of the loop depending on the language or the version of the language.

The form of the algorithm for the automatic-counter loop is the following:

```
Loop: Counter = Begin To End Step StepValue
  Instruction
  Instruction
  .
  .
  .
Loop-End: Counter
```

Here again, a bracket connects the beginning (*Loop*) and the end (*Loop-End*) of the algorithm, and the instructions are indented to improve readability, just as in the other loop structures.

The flowchart in Figure 7.7 shows the automatic-counter loop structure. *Counter* is the variable used for the counter, *Begin* is the beginning value, *End* is the ending value, and *StepValue* is the value by which the counter is to be incremented. To decrement, *StepValue* needs to be a negative number and *Begin* must be greater than *End.*

Figure 7.8 shows the decision equivalents to the automatic-counter loop structure. There are two solutions illustrated since there are two places the decision can be processed: at the beginning of the loop or at the end of the loop. Which decision equivalent

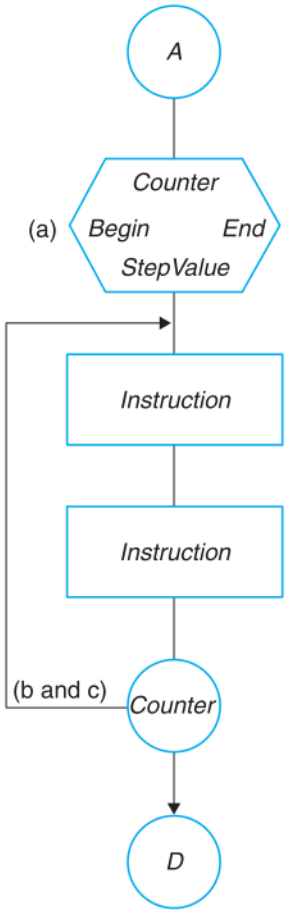


Figure 7.7 Flowchart of Automatic-Counter Loop

would be used depends on the language or the version of the language. You will not find both versions in the same language or language version.

Figure 7.9 shows the equivalent algorithms and flowcharts of the *While/WhileEnd* and *Repeat/Until* loops for the automatic-counter loop. Note that both these loops require a counter instruction (*Counter = Counter + StepValue*) and initialization of the counter (*Counter = Begin*). The decision for both loops involves the use of the ending value (*While Counter <= End and Until Counter > End*). The automatic-counter loop takes care of these instructions automatically. The *While/WhileEnd* loop executes the test at the beginning of the loop, and the *Repeat/Until* executes it at the end. When *Begin* is greater than *End* at the beginning of the loop, the *While/WhileEnd* will not execute any internal instructions, whereas the *Repeat/Until* will execute the instructions once. When you know the starting and ending values of the counter and you need a counter to run through a group of instructions multiple times, the use of the automatic-counter loop is preferred.

There are rules that the computer follows when processing the automatic-counter loop. The rules are simple, and it is important that the user understands and follows them when testing a solution.

When incrementing the counter, remember the following rules:

1. When the computer executes the *Loop* instruction, it sets the counter equal to the beginning number (see a, a_1, and a_2 in Figures 7.7 and 7.8).

