Problem Solving Basics and Computer Programming

By Ron Pasko and Matt Bauer

Solving Problems with Solutions Requiring Sequential Processing

Overview

Computer programming is not just programming language syntax and using a development environment. At its core, computer programming is solving problems. We will now turn our attention to a structured methodology you can use to construct solutions for a given problem. We will trace the following sample problem through each of the steps of our problem solving methodology:

Given the 3 dimensions of a box (length, width, and height), multiply them together to determine the volume.

Decomposition

The first step to solving any problem is to decompose the problem description. A good way to do this would be to perform **syntactic analysis** on the description. We can do this in four steps.

1. Identify all of the nouns in the sentence.

Given the 3 dimensions of a box (length, width, and height), calculate the volume.

The nouns in the problem specification identify descriptions of information that you will need to either identify or keep track of. Once these nouns are identified, they should be grouped into one of two categories:

Input (items I either already know or am getting from the user)
Output (items that I find out by manipulating the input)

Input		Output	
Dimensions		Volume	*We need to calculate this.*
Length	*We are told these,*		
Width	*dimensions are "given".*		
Height			
Box			
Them			

2. Eliminate redundant or irrelevant information.

There may be some information in the problem description that made it into our input/output chart that we really don't need to solve the problem (that is, not all of the nouns may be relevant). Also, there may be some nouns that appear redundant (information we already have in our table, just in a different form).

Input		Output
~~Dimensions~~	*We don't need the noun dimensions*	Volume
Length	*here because we already have length*	
Width	*width, and height.*	
Height		
~~Box~~	*We do not need the box to calculate volume if we know the dimensions, not needed.*	
~~Them~~	*Another word for dimensions, not needed.*	

You may ask why we eliminated "dimensions" instead of "length," "width," and "height." The rule of thumb for eliminating redundant information is to always eliminate the *most general* item. In other words, you wish to keep the most specific nouns possible in your table. When in doubt, try to piece it together logically: when figuring out the volume, which nouns would be the most useful to you?

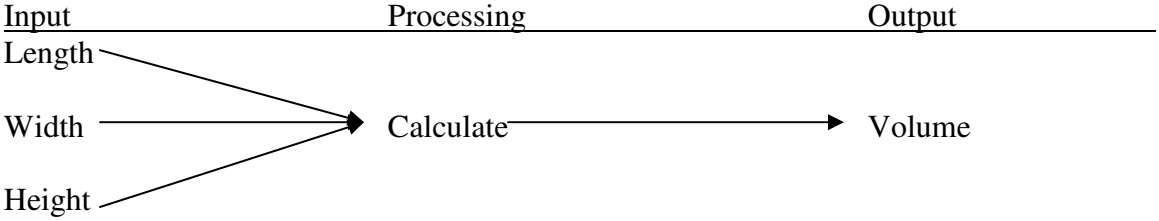
3. Identify all of the verbs in the sentence.

Given the 3 dimensions of a box (length, width, and height), calculate the volume.
The verbs in the problem specification identify what actions your program will need to take. These actions, known as **processing** are the steps between your input and your output.

Input	Processing	Output
Length	calculate	volume
Width		
Height		

4. Link you inputs, processes, and output

This step is as simple as drawing lines between the relevant information in your chart. Your lines show what inputs need to be processed to get the desired output. In our example, we need to take our *length, width,* and *height* and *multiply* them, to give us our desired *volume*.

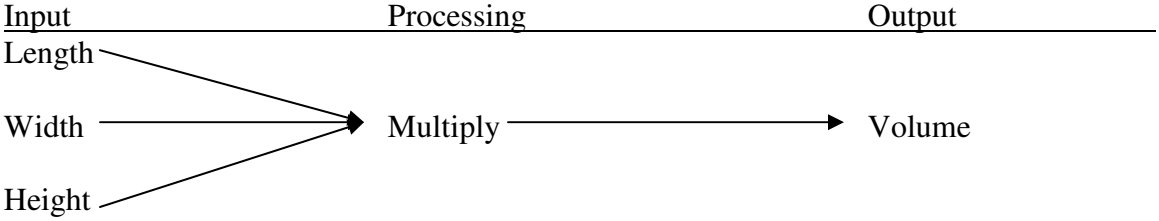


5. Use external knowledge to complete your solution

In the solution, we have used a general verb *calculate.* It is at this point at which we are required to determine what "calculate" means. In some arbitrary problem, *calculate* could refer to applying some mathematical formula or other transformation to our input data in order to reach the desired output. You must oftentimes refer to external knowledge (such as your background in mathematics) to "fill in the blanks." In this case, our elementary geometry tells us that the volume of a box can be found using the following formula:

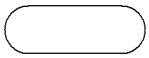
*Volume = length * width * height*

Simply apply this "new" knowledge to our previous sketch:

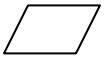


Flowcharting

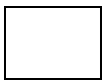
The second step in solving our problem involves the use of flowcharting. Flowcharting is a graphical way of depicting a problem in terms of its inputs, outputs, and processes. Though the shapes we will use in our flowcharts will be expanded as we cover more topics, the basic elements are as follows:



Rounded Rectangle (start/end of a program)



Parallelogram (program input and output)

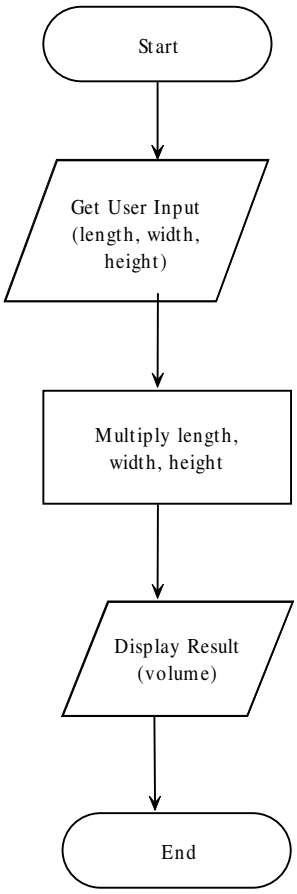


Rectangle (processing)

The flowchart should proceed directly from the chart you developed in step one. First, lay out your starting node, as every one of your programs will have these.

Next, begin adding your program elements sequentially, in the order that your problem description indicated. Connect the elements of your flowchart by uni-directional arrows that indicate the flow of your program.

According to our sample problem, we need to take in three items as input (length, width, and height). And after we have the user's input, need to process it. In this case, we must multiply the dimensions together. Finally, since our processing is complete, we should display the output for the user.



