Intro

Sometimes it's necessary to make multiple recursive calls, in order to solve one problem. Counting problems are a good example of this fact, and in this chapter we will learn how to solve them. First we'll talk about how to plan your base case, and then we'll move on to addressing the recursive calls.

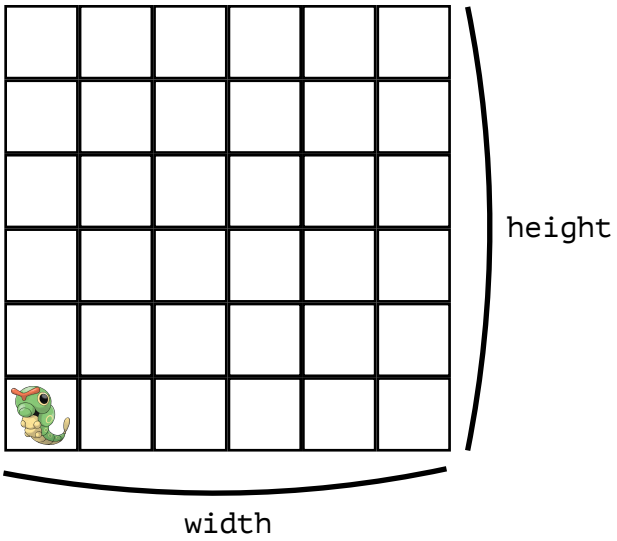
Practice: `count_paths`

Recall from the previous chapter the anatomy of a recursive problem:

- Test whether we're done.
- If we're not done, move closer to being done.
- Use the recursive call in a useful calculation.

When we only have one parameter, these are pretty well-defined. For example in `factorial`, we only had the parameter `n`, so to "be done" meant that `n == 0`. This is more complicated when we have 2 or more parameters in our recursive function. What if one parameter is big, and the other is small? What does it mean then, to "be done" with solving the problem?

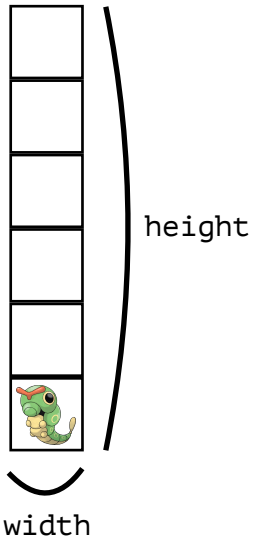
Let's look at a specific example, called `count_paths`. It takes in two parameters `height` and `width`, which correspond the size of a rectangle grid. We have an interest in a caterpie on this grid, who can only move right or up. We want to know how many different paths the caterpie can take, in order to get from the bottom left corner to the top right corner of the grid.



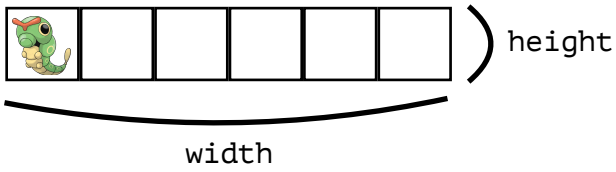

```
def count_paths(width, height):
    ...
```

In order to solve this problem, we have to recurse on *two* different parameters.

Just like before, the smallest value of our input will correspond to our base case. Now that we have two parameters for our input, we will just think of the smallest value for each of them individually.



The smallest value of `width` is 1.

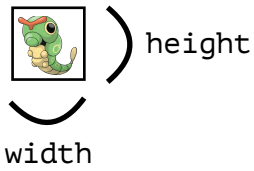


The smallest value of `height` is also 1.

In this example, the base case for each parameter is 1. This leaves us a few possibilities to account for. Either `width` is in its base case (i.e. `width == 1`), or it is not. And similarly, either `height` is in its base case (i.e. `height == 1`), or it is not. Let's plan out what to do in each of these scenarios. We'll use a chart like the one below, because it makes it easier to organize our thoughts.

	`height == 1`	`height > 1`
`width == 1`		
`width > 1`		

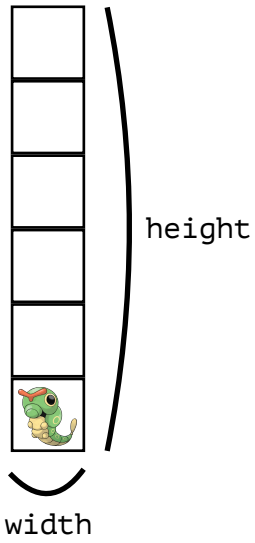
First, let's consider what the problem looks like when both parameters are in their base cases (i.e. `width == 1` and `height == 1`).



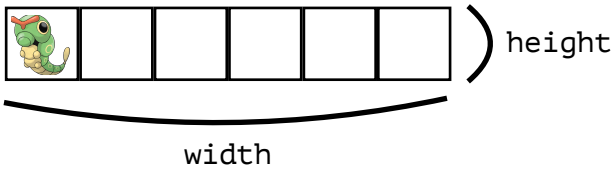
There is exactly one way for caterpie to get from the bottom left corner to the top right corner, and that is by just staying put.

