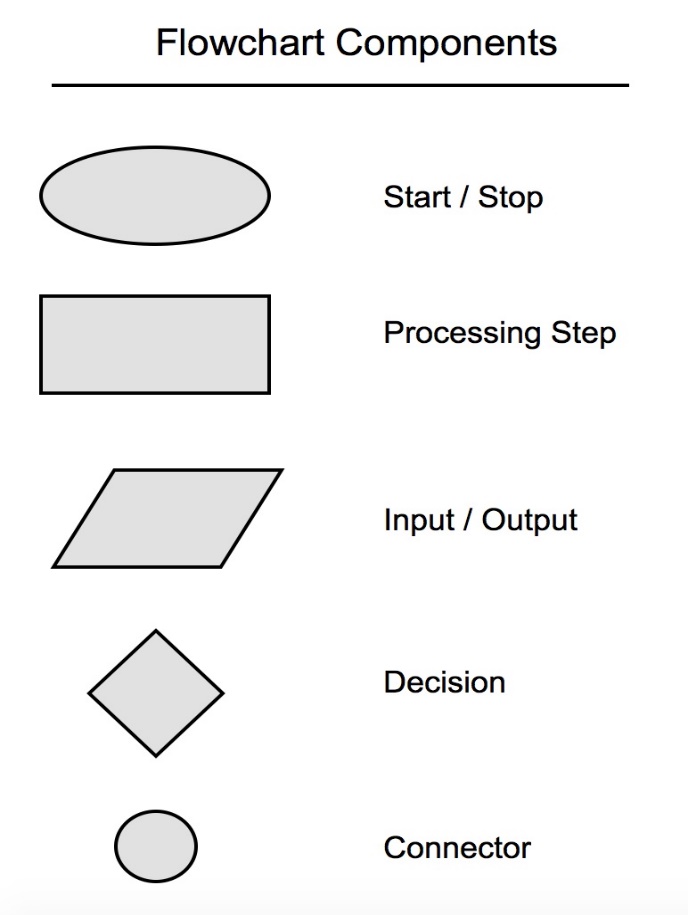
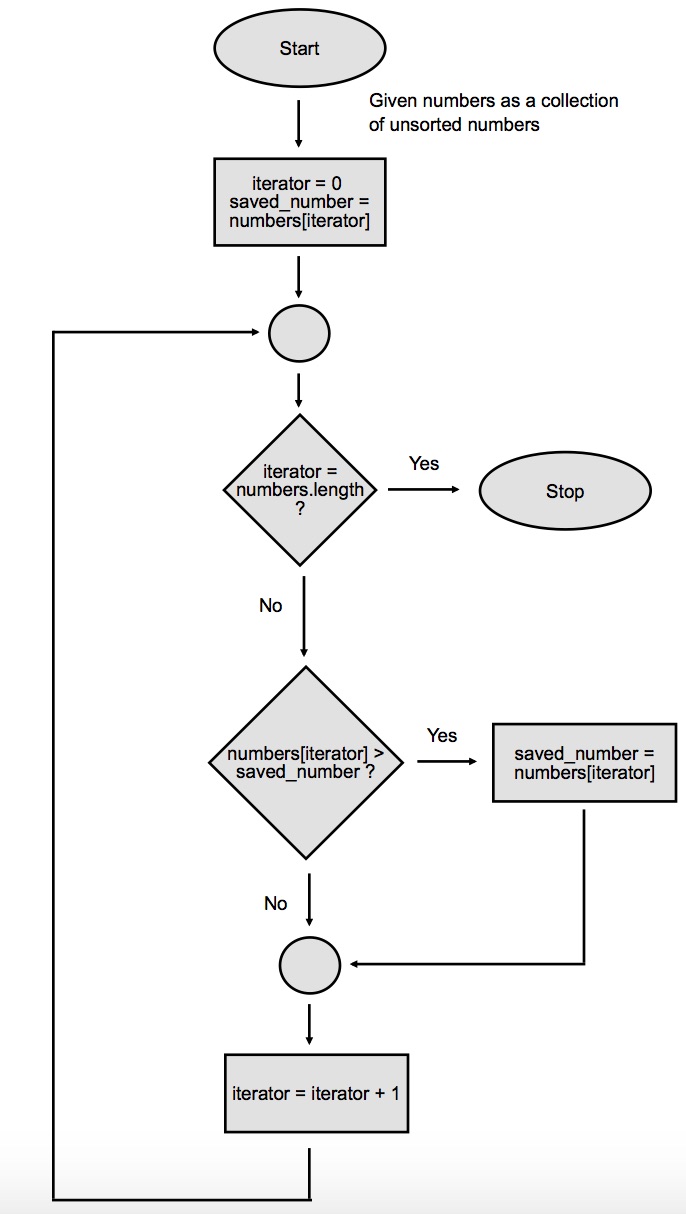
Flowchart