Pseudocode

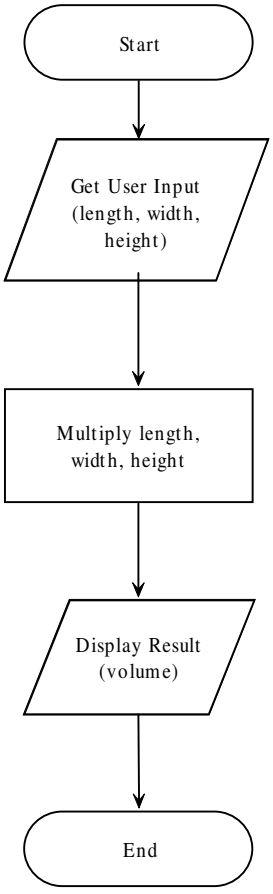
The final step in analyzing our problem is to step from our flowchart to **pseudocode.** Pseudocode involves writing down all of the major steps you will use in the program as depicted in your flowchart. This is similar to writing final statements in your programming language without needing to worry about program syntax, but retaining the flexibility of program design.

Like flowcharting, there are many elements to pseudocode design, only the most rudimentary are described here.

Get	used to get information from the user
Display	used to display information for the user
Compute	perform an arithmetic operation
+ - * / = ()	Standard arithmetic operators
Store	Store a piece of information for later use

*It is important to note that each time you compute a value that you will need later, it is necessary to **store it** even if you will need it right away.*

Here is the pseudocode for our example. It may be helpful to write out your pseudocode next to your flowchart.



Get length, width, height
Compute volume
    volume = length * width * height
    Store volume
Display volume

Now, on your own, work through the three steps of **decomposition, flowcharting, and pseudocode** for the following example.

You have a store that sells lemons and oranges. Oranges are $.30 each and lemons are $.15 each. Your program should get from the user the numbers of oranges and lemons he/she wants and outputs the total amount of money they owe you.

Solving Problems with Solutions Requiring Selection

Overview
Up to this point, you have solved problems with solutions which were strictly linear in nature, or sequential. In other words, from the start to the end of your pseudocode (or flowchart), each line (or figure in the flowchart) is executed once, in order. However, this raises one important question: What happens if a problem requires a solution that has alternate paths through the pseudocode depending on the input? How can I make a particular line (or block) of pseudocode optional?

Consider the following:
Write a program that will accept as input from the user, an answer to the following question: Is it raining? If it is raining, tell the user to get an umbrella.

Currently, we have not covered anything in problem solving that can help us handle conditions like "If it is raining". Fortunately, we are not left out in the cold and the rain; there is a concept known as **logical decision-making**

Decomposition - When to use logical decision-making
It is relatively trivial to identify when to use decision-making when solving a problem. Simply put, whenever you would need to make a real-world decision (such as whether or not to tell the user to bring an umbrella), you will need to implement a logical decision-making structure in your solution to the problem. In order to identify such situations in a problem statement, you first look for cues. From these cues, you can decipher the condition that the decision will be based on, and the actions that will be taken when this condition is true or false.

These cues may be obvious:

> *If it is raining, then tell user to get an umbrella*

Condition: If it is raining
Action: Tell the user to get an umbrella

or they may be subtle.

> *Based on the temperature, either tell the user to bring a heavy jacket (colder than 32 degrees), light jacket (between 32 and 50 degrees), or no jacket at all.*

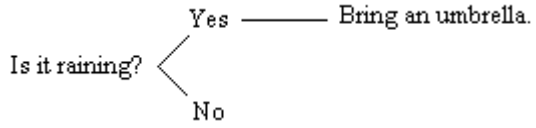
Condition: If the temperature is less than 32 degrees
Action: Tell user to bring a heavy jacket

Condition: If the temperature is between 32 and 50 degrees
Action: Tell user to bring a light jacket

Condition: If the temperature is greater than 50 degrees
Action: Tell user not to bring any jacket

Note, for the more subtle cues, look for **cases**; they are usually constructs that can be reworded into *if statements***.** In the above problem statement, you have three cases: **if** temperature is less than 32 degrees, **if** it is between 32 and 50 degrees, and **if** it is above 50 degrees.

It may be helpful to make a cursory sketch of all of the decisions you will need in your program before you go any further. For instance, an initial sketch of our sample problem yields the following:

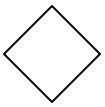


Note: our program description did not tell us what to do if the user says it is not raining, therefore, we do not put anything in the "No" branch of our decision.

Flowcharting

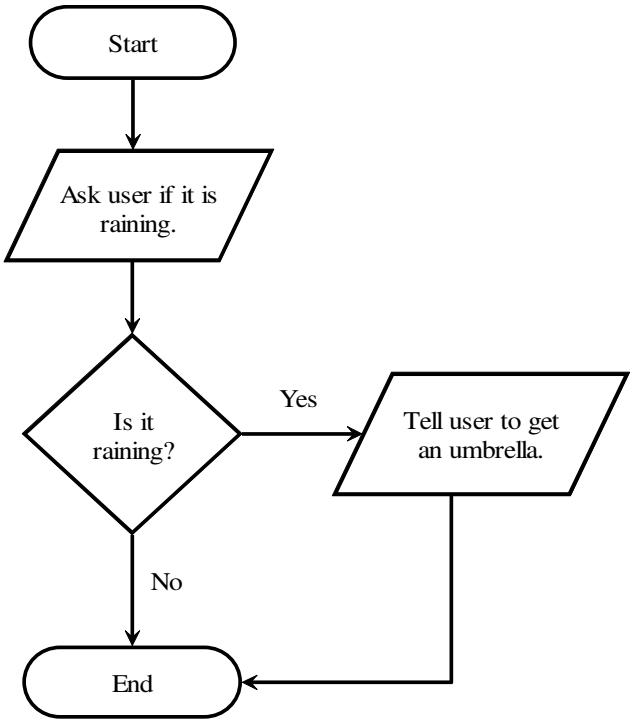
After we have made cursory sketches of all of the decisions in our program, we can immediately move on to flowcharting. Drawing a flowchart for a program containing logical decisions is quite helpful, as it helps the programmer see the "big picture" – how all of the decisions interact and affect each other.

For decision-making, we have a brand new flowcharting tool, the **diamond.**



Diamond (used for decisions). The "question" being asked goes inside the diamond. An arrow for each "answer" protrudes from the diamond.