That means we should return 1, if `width` and `height` are both passed in as 1. We will fill in the first value in our chart correspondingly.

	height == 1	height > 1
width == 1	return 1	
width > 1		



Now let's think about what to do when we're in the top right box — that is, `width == 1` and `height > 1`. There is exactly one way for caterpie to get from the bottom of the grid to the top of the grid.



What if we're in the bottom left box? `width > 1` and `height == 1`, so once again there is exactly one way for caterpie to get from the far left of the grid to the far right of the grid.

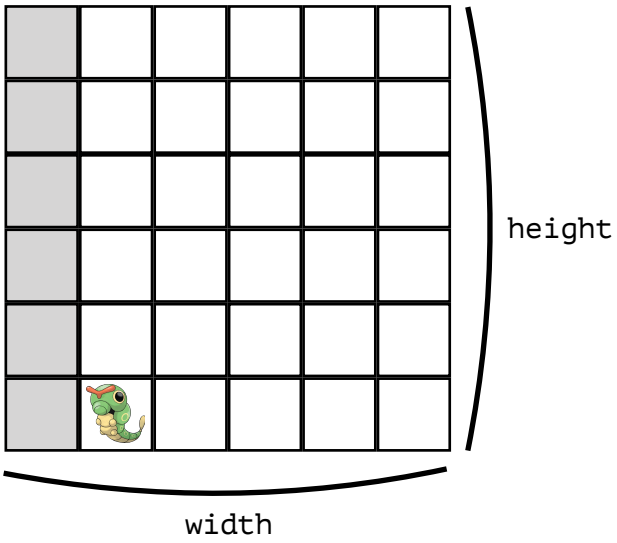
Let's put these values into our chart.

	height == 1	height > 1
width == 1	return 1	return 1
width > 1	return 1	

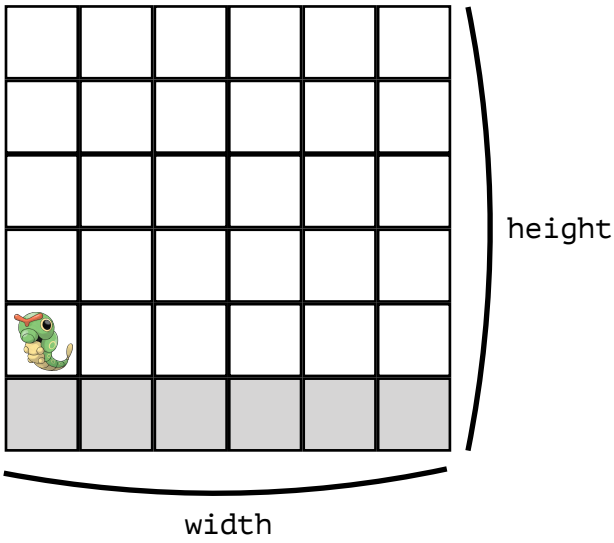
Now we are almost done with the problem. Let's recap. First we determined a base case for each parameter, individually. Then we made a chart that maps out whether each parameter is in its base case or not. And now, we also have finished filling in all the boxes where at least one of the parameters is in it's base case.

The only box left corresponds to both `width` and `height` being larger than `1`. This is also the trickiest box, because it's the one where the recursion comes in. We need to split the problem into smaller ones — but how?

Remember, the size of a recursive problem is just the size of its inputs. In order to make the problem smaller, we have to make the inputs smaller. In this example, there are exactly two ways to do that. We can either shrink `width`, or we can shrink `height`. Both of these possibilities were given to us in the problem description, where we learned that caterpie can either go right (shrinking `width`) or go up (shrinking `height`).



If caterpie moves right, then `width` decreases by one because there's no way to reach any of the squares on the left column anymore. This corresponds to a recursive call `count_paths(width - 1, height)`.



If caterpie moves up, then `height` decreases by one because there's no way to reach any of the squares on the bottom row anymore. This corresponds to a recursive call `count_paths(width, height - 1)`.

Each possibility corresponds to one recursive call. If caterpie moves right, then there are `count_paths(width-1, height)` ways for caterpie to get to the top right corner. If caterpie moves up, then there are `count_paths(width, height-1)` ways for caterpie to get the top right corner. Since we want to calculate all possibilities total, we will add together the results of these recursive calls

	`height == 1`	`height > 1`
`width == 1`	`return 1`	`return 1`

`width > 1`	`return 1`	`count_paths(width - 1, height)`
		`+`
		`count_paths(width, height - 1)`

Now the problem is done. The solution is literally just the filled-in chart. All that's left to do is translate the solution into Python. Each box could be its own `if` case, like so:

```
def count_paths(width, height):
    if width == 1 and height == 1:
        return 1
    if width == 1 and height > 1:
        return 1
    if width > 1 and height == 1:
        return 1
    if width > 1 and height > 1:
        caterpie_goes_right = count_paths(width - 1, height)
        caterpie_goes_up = count_paths(width, height - 1)
        return caterpie_goes_right + caterpie_goes_up
```

This is entirely correct, but it's also a little repetitive. We can make it look a little nicer, without changing how it works.

```
def count_paths(width, height):
    if width == 1 or height == 1:
        return 1
    else:
        caterpie_goes_right = count_paths(width - 1, height)
        caterpie_goes_up = count-paths(width, height - 1)
        return caterpie_goes_right + caterpie_goes_up
```

There we go! Make sure you understand this solution before moving on.