Using a flowchart helps us map out the logical sequence of a possible solution in a visual way. We'll take a stab at drafting a flowchart for the same problem we saw in the previous assignment. But first, let's take a look at the various components at our disposal in a flowchart.



Using the components above, here's what a flowchart would look like for our approach to finding the largest number in a collection. Note that the decision (the diamond) component should only have two branches. If you have a decision condition that has 3 (or more) branches, use multiple diamonds.



Note that the arrows show the "flow" of the logic, and that we're taking great pains to specifically draw out the iteration logic. This helps us map out the step-by-step logic our program would need to solve this problem. This is called the *imperative* or *procedural* way to solve a problem. In many higher level programming languages, basic concepts (like for example, iterating over a collection) are encapsulated into a method. In Ruby, for example, we have a handy each method that allows us to iterate over an Array collection. Using each is the *declarative* way of solving a problem.

When working with flowcharts, we are going to be imperative in our approach, and we'll visually show how to loop manually, rather than using any declarative constructs built in to the language.

Doing this will force you to understand the logic much better, and also force you to "think like a computer", which will help debug logical errors in your code. Over time, as you get better at "thinking like a computer", you can reach for the higher level declarative syntax.

A Larger Problem

Now, let's suppose that the above flowchart is mapping out a solution to part of a larger problem. Let's suppose that in our larger problem, we're asking the user to give us N collections of numbers. We want to take the largest number out of each collection, and display it.

How would we approach coming up with a solution to that type of problem?

Let's try to high level pseudo-code this.

while user wants to keep going

- ask the user for a collection of numbers

- extract the largest one from that collection and save it

- ask the user if they want to input another collection

return saved list of numbers

That seems reasonable, but you can see the line extract the largest one from that collection is a sub-process that itself contains a lot of logic. The functionality there is pretty self-contained, so it's a great candidate for a sub-process. We can, of course, turn that sub-process into a method. But let's not even think about "methods" yet. Let's keep our train of thought at the logical level and not think about code.

Now, you may be thinking that we were only able to see the "extract the largest one from that collection" as a sub-process because we did the previous assignment first. If we were tackling this larger problem first, it may not be obvious. That's a fair point. In that scenario, we'd just include the full pseudo-code from the previous assignment, like this:

while user wants to keep going

- ask the user for a collection of numbers

- iterate through the collection one by one.

- save the first value as the starting value.

- for each iteration, compare the saved value with the current value.

- if the saved value is greater, or it's the same

- move to the next value in the collection

- otherwise, if the current value is greater

- reassign the saved value as the current value

- after iterating through the collection, save the largest value into the list.

- ask the user if they want to input another collection

return saved list of numbers

When pseudo-code gets long, it becomes very hard to trust the accuracy of the logic (remember, you can only verify the logic by running actual program code). Therefore, it's prudent to extract a logical grouping into a sub-process, and to tackle the various pieces separately.

Let's take our shortened pseudo-code from the beginning and translate this into formal pseudo-code.