Mark each of these arrows with the appropriate "answer." The decision diamond in your flowchart should look very much like the rough sketch of your decision. The full flowchart for our example is below.

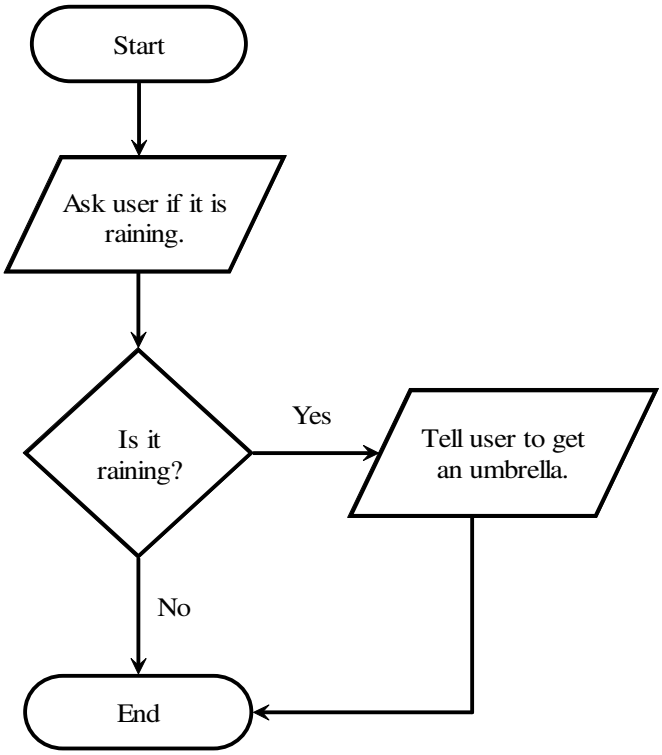


Note: Just like in our rough sketch, nothing of note occurs if the user answers "no" to "is it raining."

Pseudocode

The pseudocode for decision making brings us much closer to the actual implementation of this construct in our programming. Most decisions will begin with the operative word **if**, as all decisions must be declarative statements. The general structure of a decision should make sense in your mind; **if** it is raining, get an umbrella.

The pseudocode for our example is below, matched with our flowchart for clarity

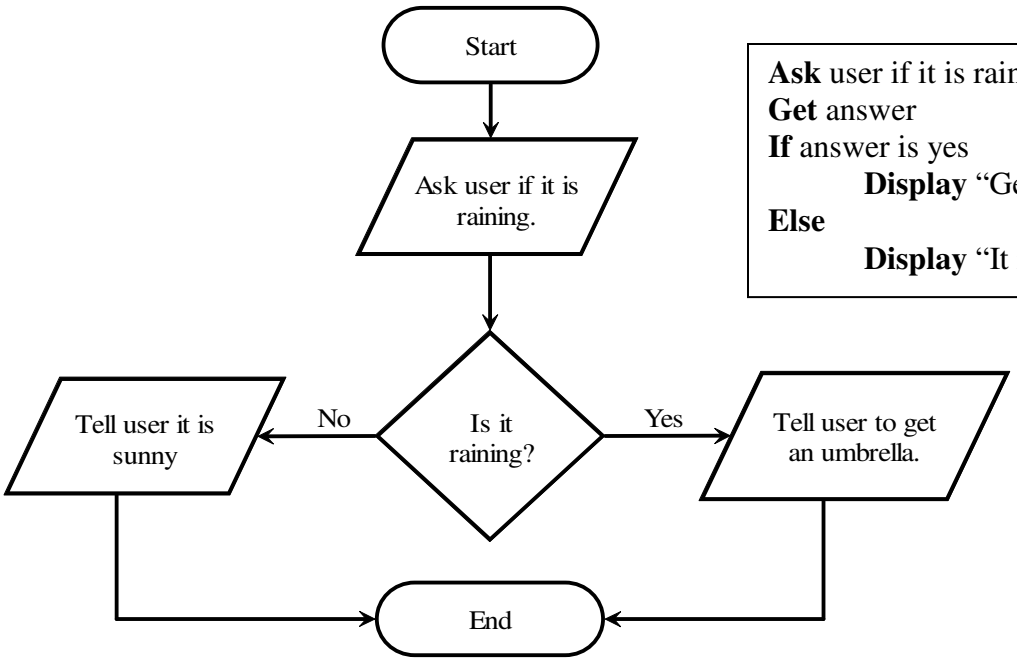


Ask user if it is raining
Get answer
If answer is yes
    Display "Get an umbrella"

The following is the pseudocode and flowchart for a modification of our example: *If it is raining, tell the user to get an umbrella. Otherwise, say it is sunny.*

The extension of this concept in our flowchart is trivial, we simply do *something* with the "no" branch. The only change is in our pseudocode, we have added the **else**, or condition not true, case. Again, the pseudocode reflects English rather well:

***If** it is raining, tell user to get an umbrella… **else** tell the user that it is sunny.*



Ask user if it is raining
Get answer
If answer is yes
    Display "Get an umbrella"
Else
    Display "It is sunny"

Now, on your own, work through the three steps of **decomposition, flowcharting, and pseudocode** for the following example.

Given 2 numbers, determine whether or not their sum is greater than 100.

Conditions
Most programming languages provide the following relational operators (sometimes with slightly different syntax) to make mathematical comparisons between data within your conditions:

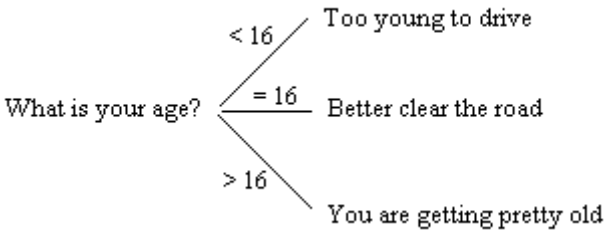
>	greater than	<	less than
>=	greater than/equal to	<=	less than/equal to
==	equal to	~=	not equal to

Also, the following logical operators (or similar) are usually provided:

&&	and	||	or	!	not

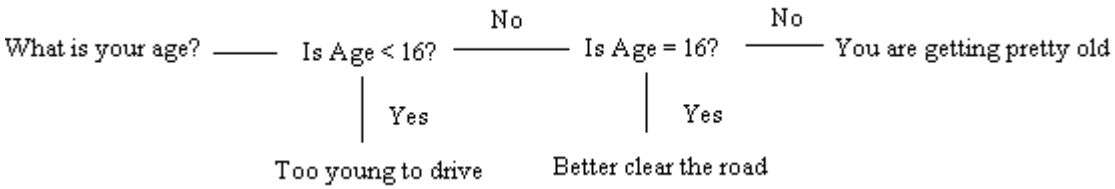
Though the relational operations (>, >=, etc.) are self-explanatory, the **and, or** and **not** operations deserve a brief explanation. Both **and** and **or** allow you to create compound conditions. **And** implies that a condition is true only if **all** of its components are true. **Or** implies that a condition evaluates to true if **any** of its components are true. **Not** reverses the truth value of a condition. This will be discussed in greater depth later.

