Step 3: Generalize Your Steps

Having solved one or more problems from the class we are interested in and written down the particular steps we executed to solve them, we are ready to try to generalize those steps into an algorithm. In our Step 2 steps, we solve particular instances, but now we need to find the pattern that allows us to solve the whole class. This generalization typically requires two activities. First, we must take particular values that we used and replace them with mathematical expressions of the parameters. Looking at our Step 2 steps for computing 3^4, we would see that we are always multiplying 3 by something in each step. In the more general case, we will not always use 3—we are using 3 specifically because it is the value that we picked for *x*. We can generalize this slightly by replacing this occurrence of 3 with *x*:

```
Multiply x by 3
  You get 9
Multiply x by 9
  You get 27
Multiply x by 27
  You get 81
81 is your answer.
```

The second common way to generalize steps is to find repetition—the same step repeated over and over. Often the number of times that the pattern repeats will depend on the parameters. We must generalize *how many times* to do the steps, as well as what the steps are. Sometimes, we may find steps which are *almost* repetitive, in which case we may need to adjust our steps to make them exactly repetitive. In our 3^4 example, our multiplication steps are almost repetitive—both multiply *x* by "something," but that "something" changes (3 then 9 then 27). Examining the steps in more detail, we will see that the "something" we multiply is the answer from the previous step. We can then give it a name (and an initial value) to make all of these steps the same:

```
Start with n = 3
n = Multiply x by n
n = Multiply x by n
n = Multiply x by n
n is your answer.
```

Now, we have the same *exact* step repeated three times. We can now contemplate how many times this step repeats as a function of *x* and/or *y*. We must be careful not to jump to the conclusion that it repeats *x* times because *x* = 3—that is just a coincidence in this case. In this case, it repeats *y* - 1 times. The reason for this is that we need to multiply 4*3s together, and we already have one in *n* at the start, so we need *y* - 1 more. This would lead to the following generalized steps:

```
Start with n = 3
Count up from 1 to y-1 (inclusive), for each number you count,
  n = Multiply x by n
n is your answer.
```

We need to make one more generalization of a specific value to a function of the parameters. We start with $n = 3$; however, we would not always want to start with 3. In the general case, we would want to start with $n = x$:

```
Start with n = x
Count up from 1 to y-1 (inclusive), for each number you count,
  n = Multiply x by n
n is your answer.
```

Sometimes you may find it difficult to see the pattern, making it hard to generalize the steps. When this happens, returning to Steps 1 and 2 may help. Doing more instances of the problem will provide more information for you to consider, possibly giving you insight into the patterns of your algorithm. This process is often referred to as writing 'pseudo-code', as you are working to design an algorithm programmatically with no particular target language. Nearly all programmers make use of this method to ensure their algorithm is correct before writing any actual code.