START

SET large\_numbers = []

SET keep\_going = true

WHILE keep\_going == true

GET "enter a collection"

SET collection

SET largest\_number = SUBPROCESS "extract the largest one from that collection"

large\_numbers.push(largest\_number)

GET "enter another collection?"

IF "yes"

keep\_going = true

ELSE

keep\_going = false

IF keep\_going == false

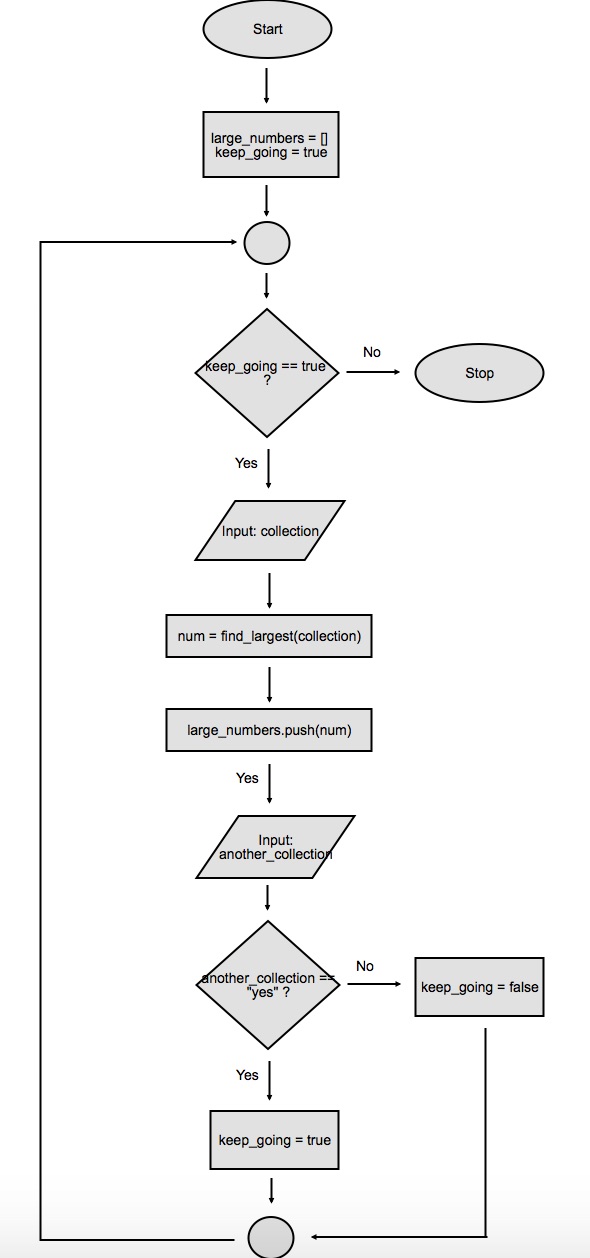
exit the loop

PRINT large\_numbers

END

Notice that we have a SUBPROCESS keyword to show that there's some other *thing* that will take care of extracting the largest number out of a collection. Just like before, we could have included the entire formal pseudo-code from the previous assignment, but that would have made it very long, and our confidence in such long pseudo-code cannot be very good. Extracting to sub-processes helps us focus on a well defined narrowly scoped set of pseudo-code.

Next, we'll try to use a flowchart to help us organize the logical flow a bit more.



The interesting part about this flowchart is the processing square in the middle num = find\_largest(collection). This is our sub-process. You can think of this square as the zoomed-out high level view of the very first flowchart from the top of this assignment.

Interestingly, when we move logic to sub-processes, we are able to use a *declarative* type of syntax, rather than *imperative*. In other words, we can say "find\_largest", rather than outline step by step how the logic should be. Thinking about the high level logic flows allows you to create sub-processes to narrow down the scope of your application. From a high level, writing declarative code segments our program into logical sections, allowing us to focus on one section at a time. For example, if we wanted to add a validation feature to our program, we could have a sub-process that returns true or false given an input. We can just call it validate\_input and represent it as a square in our flowchart, without having to layout the step-by-step imperative logic that's required to validate the user's input. From a high level, we can trust that this sub-process will do its job -- it will only return true or false. When we're ready to work on the logic in that validate\_input sub-process, we can forget about the larger program, and just focus on the responsibilities of this sub-process.