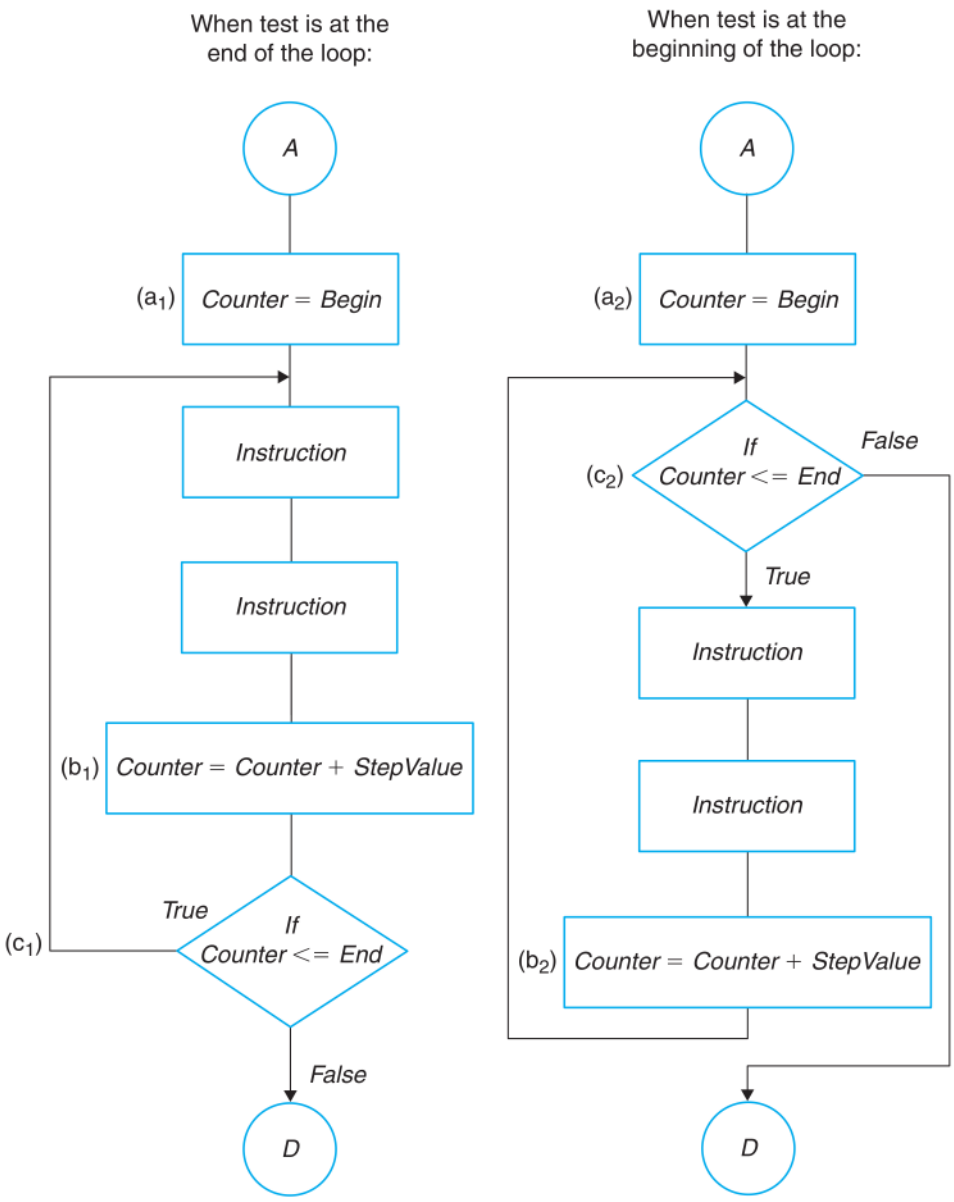


Figure 7.8 Decision Equivalents to Automatic-Counter Loop

2. When the computer executes the *Loop-End,* it increments the counter. Since the counter is usually incremented before the condition is processed, this book assumes the incrementing of the counter at *Loop-End* (see b, b_1, and b_2 in Figures 7.7 and 7.8).
3. When the counter is less than or equal to the ending number, the processing continues at the instruction that follows the *Loop* instruction. When the counter is greater than the ending number, the processing continues at the instruction that follows the *Loop-End* instruction (see c, c_1, and c_2 in Figures Figures 7.7 and 7.8).

There are two differences in these rules when decrementing the counter. First, the counter is decremented instead of incremented at the end of the loop. Second, when the counter is greater than or equal to the ending value, the loop continues, instead of when

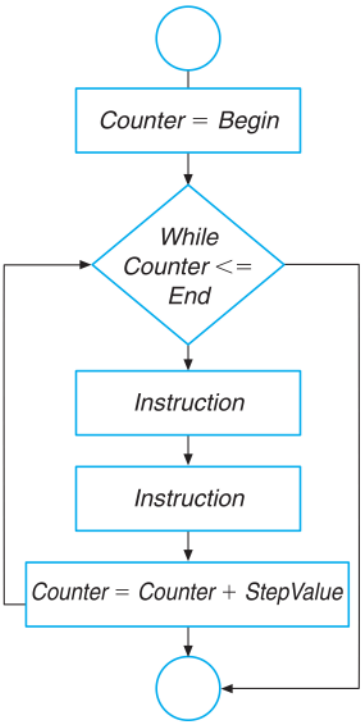
Algorithm	Flowchart	Pseudocode
10. *Counter = Begin* 11. *While Counter <= End* *Instruction* *Instruction* *Counter = Counter + StepValue* *WhileEnd* 12		*Counter = Begin* *While Counter <= End* *Instruction* *Instruction* *Counter = Counter + StepValue* *WhileEnd*

