Cracking Count Change

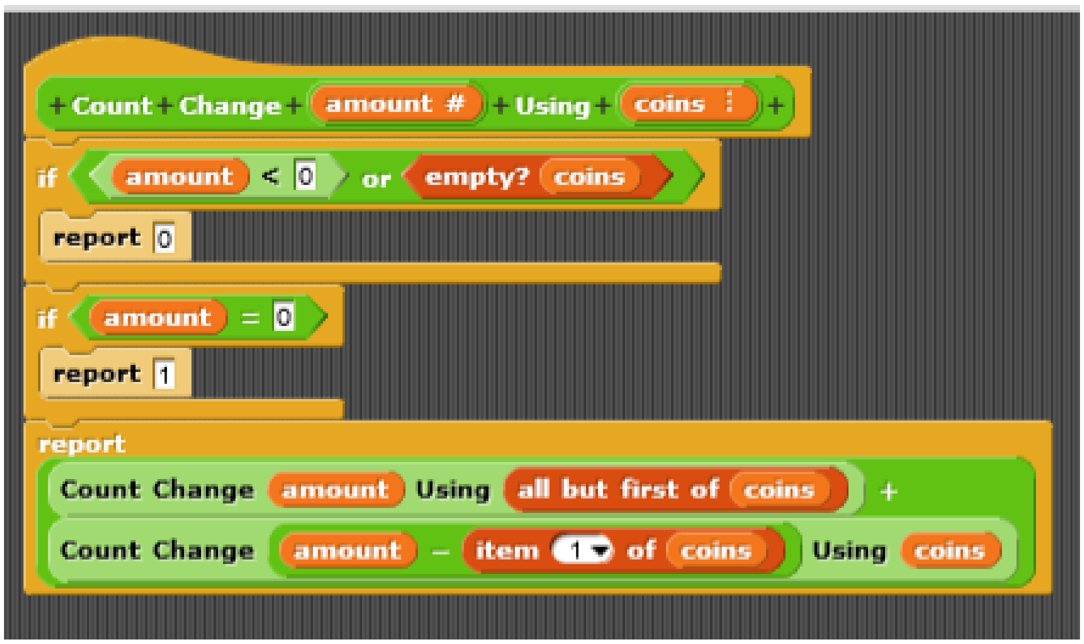
Count Change can be confusing. Really confusing. But that is only *if* you let it beat you. This guide is designed to ensure that does not happen. Count Change can be understood, I promise you. Let's see how.

Understanding the Concept

Before we get to the code itself, let's consider the general concept it embodies. You've learned about recursion, but Count Change utilizes a specific type of recursion known as *tree recursion*.

Tree recursion, in its most basic form, is simply when a function calls itself more than once (usually twice, though not necessarily). However, when it comes to actually coding a tree-recursive function, it can be useful to think about it in the following way: whenever you are faced with a problem where you have to consider ALL POSSIBLE CASES of a situation that seems to branch (ha, tree pun) off endlessly in all these directions that seem impossible to keep track of, it's a safe bet that you will have to utilize the technique of tree recursion.

Okay, now that we have a basis from which to view it, let's consider the actual Count Change function. Here it is, in all its mind-blowing glory:



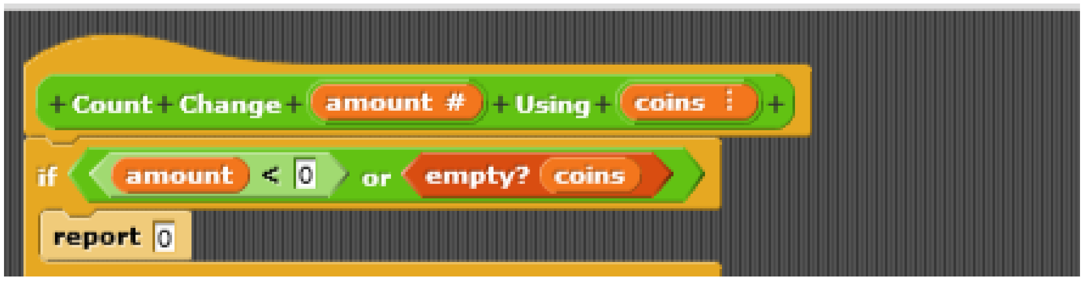
```
+Count+Change+ amount # +Using+ coins ⋮ +
if < amount < 0 > or < empty? coins >
    report 0
if < amount = 0 >
    report 1
report
    Count Change amount Using all but first of coins +
    Count Change amount − item 1 of coins Using coins
```

Now, before we start breaking it down piece by piece, let's think about what the function's overall goal is. We take in two inputs: *amount* and *coins*. Amount is some integer value, and coins is a list of coins we can use to make change for that amount.