All conditions in computer programming must evaluate to a Yes/No (or True/False) question. You cannot, for instance, have a condition which branches like this:



← You cannot do that in a single condition in a program!

Computers only understand two things: 0 and 1… true and false. Thankfully, we can rewrite all conditions into a string of Yes/No questions. The above can be translated into the following:

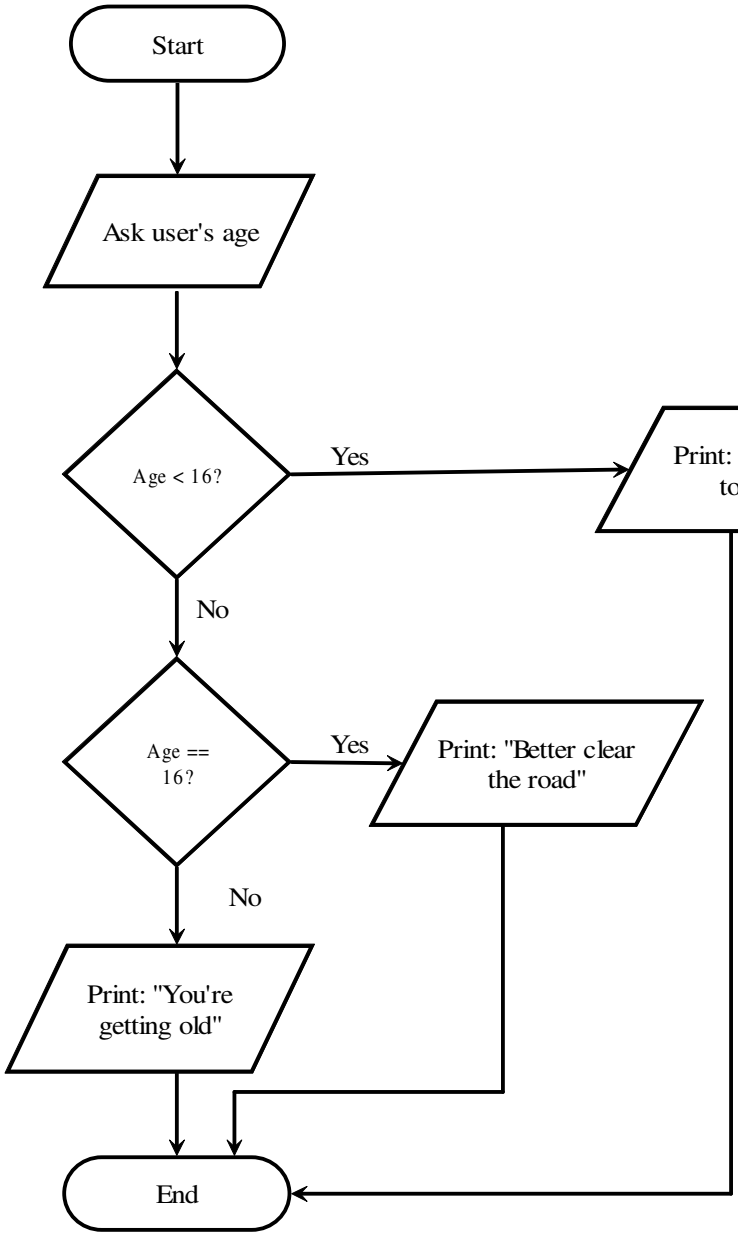


Notice that the "> 16" case does not appear in the conditional expression (if a number is not less than 16 and does not equal 16, it must be greater than 16). Because of this obvious consequence, our flowchart and pseudocode do not have to explicitly state the final case. This is known as the **default case.**

Again, a flowchart an its resulting pseudocode should be relatively easy to discern from the above picture.

Note the default case explained by the final ELSE in the pseudocode. If neither of the above if statements are true, the else is invoked by default. Furthermore, this if/else chain is **mutually exclusive.** In other words, if one condition is true, none of the following conditions are tested. This is clear from the flowchart.

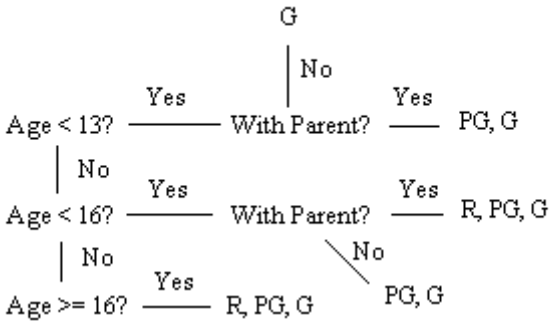
In the case where multiple conditions would be true, your flowchart would look much different. Consider the following:



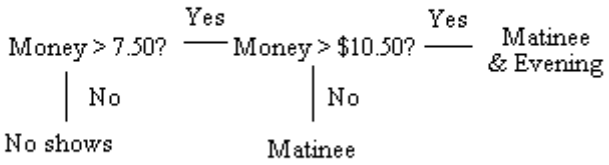
Ask user for age
Get age
If age < 16
    Display “Too young to drive”
Else if age = 16
    Display “Better clear the road”
Else
    Display “You’re getting old”

Write a program that tells the user what type of movie they can attend based on their age, if they are with their parents, and their amount of money.

