In-lab activities for 10 using Visual Logic

Note that many of these activities use portions of the Visual Logic textbook, so you should bring that to lab every week.

Activity A: Introduction to Visual Logic

In the lecture portion of CSc 10, you will be learning to write and desk-check algorithms, using pseudocode. In the lab component of CSc 10, you will see an alternate way to represent algorithms, using a flowchart.

Flowcharts and pseudocode are both ways to represent an algorithm. (An algorithm is a finite set of instructions that solves a problem.) Flowcharts are easy to understand, because they graphically represent the logical steps. Most modern programmers prefer to use pseudocode rather than flowcharts to help them come up with the right steps to solve a problem, for reasons which will be touched on later in the semester, but everyone recognizes that both flowcharts and pseudocode are useful.

To introduce you to Visual Logic, we'll start with a very simple algorithm, expressed in pseudocode. You don't have to understand what pseudocode is, or even what an algorithm is, to understand this example.

I) First algorithm

DISPLAY "Hello, world!"
DISPLAY "See ya"

There. That’s an algorithm. Not much of an algorithm, but an algorithm all the same. It is a complete, finite set of instructions to solve the problem of displaying a welcoming message.

A flowchart to represent that same algorithm would look like this:
(Some flowcharts use ovals for “Begin” and “End”, but we’ll use rectangles, simply because the tool we are going to use does that.)

In the lecture portion of CSc 10, you’ll learn how to read and desk-check algorithms represented with pseudocode. When you read an algorithm that is represented by a flowchart, you begin reading at the “Begin” rectangle, and you follow the arrow to see where to go next. Continue until you reach “End”.

II) A flowchart using Visual Logic

Now we’ll create a flowchart using the Visual Logic program.

1) Begin the Visual Logic program. (Your instructor will tell you how to start the program.)

2) As you see, the “Begin” and “End” rectangles are already there for you. All you are going to have to do is put in the two “Output” parallelograms. For the first one, click the
mouse somewhere along the arrow that runs from “Begin” to “End”. (When using Visual Logic for this operation, you can right-click or left-click.)

3) Look at the group of flowchart operation symbols presented. Find the one for “Output”, and select it with the mouse. (Left- or right-click)

4) You should get a parallelogram with Output and <expr>. Now all we need to do is figure out how to put “Hello, world!” in place of <expr>. (“expr” is short for “expression”.)

   **Double-left-click** anywhere inside the parallelogram. (Alternately, right-clicking and selecting “Edit...” would accomplish the same thing.)

5) Inside the window that pops up on your screen, enter the expression that we want to output. Specifically, enter "Hello, world!" **You must include the double-quotes.** Then click OK.

6) Now repeat that same steps for the second output operation. Be sure you click on the arrow in the right place – we want to insert this second operation after the first one, so you have to click somewhere along the arrow between the first output operation and the “End” symbol. Create the output box and put “See ya” as the output expression.

The flowchart is now complete. In the old days, that would be the end of the flowcharting – you draw a flowchart (usually with pencil and paper), and that’s that. But this isn’t the old days! This is the digital twenty-first century!

The Visual Logic program does much more than helping you to draw a flowchart. It can actually **execute** the logic of the flowchart. That’s why we’ll be using it all semester. We can learn all about creating algorithms during lecture and from the lecture textbook. In lecture, you’ll also learn to “desk-check” your algorithms – sort of “play execute” them.

But with Visual Logic, you can actually get the computer to carry them out!

And it is time to do just that.

7) On the Visual Logic menu, choose “Debug”, then choose “Run/Continue”.

   Voila! The first output shows up on the screen! (If you got a red error message instead, you didn’t follow the instructions exactly. If you can’t figure out how to fix your flowchart, call for help.)

8) Assuming you were successful up to this point, close the output box (click OK). The execution continues to the next operation (the second output).

9) Close that second output box.
The flowchart has been executed. Your algorithm was designed to display:

    Hello, world!
    See ya

on the screen, and it did exactly that. Cool!

Shortcut key

10) There’s another way to execute the flowchart, a “shortcut key” – some of you may prefer this alternate way. It doesn’t matter which way you use, the menu or the shortcut key. Press the <F5> function key. This should bring the same result as when you used the Debug menu’s Run/Continue option.

III) Syntax errors.

As the semester progresses, you’ll learn that there are three different kinds of errors which can occur when creating and executing programs (and “executable flowcharts”). The first kind, and the only kind we’ll talk about right now, is a “syntax error”. In a “syntax error”, you provide some sort of command to the computer (specifically, here, to the Visual Logic program), but Visual Logic does not understand what you are asking it to do, that is, your command does not conform to the grammar (syntax) rules of Visual Logic.

When using Visual Logic, since you select all of the operations from a graphical list, you won’t be making any syntax errors by putting in the wrong command. (Alas, it is all too easy to make such an error when writing programs in various programming languages.) But it is still possible to make a syntax error when you provide the <expr>, that is, the expression. Let’s make one, just to see what happens.

1) Using the current flowchart, double-click inside the first Output parallelogram (the one with "Hello, world!"). This opens the Expression dialog box.

2) Change the expression. **Remove the opening double-quote mark.** Then click “OK”.

3) Execute the flowchart. (Use the Debug menu or the <F5> key.)

Uh-oh. Visual Logic is not happy. It does not understand the expression you put in. It doesn’t care what you want to display, but it wants it enclosed in double-quotes.

4) Fix it and execute again. All should be well.
IV) Output of numeric values.

Here’s a new, very small, algorithm (in pseudocode).

\[
\text{DISPLAY 25}
\]

Hmm. There are no quotes around 25. Let’s see what this looks like as a flowchart.

1) Let’s create a \textbf{new} Visual Logic flowchart. Select \textit{New} from the \textit{File} menu. It will ask you if you want to save the current flowchart. Just say “no”.

2) Put an output operation into the flowchart, just like you did last time. For the expression, type:

\[
25
\]

(No quotes.)

3) Execute the flowchart.

Hmm. That worked. Maybe we really don’t need those double-quotes after all.

3) Re-open the expression dialog for the Output operation and replace 25 with \textit{Hello}. No quotes.

4) Execute the flowchart.

Aack! No syntax error – the program ran. But it sure didn’t do what we wanted. It printed zero instead of Hello.
You’re going to have to wait awhile before you get a full explanation of why it displayed a zero. It has something to do with variables, and we’ll be covering those next time. For now, you need to know the following:

If you want to print words and stuff like that, put the expression in quotes.
If you want to print numbers, you don’t need the quotes.

Well, gee, what if it is some of both? My old address, when I lived in Florida way back in the last century, was 6645 8th Avenue North. That’s got numbers and words.

5) Change the output expression to:
   6645 8th Avenue North
   (No quotes)

6) Execute the flowchart. Nope, it doesn’t like that.

7) Put quotes around the whole expression.

8) Execute.

There, that worked.

I wonder what would happen if we put numbers (without words) in quotes.

9) Change the expression to “6645”. Use the quotes.

10) Execute.

It looks like we can have numbers as the output expression with or without quotes. And that’s true, uh, sort of.

The real truth is that numbers are no different than any other characters (like the characters that make up “Hello, world!”), unless we want to do some arithmetic on them. If we want to do arithmetic, then numbers inside quotes aren’t going to work. The computer treats anything inside quotes as just “characters”, keystrokes on the keyboard. If you want them to actually be numeric values, don’t put them in quotes.

(On a slightly more subtle level, if we mean for them to be “numeric”, don’t put them in quotes. If we mean for them to just be characters, do put them in quotes. The “8” in “8th” avenue in my old address is just a character. It could just as easily have been Avenue Q, or something like that.)
Definition: **string** (also called *character string*)
In computer programming, we refer to a group of characters ("Hello, world!" or "6645 8th Avenue North") as a *string*.
If we put a number in quotes, like "8", that’s a *string*, too. It treats it just like any other characters.

Note: Next time, when we start working with *variables*, you’ll find that there’s a little more to this “when to use double quotes” stuff. Can’t wait, can you?

**IV) Let’s try some arithmetic.**

1) Change the output expression to:

   \[
   12 + 7
   \]

   (No quotes)

2) Execute the flowchart. Did it treat the twelve and the seven as “numeric”? Sure it did.

3) Put the expression in quotes:

   \[
   "12 + 7"
   \]

4) Execute.

That should make sense. They aren’t numbers anymore. It’s just a *string* with the keystrokes 1 and 2 and space and plus sign and space and 7.

**V) Joining strings together.**

Some of you might wonder what would happen if you put each of the numbers in quotes, i.e., you had an expression that looked like:

\[
"12" + "7"
\]

Well, you are welcome to try it, but what it does isn’t exactly what the Visual Logic flowchart textbook will teach you. What it does is combine the two strings into the single string “127”. Not entirely unreasonable, but as you’ll see in the book, there is another symbol recommended for joining together two strings rather than the plus sign. That symbol is an ampersand (&). Let’s try it.

1) Change the output expression to:

   \[
   "Hello" & ", " & "world" & "!"
   \]

2) Execute the flowchart. There shouldn’t be any surprises. If it doesn't print

   *Hello, world!*
you probably typed something wrong. Go look carefully at where the quotes and ampersands are.

**Definition: concatenation**
Joining two strings together to form a single string is called *concatenation*.

But why would you go to all of that trouble?
Well, since you asked....

It turns out that you can concatenate numbers (actual numeric values) and strings to form a string.

3) Change the expression to:
   "XYZ" & 25

4) Execute the flowchart.

Visual Logic (and some programming languages) will allow you to concatenate that way. Visual Logic converts the number to a string and concatenates. No big deal, of course, you could have made the expression "XYZ25" and gotten the same result. But....

5) Change the expression to:
   "Twice my age is " & <put your age here> * 2
(Put your own age in place of <put your age here>.)
Note that in computer programming, we use an asterisk for multiplication. And be sure to leave a blank at the end of the string (right after *is*, inside the quotes).

6) Execute. There. Concatenation of a calculated number and a string! Whoopee!!

I know this doesn’t seem like all that big a deal, but as soon as we start using variables, it will become a very big deal. That’ll have to wait for next time.

**TO TURN IN**

Create a new flowchart which displays your own name followed by the words “is learning to program using Visual Logic.” This should all be in one output operation. Follow that with a separate output operation that displays, “Whoopee!”.

Execute your flowchart to be sure it works correctly.

Then, **change both of the output operations as follows:**
   Open the output dialog box (double click on the output parallelogram).
Click on the **More>>** button. Change the selected radio button from *Dialog Box* to *Console* and click OK. (Don’t worry if a funny squiggly symbol shows up at the end of your string. That’s the console end-of-output symbol.)

After you’ve changed both output operations, execute the flowchart again. The output should show up in a large text window, called the *Console Window*. And it will all be on the same line. (The squiggly end-of-output symbol does not show up in the Console Window.)

Let’s fix the “all on the same line” issue.

Re-open the first output dialog box. Put your cursor just in front of the squiggly end-of-output symbol, and press <enter>, then click OK. Since the end-of-output symbol shows where the next output will start, we’ve just moved it to the next line, so the next output will start there.

Execute again and make sure the Console Window now has “Whoopee!” on the second line.

Now it is time to figure out how to get “hard-copy”, i.e., output printed on paper, for your in-lab activities. The instructions that follow will be used for every “To turn in” section throughout these exercises. (Sometimes there will be more to turn in, but at the least, there will always be two parts: the output, which is what is in the Console Window, and the flowchart itself.)

See next page for the instructions:
GENERAL INSTRUCTIONS FOR TURNING IN VISUAL LOGIC EXERCISES

The following approach will be used throughout the semester whenever you are asked to turn in something from Visual Logic.

1) Set up your output operations to send the output of the flowchart to the Console. (Later in the semester, when there is also Input, you'll do the same for the input -- that is, you'll select Console in those boxes, too.)
2) Execute the flowchart. (If there is input to do, we'll often tell you what data to use. There isn't this time.) Leave the Console Window open after you execute the flowchart.
3) Next, in the main Visual Logic window, select View / Create Flowchart Metafile
4) You will be prompted for a place to store the Flowchart Metafile and a name for the flowchart. You can save it on a floppy disk, or you can save it in the Temp folder on the C: drive. For this first exercise, name it Activity_A. (You don't have to provide an extension -- it will automatically append .emf.)
5) Open Microsoft Word, and put your name, date, and section at the beginning of a new document.
6) Go to the Console Window and highlight all of the output from the execution of your flowchart. Then just press the < Enter > key. This will copy the selected text to the clipboard.
7) Return to the Word document, and paste the selection a few lines below your name, date, and section.
8) Put a couple of blank lines after the inserted output.
9) In Word, select Insert / Picture / From File
10) Find the Flowchart Metafile (it is saved wherever you saved it with whatever name you gave it, followed by .emf). Select it and click Insert. This should put the flowchart into the Word document.
11) In general, I recommend that you save the Word document on your floppy disk, so that if your instructor accidentally fails to record your score, you can show that file as evidence that you completed the activity. That's probably not a big deal today if you don't have a floppy disk with you (but bring one in the future).
12) PRINT the Word document to hand in. It won't look great on the dot-matrix printers in the lab, but it will work. (But see next paragraph first!)

NOTE: If you are using a laser or inkjet printer, the output will look just fine. But if you are using the lab dot-matrix printer, you might want to make the flowchart easier to read. You can make it larger before you print it. In the Word document, click somewhere inside the area where the flowchart is displayed. A rectangle should appear around the picture. Move your mouse over the black dot in the middle of the bottom line of the rectangle. Click and drag to increase the size of the rectangle. The size of the picture increases, too. Unfortunately, there doesn’t seem to be an easy way to increase the size to cover more than one full page, so when our flowcharts get bigger, this may not work as well, but for now, make your instructor happy with a more readable flowchart printout.

Continue to next page….
**Saving your work**

There will be times this semester when you do not finish your entire Visual Logic exercise during a single lab session. When that happens, you will want to save the flowchart you have created, so you can continue working on it later. (We’re talking here about actually saving the executable flowchart, not just the “picture” of the flowchart we used for printing.)

This is very easy to do -- it is just like any other Windows application. Just use the File/Save menu option. You should save your work on a floppy disk (A:). Remember to bring a floppy disk to lab in the future.

You probably don’t need to save this week’s work. It is pretty short -- you should be able to finish it during lab. And you probably didn’t bring a floppy disk today anyway.
Activity B: Exercise with variables, assignment statements, more output

Concept of variable and assignment statement

I) Simple arithmetic

1) Create a new Visual Logic flowchart.

2) Insert an Assignment Operation into your flowchart. A rectangle should show up, with:

   <var> = <expression>

in it.

3) Open the Assignment dialog box the same way you did with Output dialog boxes during Activity A. (Double-click)

4) In the Variable box, type:
   
   First

5) In the Expression box, type:
   
   5

You have now created the assignment statement: 
   
   First = 5

6) Click OK.

7) Create another assignment operation following the first one. Assign 3 to Second.
   
   Second = 3

8) Create another assignment operation following the first two. Assign sum of First and Second to variable Sum.
   
   Sum = First + Second

9) Create an output operation and output the value of Sum. (Put the variable name Sum as the output expression. Do not put it in quotes.)
   
   Execute and observe the results. If you did everything correctly, the output should be 8. If something else happens, you’ve made a mistake. Try to solve it. If you can’t, call for help.

II) Value of a variable

1) Change the output box to output not the value of Sum, but the word Sum. (Put it in quotes.)
Execute and observe the results.

2) After you see the result, change the output box back to the correct approach. If you want the value of a variable, you use the variable name. If you put a name in quotes, you are no longer dealing with the value of the variable, just a bunch of letters, i.e., a string.

Re-read 2) above. It is important! Don’t go on until you understand the difference between outputting “Sum” and outputting Sum.