- Given this list of coins, we want to figure out how many different ways we can make change for the amount input using the coins we have. **We assume that for any given coin in the list, we can use as many of it as we would like.**

Part 1: The Base Cases

Now that we understand what this function is meant to do, let's look into it a bit further, starting with the first base case:



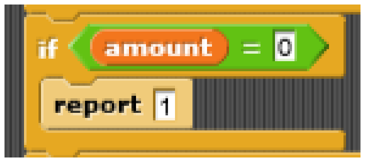
```
+ Count + Change + amount # + Using + coins ⋮ +
if < amount < 0 > or < empty? coins >
    report 0
```

So what is happening here? Translate it into English instead of trying to understand the code in its raw form—a great strategy for any computer science problem, by the way.

This says is that if the amount of change we are trying to make is less than 0 *or* our list of coins is empty, there are no ways to make change. Why? Let's break it down:

- **Amount < 0**: There is no such thing as a negative coin, so how are we supposed to use the coins we are given to make change for a negative number? We can't.
- **Empty Coins**: This one is a bit more intuitive. If someone asks us to make change for a certain amount, whatever it may be, how are we supposed to do it without any coins? We can't.

Okay, now we turn to the second base case:



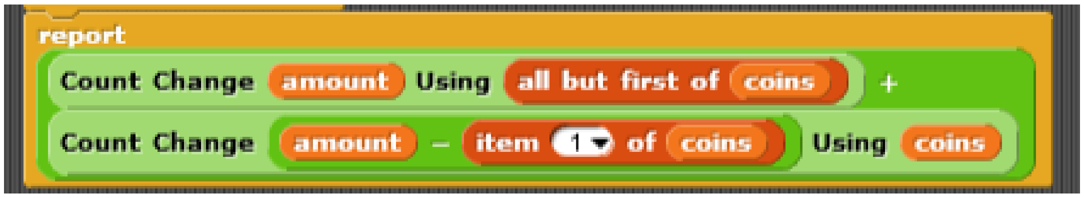
```
if < amount = 0 >
    report 1
```

Here, we only have one thing to ask ourselves according to the conditional statement: what happens when the amount of change we are trying to create is 0?

Clearly, we report 1. But *why*? Think of it like this: if our goal is to use our assortment of coins to count to the value 0, there is precisely **one and only one** way that we can do this: to look at our list of coins and use none of them. **Keep in mind that the reason we report 1 is that this function returns the number of ways to make change for a certain amount, NOT the number of coins we use to make that change.**

Part 2: The Recursive Cases:

Okay, now it gets really confusing, but we got this. One step at a time, let's understand the recursive portion of this function:



```
report
  Count Change (amount) Using (all but first of (coins)) +
  Count Change ((amount) − (item (1) of (coins))) Using (coins)
```

Okay. It calls itself once with the same amount and all but first of coins, then it calls itself again with a smaller amount and the same coins, and then it adds them, and that does … something? All right, so maybe just jumping into it head-on wasn't the best idea. Clearly, we need to approach understanding this from a different perspective.

Let's try this. For a second, forget about the function, and let's just talk about counting change ourselves with a concrete example. No weird code to deal with.

Consider the following: we want to make change for 15¢, and we have the following coins: [10¢, 5¢, 1¢]. So what are all the ways we can do this?

Change for 15¢ w/ [10¢, 5¢, 1¢]

Fifteen 1¢ coins*

Two 5¢ coins & Five 1¢ coins

Three 5¢ coins

One 5¢ coin & Ten 1¢ coins

One 10¢ coin & One 5¢ coin

One 10¢ coin & Five 1¢ coins

Clearly, there are six ways to make change for 15¢ with the coins we have been given (you can check if you don't believe me). Notice I have color coded the options above to signify two different groups. The reason for this is that when we count change for 15¢ with the coins [10¢, 5¢, 1¢], we can actually break it down into two parts: 1) counting the number of ways to make change for 15¢ with [5¢, 1¢] and 2) counting the number of ways to make change for 5¢ with [10¢, 5¢, 1¢]. We can then add the number of ways to do these two things to get our answer. Don't believe me? Take a look:

Change for 15¢ w/ [5¢, 1¢]	**Change for 5¢ w/ [10¢, 5¢, 1¢]**
Fifteen 1¢ coins	One 5¢ coin
Two 5¢ coins & Five 1¢ coins	Five 1¢ coins
Three 5¢ coins	
One 5¢ coin & Ten 1¢ coins	