Under 13:	G
Under 13 w/ parent:	G, PG
13 and Over and Under 16	G, PG
Under 16 w/ parent	G, PG, R
16 and Over	G, PG, R
Matinee:	$7.50
Evening:	$10.50



Notice our cursory sketches are getting quite complicated. In order to simplify things, always treat decisions of different types separately. For instance, the amount of money you have does not effect what rating of movie you can view, so these decisions are treated separately



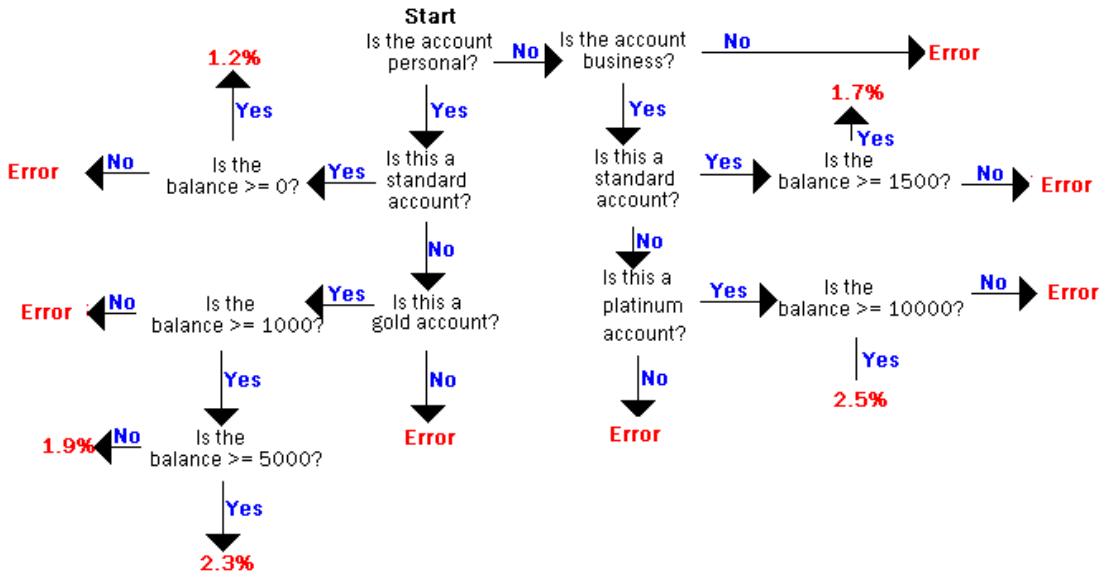

```
If age < 13
        If with parent
                Print "Can go to G & PG show'
        Else
                Print "Can to go G show"
Else If age < 16
        If with parent
                Print "Can go to G, PG & R show'
        Else
                Print "Can to go PG & G show"
Else
        Print "Can go to G, PG, & R show"

If money < 7.50
        Print "Not enough money"
Else If money < $10.50
        Print "Can to go to the matinee Show"
Else
        Print "Can go to the evening & matinee show"
```

The flowchart and pseudocode are much more complicated for this example. However, recall that we also have the ability to **combine** conditions using **and** and **or** operators. This will be demonstrated in the following example.

Write a program that will take as input the user's bank account balance and the type and level of account they have. Based on this information and the below rate table, determine the interest rate they are receiving.

Type of account	Level	Minimum Balance	Interest Rate
Personal	Standard	$0	1.2%
Personal	Gold	$1000	1.9%
Personal	Gold	$5000	2.3%
Business	Standard	$1500	1.7%
Business	Platinum	$10000	2.5%

Initially, you will notice that there are no "cue" words in the problem statement. However, it is clear that there are easily identifiable cases (just look at the table). From these cases, we can sketch our logic:



Wow. That is a very large sketch. Normally, we would need a condition for every yes/no decision in our sketch. However, using ANDs and ORs we can condense this a bit and translate into pseudocode.

Ask for user account type, account level, and balance.
Store account type, account level, and balance.

If account type is "personal" **and** account level is "standard" **and** balance >= 0
    print "Interest Rate is 1.2%"
Else If account type is "personal" **and** account level is "gold" **and** balance >= 5000
    print "Interest Rate is 2.3%"
Else If account type is "personal" **and** account level is "standard" **and** balance >= 1000
    print "Interest Rate is 1.9%"
Else If account type is "business" **and** account level is "standard" **and** balance >= 1500
    print "Interest Rate is 1.7%"
Else If account type is "business" **and** account level is "gold" **and** balance >= 10000
    print "Interest Rate is 2.5%"
Else
    Print "Error: the account information you entered is incorrect.

Now, on your own, work through the three steps of **decomposition, flowcharting, and pseudocode** for the following example.

*Write a program that will take as input the type of restaurant the user ate at, the cost of the meal, the number of people in his/her party, and how good the service was. Determine the **dollar amount** of the tip:*

Base Tip:
Diner: 12%
Good Restaurant: 15%
Fancy Restaurant: 20%

Additions/Subtractions:
Poor Service: -2%
Good Service: +0%
Excellent Service: +2%
1-5 in party: +0%
6-10 in party: +3%
more than 10: +5%

Solving Problems with Solutions Requiring Iteration

Overview

In the previous section, you received your first glimpse of solutions to problems that do not follow a strict linear form. Instead, you worked with programs that were able to branch off into one or more directions based on a certain decision or condition. However, this simple execution (where the only deviation from our "path" is a fork in the road) is not nearly powerful enough to tackle problems that are more advanced than the trivial examples we have covered thus far.

Consider the following:

Write a small program that will display the numbers 1 - 10.

Using the knowledge you currently possess, you would most likely write a program that uses individual lines of code that print out each number. The pseudocode for such an answer looks like this:

Display 1
Display 2
Display 3
Display 4
Display 5
Display 6
Display 7
Display 8
Display 9
Display 10

As far as code-length goes, programming such an application is indeed possible. However, what if the problem statement was modified thusly:

Write a small program that will display the numbers 1 - 100.

If you had not guessed it before, you should know at this point that a **good** programmer would never write (nor want to write!) an application made up of 100 lines, each saying **display** x. Indeed, there must be a way to avoid the repetition. Like any good tool, most programming languages provide us with a quick and easy way to solve such problems: iteration, also known as loops.

Decomposition - Identifying the need for a loop

Before we approach what the structure of a loop looks like, it is important to present the types of situations that you will encounter that will lend itself well to iteration. Stated simply, one should use a loop at any point where you wish to **repeat** a process, idea, or function.

For example, see if you can determine which of the following problems might be best solved using a loop:

A. Solving the equation $2x^2 + x + 5$ for all x between 5 and 10
B. Summing inputted integers until the user enters -1
C. The user enters in the current year and then his/her birth year. Your program computes the users age. Perform this task again if he or she wishes.

Trick Question! The answer is all of them. Let's briefly overview each problem to see why a loop would be necessary.

Solve the equation $2x^2 + x + 5$ for all x between 5 and 10
This problem is very similar to the one we were approached with at the beginning of this section. Instead of writing the code which computes $2x^2 + x + 5$ six times (one for each of the following x= 5, 6, 7, 8, 9, 10), we can say **repeat** this equation for each of the values of x.

