Algorithms

As we discussed earlier, an algorithm is a clear set of steps to solve any problem in a particular class. Typically, algorithms have at least one parameter; however, algorithms with no parameters exist—they are simply restricted to one specific problem, rather than a more general class. We can discuss and think about algorithms in the absence of any particular knowledge of computers—a *good* algorithm can not only be translated into code, but could also be executed by a person with no particular knowledge of the problem at hand.

Algorithms that computers work on deal with numbers—in fact the module **Types** will discuss the concept of "Everything is a number," which is a key principle in programming. Computers can only compute on numbers; however, this course will also illustrate how we can represent a variety of useful things (letters, words, images, videos, sound, etc.) as numbers so that computers can compute on them. As a simple example of an algorithm that works with numbers, we might consider the following algorithm (which takes one parameter *N*, a non-negative integer):

```
Given a non-negative integer N:
  Make a variable called x, set it equal to (N+2)
  Count from 0 to N (include both ends), and for each number (call it "i")
  that you count:
    Write down the value of (x * i)
    Update x to be equal to (x + i * N)
  When you finish counting, write down the value of x.
```

For any non-negative integer *N* that I give you, you should be able to execute these steps. If you do these steps for *N* = 2, you should come up with the sequence of numbers **0 4 12 10**. These steps are unambiguous as to what should happen. It is possible that you get the wrong answer if you misunderstand the directions, or make arithmetic mistakes, but otherwise, everyone who does them for a particular value of *N* should get the same answer. We will also note that this algorithm can be converted into any programming language quite easily—all that is needed is to know the basic syntax of the particular language you want.

You may wonder why we would want an algorithm that generates this particular sequence of numbers. In this case, it is just a contrived algorithm to show as a simple introductory example. In reality, we are going to devise algorithms that solve some particular problem. However, devising the algorithm for a problem takes some significant work, and will be the focus of discussion for the rest of this module.

Even though computers can only work with numbers, we can envision algorithms that might be executed by humans who can work on a variety of things. For example, we might write algorithms that operate on physical objects such as LEGO bricks or food. Even

though such things would be difficult to implement on a computer (we would need the computer to control a robot to actually interact with the physical world), they are still instructive, as the fundamental algorithmic design principles are the same.

One exercise done at the start of some introductory programming courses is to have the students write down directions to make a peanut butter and jelly sandwich. The instructor then executes the algorithms, which are often imprecise and ambiguous. The instructor takes the most comical interpretation of the instructions to underscore that what the students wrote did not actually describe what they meant.

This exercise underscores an important point—you must specify exactly what you want the computer to do. The computer does not "know what you mean" when you write something vague, nor can it figure out an "etc." Instead, you must be able to describe exactly what you want to do in a step-by-step fashion. Precisely describing the exact steps to perform a specific task is somewhat tricky, as we are used to people implicitly understanding details we omit. The computer will not do that for you (in any programming language).

Even though the "sandwich algorithm" exercise makes an important point about precisely describing the steps you want the computer to perform, it falls short in truly illustrating the hardest part of designing an algorithm. This algorithm has no parameters, so it just describes how to solve one particular problem (making a peanut butter and jelly sandwich). Real programming problems (typically) involve algorithms that take parameters. A more appropriate problem might be "Write an algorithm that takes a list of things you want in a sandwich and describes how to make the sandwich."

Such a problem is much more complex but illustrates many concepts involved in devising a real algorithm. First, our algorithm cannot take a list of just anything to include in the sandwich—it really will only work with certain types of things, namely food. We would not expect our algorithm to be able to make us a "car, skyscraper, airplane" sandwich. These items are all the wrong *type*. We will learn more about types in programming later in this course.

Our algorithm may also have to deal with error cases. Even if we specify the correct *type* of inputs, the particular values may be impossible to operate on correctly. For example, "chicken breast" is food, but if the chicken breast has not been cooked yet, we should not try to make a sandwich out of it. Another error case in our sandwich creation algorithm might be if we specify too much food to go inside the sandwich (how do you make a sandwich with an entire turkey, 40 pounds of carrots, and 3 gallons of ice cream?). Of course, if we were writing this sandwich algorithm for humans, we could ignore this craziness because humans have "common sense"—however, computers do not.

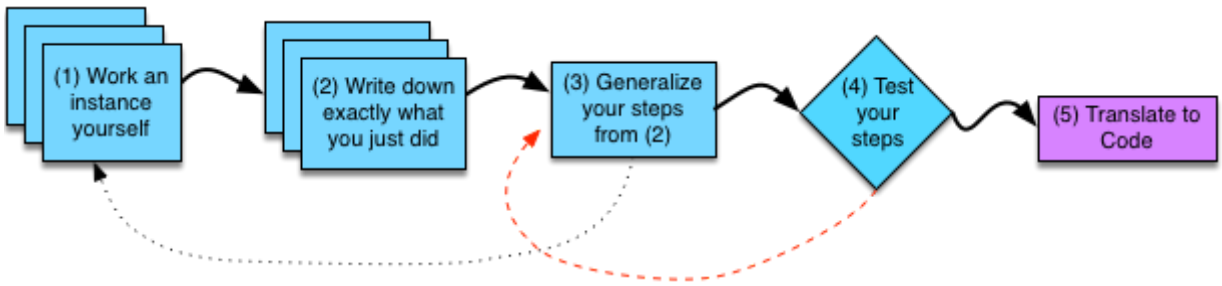
Even if we ignore all of the error cases, our general algorithm is not as simple as just stacking up the ingredients on top of bread in the order they appear in the input. For example, we might have an input of "chicken, mustard, spinach, tomatoes." Here, we

probably want to spread the mustard on the bread first, then place the other ingredients on it (hopefully in an order that makes the most stable sandwich).

It would seem that writing a correct algorithm to make a sandwich from an arbitrary list of ingredients is quite a complex task. Even if we did not want to implement that algorithm in code, but rather have it be properly executed by a person with no common sense (or a professor with a comedic disregard for common sense), this task is quite challenging to do correctly. How could we go about this task and hope to get a good algorithm?

The *wrong* way to write an algorithm is to just throw some stuff on the page, and then try to straighten it out later. Imagine if we approached our sandwich example by writing down some steps and having someone (with no common sense) try them out. After the kitchen catches on fire, we try to go in and figure out what went wrong. We then tweak the steps, and try again. This time, the kitchen explodes instead. We repeat this process until we finally get something that resembles a sandwich, and the house did not burn down.

The previous paragraph may sound silly, but is exactly how many novice (and intermediate) programmers approach programming tasks. They jump right into writing code (No time to plan! Busy schedule!), and it inevitably does not work. They then pour countless hours into trying to fix the code, even though they do not have a clear plan for what it is supposed to do. As they "fix" the code, it becomes a larger, more tangled mess. Eventually, the program sort-of-kind-of works, and they call it good enough.



Instead, you should devise an algorithm in a disciplined fashion. The above figure shows how you should approach designing your algorithm. We will spend the next few sections discussing each of these steps in detail. However, note that "translate to code" comes only after you have an algorithm that you have tested by hand—giving you some confidence that your plan is solid before you build on it.

If you plan well enough and translate it correctly, your code will just work the first time. If it does not work the first time, you at least have a solid plan of what the code *should* be doing to guide your debugging.