III) Deleting from a Visual Logic Flowchart

1) Delete everything from the current flowchart. (Use the mouse to highlight everything between Begin and End. Experiment to find the best way to do this. Notice that Visual Logic will always retain the Begin and End. You should find that you can "box" the group of flowchart symbols to highlight all of them at once – then click the delete key.)

(This gets you to the same place you’d be if you had just started a new flowchart without saving the old one. But the advantage of learning to do this is that later you can use the approach to delete just particular sections of an existing flowchart -- that’ll save you a little time.)

IV) Accumulating a "running total"

1) Create an assignment operation that assigns 1 to Count.

2) Follow that with an assignment statement that assigns the current value of Count plus 1 to the variable Count. It should look like:
   \[
   \text{Count} = \text{Count} + 1
   \]

Carefully study that statement! It is one of the most important concepts in programming.

3) Add an output box to output the resulting value of Count. Execute and observe the results.

The secret of the assignment statement: The computer will first evaluate the right-hand side of the assignment operator. If necessary, it will retrieve the values of any variables in the expression on that right-hand side. After it has calculated the result of the right-hand side, it stores that result into the variable named on the left side of the assignment operator, replacing whatever was there before.

4) After the output box, add another assignment operation, again adding one to the value of Count.
5) Add an output box to display the final result. (Learn to do this by copying and pasting the other output box! It will save you lots of time.) Execute and observe the results.

**V) Output Labels**

If you were actually writing a program for a human end-user, that user would not be a programmer – he/she would not read the program. They would just look at the result. Therefore, you need to make the results "meaningful."

1) Delete the current flowchart contents or start a new one.

2) Put in three assignment operations. The first stores 5 to the variable First. The next stores 7 to Second. The third stores the sum of First and Second to variable Sum.

3) Put an output box to display the Sum. Execute and observe the results to be sure you’ve done it correctly.

4) Now change the output operation so that it prints the sum, but preceded by the words, "The sum is " and the ampersand (as you learned previously).

   "The sum is " & Sum

   Execute and observe the results. This is what is meant by an "output label".

5) Change the first assignment operation to store 11 to First.

   Execute and observe the results. If you get "The sum is 18", you did it correctly. (If not, fix it!)

Think carefully about what you've done. You could have gotten the right original result by using an output box with "The sum is 12" in it. But when the program changed (when we changed the value of First), the result would have still said that the sum was 12. That's why you use a combination of literal strings (the stuff inside the quotes) and variables (outside quotes) and the ampersand to join them (concatenate them) for output.

**VI) Improving the output label**

1) Change the output operation so that it results in this output:

   The sum of 11 and 7 is 18

Be sure you are using an appropriate combination of literal strings and variables. The 11, the 7, and the 18 should all be values of variables.

   Execute and observe the results. **Make sure you have appropriate blanks in there.** (Blanks inside quotes are just like any other characters.)
2) To make sure you have made correct use of variables, change the second assignment operation so that it stores 9 to Second.

Execute and observe the results. If you get "The sum of 11 and 9 is 20", you're on the right track!

NOTE: This output operation demonstrates a concept called "echoing" – we "echo" the values used in the calculation in the actual output (in this case, the values 11 and 9). Often, it is a very useful thing to do, and we'll do it quite a bit. There are times when it is not as useful, but for now, consider it a good thing to echo the values.

VII) Uninitialized variables

In some programming languages, and in the Visual Logic tool, if you don't assign a value to a variable, the language automatically initializes the variable (to 0, in the case of Visual Logic). Relying on that automatic initialization gets you into bad habits. You need to be in charge. If you want a value in a variable, put it there. Don't let the tool push you around!

For purposes of CSc 10 (and many future courses), any variable which is supposed to have some initial value must get that value by you explicitly assigning it. Failure to do so will be considered an error, even if the program works!

1) Start a new flowchart (or erase everything in the current one).

2) Assign 5 to First.

3) Assign First plus Second to variable Sum.
   (Notice that you have not assigned anything to the variable Second.)

4) Output Sum (no label needed this time).
   Execute and observe the results. You should see that Second was "automatically" initialized to 0. That's why the when you added First (5) to Second (0), you ended up with 5. Fine. Now that you see it, don't let it happen again! Uninitialized variables are errors!

   (By the way -- back in activity one, you had an output operation that tried to print Hello, with no double-quotes. Maybe now you see why it printed zero. Visual Logic treated Hello as a variable, and since it had never been given a value, it printed zero for the value.)
DECLARING VARIABLES
In most modern programming languages, it is required that you "declare" your variables before you can use them. That is, you have to tell the program what kind of data you plan to put into a particular variable. (So far in this exercise, you've only dealt with variables that store numbers, but variables can store individual characters, strings, and even some other stuff.) The ability to "declare" variables is very important in helping to prevent certain kinds of errors, but is not really critical for the "logic" issues we are dealing with in CSc 10. Visual Logic does not have the capability for declaring variables. (I wish it did!)

VII) String variables
Not all variables store numbers. Here's an example using variables that store "strings". As you learned previously, string in computer programming means any combination of alphabetic and numeric characters, plus punctuation, spaces, etc. In most languages, and specifically in Visual Logic, literal strings are enclosed in double-quotes.

1) Start a new flowchart. Create two assignment operations. The first assigns "Hi" to the variable Greeting. The second assigns your own name (first and last) to the variable MyName. Don't forget quotes.

2) Now provide an output operation that looks like this:
   Greeting & ", my name is " & MyName & "."

   Execute and observe the results.

TO TURN IN
Create a Visual Logic flowchart for the following algorithm:

Name ← < put your name here, in quotes >
Age ← < put your age here – NOT IN QUOTES >
Major ← < put your major here >
DISPLAY Name, Age, Major

In your flowchart, include a meaningful sentence for the output. It should look like the following (but, of course, using your data):

   My name is Joe Student, I'm 19 years old, and my major is Pre-Computer Science.

Send the output to the Console.
Then follow the procedure shown in activity one to provide a printout to turn in. Remember to include your name, section, and date at the top, and include the output and the flowchart diagram picture.
Activity C: Input (and how to prompt the user when using Visual Logic).

Up to now, the Visual Logic executable flowcharts have really been pretty useless. Every time you execute the flowchart, you get exactly the same output. That's because you've used only assignment statements to give values to variables. In order to give a variable a different value, you would have to go in and change the flowchart.

The missing element is input. Input gives us a way to put values into variables while the program is executing, rather than only while we are writing the program. This way, we can execute the same program (executable flowchart in Visual Logic) twice, but get completely different output, because each time we execute the program, we put in different values.

1) Input instead of assignment operation.

We'll start with a simple flowchart that uses only assignment operations.

1) Start a new flowchart.

2) Put in one assignment operation, assigning a value to a variable named Age. For this example, use 99 as the age.

3) Add an output operation that uses the variable to print a statement that says:
   I am 99 years old.

   Execute and observe the results. Then execute again. Will the results change? Will the results ever change (as long as we don't change the assignment statement)?

4) Now delete the assignment operation and replace it with an Input operation. The variable to be input is Age.

   Execute. When the prompt comes up saying to please enter value for Age, enter 99.

   Hmm. This gives exactly the same result we got in the previous version.

5) Execute the flowchart again, but this time, enter your own age (unless you happen to be 99 years old, in which case, lie).

   As you see, the output is different this time. And in fact, every time you execute, you might get different results. (I say might, because, as I'm sure you understand, if you enter the same age twice, you'll get the same result twice.)
II) A more involved algorithm using input and assignment operations.

1) Start a new flowchart.

2) Begin with an input operation. The variable name should be \textit{Diameter}.

4) Next, put another assignment operation. Assign $3.14 \times \text{Diameter}$ to \textit{Circumf}.

5) Add an output operation that displays the circumference of the circle, appropriately labeled. Echo the input value (but not the value of Pi), so your output looks like:

The circumference of a circle with diameter \textit{nn} is \textit{mmm}.

Execute, putting in the value 10 as the input.

NOTE: Here's an interesting feature of Visual Logic. It isn't critically important, but it might be useful to know.

6) Change the assignment statement to:

\begin{equation}
\text{Circumf} = \text{Pi} \times \text{Diameter}
\end{equation}

Based on the earlier discussion of uninitialized variables, you'd think this would give you 0, since we did not give \textit{Pi} a value. But Visual Logic has decided to include the key word \textit{Pi} as a part of the language – it gives it the right value. Execute the flowchart and see.

Incidentally, if you had tried to assign a value to \textit{Pi}, you would have gotten a weird error message. Visual Logic is telling you that it already has a meaning for the name \textit{Pi} and you can't use it for a variable name. Let's try it.

7) Add a new assignment operation in front of the assignment operation you have now.

\begin{equation}
\text{Pi} = 3.14
\end{equation}

Execute. You should get an error message. (Not the world's most helpful error message, but an error message all the same.) You are not allowed to use "reserved words" (also called "key words") as variables. Too bad the book doesn't tell you what all the reserved words are.

III) Variable names and end users

In the previous two examples, we chose very "good" variable names for the input variables. That is, the names were very descriptive, understandable to anyone.

While it is important to use good variable names, it is also true that a good variable name from the point of view of a programmer may not necessarily be particularly meaningful to an end user – the person who buys your program to use in his/her home or business.
This creates a problem in the Visual Logic flowcharts we've created so far. When we do an input operation, Visual Logic "prompts" the user by asking them to put in a value for <variable name>.

Here's an example of the problem.

1) Start a new flowchart.

2) Start with an input operation for a variable named StudentName. (Variable names in Visual Logic, and in most programming languages, cannot have any blanks in them.)

3) Add another input operation, for a variable name YearInSchool.

4) Finally, put in an output operation that prints something like:
   Joe Student is in year 3 in school.
   Of course, use the variables for the name and year, not those actual values.

Execute, entering any name you like and any year.

In Visual Logic, when you input a string, you are required to put the string in quotes. This is a bit odd – most programming languages don't have this requirement. You're just going to have to remember to use the quotes this semester.

For the year, don't use quotes. It is a numeric value.

The problem here is that the prompt that appears on the screen for the user says something like:
   Please type a value for StudentName.

Well, that looks perfectly reasonable to you, you wrote the program. But it would look a bit strange to some end-user who bought your wonderful program to use at home. They don't know anything about variables or variable names. They expect a prompt that, at the very least, doesn't look like a typing error!

But it gets worse.

5) Change the first input operation – instead of StudentName, use the variable name N.
Now, I'm not telling you that such a variable name is a good idea (or would even be tolerated by your instructor), but it certainly is okay with Visual Logic, or any other programming language out there.

6) Change the output operation to use N instead of StudentName.

Execute the flowchart. Ugh! That's a terrible first message to present to the end-user. (Enter any name and year you like – the program should behave exactly as the previous one did.)
Most programming languages have a way to incorporate a nice, user-friendly prompt with an input operation, and none that I know would ever display the variable name. Visual Logic also has a nice way to include prompts for input.

**IV) Input with user-friendly prompt**

1) Change the existing flowchart back to the way it was – replace \( N \) with StudentName in the first input operation and in the output operation. I don't want you to get into the habit of using poor variable names!

2) Open the input operation box. You'll see a button labeled More>>. Click that. It opens up some other options (such as the "Console" option we used to get printable output). But you'll also see that there is a place to enter a Prompt: In that box, put: "Please enter the name for this student." and click OK.

3) Put a prompt for the second input operation. Make up a nice, user-friendly prompt for the YearInSchool input. (And tell them not to use quotes for this one.)

   Execute, entering any name and year you like.

**TO TURN IN**

A) Here is an algorithm, in pseudocode. You're going to create a Visual Logic flowchart to implement this algorithm. And you are going to come up with good, friendly, prompts for the inputs and a well-formed output statement for the results. The output should look like this (but, of course, using whatever input values the user enters).

\[
\begin{align*}
3 + 4 &= 7 \\
3 - 4 &= -1 \\
3 \times 4 &= 12 \\
3 / 4 &= 0.75
\end{align*}
\]

Send output to the Console!

Here's the algorithm:
INPUT Num1
INPUT Num2
Sum ← Num1 + Num2
Difference ← Num1 – Num2
Product ← Num1 * Num2
Quotient ← Num1 / Num2
DISPLAY Sum, Difference, Product, Quotient

Test your flowchart with various input values (checking the calculations on paper or with a calculator). When you are convinced it is correct, and that the prompt(s) and output(s) look good, execute the flowchart using the values 7 and 12.

Copy the output and paste it into a Word document. Add the flowchart. Be sure to include your name, section, and date at the top.

Next – execute one more time. This time, input the numbers 5 and 0. In your Word document, explain what happened when you ran that. Explain why it happened.

(Later in the semester, we'll come up with ways to prevent such "run-time errors".)
Activity D: Arithmetic, precedence of operators. Storing calculation results vs. calculating within the output operation.

I) Arithmetic operator precedence

Before doing anything on the computer, think about this:

What is the result of $5 + 2 \times 3$?
It could be 21, or it could be 11. Which is it?

Hopefully, you recognize that the right answer is 11. This is because of the "precedence" of operators. The multiplication operator (an asterisk in almost all programming languages) has **higher precedence** than the addition operator. There is more discussion of this issue in the lecture textbook.

1) Start a new flowchart, and create an output operation that prints the result of $5 + 2 \times 3$.
(Put the arithmetic expression right into the output operation. Like most languages, an output operation will perform arithmetic.)
   Execute and observe the results. Should be no surprises.

2) Change the expression by putting parentheses around the addition operation.
   $(5 + 2) \times 3$.
Think about what the result will be, then execute and see if you were right.

3) Change the expression – let's do a more complicated one. Before you run this, be sure you have determined what the result ought to be, based on the rules of arithmetic. Recall that multiplication and division have higher precedence than addition and subtraction, and that if you have two operations that have the same precedence, they are taken from left to right.

$$4 - 3 \times 8 + 6 / 2 \times 3 - 9$$
   Execute and observe the results. Hopefully you knew it should be -20, and that's what you got.

4) Some languages (and Visual Logic) provide an exponentiation operator. (Others don't, by the way.) Many languages use the symbol "^" for that purpose.
   Change the expression in your output operation to $3 \times 2$.
   Execute and observe the results.

5) Try this one:
   $3 + 4 \times 2 \times 3$
   Execute and observe the results. Which has higher precedence, multiplication or exponentiation?
6) Change the expression to force it to do the multiplication before the exponentiation. (Use parentheses.) Then execute and see if it worked correctly. Should be 515.

II) Using intermediate calculations

1) Create the following (new) flowchart. The first assignment operation has the following expression:
   \[
   \text{Result} = (3 + 4) \times \left( \frac{5 - 1}{2} \right)^{2 + 1}
   \]

2) Add an output operation to output the result. (No label needed.)

Execute and make sure you did it right up to here – you should get 56.

3) Now, leaving the output operation unchanged, we're going to modify the flowchart so that you can carry out all of the same arithmetic, in exactly the same order, without having to use any parentheses. The trick is to break the one large assignment operation into a series of "intermediate" assignment operations.
   a) Begin with an assignment operation that adds 3 plus 4, assigning the result to Sum.
   b) Create the next assignment operation to subtract 1 from 5, assigning the result to Diff.
   c) For the next assignment statement, divide Diff by 2, assigning the result to Quotient.
   d) Assign 2 plus 1 to Exponent.
   e) Now we can put it all together and rely on correct operation precedent to finish the problem. Assign Sum times Quotient raised to power Exponent to Result.
   f) Make sure you delete the original, complicated assignment statement used in the previous version.

At this point, Result should have the same answer it had in the original version.

Execute and see if you got it right.

Carefully examine the set of assignment statements to be sure you see that it is really doing exactly what the original one did!