Figure 7.9a *While/WhileEnd* Loop Equivalent of the Automatic-Counter Loop

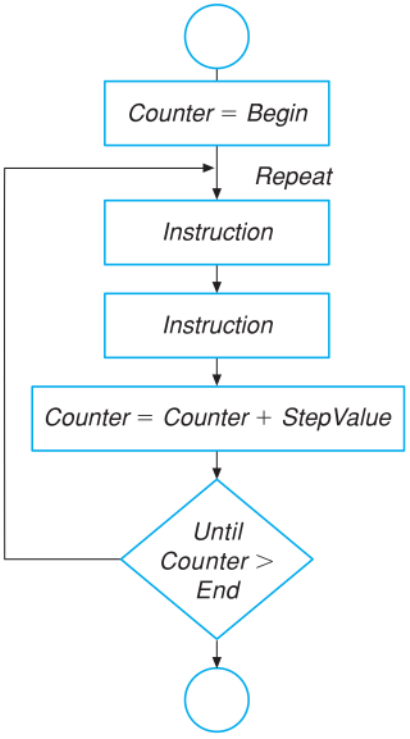
Algorithm	Flowchart	Pseudocode
10. *Counter = Begin* 11. *Repeat* *Instruction* *Instruction* *Counter = Counter + StepValue* *Until counter + End* 12		*Counter = Begin* *Repeat* *Instruction* *Instruction* *Counter = Counter + StepValue* *Until counter > End*

Figure 7.9b *Repeat/Until* Loop Equivalent of the Automatic-Counter Loop

it is less than or equal to the ending value when incrementing. It is important to understand these rules in order to use this structure correctly.

Use the automatic-counter loop structure when you know from the start the number of times the loop will be executed—for example, when you know the exact number of people in the group for the average-age problem. Do not use a decision to exit from this loop. A decision cannot be used to exit a loop since to do so would require a *GoTo* instruction. *GoTo* instructions tend to decrease readability, so they are used as little as possible in structured programming. If a decision instruction is necessary, then consider using one of the other types of loop structures for your solution. Counter loops are overused because they are easy to code. A good programmer is careful not to fall into this trap. It is important to choose the best loop structure for the problem.

Putting It All Together

Example

This section illustrates the same problem, using the automatic-counter structure that the previous two Putting It All Together sections did, using the other two types of loop instructions. The solution calculates the average age of a group of people. The programmer would use the automatic-counter loop structure for this solution only if the number of students in the class—in this case, 12—is known.

In the automatic-counter loop in Figure 7.10, as in the other two loop structures, the first two instructions initialize the variables *Sum* and *Counter.* The next instruction begins the loop. The *A* is the counter variable. The counter begins at 1 and is incremented by 1 until the counter is greater than 12, so the loop will be repeated 12 times. Notice that there is no increment value. When the increment value is left out, the computer assumes the value to be 1. When the counter increases to a value greater than 12, the average age is calculated and printed. Also notice the absence of the primer *Read.* The primer *Read* is not necessary because there is no trip value needed since the number of students is known. Whenever the number of items to be counted is known, the automatic-counter loop is preferred. If that number is not known, then one of the other loop structures is preferred.

Nested Loops

Nested Loops

Loops can be nested like decisions can. Each loop must be nested inside the loop just outside it. The general rules regarding loops, such as where the condition is processed and how indentation and brackets are used, hold true for nested loops as well as single loops. The inner loops do not have to be the same types of loop structures as the outer loops; that is, a *While/WhileEnd* may be nested inside a *Repeat/Until* loop, or vice versa.

