Step 3: Repetitions

Generalizing Repetitions

Another important part of generalizing an algorithm is to look for repetitions of the same (or similar) steps. When similar steps repeat, you will want to generalize your algorithm in terms of how many times the steps repeat (or until what condition is met). To examine this aspect of generalizing, we will deviate from our rectangle example (which does not have this type of repetition), and consider a slightly different problem for a moment:

Given an integer N (>0), print a right triangle of *s, with height N and base N.

For example, if N = 4, you would print

```
*

**

***

****
```

We might work an example with N=5, and end up with the following result from Step 2:

Print 1 star

Print a newline

Print 2 stars

Print a newline

Print 3 stars

Print a newline

Print 4 stars

Print a newline

Print 5 stars

Print a newline

Here, we are doing almost the same thing (Print i stars; Print a newline) 5 times. Once we observe the repetition, we can take one step towards generalizing the algorithm by re-writing the algorithm like this:

```
Count (call it i) from 1 to 5 (inclusive)
Print i stars
Print a newline
```

Notice that the way we have re-written the algorithm here gives us two new constants to scrutinize: the 1 and the 5 in the range that we count from/to. Careful consideration of these would show that 1 is truly a constant (we *always* start counting at 1 for this algorithm), but 5 should be generalized to N:

```
Count (call it i) from 1 to N (inclusive)
Print i stars
Print a newline
```

This algorithm is correct for the triangle-of-stars problem. Sometimes it takes a little more work to make the steps of your algorithm match up so that you can describe them in terms of repetition. For example, consider the following problem:

Given a list of numbers, find their sum.

We might work this problem on the list of numbers 3, 5, 42, 11, and end up with the following result from Step 2:

```
Add 3 + 5 (= 8)
Add 8 + 42 (= 50)
Add 50 + 11 (= 61)
Your answer is 61
```

Scrutinizing each of these constants might lead us to the following more general steps:

```
Add (the 1st number) + (the 2nd number)
Add (the previous total) + (the 3rd number)
Add (the previous total) + (the 4th number)
Your answer is (the previous total)
```

Here, we *almost*, but not quite, have a nice repetitive pattern. We can, however, make the steps match up:

```
previous_total = 0
previous_total = Add previous_total + (the 1st number)
previous_total = Add previous_total + (the 2nd number)
previous_total = Add previous_total + (the 3rd number)
previous_total = Add previous_total + (the 4th number)
Your answer is previous_total
```

Note that mathematically speaking, what we did was exploit the fact that 0 is the additive identity—0 + N = N for any number N. We will also note that starting with the identity element as our answer before doing math to the items in a list is typically a good idea, since the list may be empty. Often, the correct answer when performing math on an empty list is the identity element of the operation you are performing. That is, the sum of an empty list of numbers is 0, the product of an empty list of numbers is 1 (the multiplicative identity). Now that we have re-arranged our steps, we can generalize nicely:

```
previous_total = 0
Count (call it i) from 1 to how many numbers you have
previous_total = Add previous_total + (the ith number)
Your answer is previous_total
```

In this example, we also did something that will make Step 5 (translating to code) a bit easier—naming values that we want to manipulate. In particular, we gave a name to the running total we compute, which means that not only is it clear exactly what we are referencing when we say **previous_total**, but also that when we reach Step 5, this will translate directly into a variable.