Sometimes it is useful to break complicated arithmetic expressions down in this way to help you (the human) understand them. Other times, you'll be happy with the parenthesized, and somewhat more complicated, combined expressions. There is no real rule for which is best. Whatever makes it easier for you (and others) to understand is the best choice.
III) Operator precedence – example with variables and input.

I know many of you are looking at that last exercise and thinking that it is really dumb – why not just start by doing the simple arithmetic in your head, so you have something like:

\[ \text{Result} = 7 \times 2^3 \]

Sure, that makes good sense. But in the real world of programming, you are usually dealing with variables, not the actual numbers. So reconsider what this would have looked like if it had begun as:

\[ \text{Result} = (x + y) \times \left( \frac{(a - b)}{2} \right)^{(c + 1)} \]

Assuming all of those variables had been given values previously (through a combination of assignment operations and input operations), you could not have "simplified" it by doing the arithmetic in your head. You'd need to use either parentheses or intermediate calculations.

Let's try one with relatively simple arithmetic, but still dealing with operator precedence issues.

1) Start a new flowchart.

Here's the pseudocode for an algorithm to calculate a bowling average, given three games. We will avoid any issues of data integrity (for example, a bowling score must be between 0 and 300, but we will ignore that for this week).

\[
\begin{align*}
\text{NumberOfGames} & \leftarrow 3 \quad // \text{Named constant} \\
\text{INPUT Score1} \\
\text{INPUT Score2} \\
\text{INPUT Score3} \\
\text{Average} & \leftarrow (\text{Score1} + \text{Score2} + \text{Score3}) / \text{NumberOfGames} \\
\text{DISPLAY Average}
\end{align*}
\]

Implement this exact algorithm using a Visual Logic flowchart. (You should already have covered named constants in lecture. If not, ask your instructor to explain them. Unfortunately, there is no facility within Visual Logic for named constants, so we'll just use ordinary assignment statements for them.)

Include a suitable prompt for input and a suitable label for the output.

To test your Visual Logic flowchart, you should come up with three bowling scores, and determine (using a calculator) what the output should be. Then go ahead and execute the flowchart using that input data and see if it gets the answer you expected.

(If it doesn't, either your flowchart is wrong or your previous calculation was wrong. Figure out which. If it is your flowchart, fix it!)
2) Now modify the flowchart. Replace the assignment operation (of Average) with one intermediate calculation (of the sum) followed by the assignment to Average, using the sum.

Execute again and make sure you get the same answer.

NOTE: This is really not enough testing, but for a problem that is this short, it really shouldn't take too much testing to gain confidence that the solution is acceptable. But even with this simple case, think about this:
   Would you be as confident in the correctness of your algorithm if the only test you tried was the scores 100, 100, 100 and it gave you an average of 100? If you would be as confident, you need to reconsider what testing is about. After all, an algorithm that said:
   INPUT Score1, Score2, Score3
   DISPLAY Score3
   would have gotten the "right" average for that one poor test case. But it certainly is not the correct algorithm!

IV) Directly printing the result of an arithmetic calculation vs. storing it in a variable

1) There are two ways to print the result of an arithmetic calculation. Here's the first: Start a new flowchart. Create three input operations, one for each of three exam scores. (Name the variables FirstExam, SecondExam, FinalExam.)

2) Now add an output operation, as follows:
   "The average of " & FirstExam & ", " & SecondExam & ", and " & FinalExam
   " is " & (FirstExam + SecondExam + FinalExam) / 3

Note that we are carrying out the arithmetic within the output operation (as you did earlier in this set of exercises).

Execute, using "reasonable" test data, and see if you get the answer you expected.

3) Now make a change. Add an assignment operation in between the third input and the output operations.
   Average = (FirstExam + SecondExam + FinalExam) / 3

4) Modify the output operation – instead of doing the arithmetic within the output operation, use the variable Average.

Execute with the same data. You should get the same result.
NOTE: In this second example, you did the calculations separately from the output operation, saving the result (and then printing that saved result in the output operation).

Both examples worked, and, frankly, both are probably just as good. Some of you will think the first version is clearer, some will prefer the second. It really doesn't matter, in this example.

But here's an example where it does matter.

5) Using the flowchart you just completed, add a named constant at the beginning of the algorithm. Name it ExamWeight and assign it the value 0.6.

6) Then add one more output operation, at the very end. This should say:

```
"Exams are worth " & (ExamWeight * 100) & "% of your grade. Your average gets you " & (ExamWeight * Average) & " out of that total."
```

Now reconsider the first way you solved the exam average problem. You calculated the average within the print statement, never assigning it to a variable. If you had retained that approach, you would have a problem in this second output operation. **You'd have to re-calculate the same average again** to calculate the weighted points.

Think about that.

So, a general rule is that, when you want to calculate a value **just to print it**, you can calculate it using an assignment statement and then print the variable you assigned it to, or you can just calculate it directly within the output operation. **But if you are going to need that same calculated value again later within the algorithm, always use the assignment operation approach!**

**TO TURN IN**

The rules for determining a golf handicap (at least, the rules we'll use in this exercise) state that a handicap is calculated by first obtaining the average of the most recent five golf scores (which will be inputs). Then subtract a "course rating" from that average – in this example, the course rating will be 72. Finally, multiply the result of that subtraction by an adjustment factor, which, for this example, will be 85%.

(There's more to the problem – anyone whose average of those five scores is less than or equal to the course rating does not get a handicap. But you don't yet have the tools to solve that part – that's coming in a few weeks. So don't worry about it for now.)
Here is an algorithm, written in pseudocode, to calculate a golf handicap according to the rules stated:

\[
\begin{align*}
\text{CourseRating} & \leftarrow 72 \\
\text{AdjustmentFactor} & \leftarrow 0.85 \\
\text{INPUT} & \text{ Score1, Score2, Score3, Score4, Score5} \\
\text{Handicap} & \leftarrow (((\text{Score1} + \text{Score2} + \text{Score3} + \text{Score4} + \text{Score5}) / 5) - \text{CourseRating}) \times \text{AdjustmentFactor} \\
\text{DISPLAY} & \text{Handicap}
\end{align*}
\]

A) Your first task is to convert that pseudocode into a Visual Logic flowchart. Include useful prompts for input and output the handicap with a nice label that **echoes the five scores** as well as the calculated handicap. **Input and output must both use the Console.**

Test your flowchart using some different sets of inputs, and be sure to calculate ahead of time what you think the output should be. Once you are satisfied that your flowchart is correct, execute it using this data:

79, 75, 81, 76, 76

Open a Word document, put in your name, section, and date. Follow that with the copy-and-pasted output from the data in the previous paragraph. And insert the flowchart.

B) Once you have the flowchart and result in a printable form (and you have saved that somewhere, either on your floppy or in C:\temp), make a few changes to the flowchart. Leave the same constants and five input operations for the scores, and leave the output operation as it is. But instead of using one (rather long and complicated) assignment statement to obtain the handicap, rewrite that operation using **intermediate assignment operations** such that your flowchart produces the same correct result, but such that it does not need any parentheses.

As with part A, create a flowchart metafile and create a printable file with the flowchart and the algorithm output (using the same set of five scores).

**TURN IN THE PRINTOUT OF BOTH FLOWCHARTS AND THE OUTPUT FROM EXECUTING EACH OF THE VISUAL LOGIC FLOWCHARTS.**
Activity E: Putting together the first four activities!

I) Using the Visual Logic textbook

Use your Visual Logic textbook. Read through chapter 1, except that you may begin at 1-1 (i.e., skip the "Grocery Checkout" case study), and you may also skip Section 1.7, Debugging. (There is a minor problem with the way Visual Logic handles its “debugging”. We’ll have a small activity later today showing how it works, but other than that, we won’t use it this semester.) You may also skip the Case Study at the end of the chapter (Grocery Checkout).

For each example in the text, create the appropriate Visual Logic flowchart and carefully read what the book has to say. (You will be expected to turn in the example shown in section 1-4 through 1-6, the "Weekly Paycheck Program"). Reading the book and doing the examples should act as a very good review of what you’ve learned in lab so far this semester. In addition, it will teach you several new concepts, specifically, ways to “format” numeric output to look nice, and the use of some other intrinsic functions that might come in handy.

Answer the questions in Quick Check 1-A and 1-B (page 11) to turn in. (These must be typed -- use Microsoft Word.)

II) Using the debugger

This activity will show you how “debuggers” work. You’ll probably be using them in future classes. You might even choose to use the one in Visual Logic, but, at least in the version I used to write these activities, there is a minor problem that makes this particular debugger a bit awkward.

1) Create a new flowchart. (This must be a NEW flowchart -- don't just erase whatever was in the last one.) This will be a very short, simple flowchart. Be very careful. I want you to create this flowchart without any mistakes -- if you accidentally put in an incorrect variable, this exercise isn’t going to work correctly. Start over with a new flowchart.

You need only three operations. Start with an INPUT operation, for the variable Num. Then put an ASSIGNMENT operation. Assign Num * 2 to variable Answer. Finally, add an OUTPUT operation, outputting the value of Answer. Don’t bother with input prompts or labels for output.

2) Execute the flowchart, using 6 as the input, and make sure it correctly displays 12 as the result.
Now, it’s time to see what a debugger is. Think about the desk check tables that you have been creating in the lecture portion of CSc 10. In a desk check, you “mentally” execute the first statement of your algorithm, and you show the values of all of the variables in your algorithm after that statement has been executed. Then you do the next statement, and the next, etc.

Well, that’s exactly what you can do with a debugger.

3) Go to the Debug menu, and select Step Into. This tells Visual Logic to move to the first operation, the INPUT operation. You should notice that the INPUT symbol is highlighted with a flashing border. That means that the INPUT operation is the next operation to be executed. It has not yet been executed.

You’ll also see that your variables pop up in a window. At present, they do not show any values, because (exactly as with your desk check), they have not yet gotten any known values put in them yet.

4) Again, go to the Debug menu and select Step Into. (You can use the shortcut key combination if you like.) When you select Step Into, the debugger executes the INPUT operation. As you see, the Input Dialog box pops up. You’ll probably have to select the Input Dialog box before you can type in your number. Go ahead and put in a value (let’s use 6 again) and click OK.

Wow! It’s just like a desk check table. When you executed the INPUT operation, the variable Num is shown with its new value. Make sure you see how that is just like desk checking (only the Visual Logic debugger is doing it for you).

5) The highlighted operation is the ASSIGNMENT operation, since that is the next operation that will be executed. Debug/Step Into.

Look -- the assignment operation is carried out, and the variable Answer now gets the calculated value of 12. Cool!

6) Do one more Debug/Step Into. This will execute the OUTPUT operation.

Well, okay, that’s what a debugger does. Actually, most debuggers, including this one, have lots of other things they can do, but the basic idea is that you can execute your operations and see what is going on in memory. We use debuggers when we have a problem in our program to help find the problem. In much more complicated programs, it is often the case that what you think the program is going to do doesn’t turn out to be what it actually does. Using the debugger lets us watch to try to find out what went wrong. Until we get to more complicated programs, it is kind of hard to make much of a case for how useful debuggers can be, but at least you now have a little introduction.

So, why did I say there was a problem with the way the debugger in Visual Logic works? (Again, we’re assuming they haven’t fixed it between the time I wrote this and the time
you tried it. Your instructor will let you know if that’s the case, but you can really find out right now, because I’m going to show you the problem.)

7) Close the Variable Watch window.

8) Delete the last two operations (ASSIGNMENT and OUTPUT) from your flowchart.

9) Now start debugging again. Debug/Step Over.
The Variable Watch window opens again. But, darn, it still thinks the variable Answer is in the flowchart. Oops!

That’s it. If you can live with that problem, feel free to use the debugger when you run into trouble as the algorithms get more complicated. It really is very helpful to have an “automatic desk checker”. But I don’t like the confusion of having variables in the window that don’t exist. So I won’t talk about it anymore. (Well, I might mention it when we start dealing with IFs and LOOPs -- it’s really neat to watch the execution jump around to the appropriate operation.)

TO TURN IN

A) The answers to the Quick Check questions from the first section of this week’s activity. These must be typed in, using Word. Be sure to start the document with your name, section, and date.

B) Printout of the flowchart for the "Weekly Paycheck Program". (That's the one from the book, sections 1-4 through 1-6.) Just print the flowchart (using the View/Create Flowchart Metafile approach). You don't need to show output.

C) Create an algorithm (in pseudocode) to calculate and display the final (after sales tax) price for an item. The initial price is an input. The sales tax rate is 8 ½ %. Use a named constant for the sales tax rate. (Be sure to use the correct decimal value, since you need to use it in the arithmetic calculation.)

(You can write the algorithm longhand, but eventually, you will have to type it, using Word. Work it out in pencil first, though.)

Then create a Visual Logic flowchart for that algorithm. This should include a meaningful prompt for input. The output should echo the initial price and tax rate, along with the final calculated price. All should be labeled nicely. And all should be displayed using the appropriate Intrinsic Functions (for percentage and for currency values).

TURN IN THE ANSWERS TO PARTS A (typed, using Word) and B (flowchart), plus the PRINTOUT OF THE PSEUDOCODE, THE FLOWCHART, AND THE OUTPUT FROM EXECUTING THE VISUAL LOGIC FLOWCHART for the sales tax problem
(C), using input of $50.00 as the pre-tax price.
Activity F: Conditional Control Structure

For this exercise, we’ll go back and forth between the Visual Logic textbook and the exercises in this handout. Start with the textbook.

I) Introduction to IF

1) Read textbook sections 2-1 and 2-2. Create the first Visual Logic flowcharts described (Figure 2.1). (You don't need to create figure 2-2, but you do need to read it and be sure you understand it.)

2) Answer the Quick Check 2-A questions (to hand in).

In lecture, you have learned to use pseudocode to represent IF/THEN/ELSE. The same control structure is represented in flowcharts with a diamond to represent the IF condition, and a "True" side for when that condition is true, a "False" side for the ELSE, i.e., for when the condition is not true.

II) Consider this pseudocode:

```
INPUT Number
IF Number > 0 THEN
    DISPLAY "Positive"
ELSE
    DISPLAY "Not positive (negative or zero)"
ENDIF
```

To represent that exact same algorithm using Visual Logic:

1) Start a new flowchart.

2) Create an input operation for the variable Number.

3) Create an "If Condition" after the input. Double-click the diamond, and inside, put the condition:
   Number > 0

4) Click along the "True" arrow, and create an output operation to display "Positive".
5) Click along the "False" arrow and create an output operation to display "Not positive (negative or zero)".

Execute the flowchart. Input 3 and watch what happens. Try again with -5. And then try again with 0. Make sure you understand why the 0 input behaves as it does.

III) An IF does not need an else.

Consider this algorithm to print the absolute value of an input:

INPUT Number
IF Number < 0 THEN
    Number = Number * (-1)
END IF
DISPLAY Number

As you see, if the number input by the user is negative, we multiply by minus one to make it positive. If the input was positive, we don't have to do anything to it.

1) Create a new flowchart.

2) Implement the logic of the absolute value algorithm. Be sure you recognize that the output operation (DISPLAY Number) occurs after the END IF, that is, after the True and False paths rejoin. Include a nice prompt to the user and label the output. (Don't try to echo the input in the output, though. You might want to think about the problem that would cause. We'll solve it next.)

Execute your flowchart a couple of times, once with a positive input, once with a negative input. (You could try zero, too, if you like.)

IV) Improving the label for the absolute value algorithm. (First version)

1) As you probably realized, we couldn't echo the input after the IF condition had finished, since the original input might not be there anymore (i.e., we might have changed the value of Number by the time we got to the output operation). One solution is to save the original input in a separate variable for use in the output statement.

Right after the input, create an assignment operation and store the value of Number into a variable named Original.