Examples of nested loops are found in Figure 7.11. Notice that the nested loop in each flowchart results in the same output. The output for all four nested loops is the following:

OuterLoop *(outer loop counter)*	InterLoop *(inter loop counter)*
1	1
1	2
1	3
2	1
2	2
2	3

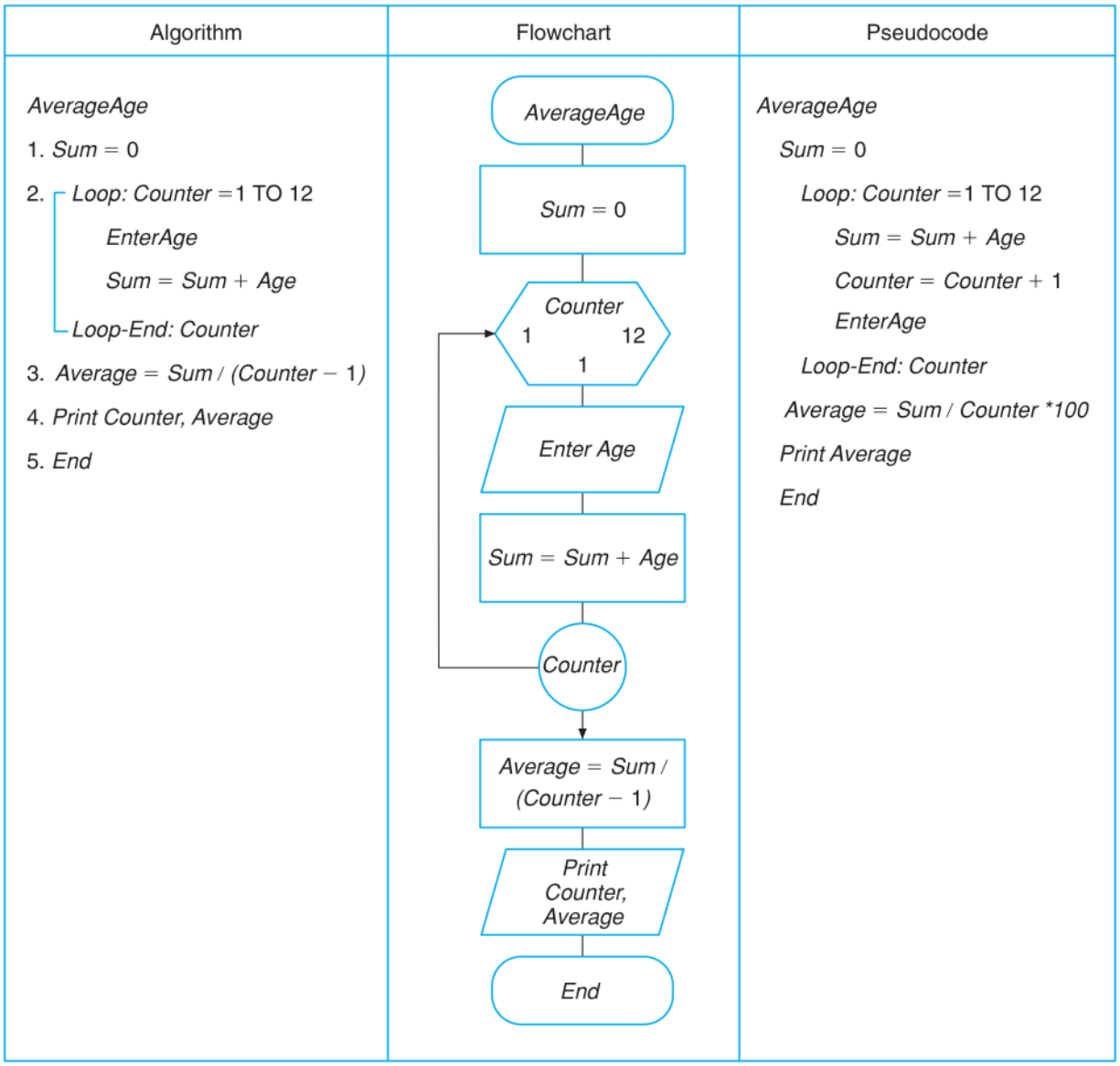


Figure 7.10 Example of an Algorithm and Flowchart for a Problem Using an Automatic-Counter Loop

In the first flowchart in Figure 7.11, the nested loops are both *While/WhileEnd* structures. The inner loop will be processed as long as the value of InnerLoop is less than 3. The inner loop increments InnerLoop and, later, prints the values of OuterLoop and InnerLoop. During the processing of the inner loop, the value of OuterLoop does not change. When InnerLoop becomes equal to 3, the inner loop has completed the processing, and the processing continues with the outer loop. The OuterLoop increments OuterLoop, sets InnerLoop equal to 0, and then processes the InnerLoop. When OuterLoop becomes equal to 2, processing continues at *B*. This whole process is essentially what happens in the other three examples of nested loops. The outer loop increments OuterLoop, sets InnerLoop equal to 0 or 1, and processes the inner InnerLoop loop. The InnerLoop increments InnerLoop and prints the values of OuterLoop and InnerLoop.