Summing inputted integers until the user enters –1.
Without loops, this program is impossible: In essence, (without loops) you would need to have an infinite number of read statements all in a row to perform this task. Instead, we notice that as long as the user has not entered –1, repeat the addition and read statements. Remember, always look for indications that you will be repeating something.

The user enters in the current year and then his/her birth year. Your program computes the user's age. Perform this task again if he or she wishes.
This is a less intuitive use for loops. On the outside, it appears you are only performing one task: finding out the number of years the person has been living. However, you'll notice by reading the problem statement carefully that, if the user chooses, you should run the program again. In essence, you will be **repeating** your entire program. In this case, the "something" that you will be repeating is not a single statement or equation, but a large block of code.

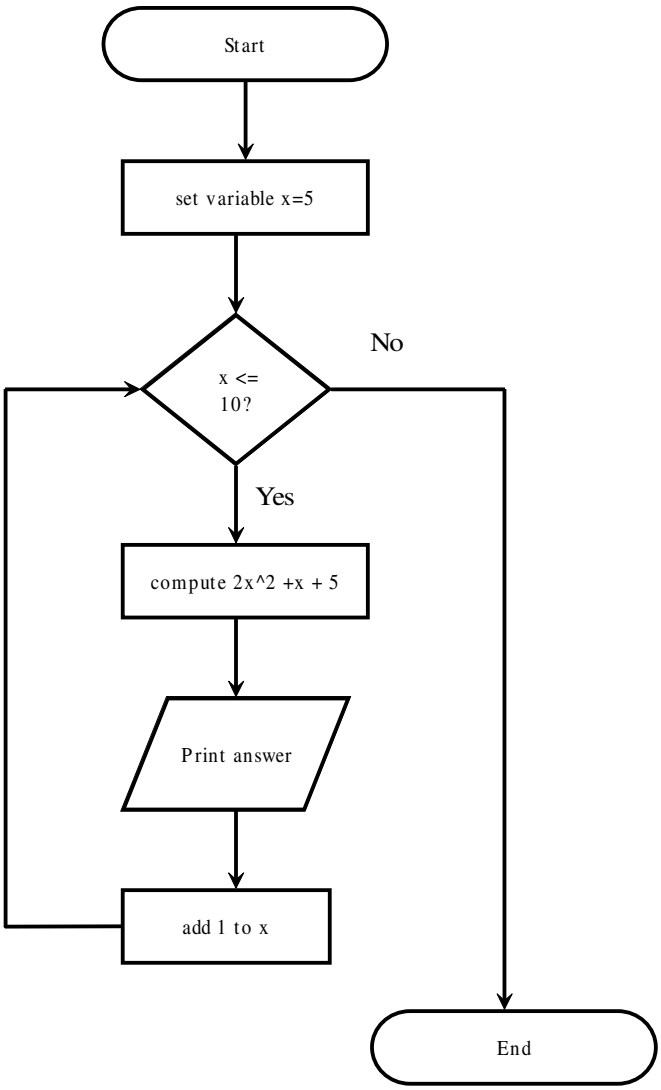
Flowcharting & Pseudocode
Loops are quite powerful: they allow us to do a massive amount of work with a minimal amount of programming. Amazingly, there is no need to learn any additional structural tools in order to think about loops. You have everything you need to know already at hand. If you are comfortable with logical decision-making, iteration should be easy.

Let's create a flowchart for the problem A (Solve the equation $2x^2 + x + 5$ for all x between 5 and 10).

The first task is to step back from the problem and plan how you would like to attack it. After you have devised this general plan of attack, you may proceed with the flowcharting and pseudocode.

Consider very carefully how you approach such a problem on a high-school mathematics exam. Your program should take **x** and start it at 5. Next, substitute 5 for each value of x in the equation ($2(5)^2 + (5) + 5$) and then solve (answer = 60). Now take the next x, which is obviously 6 (5 + 1). Solve the equation for x = 6 and continue on. The last value of x you will do this for is 10.

Now that we know how to approach the problem, let's sketch it out. We will discuss the flowchart and pseudocode in detail for this first example.



Set x to 5
While x <= 10
    Compute $2x^2 + x + 5$
    Store answer
    Print answer
    Increment x
    Return to While Statement
Quit

Again, there is very little structurally different about this flowchart compared to others we have studied. The only subtle change is that a certain execution path will lead you, not farther into the program, but to a previous section of code (note that after you "add 1 to x" you proceed back to the decision "x <= 10?")

In the pseudocode, we represent the lines of code we wish to repeat by indenting them (much how we did with if/else statements in the previous sections). When we hit the *return* statement in the pseudocode, we only repeat those lines that have been indented (note that we do **not** repeat "set x = 5").

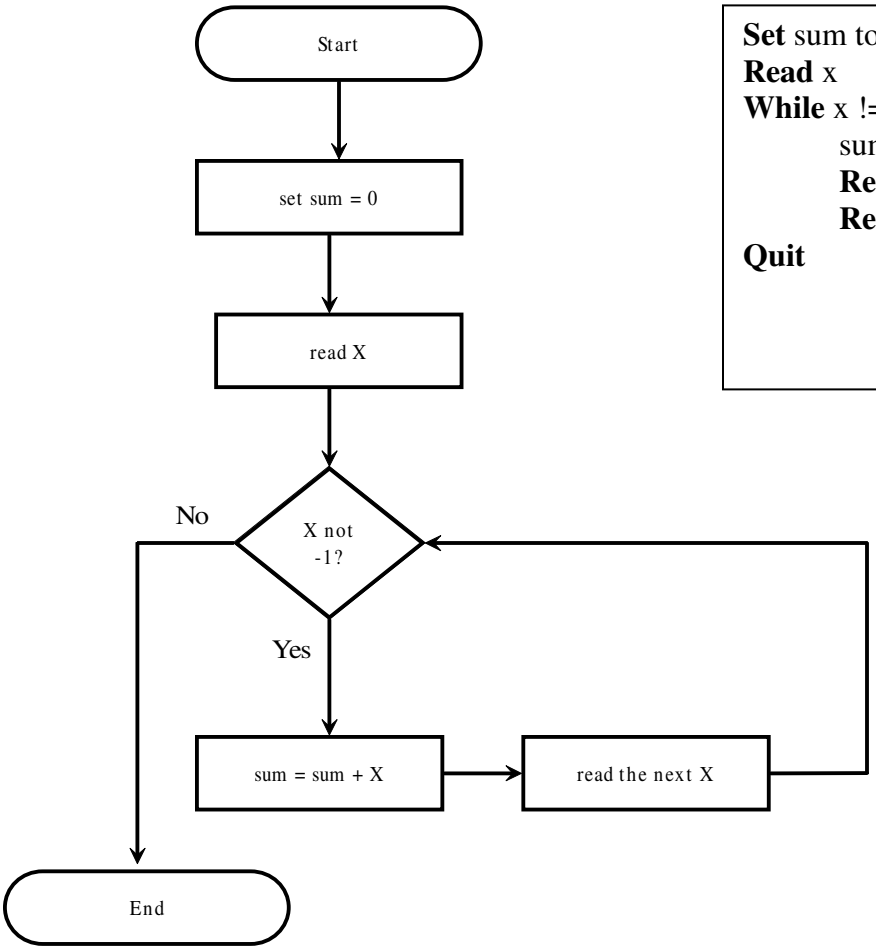
Also note that instead of an **if** statement, we are making a decision using the term **while**. In the pseudocode, **while** indicates that we will be repeating a certain chunk of code (that which is indented) until the condition (in this case "x <= 10") becomes false. Once this condition becomes false, we **skip** all of the indented code. In other words, when x = 11, we skip to the **quit** line.