2) Now modify the output operation to include echoing the input, using Original as the input value you echo. For example, if the user had entered -4, your output should look like:
The absolute value of -4 is 4.

Execute, using positive and negative values. Make sure the echoed input is correct (and, of course, make sure the absolute value is correct, too).

V) Another approach to improving the label for the absolute value algorithm.

1) Delete everything in your flowchart except input operation and the IF condition. (That is, leave input operation and the IF, but remove the assignment statement that assigned Number to Original, remove the multiplication assignment statement, and remove the output operation.)

2) Recall that your IF condition determines if the number is negative. So, in the "True" part of your IF, create an assignment operation that assigns the value of Number multiplied by negative one to Result.

3) In the "False" part, create an assignment operation to just assign Number to Result. (In other words, if Number was not negative, we don't have to change it, but we still want the final answer in the variable named Result.)

4) Now create an output operation (after the IF is ended) to print the absolute value, with a label that echoes the input. (The output should look like the one you created in part D above.)

Execute and be sure it operates the same way the previous version (D) did. It should if you did it right. Make sure you try a positive input, a negative input, and zero.

VI) The mod operator

Consider how you would determine if a number is odd or even. If a number is evenly divisible by 2, it is even. Otherwise it is odd.

So your first try at an algorithm might be something like:

```plaintext
INPUT Number
IF Number is evenly divisible by 2 THEN
   DISPLAY “Even”
ELSE
   DISPLAY “Odd”
ENDIF
```

Yeah, you might try something like that. But you’d get into trouble when it came time to try to implement that algorithm directly in Visual Logic (or any programming language).
That’s because there is no condition “is evenly divisible by 2” built into Visual Logic (or any programming language).

Think back to grade school. You learned to do long division with remainders. Remember??
A number is evenly divisible by 2 if, after dividing the number by 2, you get a remainder of zero.

Now if only there were a way to get the remainder of a division. Well, there is, and you've already seen it in the Visual Logic textbook. The operation is called mod. Different languages have different symbols for it, but in many languages, and in Visual Logic, it is just that, mod.

So the result of (13 mod 5) is 3, the remainder of 13 divided by 5.

Here’s an algorithm to determine if a number is even or odd.

```
INPUT Number
IF (Number mod 2) = 0 THEN
   DISPLAY “Even”
ELSE
   DISPLAY “Odd”
ENDIF
```

1) Create a new flowchart.

2) Implement the algorithm, including a prompt for input and a nice label for the output, one that echoes the input.

Execute the algorithm and make sure it works for odd numbers and for even numbers. Check zero. Check negative numbers.
(We can argue about whether zero is really an even number, but according to the definition we gave earlier, it is, and your flowchart should declare it so. Does it?)

TO TURN IN

You probably learned somewhere that a “leap year” is a year that is evenly divisible by 4. While you will learn later that there are really several other rules, that simple rule is good enough for the next 90 years or so, so let’s use it for this exercise.

Write an algorithm (in pseudocode) to determine if a year (input by the user) is a leap year.
Then create a Visual Logic flowchart for that algorithm. Send the output (which consists of the input year and the statement “is a leap year” or “is not a leap year”) to the console window. (Input should also be from the console window.)


Turn in a Word document containing your name, section, and date, the algorithm (in pseudocode), the output from the three test cases, and the flowchart.
Activity G: Conditional Control Structure (continued)

I) Nested If

1) In the Visual Logic textbook, read section 2-3. Carefully study the two Nested If example flowcharts (figures 2.3 and 2.4). You do not have to actually create these flowcharts.

II) Nested If pseudocode and flowchart

As you’ve learned in the lecture text (or will learn when you read it), it is often useful to “nest” IFs. Consider this algorithm to decide whether a number is positive or negative.

```
INPUT Number
IF Number > 0 THEN
   DISPLAY “Positive”
ELSE
   DISPLAY “Negative”
ENDIF
```

Is that right? Think about it. Would it work for 6? How about -6? How about 0? Uh oh, I don’t think it works for zero. Zero is not negative. Hmm. It really isn’t positive either. Time for a nested IF.

```
INPUT Number
IF Number > 0 THEN
   DISPLAY “Positive”
ELSE
   IF Number < 0 THEN
      DISPLAY “Negative”
   ELSE
      DISPLAY “Zero”
   ENDIF
ENDIF
```

Carefully consider this algorithm. If the first test is true (Number is greater than 0), it displays “Positive” and then skips the entire ELSE. But if that first test is false, it goes to the first ELSE. Inside that ELSE, it checks to see if the number is less than zero, and if so, displays “Negative”. If that second test is also false, it goes to the corresponding ELSE for that IF, and displays “Zero”.

To think about (and probably see on a quiz or exam some day): Why don’t we need a third test to see if it is zero?

Let's implement that algorithm in Visual Logic.
1) Start a new flowchart.

2) Set up the input, with a prompt.

3) Create the first IF, and inside, put (Number > 0).

4) On the True arrow, put an output operation to display “Positive” (labeled nicely, echoing input).

5) On the False arrow, put another IF. That IF tests (Number < 0).

6) On the True arrow for that second IF, put an output operation to display “Negative” (labeled, echoing input).

7) On the False arrow for that second IF, put the output operation to display “Zero” (yeah, labeled….).

Execute. Test with positive, negative, and zero values.

---

III) Nested IF again

Consider this algorithm to determine if an exam score is a valid value:

```plaintext
INPUT Score
IF Score >= 0 THEN
    IF Score <= 100 THEN
        DISPLAY “Valid score”
    ELSE
        DISPLAY “Not valid”
    ENDIF
ELSE
    DISPLAY “Not valid”
ENDIF
```

This is a correct algorithm, and will work for all cases.

1) Create a new flowchart and implement exactly the logic of this algorithm.

Execute, testing with scores of -1, 0, 1, 99, 100, 101.

It should display “Valid score” for 0, 1, 99, and 100, and “Not valid” for the others.
IV) Compound conditions

1) Read section 2-4 of the Visual Logic textbook. You may ignore “XOR” in table 2.2.

2) Answer the Quick Check 2-B questions (1 through 4), on paper to hand in.

V) Based on Section 2-4 of the Visual Logic textbook, here's another alternative for “valid score” algorithm. “AND”

Reconsider the algorithm in part C of this week’s activity. What it is really saying is that in order to display “Valid score”, you need two conditions to be true -- the score must be >= 0 AND the score must be <= 100. If either of those conditions is not true, then the score is not valid.

Here’s an alternative algorithm that accomplishes the same thing.

```
INPUT Score
IF Score >= 0 AND Score <= 100 THEN
   DISPLAY “Valid score”
ELSE
   DISPLAY “Not valid”
ENDIF
```

1) Create a new flowchart (or modify the previous one if you think that's easier) and implement the logic of this algorithm.

Inside the IF, your condition must use parentheses for the two conditions separated by the “AND”. It looks like:

(Score >= 0) AND (Score <= 100)

Execute and test with the same values you used in the previous example. You should get the same results.

2) Just to prove the point, remove the parentheses in the IF condition.

   Execute. Test with 0, 1, 99, and 100.

Hmm. It looks like it works. Maybe I was wrong to say you needed the parentheses.

   Execute one more time. Try -1.

Uh oh. You really need the parentheses!

(Incidentally, that’s not true in all languages -- it just happens to be the way Visual Logic does the interpretation. It has to do with precedence, but it is not normal arithmetic precedence, and we won’t worry about it here -- just remember to put in the parentheses.)
VI) OR

In addition to “AND”, you can use “OR” in your logic. Here’s exactly the same problem solved with a different algorithm, one which uses OR.

```
INPUT Score
IF Score < 0 OR Score > 100 THEN
    DISPLAY “Not valid”
ELSE
    DISPLAY “Valid score”
ENDIF
```

Think carefully about this -- make sure you see why it is an equivalent solution to the problem.

1) Create a new flowchart (or modify the existing one) to implement the logic of this new algorithm.
   Execute and test with same values and make sure you get the same results.

VII) Precedence of AND and OR

Just like there is precedence with arithmetic operators (multiplication before addition, for example), there is precedence with AND and OR.

1) Create a new very simple flowchart.

Begin with an IF. The condition is:
   
   \[(2 > 1) \text{ OR } (3 = 4) \text{ AND } (5 < 10)\]

2) In the True path, output "Yes". In the False path, output "No".

Before you execute the flowchart, let's analyze this.
The first condition is true, the second and third are false.
We have:
   True OR False AND False

Now, there are two ways this could be interpreted. If OR has higher precedence than
AND, or, for that matter, if there is no precedence and they are just taken left to right, we
would have:
   True OR False AND False
   True AND False
   False
On the other hand, if AND has higher precedence (gets evaluated first), we have:

True OR False AND False
True OR FALSE
True

So if OR has higher precedence, your flowchart should print "No" (i.e., should be false). If AND has higher precedence, it should print "Yes".

Go ahead and execute the flowchart. Which is it?

Okay, now you know that AND has higher precedence. Let's use parentheses to change the precedence. Put a pair of parentheses around the OR part, to make it get evaluated first. The condition should now look like:

( (2 > 1) OR (3 = 4) ) AND (5 < 10)

Execute again. This time, it should print "No", because it first evaluated False OR True (which is True), and then that result (True) AND False, which is false.
VII) Mutually exclusive cases

Sometimes the logic for a problem takes the form where there are several possibilities, but only one can possibly be true at any one time. For example, consider inputting a number and then testing to see if it is 0, 1, 2, 3, or none of the above. You can solve this problem (with some difficulty) with a bunch of separate IF statements, but it gets ugly. It would look like:

```
INPUT Number
IF Number = 0 THEN
    DISPLAY “Zero”
ENDIF
IF Number = 1 THEN
    DISPLAY “One”
ENDIF
IF Number = 2 THEN
    DISPLAY “Two”
ENDIF
IF Number = 3 THEN
    DISPLAY “Three”
ENDIF
IF Number <> 0 AND Number <> 1 AND Number <> 2 and Number <> 3 THEN
    DISPLAY “None of above”
ENDIF
```

That works, but it is ugly (that last IF is no fun, and would be even less fun if we were checking for 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10). Moreover, it is very inefficient. Consider what happens if the very first IF is true. It displays “Zero”, but then it has to go ahead and test all of the others anyway, even though it is impossible for any of the others to be true. That wastes computer time.
Here’s a better approach:

```
INPUT Number
IF Number = 0 THEN
    DISPLAY “Zero”
ELSE
    IF Number = 1 THEN
        DISPLAY “One”
    ELSE
        IF Number = 2 THEN
            DISPLAY “Two”
        ELSE
            IF Number = 3 THEN
                DISPLAY “Three”
            ELSE
                DISPLAY “Something else”
            ENDIF
        ENDIF
    ENDIF
ELSE
    DISPLAY “Something else”
ENDIF
ENDIF
```

(In your lecture text, you’ll see some alternatives to this algorithm, and some meaningful types of problems that this approach can solve, but for our purposes, this one is the one we’ll use.)

1) Implement this algorithm in a new Visual Logic flowchart. Execute, and test with 0, 1, 2, 3, 4, and -1.
IX) "Factoring"

Consider this flowchart.

Continued on next page:
This flowchart will execute correctly and do what it is supposed to do. (I'll admit, it doesn't do a heck of a lot, but that's okay.) However, the flowchart could accomplish exactly the same task with fewer pieces.

The important thing to notice is that there is an operation (the input for name) that is exactly the same in the true path and in the false path. And that operation happens before the only part that is different (the part that says you are or aren't old enough to vote).

So that operation (input name) really don't have to wait to find out whether the if is true or false. It can be done before the if operation, and in that way, it doesn't need to be repeated in two places in the flowchart.

There are situations where factoring works and situations where it does not. In the lecture text, there is more discussion of this issue.
X) Case Study

Read the Case Study Solution at the end of chapter 2 in the Visual Logic textbook. There are four alternative algorithms to solve the same problem. This is an excellent opportunity to learn about algorithm design and to compare alternative designs. Read it carefully!

The fourth alternative is particularly important. It will be the approach you will use next lab when we try to solve the same problem for 20 values (instead of just 3), and it is the solution you will use this week in the "to turn in" section to solve the problem for 4 values. You’ll have to wait until we get to loops to see the 20 value version, but if you understand this solution, that one will be easy.

TO TURN IN

A) The answers to Quick Check 2-B (1 through 4).

B) Modify the flowchart in Solution 4 of the chapter 2 Case Study (Smallest Number). Make it work for four inputs. Input from the console window, and send the output to the console window. (Include prompts and a meaningful label for the output.) (Some creative copy-and-paste can save you lots of time, but be careful.)

Turn in the printed output, using values 43, -2, -9, and 15. Include printed flowchart.
Activity H: Loops

NOTE: Section 3-1 of the Visual Logic text does a nice job of explaining the use of Console input and output. We've already discussed that at the very beginning of this handout, but if you want a little refresher, feel free to consult that section. (For now, do not read the rest of that chapter. We'll use parts of it in the following two activities, but there is one section in 3-3 which does not follow the approach shown in your lecture text. While that doesn't make it wrong, it probably would be more confusing for you if you read it.)

I) Simple pre-test loop

In a previous exercise, we checked to see if an input value represented a valid exam score. An algorithm we used said:

```
INPUT Score
IF Score <0 OR Score > 100 THEN
    DISPLAY “Not valid”
ELSE
    DISPLAY “Valid Score”
ENDIF
```

In today's first exercise, we'll consider the same problem, but instead of just displaying whether the score is valid or not, we'll force the user to enter a valid score.

But first, we'll do it wrong!

Here's the algorithm we'll try.

```
INPUT Score
IF Score <0 OR Score > 100 THEN
    INPUT Score    // Prompt user telling him/her why entry was invalid.
    ENDIF
DISPLAY Score & " is valid"
```
1) Create a new flowchart and implement that exact algorithm. Be sure to echo the value of the input in the final output operation. And don't forget that you need parentheses for the two parts of the OR.

(Note that there is no ELSE -- that is, your flowchart will not have anything in the False path.)

It sort of looks reasonable. If they enter an invalid score, we'll tell them they made an error and what the error was (as part of the prompt for the next input), and then we'll make them re-do the input. (Note: The "prompt" approach in Visual Logic appears to have a small problem -- if the prompt gets longer than a line or so, it sort of gets lost when displayed. Keep your prompts short enough.)

Let's try it and see.

Execute the flowchart. For your first input, enter 90 (a valid score). Well, that should have worked. It should have printed "90 is valid".

Now execute again. This time, enter an invalid score (101). Well, that looks good, too -- it said we had entered an invalid score and asked us to do it again. So obey the rules -- enter a valid score (90).

Worked again. Printed "90 is valid", as it should have for 90.

Hmm. Maybe I'm wrong. Maybe it really is a good solution to the problem. But let's try it just one more time.

Execute, and this time, enter the invalid value (101). When prompted to re-enter, enter another invalid value, -1.

Uh oh! It says that -1 is a valid score.

Back to the drawing board! What we need is a way to make them re-enter if they enter an invalid score, but then to continue to make them re-enter, time after time, until they finally enter a valid score. We need a loop.
II) Corrected data validation algorithm

INPUT Score
WHILE Score <0 OR Score > 100
    INPUT Score // with appropriate error message prompt
ENDWHILE
DISPLAY Score & "is valid"

Look carefully. The loop says that as long as the value of Score is invalid, we'll keep on looping (and within the body of the loop, re-entering the value for Score).

1) Start a new flowchart. (You can try to re-use the existing one, but I think you'll find it more trouble than it is worth.)

2) Start with the input.

3) Below the input, select the "While Loop".