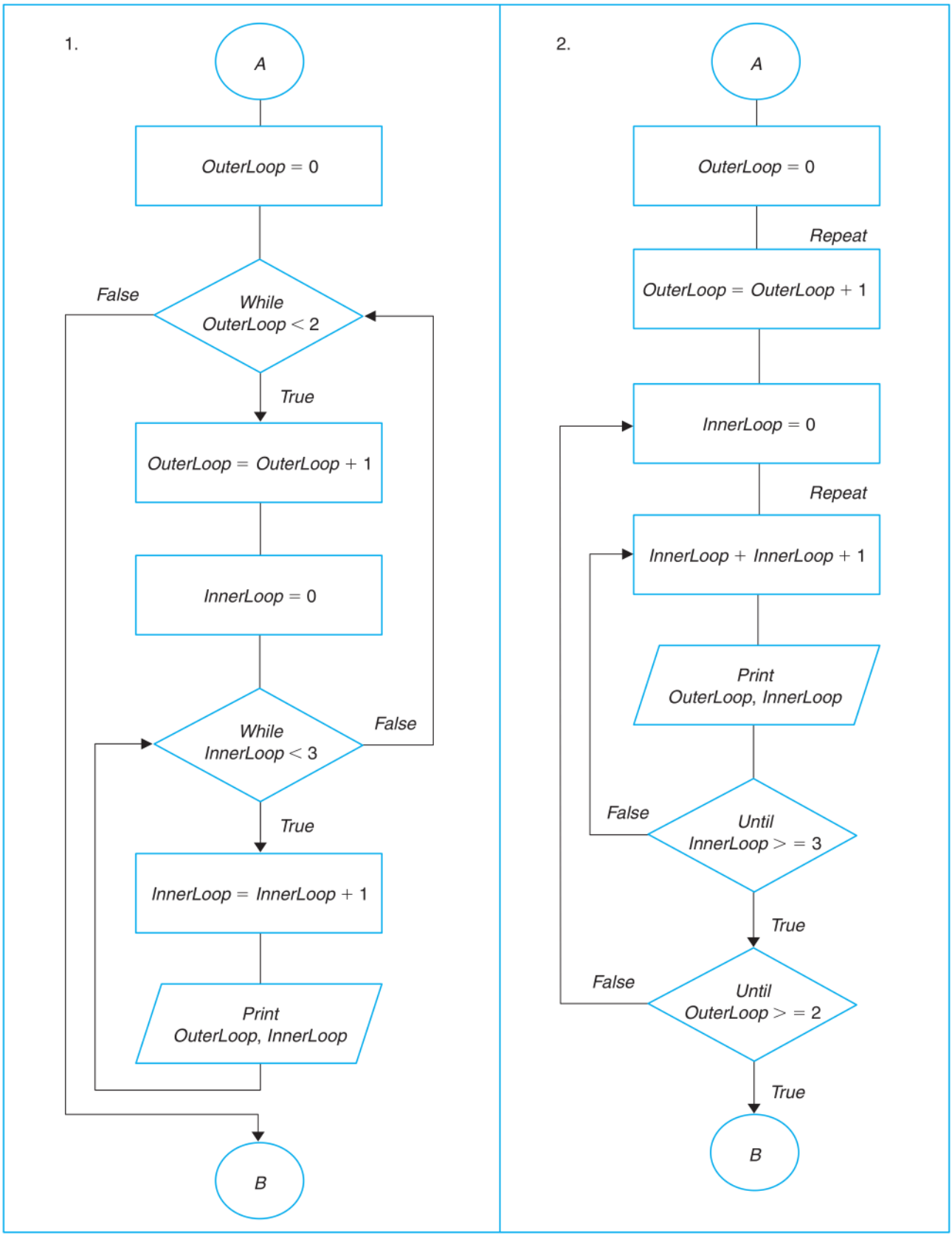


Figure 7.11 Nested Loops*(continued on page 166)*

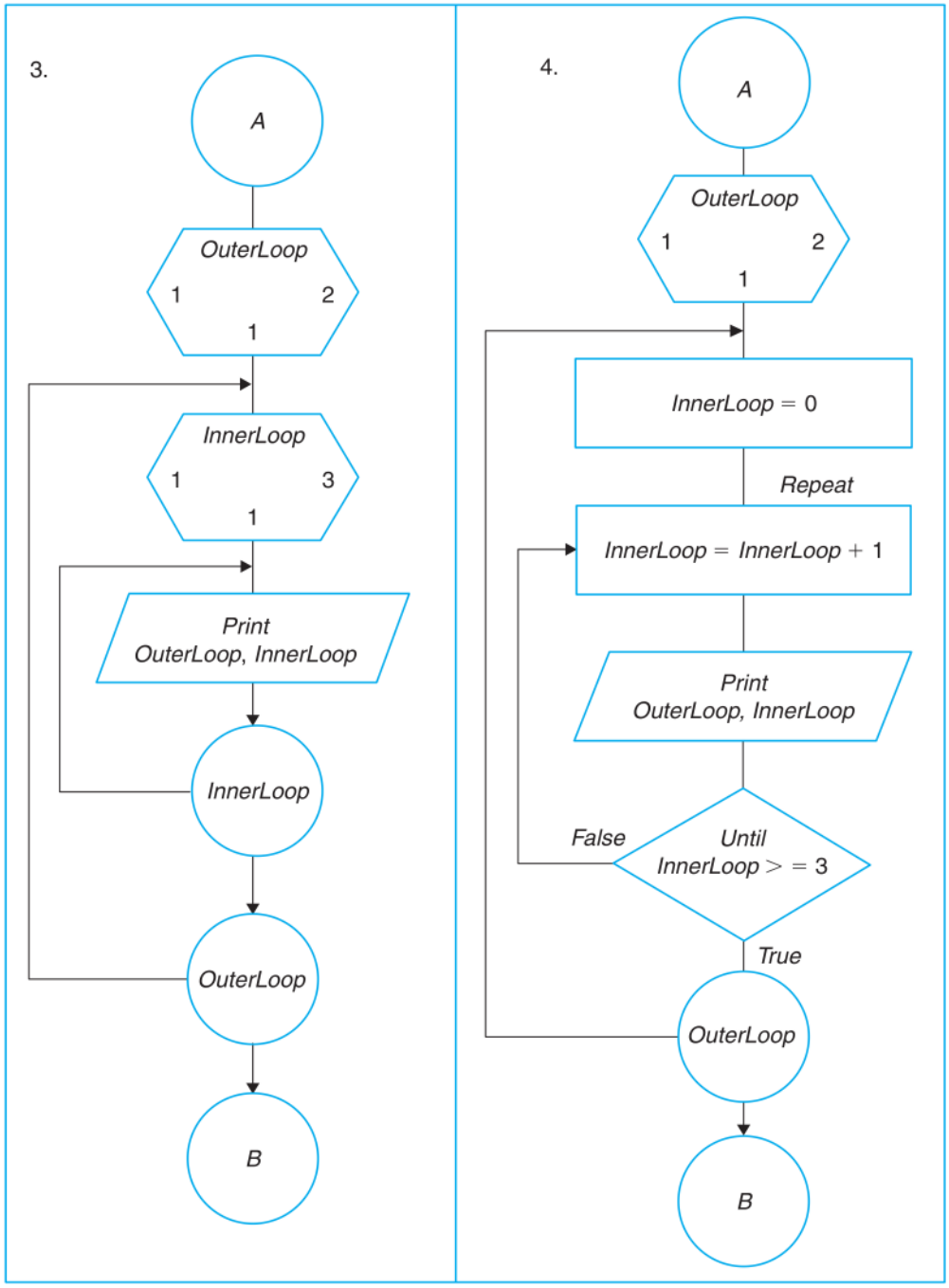


Figure 7.11 Nested Loops *(continued from page 165)*

Indicators

indicators

Indicators are logical variables that a programmer sets within a program to change the processing path or to control when the processing of a loop should end. They are sometimes called *flags, switches,* or *trip values.* The user has no knowledge of these indicators during processing. These indicators are used to detect changes in the output line, in the processing of data, and so forth. An error indicator designates that an error has occurred in the input or the output. An end-of-data indicator designates that there are no more data to be entered.