As you use pseudo-code and flowcharts to help you dissect the logic of a problem, you'll be constantly trying to figure out how detailed the chart and words should be, and what can be extracted to sub-processes. This is exactly what a programmer should be thinking about when designing the solution to a problem. You won't get it right the first time.

Start at a high level, using declarative syntax. For example, if you're working on a calculator, you can start with something like this:

- Get the first number

- Make sure it's valid, otherwise, ask for another

- Get the second number

- Make sure it's valid, otherwise, ask for another

- Get the operator

- Make sure it's valid, otherwise, ask again

- Perform operation on the two numbers

- Display result

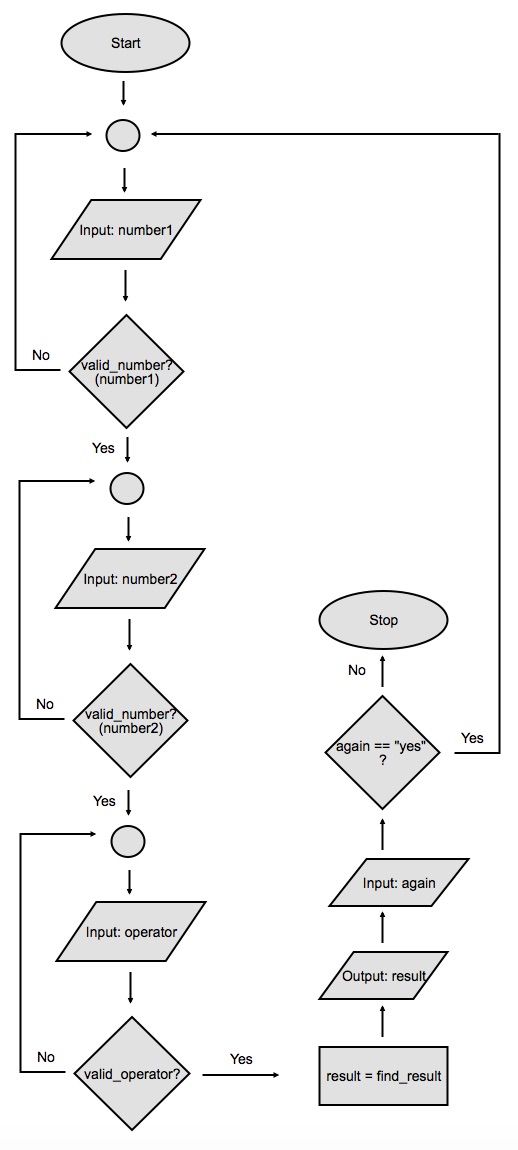
- Ask if user wants to do another calculation

In the above, you're not yet outlining exactly how to validate the inputs. No specifics or imperative style pseudo-code yet. Once you have the high level steps, it's time to drill down a level into imperative pseudo-code and outline specifics.

This is going to take some practice, and you do not have to use either pseudo-code or flowcharts at all. If you've been programming for a while, it's likely you don't need these aids. If you're new to programming and have trouble "thinking like a computer", try using these tools before you jump straight into program code.

Flowcharting the Calculator

Taking the high level pseudo-code above, we can come up with a flowchart that looks something like this.



We're only able to come up with this high level flowchart by introducing three sub-processes: valid\_number?, valid\_operator?, and find\_result. By not worrying about the low-level details of *how* those sub-processes will be implemented, we can think at a higher level about our overall application logic. When we're ready to dive into how each of those sub-processes should work, we can go and create detailed pseudo-code and flowcharts for each of those sub-processes.

Hopefully this gives you an idea of one way to approach tackling the logical side of problem solving.