4) Open the While Loop to enter the condition. Use exactly the same condition you used in the previous version. The rules for conditions are the same for WHILE LOOPS as they are for IFs. (Including the parentheses…)

Notice that the default "Loop Type" is "Pre-test". That's good. We'll use mostly pre-test loops. Pre-test means that the condition will be evaluated before we enter the loop -- look at the pseudocode -- if the condition is true, we enter the loop, displaying the error message and getting another input. But if the condition is false, even on the very first time we reach the loop (in other words, if the user's first entry is valid), we just skip the entire loop!

5) In the True path, put the new input operation, being sure to prompt the user with a message telling him/her that they made an error (keep it to one line).

6) In the False path (i.e., the path that will be taken when the loop condition is not true), put the final output operation.

Execute. Try all of the possibilities we tried before. Execute and enter a valid score the very first time. Then execute again, entering an invalid score first, followed by a valid one. Then execute again, entering an invalid score, followed by another invalid score. That's the case that didn't work right with the IF version. But it works just fine now. We're not getting out of this loop until we finally enter a value that makes the WHILE condition false, that is, until we enter a valid score. Try several more invalid scores. Finally, enter a valid score.
The loop exits (if you did it right) and the correct message is displayed.

III) Another example of data validation

Look at Figure 3.9 in the Visual Logic textbook. Make sure you see how the approach used in this flowchart is exactly the same as the approach you just learned in the preceding example in this handout. (NOTE: The first portion of section 3-3 in the Visual Logic text is good (the introductory paragraph and the "Validating Input" section). However, the discussion of the "sentinel" loop that begins at "Grocery Checkout Revisited" is not handled the way we do it in the lecture text, so you probably should avoid confusion by not reading the remainder of section 3-3. (We'll use sections 3-2 and 3-4 later in this handout.)

IV) Other uses of pre-test WHILE LOOP.

All pre-test WHILE LOOPs operate just like the first exercise. The condition is evaluated before entering the loop. If it is true, go into the loop, then back up to the condition. The flowchart does a really nice job of demonstrating just exactly what is going on. (Better than pseudocode, I think. The arrows really make it clear.) But, of course, the algorithm is the same whether you represent it in pseudocode or with the flowchart!

Here's a common loop situation. We want to add up a bunch of bowling scores. Any time you want to do something a bunch of times, you probably need a loop. But with a WHILE LOOP, you must have some way to end the loop, i.e., to get out of the loop -- something within the loop body must cause the condition to become false.

One approach is to have a special number at the end that is not considered one of the numbers we want to add, but is, instead, just a special value that means we're done entering numbers. This is called the sentinel loop approach. Since the idea of this loop is to add bowling scores, we'll use the special value negative one to mean that it's time to quit. (We'd never have a -1 for a bowling score, even if we were really bad bowlers, so -1 becomes a good choice for the sentinel.)

To add up the scores, we'll enter one score each time through the loop. And we'll add that new score to a running total so that when we're done, we end up with the total of all of the scores (except we don't add in that special sentinel value, since it's not really one of the bowling scores).

We have to make sure the running total starts at zero (that is, it is zero before we add any scores to it). You were told earlier this semester that Visual Logic would automatically make any variable zero, but you were also told that we would not allow that approach.
Here's an algorithm, but it's **not a good algorithm**:

```
Total ← 0
WHILE Score <> -1
    INPUT Score
    Total ← Total + Score
ENDWHILE
DISPLAY Total
```

Don't make your flowchart yet! We have a problem with this algorithm. Look at the WHILE condition. The very first time we get there, is this condition True or False?

The answer is that we don't know! Because Score has no value yet!

Well, how about this?

```
Total ← 0
INPUT Score
WHILE Score <> -1
    Total ← Total + Score
ENDWHILE
DISPLAY Total
```

This time, we do have a value for Score when we get to the WHILE condition. I still don't think this is going to work, but maybe I should let you find out for yourself.

1) Implement **exactly** this algorithm in a new Visual Logic flowchart. Include a prompt to the user with the input operation that tells them to enter a bowling score, or to enter minus 1 when they are done entering all of their scores.

2) **Use the debugger to watch what happens**! If there are some extra variables that show up in the variable watch window, don't worry about them. You just need to watch what is happening to Total and Score, and, most importantly, you need to watch each statement get executed so you see what is going on.

   So, press the function key <F8> to start stepping through the flowchart.

3) On the first press of <F8>, you'll see that the focus moves to the Total = 0 assignment operation. Press <F8> again. You should see a zero show up for Total in the variable watch window.

4) Press <F8> again. This executes the input operation. Enter -1. (You will probably have to click in the input box to move the focus there so it will accept your entry.) You should see the -1 in the watch window for Score, and execution now moves to the WHILE (but recall from earlier in the semester, the highlighted operation is the one that is **about to be executed**).
5) Press <F8>. This gets you to the WHILE. Press <F8> again. The WHILE condition is evaluated, and since it is false (that is, it is not true that Score is not equal to -1), the entire loop is skipped and execution moves to the output operation on the False path.

6) Press <F8>. The output is displayed. (The correct value for Total, 0, is displayed.) One more <F8> and the execution of the flowchart ends.

Hmm. I don't know that this showed you much new. Let's try again.

1) Press <F8> to get to the assignment operation, press <F8> to execute that assignment and get to the input. <F8> again to execute the input operation. This time, enter 250. (That's my typical bowling score, of course.)

2) You should see that in the variable watch window, Total is 0 and Score is 250. And the highlighted next operation is the WHILE. Press <F8> to evaluate the WHILE condition, which is True this time. (For some reason, it sometimes takes one more <F8> to get the WHILE to evaluate. Not sure why.)

3) You are now in the loop body, ready to execute the assignment statement that adds Score to Total. Press <F8>, and you'll see Total get 250 added to it. And we're back to the WHILE condition.

4) Press <F8> again. Well, Score is still 250, so the WHILE condition is still true. Press <F8> to execute the assignment statement. Now Total is 500. Try <F8> again, and again, and again, and again, and…..

Hmm. This is not good! You are in the dreaded infinite loop. That is, there is no way to get out, because nothing in the loop body can cause the WHILE condition to ever be false.

5) Terminate the execution by using the Debug menu's Terminate option.

We need to fix this. Hopefully you saw that, once we got the first score (which we did before ever getting to the loop), we never got another score.

1) Fix the algorithm. Add another input (for Score) right after the assignment operation that added Score to Total. The algorithm (in pseudocode) now looks like this:

```
Total ← 0
INPUT Score
WHILE Score <> -1
    Total ← Total + Score
```
INPUT Score
ENDWHILE
DISPLAY Total

Make your flowchart match this algorithm. (The prompt for input must always remember to tell the user not only what they are supposed to enter, but also how they can get out -- that is, tell them to enter -1 when done.) You can use some copy-and-paste to save typing.

Let's use the debugger approach again.
2) Press <F8> until you get to the first input, and enter 250 for the score. Watch the values in the variable watch window and you press <F8> again and again and… until you get to the next input. This time, enter 300.

3) Again, repeatedly press <F8> until you get to the next input (and keep an eye on the variable watch window -- there's a lot to be learned from watching how the values change). This time, enter 225.

4) Again, <F8> until the next input. This time, enter -1. Now press <F8> and step through until the execution ends, watching which instructions are being executed.

That should show you an awful lot about WHILE loops (and accumulating totals).

V) Improving the bowling score algorithm.

It really isn't very useful to know the total of all of your bowling scores. But it might be useful to know the average of your scores. To do that, we need to add another piece -- we need to keep track of how many scores got entered (again, not counting the sentinel, which isn't really a score at all).

1) Put an assignment operation as the second operation in the flowchart (that is, right under the one that sets Total to 0). In that operation, set the variable Count to 0.

2) Inside the loop, right after the assignment operation that added Score to Total, put a new assignment operation that adds one to the value of Count. (Count = Count + 1)

This will keep track of the number of times we've gone through the loop.

3) Now all you need to do is change your final output operation -- instead of displaying the total, display the total divided by the count!

Execute. We can abandon the <F8> debug approach. Just run in the usual way. Enter scores of 250, 300, and 225 again. (And end with -1.)

If all goes well, you should have an average of (250 + 300 + 225) / 3, or 258.333…
4) Execute again, this time using just two scores, 200 and 300 (and then -1). That should work fine, too.

5) One more time. This time, just one score, 275 (and -1). Still should do just fine.

VI) BIG TROUBLE!

Execute the flowchart one more time. This time, start off with -1 as the very first entry. (In other words, the user ran the program, but realized he hadn't bowled any games.)

What happens?
More importantly, WHY???

Somewhere in your math education, you should have learned that division by 0 is undefined. And the Visual Logic program can't do something that's undefined. But that's just what you told it to do. When the user enters -1 as the first entry, the loop works just fine. That is, it just skips over the whole loop, since the WHILE condition is false.

But then it gets to that final output operation, which attempts to divide Total by Count. And Count is zero. (So is Total, but that's not the problem -- it's okay to divide zero by something, as long as you don't divide it by zero. You cannot divide ANY number by zero.)

Time to fix the problem!

1) There's nothing wrong until we get to that last output operation. So, immediately before that operation, put an IF. The condition must check to be sure Count is not 0. (Count <> 0)

2) On the True path, put your current output operation. (Just click and drag the operation.) Because, if it is true that Count is not zero, it is okay to do the division and print the result.

3) On the False path, put an output operation that just prints a message saying you cannot calculate the average since there were no games played.

Execute -- try a bunch of combinations. Multiple scores, one score, no scores. It should work for all cases!
VII) Another use of WHILE LOOP -- a "counter-controlled" loop

Sometimes we have a situation in which we have some action that has to happen some specific number of times. For example, perhaps you have a Computer Science course which has five quizzes. We want to get the quiz average. Rather than using the sentinel approach you learned previously, we'll use a counter-controlled loop to execute the loop exactly five times.

Here's the algorithm:

```
Max ← 5    // this is a named constant
Total ← 0
Count ← 1
WHILE Count <= Max
    INPUT Score
    Total ← Total + Score
    Count ← Count + 1
ENDWHILE
DISPLAY Total / Max
```

Notice that when it is time to calculate the average we divide by Max, not Count. Look carefully at this algorithm -- see if you can figure out what the value of Count would be when we reach that DISPLAY statement.

1) Implement the logic of that algorithm in a Visual Logic flowchart.

2) Execute the flowchart. I don't think you really need to use the debugger, but if you enjoyed watching the execution one statement at a time, go ahead and use the debugging approach.

Use any values you like for the scores, but be sure to add them up yourself to make sure the flowchart is executing correctly. It should print the total divided by 5.

This should have worked just fine.

VIII) Slight variation on counter-controlled loop

What if we don't know at the time we write the algorithm exactly how many quizzes there will be? No problem. Instead of using the named constant with the predefined value of 5, we'll just let the user input how many quizzes there were.

1) Change your flowchart. Remove the assignment operation that assigned 5 to Max, and instead, do an input operation to obtain the value for Max from the user. Provide a helpful prompt to ask the user how many quizzes there are.
2) Execute the flowchart. When asked, enter 3 as the number of quizzes. Then see if it correctly gets exactly three scores and divides the total by 3 to get the average. It should.

BUT TROUBLE ONCE AGAIN REARS ITS UGLY HEAD

3) Execute again. But this time, enter 0 as the number of quizzes. Uh oh! Division by zero again. (Be sure you understand why this could not have been a problem with the previous version, where the number of quizzes was a constant.)

4) **THIS IS TO BE TURNED IN.**
Fix the problem, using exactly the same approach shown previously to prevent division by zero. Change the input and output operations to use the terminal.

Execute the flowchart using 3 quizzes, then execute again using 0 quizzes. Print the output for both cases, and create the printable version of the flowchart.

IX) YES/NO Loop

What if we don't know how many loops to do, and the user won't know how many there will be when it is time to run the "program", and there is no possible special *sentinel* value because all values are reasonable data?

Consider an algorithm to add up all the numbers entered by the user -- any kind of number, any number of numbers.

Since any number is okay, we can't use a special number as the *sentinel*. And since the user can just enter numbers until he/she feels like quitting, we really can't ask how many there are going to be. For this case, we need a different WHILE loop approach. We'll just ask the user each time whether or not he/she wants to put in another number. And the condition for the WHILE loop will be that we'll stay in the loop until he/she says "No".

With this kind of situation, it is usually a good idea to give the user the opportunity to quit at any point, including before entering any numbers. Here's an algorithm:

```
Sum ← 0
INPUT Again
WHILE Again <> "N"
   INPUT Number
   Sum ← Sum + Number
   INPUT Again
ENDWHILE
DISPLAY Sum
```
1) Implement a Visual Logic flowchart for this algorithm. In your input prompts to the user, be sure to explain what you expect them to answer. That is, tell them to enter "N" to quit, anything else to continue.

2) Execute the flowchart. Try different sets of data -- no numbers (in other words, enter "N" the very first time), one number, several numbers. It ought to work for all cases.

This approach will work in any similar situation. But it should be the last alternative -- it is not as good as the other two approaches. As part of the "TO TURN IN" section for this activity, you'll be asked why.

TO TURN IN:

1) The quiz score flowchart and two sets of output from part VIII.

2) a) Reconsider the YES/NO loop -- section IX. If the user had 100 numbers to enter, how many times would he/she have to put in an input? (Answer on paper to turn in.)

b) Now, what if we had a similar loop, but we used the counter-controlled approach. The user is still going to want to add up 100 numbers. How many times would he/she have to put in an input? (Assume we use a named constant for the maximum number of entries.)

c) Finally, consider the sentinel loop approach. Assuming we could come up with a sentinel-type approach for this problem, exactly how many times would the user have to put in an input?

I hope you see why the YES/NO loop should be used only when neither of the other approaches can be made to fit.

Put your answers to these three questions together with your flowchart and output for part 1 to turn in.
Activity I: More loop stuff (for loop, pre-test, post-test,)

In the previous in-lab activity, you learned about several uses of pre-test WHILE loops. One of those was the "counter-controlled" loop.

It turns out that this is such a common situation in programming that most programming languages (and Visual Logic) have "shorthand" for such loops.

The most important thing to understand is that this is just shorthand for the counter-controlled loop you learned in the previous exercise. It works exactly the same way -- it just saves you some typing.

I) FOR loop as alternative for counter-controlled WHILE loop.

Here's a typical counter-controlled WHILE loop algorithm. You saw it in the previous exercise.

```
Max ← 5     // named constant
Total ← 0
Count ← 1
WHILE Count <= Max
    INPUT Score
    Total ← Total + Score
    Count ← Count + 1
ENDWHILE
DISPLAY Total / Max
```

Now, here's the **exact same algorithm**, but using a FOR loop instead of WHILE.

```
Max ← 5
Total ← 0
FOR Count = 1 to Max
    INPUT Score
    Total ← Total + Score
ENDFOR
DISPLAY Total / Max
```

Look carefully! There are two statements in the WHILE version that are not in the FOR version. The line (Count ← 1) is missing, and the line (Count ← Count + 1) is missing.

That's because the FOR loop *automatically* does those two actions. And it does them at *exactly the same places* where they get done in the WHILE example. There's no difference -- it just saves you a couple of lines of typing.
Use section 3-2 of the Visual Logic text as your guide for this activity:

1) Implement the FOR loop algorithm using Visual Logic. You'll see that there is a selection for that. When you open up the For Loop box, you'll see that there are places for the name of the variable (we're using Count), the Initial Value (we want to start the value of Count at 1), and the Final Value (we want to end when Count reaches Max). There's also a place for Step. Step is used to tell how much to add to Count each time we reach the end of the loop. We've been using 1, so just leave the 1 that is there by default.

2) Execute the flowchart. Use the values 90, 80, 95, 85, 90. The average for that set of data should be 88. Make sure that's what you get.