Approach to Counting Problems

The same approach applies to any problem where you're counting some quantity — whether that's how many paths caterpie can take across a grid, or how many ways you can add coins to make a dollar. Here's the general formula:

1. Base Case
   - Find each parameter's individual base case. This is the smallest value that could be passed in for each parameter.
   - Make a chart that maps out whether or not each parameter is in its base case.
   - Fill in all the boxes where at least one parameter is in its base case. Be careful with this one; sometimes it can be tricky. For example, think of `count_paths(1, 1)`. Many students are inclined to say there are zero ways for caterpie to get to the top right corner, when in fact there is one way: for caterpie to not move at all.
2. Recursive Call
   - Look at the problem definition. It will explain how you're allowed to break the problem into smaller ones. Typically you can make a problem smaller by making some sort of decision. Does caterpie go up, or right? Do you use a dime, or something else?
   - For each smaller problem, make a recursive call with parameters that correspond to what the decision was.
   - Combine those recursive calls somehow, depending on what you want to do. For example you might add them up, multiply them, or put them in a list.

Practice: `knapsack_count`

There's a famous problem in computer science, called the knapsack problem. We have a knapsack which has a maximum weight that it can carry. We also have a list of items, which all have their own worth and weight. In the next chapter, we'll discuss this problem more and talk about how to find the best set of items to put in the knapsack, so that we maximize worth without going over the weight limit. For now, we'll just focus on counting how many valid solutions exist.

```
def knapsack_count(weight, items):
    # weight: the maximum capacity of our knapsack
    # items:  [(worth, weight), (worth, weight), ...]
    ...
    # return: how many ways we can fill the knapsack
```

To get started, let's follow the formula above. The first step is to find each parameter's individual base case. The smallest capacity for our knapsack is `0`, so that will be the base case for `weight`. `items` is a list, and it will be smallest when it is an empty list. Let's make a chart.

	`weight == 0`	`weight > 0`
`items == []`		
`items != []`		

Now we start filling things in. If `items == []`, then we have no items that we can put in our knapsack. There is exactly one valid way to complete the problem, which is to leave our knapsack empty.

	`weight == 0`	`weight > 0`
`items == []`	`return 1`	`return 1`
`items != []`		

Now let's consider the case where `items != []` and `weight == 0`. This means we still have some items that we could put inside, but we have reached out weight limit. It is tempting to say we should return `1`, because the only valid move is to not add in any more items. However, suppose it is possible for some items to have zero weight. There could be many different ways to continue, if there are many weightless items left in `items`. This tells us that we have the wrong base case, since the base case should be simple enough that it doesn't depend on what items we have available to us. In fact, the correct base case is `weight < 0`. If this is the case, then we have exceeded the weight limit of our knapsack, and (assuming there are no negative-weight items) we have no valid ways to proceed. We return `0`, because even by leaving our knapsack empty we will exceed the weight limit.

	`weight < 0`	`weight >= 0`
`items == []`	`return 0`	`return 1`
`items != []`	`return 0`	

We have reached the box where the recursion comes in. Now is a good time to refer back to the problem definition, because that will usually tell you what you can do to make the problem smaller. In particular, remember that shrinking the problem corresponds to making a decision. Here, we're trying to decide what items to put in the knapsack. One way to make that smaller is by making the decision about just one item: do we take the first item in the list, or do we not?

If we decide not to take the first item in `items`, then that corresponds to a recursive call where our `weight` parameter stays the same and our new list of items is `items[1:]`. Alternatively, if we decide to take the first item in `items`, then that corresponds to a recursive call where our new maximum weight is `weight-items[0][1]` and our list of items is again `items[1:]`. Since we want to account for both possibilities, we will sum these recursive calls together.

	weight < 0	weight >= 0
items == []	return 0	return 1
items != []	return 0	knapsack_count(weight-items[0][1], items[1:]) + knapsack_count(weight, items[1:])

Examine how this problem works. On each recursive call, we make a decision about just one item in the list. Each decision corresponds to certain parameters. We keep going, making a decision about one item at a time, until either we exceed maximum capacity or we run out of items to bring.

```
def knapsack_count(weight, items):
    if items weight < 0:
        return 0
    if items == []:
        return 1
    with_first_item = knapsack_count(weight-items[0][1], items[1:])
    without_first_item = knapsack_count(weight, items[1:])
    return with_first_item + without_first_item
```

Closing Remark

In this chapter we used charts to map out each parameter in a recursive function. This turns out to be a pretty useful strategy to organize your thoughts, whether you're solving a counting problem or not. Try it out for the tree recursion problems in the next chapter, and see if you find it helpful.