Oftentimes, working through loops can be rather tricky. It may be helpful to maintain a table to help you makes sense of loop execution by keeping track of decisions and variable values. For instance, we can trace through our entire program in the following manner:

Code segment currently executing	What's going on	x	answer
Set x to 5	We make x = 5	**5**	
While x <= 10	Check to make sure x <= 10. If so, continue with indented loop code. If not, skip indented loop code.		
compute $2x^2 + x + 5$	$2 * 5^2 + 5 + 5 = 60$		**60**
print answer	The answer is 60, print 60		
increment x	Make x = x + 1… x = 5 + 1… x = 6	**6**	
return to while statement	return to while statement		
While x <= 10	6 < = 10, so continue		
compute $2x^2 + x + 5$	$2 * 6^2 + 6 + 5 = 83$		**83**
print answer	The answer is 83, print 83		
increment x	Make x = x + 1… x = 6 + 1… x = 7	**7**	
return to while statement	return to while statement		
While x <= 10	7 < = 10, so continue		
compute $2x^2 + x + 5$	$2 * 7^2 + 7 + 5 = 110$		**110**
print answer	The answer is 110, print 110		
increment x	Make x = x + 1… x = 7 + 1… x = 8	**8**	
return to while statement	return to while statement		
While x <= 10	8 < = 10, so continue		
compute $2x^2 + x + 5$	$2 * 8^2 + 8 + 5 = 141$		**141**
print answer	The answer is 141, print 141		
increment x	Make x = x + 1… x = 8 + 1… x = 9	**9**	
return to while statement	return to while statement		
While x <= 10	9 < = 10, so continue		
compute $2x^2 + x + 5$	$2 * 9^2 + 9 + 5 = 176$		**176**
print answer	The answer is 176, print 176		
increment x	Make x = x + 1… x = 9 + 1… x = 10	**10**	
return to while statement	return to while statement		
While x <= 10	10 < = 10, so continue		
compute $2x^2 + x + 5$	$2 * 10^2 + 10 + 5 = 215$		**215**
print answer	The answer is 215, print 215		
increment x	Make x = x + 1… x = 10 + 1… x = 11	**11**	
return to while statement	return to while statement		
While x <= 10	11 <= 10 **is false!**		
compute $2x^2 + x + 5$	SKIP		
print answer	SKIP		
increment x	SKIP		
return to while statement	SKIP		
QUIT	**END OF PROGRAM**		

We can approach the other two problems in the same manner. Keep in mind **what** you want to repeat: it is that block of code that will be "inside" the loop.

B. Summing inputted integers until the user enters –1.
Approach: Read integer, check if it is –1, if so quit. If not, add this number to the sum and repeat read. Remember that the symbol '!=' is standard in most programming languages for "not equal to."



Set sum to 0
Read x
While x != -1
    sum = sum + x
    Read next x
    Return to While Statement
Quit

There are two things worthy of discussion in the above example. Firstly, notice how we read user input once outside the loop and then again inside the loop. The reason we do this is because the decision (x != -1?) wouldn't make any sense if x did not have a value. Before this decision can be made, you must input the first number (outside the loop). If the condition is true, we can add this number to the sum and read the next one. The second and subsequent times user input is read, it is done inside the loop.

The second important note is the (sum = 0) statement at the beginning of the program. It is a very good idea to set all of your variables in a program to some initial value before you use them; if they are not given values by you, they will be left with whatever "garbage" value the computer had stored in its place. This is especially true of "accumulators" (variables which are used to accumulate a value, such as a counter or a sum). For instance, if you remove the (sum = 0) statement from the program and the computer had the number –5515 stored in sum's memory location, the sum would be meaningless (if the user entered the first number as **5,** the updated sum would then read
–5510 [-5515 + 5]).