3) Let's add one more piece, just to make sure you see what is happening to the Count variable. Inside the loop body, put this prompt for the Input operation. Put exactly this in the prompt:
   "Please enter score number " & Count

Execute, and make sure you see that the value displayed for Count is 1 the first time, 2 the second, etc. That's exactly what would have happened if you had used the "longer" version, the WHILE version, and had displayed the value of Count at that point. Make sure you see that the FOR loop really does have a variable named Count, and that it really does get the value 1 at the beginning of the loop, and that it really does get 1 added at the end of each iteration of the loop!

II) **Nested loops**

You can put loops inside loops, just like you can put IFs inside IFs.

Let's return to the bowling scores problem we solved in the previous in-lab activity, but with a couple of important changes. In the previous version, we used a sentinel loop -- having the user enter -1 when they had entered all the scores they had.

This time, we're going to assume the user is always going to enter exactly three scores. (That's actually typical in a bowling situation, if my memory serves. I haven't bowled in about 20 years. But I recall that three games comprised a *series.*)

We want to calculate the average for three games.

Here's our starting point, then:

\[
\begin{align*}
\text{Max} & \leftarrow 3 \\
\text{Total} & \leftarrow 0 \\
\text{FOR Game} = 1 \text{ to Max} \\
& \quad \text{INPUT Score}
\end{align*}
\]
Total ← Total + Score
ENDFOR
DISPLAY Total / Max

1) Go ahead and create the Visual Logic flowchart for this algorithm. (You can probably save a little time if you modify the previous flowchart, but do it very carefully.) Be sure to put a nice prompt that specifies that these are bowling scores. Execute it and make sure it correctly calculates the average for exactly three games.

Now we're going to improve this algorithm a bit. It turns out that bowling scores must be between 0 and 300. (Don't ask why, that's just the way the game is.)

You learned in the previous in-lab activity that you can use a WHILE loop to force a user to enter valid inputs. Let's add a WHILE loop to make the user enter a score between 0 and 300.

In your flowchart, within the FOR loop body, we're going to do a "validation" loop for each input. Recall that the logic for such a loop looks like:

INPUT Score
WHILE Score <0 OR Score > 300
    INPUT Score  // be sure to include a good "error message" prompt
ENDWHILE

You should have learned that such a loop will ensure that, by the time that WHILE loop finally exits, a valid score has been entered. It might have been entered on the very first try (in which case the WHILE loop does not execute at all) or on the second try, third try, whatever. But before we get past that ENDWHILE, there is a valid score.

2) Add this loop inside the body of the FOR loop. In other words, right after the Input operation, add the WHILE loop. The loop should end before you add the score to the total -- that way, the score you add to the total will be a guaranteed valid score. Your flowchart should now look like:
Begin
Max = 3
Total = 0

Game 1 to Max

Input: Score
(Score < 0) OR (Score > 300)
True
False

Total = Total + Score

Output: "Average is " & Total / Max

End

Be sure to put a prompt in the new Input operation – it should say something like "Invalid score. Please re-enter."

(Note: Visual Logic doesn't display a multi-line prompt. If it did, we would want a better error message, telling the user that the score must be between 0 and 300. But we'll have to live without that for now.)

Voila! You have now seen your first nested loop. The FOR loop is the outer loop. It will execute exactly three times. But each time through that outer loop, when you get a bowling score input, you come to the inner WHILE loop. (Of course, like any WHILE loop, it might not execute at all, if the user entered a valid score, but it might execute. In fact, it might execute any number of times -- once for each invalid score entry.) Once the user has entered a correct (valid) score, the score gets added into the total, and we get to the next outer loop. Within that next loop, we might again do some number of iterations of the WHILE loop. And so on. That's a nested loop.

Execute the flowchart.
For the first score, enter a valid value. The WHILE loop is skipped, and we go right to the second score. This time, enter an invalid score (301). The WHILE loop is entered, and you are asked to re-enter. Enter a valid score. The WHILE loop terminates, and we add the score to the total, and on to the third outer loop.

For the third loop, enter quite a few invalid scores -- finally, you'll have to enter a valid score to get out of the WHILE, and, since that's the third score, the FOR loop ends and we calculate and print the average.

III) Another nested loop

Let's do a little nonsense flowchart -- this won't really solve any problem, but it might give you a little more insight into nested loops. This one is going to have nested FOR loops.

Read section 3-4 of the Visual Logic text.

1) Create the flowchart in Figure 3.14 of the Visual Logic text.

Make sure you use console output for your output statement. And make sure you put a blank line at the end of the output (just before the squiggly end-of-output mark) so you get each output on a new line.

2) Execute the flowchart and carefully evaluate the output. It should show you a lot about nesting loops.

IV) Pre-test vs. Post-test loops

The WHILE loop we've been using, and the FOR loop, are both pre-test loops.

Definition: Pre-test loop
A pre-test loop is one in which the loop condition is tested before entering the loop. If the condition is true, the loop body is executed and execution returns to the loop condition test. In a pre-test loop, if the loop condition is false the very first time, the statements in the loop body do not get executed at all.

Definition: Post-test loop
In a post-test loop, the test of the loop condition occurs after the body of the loop has been carried out one time. If the condition is true, execution returns to the top of the loop and the loop body is executed again. In a post-test loop, the statements in the body of the loop are always executed at least one time.
We are not going to spend much time on post-test loops. There are occasional situations where they are useful, but for our purposes, we can write all of the algorithms we're going to see this semester with pre-test loops (WHILE or FOR).

However, you ought to see how a post-test loop works, and how it compares to a pre-test loop, so we'll do one example where we solve the same problem with each of the loop types.

You have (several times) seen how to use a pre-test WHILE loop to "validate" data, that is, to force the user to enter a valid value. Here's an example from a previous exercise.

```
INPUT Score
WHILE Score <0 OR Score > 300
   INPUT Score // be sure to include a good "error message" prompt
ENDWHILE
DISPLAY "Valid"
```

This is a pre-test loop – it first checks to see if Score is outside the valid range, and only if that condition is true, we will carry out the loop body (which, in this case, consists only of the second Input statement). As you know, if the condition is false the first time (that is, if the very first Input, entered before the loop, is not out of range), we'll never get into the loop, just skipping over the loop entirely and continuing after the ENDWHILE.

1) Create a Visual Logic flowchart that matches the logic of the pre-test loop algorithm. **Be sure to include appropriate prompts.** The first Input has a prompt that just says to enter a bowling score. The second Input has a prompt which says you entered an invalid score.

2) Execute the flowchart, trying it a couple of times. One time, enter a valid score as the very first entry. Notice that you don't see any error message. Another time, enter some invalid scores before entering a valid one. Notice how the error message comes up when you entered an invalid score.

Here is a post-test loop used to solve the same problem.

```
DO
   INPUT Score
   WHILE Score < 0 OR Score > 300
   DISPLAY "Valid"
```

Notice that this approach looks a bit simpler – there's only one Input operation. The loop body contains just that one statement – just as in the pre-test version. But there is no
need to have an additional Input operation before the loop, as there was in the pre-test version.

When this gets executed, it will always enter the loop the first time – the Input statement in the loop body gets executed. Then it reaches the condition – if the condition is true (the Score is out of range), we return to the top of the loop (the "DO") and execute the loop again.

When (eventually) the user enters a valid score, the condition is false, and we leave the loop, going on to the DISPLAY statement outside the loop.

1) Create a new Visual Logic flowchart for this algorithm. To create the post-test loop, select WHILE loop icon, just as you did for pre-test loops. But when you open it up to enter the condition, choose the radio button that says "Post-Test".

Put the Input operation within the loop, i.e., on the True path. For the input prompt, say "Enter a bowling score."

The False path is the exit to the loop – just as in the pre-test approach. Put the DISPLAY on the False path.

2) Execute the flowchart. You'll see that the first thing that happens is that you are asked to enter a score. Enter a valid bowling score. Execution continues to the condition, and since the condition is false, we leave the loop.

3) Do the same thing again, but this time, enter two or three invalid scores before entering a valid score. You should get a clear picture of just how this loop operates.

Well, it certainly looks like post-test loops are fine for this kind of problem, and they are a bit easier to write, since there is one less step (as compared to the pre-test loop version).

But I don't think this works well at all!

4) Execute the flowchart and enter a couple of invalid scores. Look carefully at the message you get each time – it just asks you to "Enter a bowling score. The earlier (pre-test) version would have told you that you had made a mistake when you entered an invalid score.

Try to figure out a way to provide an error message. But, of course, you don't want to tell them they made a mistake for the first entry – they haven't made a mistake yet.

Here's a modification to the algorithm which will solve this problem.

    DO
    INPUT Score
IF Score < 0 OR Score > 300 THEN
    DISPLAY an error message
ENDIF
WHILE Score < 0 OR Score > 300
    DISPLAY "Valid"
ENDWHILE

5) Add the logic of that algorithm to your flowchart. (That is, add an IF with an Output of an error message on the true path, nothing on the false path.) Of course, the IF must follow the Input operation.

6) Execute the flowchart, and enter a valid score the first time. That should work nicely – you won't see an error message, and you get the ending message saying the score is valid.

7) Execute again – this time, enter several invalid scores before entering a valid one. That works, too. You get the error message for every invalid score. This now works just like the pre-test version we had before – an error message for every invalid input.

Problem solved!

But do you still think the post-test approach is simpler than the pre-test one?
(If you answered "yes" to that question, you'll probably be one of those programmers who uses post-test loops most of the time. I'm not. I almost never use one.)

TO TURN IN: There is nothing to turn in from this activity. In the next activity, you'll get a chance to work with loops in several different ways (to turn in).
Activity J: Loop problems for you to solve

This week's activity is entirely problems for you to solve and hand in. Consider it a "review" of loops.

1) Three solutions, one problem!

For the first portion of this week's activity, you are going to solve exactly the same problem using three different loop types. You'll use a pre-test WHILE loop, a post-test WHILE loop, and a FOR loop.

Be aware that you can always make a pre-test loop and a post-test loop solve the same problem (although you sometimes need an extra IF in the post-test version). But in order to have a problem which can be solved by all three loop types, we must specify a problem that involves some sort of counter-controlled loop, since that's the only type of problem a FOR loop is used for.

So here's the problem:

The user will enter exactly six numbers. Your flowchart should print out the number of even numbers entered and the number of odd numbers entered. Provide suitable prompts and output labels.

A) Solve this problem with a Visual Logic flowchart that uses a pre-test WHILE loop. (Remember that an even number is one which is evenly divisible by 2 – that is, when divided by 2, the remainder is 0.)

Be sure to use a named constant (Assignment operation) for the number of values to be entered (6). And be sure you actually use that constant in the loop condition.

Print the flowchart and the output. Your input should consist of six well-chosen values. Some should be odd, some should be even. Don't come up with exactly half odd, half even, because we wouldn't be able to tell if you had really solved the problem. (Think about that – you'll be asked why before this activity is done.)

Save your solution flowchart (not just the printable version, but the actual flowchart). We'll use it again in part E.

HINT: After you've saved the flowchart, leave it open. Then, when you go to work on part B, you can re-use what you have, and after making the necessary changes, use the SaveAs option to save the new version with a different name. The change to part B is very simple (one step, in fact).

B) Solve the same problem with a Visual Logic flowchart that uses a post-test WHILE loop. Use the same inputs for your test output. Print the flowchart and output.
Be sure to use a named constant (Assignment operation) for the number of values to be entered (6).

**Save your solution flowchart with a new name. We'll use it again in part E.**

C) Solve the same problem using a FOR loop. Use same inputs. Print flowchart and output. (You can try to use the previous one as a starting point, but there will be more changes than there were between parts A and B. Make sure you don't leave in any statements that are no longer necessary. Be sure to **SaveAs** with a new name.)

Be sure to use a named constant (Assignment operation) for the number of values to be entered (6).

**Save your solution flowchart with a new name. We'll use it again in part E.**

D) Once you have printed the flowcharts and output for the three cases, answer these two questions:

1) Why did we suggest that you not have exactly three even and three odd values? (Be specific.)

2) For this specific problem, which of the three loop types do you like the best, and why?

E) Change each of the three flowcharts as follows, and for each, execute with the specific data suggested below.

There is only one change. Remove the Assignment operation you used to set the value of the named constant to 6. Replace it with an Input operation – ask the user how many numbers he/she wants to enter. **Make no other changes.**

Execute each flowchart twice. The first time, enter 3 as the number of numbers to be input. Observe how each of the flowcharts behaves (that is, does it work correctly?)

The second time, enter 0 as the number of numbers to be input. Observe how each of the flowcharts behaves.
Answer this question:
Did all three flowcharts work correctly? If not, which one(s) did and which one(s) did not.

If any of them didn't work just because you forgot to use the named constant in your loop condition, fix that and re-run them. Don't consider that as a problem with one of the loop types.

F) Assuming that one of the flowcharts did not work correctly (and I think that should be the case), fix it. Keep the same loop type, but use whatever additional logic you need to make it work correctly. Keep in mind that 0 is an acceptable answer to the question of how many numbers are to be entered. It just shouldn't do any of the loop stuff if they entered 0. (It should still print the counts of odd and even numbers – which should be 0.)

Print the fixed flowchart.

II) Finding largest number

In a previous lab activity, your Visual Logic textbook showed you how to find the smallest of three numbers, and you then wrote an improved version which did the same for four numbers. You might want to review the algorithm you used (recall that there were several different algorithms in the book, but we ended up using the last one because I said we'd use a similar approach when we got to loops).

Write an algorithm (in pseudocode) to find the largest of any number of numbers entered by the user. Begin by asking the user how many numbers there will be. Use a FOR loop.

Then create a Visual Logic flowchart for that algorithm.

Execute the flowchart, using 10 numbers. For the output to print and turn in, you may pick any numbers you want to use, but make sure there are positive numbers, negative numbers, and 0. And make sure that you don't put the largest one first or last (although you should certainly test your flowchart to make sure it does work with the largest number first, last, or somewhere in between).

Print the pseudocode, the flowchart, and the output.

III) Printing figures

In the Visual Logic textbook, Figure 3.18 shows a triangle created using the letter "o". A flowchart is given to accomplish that. (You need to study that flowchart very carefully – yours is going to include some very similar stuff!)
Create your own flowchart to print a figure which is similar, except:
   a) It is upside down (as compared to the one shown in the book),
   b) the number of rows is an input from the user, and
   c) we'll print multiple figures. You'll use a **triplly-nested-loop** – the outer loop is
      a WHILE loop using the **sentinel** approach, based on the value the user enters as
      the number of rows. When the user chooses to quit, he/she will signal that by
      entering a negative one. That is the **sentinel** value – don't try to print another
      figure – just use that value as the signal that it is time to quit. Prompt the user
      appropriately.

      Then you'll use two (nested) FOR loops, very much like the example in the text,
      but with an important modification. Hint: The first (outer) of the nested FOR
      loops is going to have to count backward – consider initial value, final value, and
      **step**. (The innermost FOR acts just like the one in the book, but let's use an
      **asterisk** as the symbol to print.)

      Try to leave blank lines similar to the example I've shown below. That's going to
      take a bit more thought!

Here's a sample of what an execution of your flowchart might look like:

<table>
<thead>
<tr>
<th>How many rows do you want (or enter -1 to quit):</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>How many rows do you want (or enter -1 to quit):</td>
<td>6</td>
</tr>
<tr>
<td>*****</td>
<td>****</td>
</tr>
<tr>
<td>*****</td>
<td>****</td>
</tr>
<tr>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td>**</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>How many rows do you want (or enter -1 to quit):</td>
<td>-1</td>
</tr>
</tbody>
</table>

Turn in the printout of the flowchart and output for the values 2, 9, 0, 1, and then -1.
Activity K: Arrays

Chapter 4 of the Visual Logic textbook is excellent (with one odd exception). We'll use it as parts of this and the next in-lab activity.

I) Arrays (from the text)

**Very carefully read section 4-1.** The only difference between the arrays described in this chapter and the ones shown in your lecture textbook are that arrays in Visual Logic start at 0 (zero-based arrays), whereas arrays in the lecture book start at 1.

1) Create the Visual Logic flowchart shown in Figure 4.4 (the one that calculates the square of each index). Make sure you understand the FOR loop and the calculations (particularly how the FOR loop and the calculations relate to the array indexes).

Add another FOR loop after the end of the first loop – loop 10 times, and print the value of index, followed by a colon, followed by the value of that element. (Use `console` output, so you can see the whole picture.)

**NOTE:** Can you think of another way to write the Assignment statement that currently says, "Squares(Count) = Count * Count"? (Look at Table 1.2, from the first chapter of the Visual Logic text.)

2) Create the Visual Logic flowchart for the Reverse Order problem. (Quick Check 4-A). In the prompt to the user, tell them which value they are entering. The prompts should look like:

   Enter value number 1:
   Enter value number 2:
   etc.

Be sure the prompt starts at 1, even though the index is 0. Humans start counting at 1, no matter how the computer wants to store stuff.

(Hint: Use the index in the prompt, but do some arithmetic on it first.)

To turn in: Questions 1, 2, and 3 of Quick Check 4-A.

3) Read 4-2. This is a very typical example of a problem which cannot be solved (conveniently) without using an array. It closely parallels examples in your lecture textbook. Study the flowchart shown – but you don't have to type it in. (You'll be doing a very similar one to turn in, later.)
4) Read Section 4-3. Alas, section 4-3 has a problem. It does a fine job of showing two interesting approaches to using arrays and array indexes, and should be read carefully and understood. But you should immediately recognize the problem, based on your own work from Activity J.

You don't have to create the two flowcharts in this section, but you do have to answer a question.

To turn in: What is wrong with the two flowcharts? (Could this exact problem have been solved without using an array?) Explain.

II) A problem (similar to section 4-3) which cannot be solved without an array.

We'll return to the Visual Logic book later.

For now, consider this problem.

Find the largest number in a set of numbers input by the user. Print the largest number. Then print the difference between each number and the largest number.

For this, we're going to look at two algorithms. The first works fine for a very small number of numbers. Here's the first algorithm in pseudocode.

```
INPUT a
INPUT b
INPUT c
Largest ← a
IF b > Largest THEN
    Largest ← b;
IF c > Largest THEN
    Largest ← c;
DISPLAY Largest
DISPLAY Largest – a
DISPLAY Largest – b
DISPLAY Largest – c
```

Look this over carefully. You should see that it works fine for three numbers. But what would happen if you were now told to change it to work for 20 numbers, or 50, or 1000? You'd be writing for a long time!

Now consider the following algorithm, which uses an array. Please be sure you see that the array is not necessary to find the largest number. But it is essential for printing the differences.
NOTE: For all array examples provided by us, we'll follow the logic shown in the lecture book – the first index will be 1, not 0. (Some languages use 1, some use 0. But using 1 makes more sense, since that's how we count, as humans.) So an array declared with 10 as the upper bound will be used to hold 10 elements, element(1), element(2), …, element(10).

We'll write the algorithm to handle any number of numbers – we'll let the user say how many. Notice that the approach we use to find the largest is exactly the one used in the Visual Logic book's Figure 4.7.
INPUT HowMany
Declare an array named Numbers with HowMany as the upper bound
// Get the first element, outside the loop, and also make it Largest
INPUT Numbers(1);
Largest ← Number(1);
// Fill the array for the rest of the values, and find the highest as we're doing it
FOR i = 2 to HowMany
  INPUT Numbers(i)
  IF Numbers(i) > Largest THEN
    Largest ← Numbers(i)
  ENDIF
ENDFOR
DISPLAY Largest
// Now display the differences.
For i = 1 to HowMany
  DISPLAY Largest – Numbers(i)
ENDFOR

1) Create a Visual Logic flowchart for that algorithm. Be sure to prompt the user nicely
for input, and put a meaningful label with the output.

2) Execute the flowchart, entering 3 as the number of numbers. Then use any three
numbers you like and make sure it works. Try it a few times – try it with the largest
number first, then try it with largest second, finally with largest third. Use negative and
positive numbers.

3) Execute again for three numbers. Make sure all three of the inputs are negative
numbers. It should work. Remember that for later!

4) Execute it one more time, but this time with 10 numbers.

(Think about how big a flowchart you'd need to solve the same problem without an array.
Be sure you see that only the last part would change – but it would get ugly for 10
numbers, and really ugly for 100 numbers.)

III) A **WRONG** algorithm for the same problem

1) Make the following changes to your flowchart.
   a) Remove the Input operation for Numbers(1).
   b) Change the next operation (the Assignment operation) to assign 0 to Largest.
   c) Change the FOR loop to input **all** of the values. (That is, start the loop at 1
      instead of 2.)

2) Execute the flowchart for 3 numbers. Use 5, 50, and 11.
That should have worked just fine. (Check your output to be sure.)

3) Execute again, for 10 numbers. Use 3, 0, 6, 9, -5, -2, 3, 4, -9, and 1.

It should have worked that time, too. (Be sure.)

4) Execute one more time. This time for 3 numbers again. Enter -2, -4, and -1.

Did the algorithm find the correct "largest number"?

To turn in: Why did this algorithm fail?

(The correct algorithm, which you did first, is one of those basic algorithms that all computer programmers learn and use whenever confronted with such a problem. You should remember it.)

TO TURN IN:

A) Questions 1, 2, and 3 of Quick Check 4-A.

B) In Section I, answer the question "What is the problem? Could this exact problem have been solved without using an array? Explain."

C) In Section III, answer the question "Why did this algorithm fail?"
Activity L: More Array stuff

I) Psuedo-random numbers

Read the first portion of section 4-5 of the Visual Logic text. Stop at Quick Check 4-E. (We'll do the second portion later.)

1) Do not create the complete flowchart in Figure 4.12. Instead, create a flowchart that does exactly the same thing, but without printing the histogram. (In other words, leave out the histogram FOR loop.)

(As a matter of style, I really think the number of sides for the die, 6, should be a named constant, but I won't fight about it right now.)

2) Do Quick Check 4-E. (Notice that the book says that 40000 rolls might take a minute or two. Without the histogram, it shouldn't take more than a couple of seconds.)

3) Go back and read the last sentence of paragraph four, section 4-1. It begins with "The array elements…"

To turn in:
4) Now fix the flowchart so that it follows the excellent guidance provided by that sentence (but, alas, not followed by the book's own example).

(Yes, I know it works the same way with or without initializing the accumulator values to zero, but it's wrong! When the flowchart says, "Counter(Roll) = Counters(Roll) + 1, it is using poor logic. The initial value of one of the operands on the right side of the assignment operator, Counters(Roll), has not been established. Just because some programming language or cute tool like Visual Logic might initialize some variables automatically is no reason for you to ever depend on that in your logic. Take control!)

Execute again. Use the same values required in Quick Check 4-E. You should get nearly the same results -- "nearly", because it is still random.

Run one more time, with 10,000 rolls. Print the output from that execution, along with the corrected flowchart.

II) Parallel Arrays

Carefully read the second portion of section 4-5, Parallel Arrays (Girl Scout Cookies). The concept of parallel arrays is a very important one, although you will find out in the future that some of the newer approaches to programming have other ways of handling the same problems.
This section is only two pages long (with one of the two pages being pictures). But there's enough there to make the point. The whole idea is really shown in Figure 4.14.

Next, we'll solve a different, but similar, problem using parallel arrays. For this problem, we're going to create two arrays, one with student ID numbers, one with calculated final grades. Then we'll let the user check to see what grade any student made.

**NOTE: For this entire exercise, you will be using Social Security Number-type values as the student ID numbers. These are strings, not numbers!**

The design of this problem shows that there are two different tasks -- filling the arrays, and searching for grades. Here is the algorithm for filling the arrays.

- **INPUT NumberOfStudents**
- Declare array StudentID with NumberOfStudents as upper bound
- Declare array Grade with NumberOfStudents as upper bound
- // We will ignore element 0
- FOR Count = 1 to NumberOfStudents
  - INPUT StudentID(Count)
  - INPUT Exam1
  - INPUT Exam2
  - INPUT FinalExam
  - Grade(Count) ← (Exam1 + Exam2 + FinalExam) / 3
- ENDFOR

1) Create the Visual Logic flowchart for this portion of the algorithm. (There's more to come, but let's take it one piece at a time.) This first portion of the algorithm will calculate the final grade for each student (as the average of three exams) and store it in an array element of the Grade array that has an index corresponding to the ID of the student in the array StudentID.

Be sure to use meaningful prompts. The user should be told to enter a student's ID number, then to enter that student's exam scores. Make the prompt really nice -- have the prompt for the ID say something like "Enter the ID for student number 1", "Enter first exam for student xxx-xx-xxxx", "Enter second exam for student xxx-xx-xxxx", etc., where "xxx-xx-xxxx" is the StudentID that was just entered. (Of course, for the second student, the prompt for ID should say "Enter the ID for student number 2".)

You'll have to think a bit to get those prompts. Use the Count variable to tell the user which ID he/she is inputting. Then use the current element of the StudentID array as part of the prompt for the three exam grades.

Execute the flowchart. Enter 3 as the number of students. It should prompt you for the ID and grades of three students. **Remember that you have to put strings in quotes**
when you input them. The Student ID is a string, not a number. (If you forget the quotes, it will treat the dash as a minus sign and do arithmetic on the ID you entered!)

Unfortunately, we can't tell if everything got put in the right place. Let's solve that with another simple loop.

2) Add the following logic after the end of your FOR loop.

    FOR Student = 1 to NumberOfStudents
            DISPLAY StudentID(Student) and Grade(Student)
    ENDFOR

Send output to the Console. Be sure the subsequent outputs show up on new lines.

Execute the flowchart, again for three students. Put in the following values:
"111-11-1111"
90
80
96

"234-56-7890"
75
85
82

"987-65-4321"
60
80
70

If you did everything correctly, the console output should have the following information (hopefully formatted to look nice!)

    111-11-1111:  88.6666…
    234-56-7890:  80.6666…
    987-65-4321:  70

3) Now it is time to add the part where the user can check on the grade of any particular student. This is called a sequential search. We're going to have the computer look at each ID in sequence until it gets to the one the user wants. Once it finds the ID the user is looking for, it can use the index of that array element to display the grade for that person.
We'll put that search inside an outer loop that allows the user to search repeatedly. We'll use a *sentinel* loop for the outer loop.

(The approach we're going to use for the search is not the best. You'll get the chance to improve it for the "To Hand In" portion of this week's activity.)

```
INPUT TargetID
WHILE TargetID <> "000-00-0000"
    // Beginning of search
    FOR Student = 1 to NumberOfStudents
        IF TargetID = StudentID(Student)
            DISPLAY Grade(Student)
        ENDIF
    ENDFOR
    // That's the end of one search. Now we'll get an ID for the next search
    INPUT TargetID
END WHILE
```

Add this logic to the end of your Visual Logic flowchart. Prompt the user to enter a student's ID number, but tell them how to quit too. A typical prompt might look like:

```
Enter ID number, or enter 000-00-0000 to quit.
```

In your DISPLAY, show the student's ID along with his/her grade. Use Dialog Box output. (So the only console output at this point is the complete list of all IDs and grades.)

Execute this flowchart. Enter 5 students (each with a different ID number). Put in any scores you like.

Then when prompted, enter the ID number of a student and see if the right grade gets displayed. It should.

The "To Turn In" section will use this flowchart as the starting point -- keep it open.

(Note: You might not see how useful such a program could be, since there are only 5 students, and you can see their grades on the console. But this same program would work for hundreds of students -- then it would be really handy to be able to get any individual student's grade by just entering their ID number.)
TO TURN IN:

A) From Section I: Print the fixed flowchart, and show the output for 10,000 loops.

B) Starting from the flowchart you created in part II, Parallel Arrays:

Execute the flowchart, entering 3 students with grades.

Then try to find the grade for the first student entered.
Then try to find the grade for a student ID that is not in the list.

Two problems happen. You should see one of them -- when the user tries to get the grade for a student who is not in the list, they don't get anything. No message, no help, nothing.

The second problem is not obvious, but is very important. Imagine there were 1000 students (instead of just three). Now, again, imagine the user wants to find the grade for the student who happens to be first in the list.

Answer these questions on paper to turn in.

1) How many loops does the search do for the case where the very first one is the one we're looking for?

2) Assuming student ID numbers are unique, how many of those loops are just wasting time?

For regular credit, also turn in the flowchart from Part II (the parallel array flowchart for student grades) with output for the cases described in this section (Question B).  Do not turn this in if you are going for the extra credit!

Extra Credit:
For a little extra credit, your job is to fix both of those problems. Here's an algorithm for the search that will take care of both problems. You need to carefully examine it and then implement it in your Visual Logic flowchart in place of the search section currently there. Note that the outer loop, the sentinel loop, is not changed. Only the inner portion (the actual search) is going to be replaced. (If you aren't sure exactly where to put this part, go back to the pseudocode for the original search. The "search" part is surrounded by comments.)
Index ← 0
Count ← 1
WHILE Count <= NumberOfStudents AND Index = 0
   IF StudentID(Count) = TargetID THEN
      Index ← Count
   ENDIF
   Count ← Count + 1
ENDWHILE
IF Index = 0 THEN
   DISPLAY "Not found"
ELSE
   DISPLAY Grade(Index)
ENDIF
// now your algorithm continues with the part that inputs the next ID from the user

You should look this search algorithm over carefully. Notice that it sets Index to 0, a value that does not correspond to the index of any of the ID numbers or grades in the arrays. Then it does a loop which checks each ID in the array to see if it is the one the user entered. When it does find one that matches, it sets the value of Index to the current value of the loop counter, i.e., Count, since that is the index of the matching ID.

This causes the loop to end, since one of the two conditions of the AND becomes false -- Index is no longer zero. So, assuming we do find a match, the loop does not have to continue uselessly through all of the rest of the IDs.

Consider, though, what happens if there is not a match, that is, the user entered a target ID that is not in the list. The value of Index never changes, but the loop is also designed to quit when we finish going through all of the IDs in the array.

So when the loop finishes, either there was a match, and Index has the correct value, or there was no match, and Index is still zero. All we have to do is check the value of Index to see if we should print the corresponding grade or a "not found" message.

Modify your existing flowchart to incorporate this new search in place of the FOR loop search you had before. Change all inputs and outputs to Console so they can be printed to turn in.

Run your flowchart and make sure it works. You should put five IDs and five sets of grades in. Then try to find the grade for the first ID, the last ID, one of the ones in the middle, and one which is not in the list.

Print the results to turn in along with the complete (rather large) flowchart. The flowchart probably won't look very good on the page in Word, but it will have to do.
Activity M: Graphics

This week’s in-lab activity is really just for fun (although there will be something to turn in for the week’s grade). But you will not be held responsible for any of this material on any exam or outside-of-lab assignment.

However, the final in-lab activity, which is about the very important concept of subroutines, makes extensive use of the Graphics approach you'll learn this week.

Read Chapter 5, sections 5-1 and 5-2, of the Visual Logic textbook.

1) Create the flowchart in Figure 5.6 and execute it.

2) Create the flowchart in Figure 5.7 and execute it.

3) Create the flowchart in Figure 5.8 and execute it.

4) Create the flowchart in Figure 5.10 and execute it.

TO TURN IN:

Make the nicest drawing you can. Use color and some kind of loops. Be creative! And have fun!

After creating the drawing on the screen, and while that window is highlighted, press ALT-PRINTSCREEN. (That is, hold down the ALT key and press the PRINTSCREEN key.) This will copy the screen to the clipboard. Then open Word and paste. The window will be reproduced in your Word document. (Of course, if you then print it on our lab printers, you'll get a rather poor black-and-white reproduction, but such is life!)