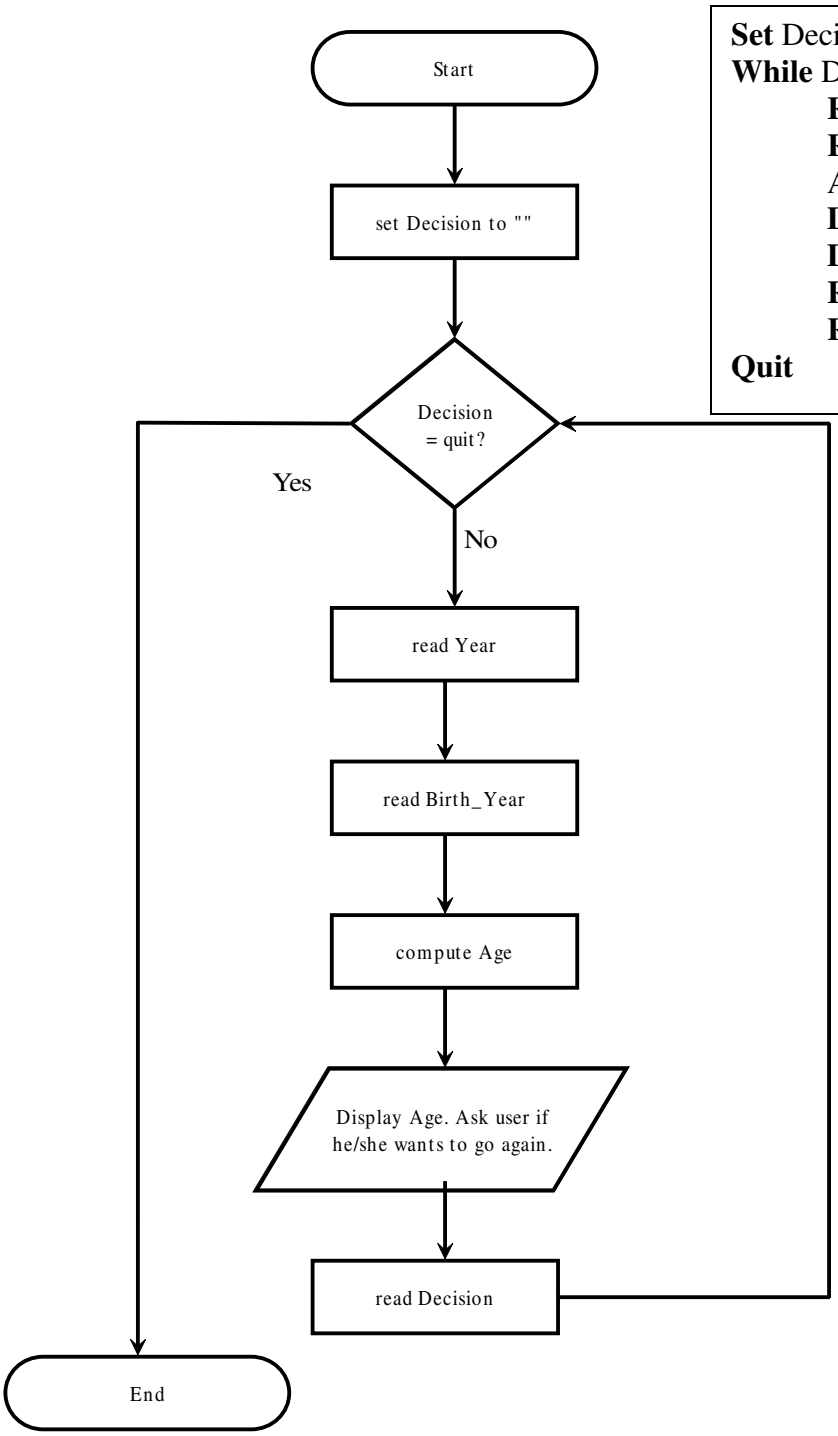
The trace table for this program is as follows: *Sample User data used:* ***5, 9, 3, 6, -1***

Code segment currently executing	**What's going on**	**x**	**sum**
Set sum = 0	sum = 0		0
Read x	user inputs 5	5	
x != -1?	5 != -1, continue		
sum = sum + x	sum = sum + 5, sum = 0 + 5		5
Read next x	user inputs 9	9	
return to while statement	return to while statement		

x != -1?	9 != -1, continue		
sum = sum + x	sum = sum + 9, sum = 5 + 9		14
Read next x	user inputs 3	3	
return to while statement	return to while statement		
x != -1?	3 != -1, continue		
sum = sum + x	sum = sum + 3, sum = 14 + 3		17
Read next x	user inputs 6	6	
return to while statement	return to while statement		
x != -1?	6 != -1, continue		
sum = sum + x	sum = sum + 6, sum = 17 + 6		23
Read next x	user inputs -1	-1	
return to while statement	return to while statement		
x != -1?	-1 != -1, **false!**		
sum = sum + x	SKIP		
Read next x	SKIP		
return to while statement	SKIP		
Quit	END OF PROGRAM		

C. The user enters in the current year and then his/her birth year. Your program computes the users age. Perform this task again if he or she wishes.
On its surface, this application appears to be the simplest of all of those in this section. However, it is covered because its loop is somewhat more difficult to visualize.

Approach: Get the current year from the user, get the user's birth year from the user, compute and display the users age. Ask if the user wishes to continue or to quit. If "continue", repeat the program. If "quit", exit the program.




```
Set Decision to ""
While Decision != "quit"
        Read Year
        Read Birth_year
        Age = Year – Birth_Year
        Display Age
        Display "Do you wish to quit?"
        Read Decision
        Return to While Statement
Quit
```

Note that decision is being set to "" for the same reason sum was initialized to 0 in the previous program.

The trace table for this program is as follows (assuming "c" indicates continue and "q" indicates quit):
Sample User data used: ***2002, 1980, c, 2002, 1990, q***

Code segment currently executing	What's going on	Year	Birth_Year	Age	Decision
Set Decision to ""	decision = ""				""
While Decision != "q"	"" != "q" true, continue				
Read Year	Get year from user, **2002**	2002			
Read Birth_Year	Get birth year from user, **1980**		1980		
Age = Year - Birth_year	Age = 2002 - 1980 = 22			22	
Display Age	Display Age, Display **22**				
Display "Do you wish to quit?	Display "Do you wish to quit?				
Read Decision	Get Decision from user, **c**				c
Return to While Statement	Return to While Statement				
While Decision != "q"	"c" != "q" true, continue				
Read Year	Get year from user, **2002**	2002			
Read Birth_Year	Get birth year from user, **1990**		1990		
Age = Year - Birth_year	Age = 2002 - 1990 = 12			12	
Display Age	Display Age, Display **12**				
Display "Do you wish to quit?	Display "Do you wish to quit?				
Read Decision	Get Decision from user, **q**				q
Return to While Statement	Return to While Statement				
While Decision != "q"	"q" != "q" true, **false!**				
Read Year	SKIP	2002			
Read Birth_Year	SKIP		1990		
Age = Year - Birth_year	SKIP			12	
Display Age	SKIP				
Display "Do you wish to quit?	SKIP				
Read Decision	SKIP				q
Return to While Statement	SKIP				
Quit	END PROGRAM				

A Note on Loop Construction Style

You may have noticed that in the flowcharts and pseudocode presented, the **While** decision always comes near the top of the chart/code. These schematics could be easily (perhaps, more easily) drawn with the **while** decision near the bottom (this would avoid, for instance, needing to read data an extra time outside of the loop). However, it is proper convention to place the decision *before* the block of code that will be repeated. Most loops in programming languages (with one notable exception) are **precondition tested,** that is, in order to execute the loop the first time, the condition the while statement is checking for must be true. In other words, the variable **decision** must *not* be "q" in order for the loop to iterate the first time. This is why decision is initialized to "".