Turn in the drawing and flowchart printouts.
**Activity N: Subroutines (also called Procedures)**

An important note!!
The Visual Logic flowchart program does not support some of the subroutine concepts you are learning in the lecture textbook. Specifically, we will not be using any **global variables**, and because there is no concept of **declaring variables**, there are no **local variables**, either.

However, the subroutines work just fine, as do the parameters. And Visual Logic supports the exact parameter approach shown in the lecture textbook, i.e., passing parameters by value and by reference.

Assuming you are going on to CSc 15 next semester, you should try to understand subroutines. They will be the "starting point" in CSc 15 -- in other words, we'll start right off talking about how subroutines are done in Java, the language taught in CSc 15. Learn what you can now!

NOTE : This activity presumes that you have completed the previous activity, the "graphics" stuff. However, if for some reason that activity was skipped, you can still follow the book here. The only thing you have to know to create the flowcharts shown in the book is that the graphics commands are all found in Visual Logic by clicking the "Graphics" menu item in the pop-up list you get when adding any items to your flowchart.

I) Your first procedure.

In the Visual Logic textbook, read section 5-3. (This book uses the term *Procedure* -- in the lecture book, we usually use the term *Subroutine*. They are the same thing, and you'll find out later that other languages and programming "cultures" have yet other names for the same thing. Java programmers, for example, call these things "Methods".)

Create the two-part flowchart shown in Figures 5.17 and 5.19. Follow the instructions for creating these flowcharts provided in the text. They are not two separate flowcharts!

(I'm not sure the book makes it completely clear how to get back to the *Main* flowchart once you've finished creating the procedure in Figure 5.17. Just go to the menu on the top of the window and select Procedures, then select *Main*. That's where you'll put the Figure 5.19 section.)
II) Parameters (arguments)

Read section 5-4. This may be the most complicated section in the entire textbook. But the more you understand about subroutines and parameters now, the better you're going to do in CSc 15. Work at it! (It's also the last section in the book that we're going to do, so there is a light at the end of the tunnel. We'll ignore section 5-5.)

Create the two-part flowchart shown in Figures 5.23 and 5.24. Carefully follow the book's instructions. The parts about the "argument boxes" require some care. (Incidentally, the term "argument" and the term "parameter" are often used interchangeably.)

1) You should create the complete flowchart set shown in figures 5.23 and 5.24. Pay particular attention to how you use the Arguments dialog box. The concept here is important. When writing programs with subroutines which have arguments, there are two steps.
   i) When writing the subroutine (procedure), you provide the formal arguments (formal parameters). This is what you are doing in Figure 5.22.
   ii) When writing the calling routine (the main routine flowchart in these examples), you must provide the actual arguments (actual parameters). This is what you are doing in the Arguments dialog box in figure 5.24.

When the flowchart executes, the value of the actual argument will be copied and used as the value for the formal argument in the subroutine. (As you'll see in a few minutes, that sentence applies for one of the two kinds of parameters, specifically, the by value parameter. We'll play with by reference parameters a bit later.)

2) Execute the flowchart several times, trying different values for SidesPerFigure and NumFigures. (If you get an error message after entering the two inputs, you probably skipped over the very important part in the text about the Arguments dialog box.)

III) Recalculating values in loops

Look carefully at the main flowchart in part II above. Assuming that the value input for NumFigures is 100, how many times does the calculation $360 / \text{NumFigures}$ get done in the main routine? Will the value be the same for every one of those calculations?

Hopefully, you see that this is an inefficient way to write a program. There is no reason to recalculate that value every time we go through that loop. Of course, the value 360 does not change. And the value of NumFigures does not change (I'm talking about during one execution of the flowchart – I know that if we finish running it and then run it again, we could put in a different value for NumFigures, but that has nothing to do with what I mean here.) So every single time we go through that loop, we do a calculation to get the same value.
To turn in: Fix the problem. Put in an Assignment operation **before entering the loop** to calculate the value of $360 / \text{NumFigures}$ and store it into a variable. (Name the variable $\text{HowFar}$.) Then replace the calculation inside the loop with that variable.

Next, go into the subroutine. The same problem occurs here. (Be sure you notice that this is a different calculation – it uses a different denominator – but the issue is the same.) Fix that, too.

Save the flowchart (or a printable representation of it) – it is part of the "To Turn In" section at the end of this activity.

**NOTE:** If you were using a full-scale programming language which provided the ability to declare variables, you could probably use the same name ($\text{HowFar}$) in both the *main* routine and the *DrawFigures* subroutine. If they are declared separately in each, they are different memory locations, even though they have the same name. This is covered in your lecture textbook, and you'll have **lots** of opportunities to deal with issues like that when you take your next course in programming concepts.

However, if you use the same name ($\text{HowFar}$) in the two routines here, there is likely to be trouble. Visual Logic does not support global variables correctly, but it does "sort of" have them – they get messed up in certain situations, which is why we are not going to use them. So if you use the same name in *main* and in *DrawFigures*, you are likely to get into trouble in this exercise. Don't!

**IV) Multiple parameters, reference parameters, multi-use subroutines, modularity.**

The Visual Logic textbook correctly makes the case that the most important reason for using subroutines is to break a large problem down into smaller problems, since as problems (and programs) get larger and larger, it becomes increasingly difficult to keep the whole picture in mind as you try to come up with algorithms. By breaking the large problem into a set of smaller (and, hopefully, relatively independent) pieces (*modules*), it becomes much easier to grasp what needs to be dealt with at any one time. (This approach is sometimes called *stepwise refinement*, and you'll also hear the term *modularity* used)

There is another reason for using subroutines. It is important, too, although not as important overall as the first reason. Sometimes there is some task you need to do in several different places within a large program. By creating a subroutine to handle that special task, you can write the algorithm/flowchart once and re-use it in all the places where that task needs to be done.

For your very last in-lab exercise of the semester, we'll deal with both of those reasons, and experiment with multiple parameters and with *reference* parameters (as well as *value* parameters, which you've already seen in the previous sections of this activity).
We're going bowling again. Yippee! We're going to enter some number of scores (any number of scores – we'll use the *sentinel* approach) and calculate the average.

Here's a "big picture" algorithm for the problem we are solving. (That is, this is going to be the *main* routine.)

```
Average ← 0
Sum ← 0
Count ← 0
Call GetScore subroutine
WHILE Score <> -99
    Sum ← Sum + Score
    Count ← Count + 1
    INPUT Call GetScore subroutine
ENDWHILE
Call CalculateAverage subroutine
DISPLAY Average
```

Now that we have the "big picture", which is, of course, the *main* routine, we need to work on the algorithms for the two subroutines.

```
GetScore (reference InScore)
INPUT InScore
WHILE ((InScore < 0) OR (InScore > 300)) AND (InScore <> -99)
    INPUT InScore
ENDWHILE
```

Think carefully about the logic of the WHILE loop. It says that we want to stay in the loop (i.e., force the user to re-enter the score) as long as both of these conditions are true:
- Condition 1: InScore is invalid
- Condition 2: InScore is not the sentinel

If we did not allow for the sentinel as an acceptable input, there would be no way to eventually exit the main loop (the one in the *main* routine).

```
CalculateAverage( value Sum, value Count, reference Result)
IF Count > 0 THEN
    Result ← Sum / Count
ENDIF
```

(Notice that the *CalculateAverage* subroutine will only calculate a value for *Result* if it is safe to do so. When we get to that point, you'll see that *main* will start with a default value for the final result (Average) of 0. So if that value doesn't get changed in *CalculateAverage*, it will remain 0, which is a good answer if there were no scores. More on that later.)
Carefully look over the complete algorithm. The main routine will call the GetScore subroutine in two different places, once before the loop, once at the end of the loop body. This is an example of using a subroutine for a task that has to be done in more than one place. The main routine will also call the CalculateAverage subroutine. This use of a subroutine is just to simplify the logic – to allow the programmer to concentrate on one task at a time.

**Now for the "sharing of data" issues:**

Consider the GetScore subroutine. It is going to input a value from the user. Does that value need to be communicated back to the calling routine (main)? Sure it does. The whole reason main "hired" that routine was so it could send that value back to main. There are three ways that GetScore could get that data back to main. One would be through a global variable (discussed in the lecture book), but since Visual Logic doesn't do a very good job with global variables, and since they are generally not the best approach anyway, we won't use that approach. The second way would be as a function return value. That, too, is discussed in the lecture text. But we won't be doing that in Visual Logic, either. That leaves one other way – a parameter.

But there are two kinds of parameters, value and reference. A value parameter can be used to send data from the caller to the called routine. You'll see that shortly for the CalculateAverage subroutine. But a value parameter cannot be used to get data from the called routine back to the caller.

That leaves reference parameters, and that's exactly what we'll do in this flowchart.

Let's look at that now. We'll leave off discussion of the CalculateAverage subroutine for awhile.

1) Start a new flowchart. (Make sure you've saved the one from part III, or at least, saved both of the flowcharts in printable form, since they must be turned in.)

2) Create a subroutine (procedure) for the following algorithm. **Name the subroutine GetScore.** When you create the new subroutine, **provide one parameter.** The name of the parameter (this will be the formal parameter) should be InScore. Make it a Reference parameter. Then create the flowchart for that subroutine.

```
INPUT InScore
WHILE ((InScore < 0) OR (InScore > 300)) AND (InScore <> -99)
    INPUT InScore
ENDWHILE
```

Be sure to provide a prompt for the input that reminds the person to enter -99 to quit.
3) Create a **stub** subroutine for `CalculateAverage`. What that means is to create the new subroutine, with the right name, but the only thing I want you to put in it right now is one Output operation which displays "This is the CalculateAverage subroutine." We'll develop that subroutine later, but for now, since `main` is going to call it, it has to be there. This **stub** does not need to have any parameters at this time.

4) Now create the flowchart for `main`. (Remember that you have to go to the *Procedures* menu and select `main`.) Here is the algorithm to implement in `main`.

```
Average ← 0
Sum ← 0
Count ← 0
Call GetScore subroutine
WHILE Score <> -99
    Sum ← Sum + Score
    Count ← Count + 1
    INPUT Call GetScore subroutine
ENDWHILE
Call CalculateAverage subroutine
DISPLAY Average
```

5) Now you need to provide Visual Logic with the name of the **actual parameter** which you will be "passing" to the `GetScore` subroutine. In **each of the two places** where you call the `GetScore` subroutine, double-click on the subroutine icon. The **Arguments** dialog box pops up. It should tell you that the name of the **formal argument** (**formal parameter**) is InScore, and that it is being passed by **reference** (**ByRef**). You need to fill in the name of the **actual argument**, the name used in `main` for that value. The name used in the pseudocode I provided is `Score`.

6) You can now execute the flowchart, **but don't!** We want to put one more operation in the `main` flowchart. What we want is an Output operation which will show us whether or not the system is working – that is, whether or not the calls to `GetScore` are actually getting scores that `main` can then use to calculate the `Sum`. (I know that we could use `CalculateAverage` to see if the average is right, but we don't want to write `CalculateAverage` yet. We don't want too many different things going on which could make it difficult to locate an error if, in fact, we had an error in the part we've written so far.) So what we're going to do is add a **diagnostic output operation** – an output of a value we only want to see to diagnose whether or not things are working right so far.

After the end of the loop, **before `CalculateAverage`**, add an Output operation to print the value of `Sum`. (We'll then be able to check to see if `Sum` is correct, and if it is, we can feel fairly confident that things are going okay up to that point.)

7) Now execute the flowchart. It should ask you for a score. Put in a valid one (200). It should ask for another.
8) For the second score, try an invalid one (-1). This should be rejected and you should be asked for a replacement. Put in another invalid score (301). This should also be rejected. Put in a valid score (150).

9) Now the program should be asking you for the third score. For this, enter one invalid score (-2). When this is rejected, enter the "valid" special sentinel value, -99. This should be accepted, which ends the loop in main, and main should then print the value of Sum before it calls the CalculateAverage subroutine (which, at this point, just reports that it has been called). Is the value of Sum correct? It should be 350, the sum of the two valid scores we entered (and, of course, without the value of the sentinel added in). (And, of course, it just prints 0 for Average, since we haven't yet dealt with calculating the average.)

NOTE: Because you used a reference parameter for the formal parameter, InScore, in GetScore, the value entered by the user got communicated back to main on each call to GetScore. Specifically, main's Score variable got the value. (Technically, what really happened was that main's Score variable and GetScore's InScore parameter shared the same location in memory, hence, any change to InScore was also a change to Score.)

10) If all went well and you got the right value for Sum, it's time to work on the last piece. The logic for CalculateAverage is pretty simple, but the parameter issues are a bit more involved.

Go ahead and change the stub CalculateAverage subroutine so that it carries out the logic of the pseudocode.

```
IF Count > 0 THEN
    Result ← Sum / Count
ENDIF
```

11) Now it's time to deal with the parameters for CalculateAverage. Look at the logic of CalculateAverage. It needs two pieces of information from the calling routine (main). It needs the values of Count and Sum. Both of those have been calculated within the loop in main. Furthermore, it needs to send back one piece of information, the calculated value (which is called Result in CalculateAverage).

Notice that the names Sum and Count are the same names used in main, but the name Result is not the same as the name used for this value in main. (In main, we're using the name Average.)

When using parameters, names do not matter! We can use the same names in the called and calling routines, or we can use different names. It makes no difference at all.
Since the values for Count and Sum are only going from the caller (main) to the called subroutine (CalculateAverage), we use value parameters. That's what value parameters are for – to communicate down to the called routine. Let's set those up.

12) Within the CalculateAverage flowchart, right-click on the subroutine heading (the box labeled CalculateAverage), and select Edit…

13) Select New in the Arguments box, and add Sum and Count, both with the Value radio button selected. (You can't add them both at once – it takes two operations.)

14) Now we need to handle Result. Like the InScore formal parameter in the GetScore method, we need to be able to get the value of Result back to main. So add it as a third parameter, but be sure to select the Reference parameter type for it.

15) Return to main. You must now tell the flowchart what actual parameters are going to be passed by main to CalculateAverage. Recall that you do this by double-clicking on the subroutine box. Fill in the actual parameters, Sum for Sum, Count for Count, and Average for Result.

You've now told the program to pass the values of main's Sum and Count down to the formal parameters (which happen to have the same name) in CalculateAverage, and to pass main's Average as the reference parameter (named Result in CalculateAverage).

16) Remove the diagnostic output (the Output operation which displays Sum). We only had that there for diagnosis of possible problems – we've now verified that Sum is calculated correctly, so we can get rid of that statement).

17) Okay, moment of truth! Execute the flowchart.
For your inputs for scores, enter the following:
   Score 1: 200
   Score 2: invalid -1, invalid 301, then valid 150.
   Score 3: 190
   Score 4: Sentinel, -99.

If all went well, the final output should be the average of (200 + 150 + 190) / 3, or 180. (If it didn't work, fix it!)

18) One more test. Execute the flowchart one last time. This time, enter -99 for the very first score. If the program is correct, it should exit the loop, call the CalculateAverage subroutine, which should not change the value of the actual parameter Average, so the program should print 0.
TO TURN IN:

A) The modified flowchart from part III above. You do not need to print output, just the flowchart. **But you need to print two flowcharts – main and DrawFigure.**

B) The three flowcharts from part IV (Bowling).
Modify the flowchart for console input and output, and execute twice. Provide the printed output for both executions.
For the first run, use these values:
Score 1: 100
Score 2: 50
Score 3: -1, followed by 60
Score 4: -2, followed by 301, followed by 90
Score 5: -99

For the second run, use these values:
Score 1: -1, followed by 301, followed by 302, followed by -99.