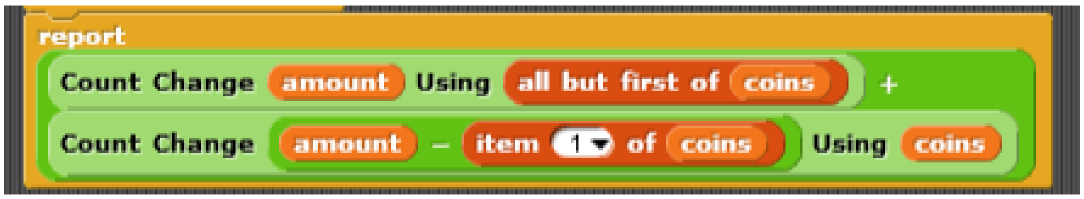
Please note that the colors here do NOT mean the values are the same as the previous page. This is clearly not the case with the 5¢ case above, as it uses no 10¢ coins. The colors signify which cases in the original problem match the broken-down version. However, also note that the 10¢ coin may look as if it is never used, but in reality was when we initially subtract 10 from 15 to get 5.

There are clearly four ways to make change for 15¢ with the coins [5¢, 1¢] and two ways to make change for 5¢ with the coins [10¢, 5¢, 1¢]. And 4 + 2 = 6! 6 is the precise number of ways to solve our initial problem: the number of ways to make change for 15¢ with the coins [10¢, 5¢, 1¢].

Okay, I admit that was kind of a lot to take in at once. For one thing, the way that we divided the problem above probably seems kind of obscure. Why did we split it that way? *Why did it work?*

When it comes to tree recursion, you always have to look for some way to split your problem into smaller parts that you can then address separately before combining them into a single output. The best strategy for doing this is to try to think of a relatively simple example (but one more complicated than the base case) and list out all of the possibilities, as we did above. Then, look for a natural way to split them up. However, the intricacies of this are not so important for this class—you'll get plenty of practice with it in later CS courses.

Our purpose is to understand Count Change. If you take another look at the recursive case, it may become easier to see how we knew where to split the 6 possibilities above:



```
report
  Count Change (amount) Using (all but first of (coins)) +
  Count Change ((amount) − (item (1) of (coins))) Using (coins)
```

It came straight from the function definition! It tells us what needs to be done in each recursive call:

- **First Recursive Call**: Count the same amount of change, but get rid of the first coin in your list.
- **Second Recursive Call**: Reduce your *amount* by the value of the first coin in your list, and count this new amount of change, but with the same list of coins.

The fundamental idea is this: with each coin we consider in our list, we have to make a decision. In reaching our total, either we ***do not use the coin (recursive case 1) or we do use the coin (recursive case 2). There is no other possibility, right? This is the key insight we use in order to figure out the "how" of the recursive case.***

However, you might still be confused as to why exactly this works. As I will stress more than once, do not try to think of extremely deep recursive cases or the subtle intricacies of the inner workings of this function, because that will only confuse you. Think of it like this: The reason this works is that, as we can see in the example above, breaking down our problem into these two smaller problems gives us the same answer as when we solve the problem wholly. Now, in terms of coding, it would be immensely difficult to try to write something that would solve this problem as a whole. So what do we do? We break it down into two parts, and then we *trust the recursion*. It seems counterintuitive, but part of writing recursive solutions to problems is trusting that you can use the function you are currently writing and that it will work properly despite being incomplete at the given moment. In this specific case, we split our change problem into two parts, and we notice that we need to count the number of ways to make change for these two problems and add them up. Do we have a function that can count change for us given a list of coins? Why, yes we do—it's the one we're currently using! And hence we use it in the recursive calls.

Note: The previous two paragraphs can be very difficult to understand after reading only once. Don't stress—it's like that for everyone. Just keep doing your best and give it time. Stare at it for a while, and at some point it will click.

Okay, there is one more subtlety worth considering with this function. Take on more look at the example split we did earlier:

Change for 15¢ w/ [5¢, 1¢]	**Change for 5¢ w/ [10¢, 5¢, 1¢]**
Fifteen 1¢ coins	One 5¢ coin
Two 5¢ coins & Five 1¢ coins	Five 1¢ coins
Three 5¢ coins	
One 5¢ coin & Ten 1¢ coins	

Some people find it confusing that the second part of the split (the red one) needs to have a 10¢ coin in its list at all. It is not as if this coin is of any use when we make change for 5¢.

Technically, this is true. However, you have to remember that this example split is a simplification in order to help explain how Count Change does what it does. In reality, *nothing is actually computed until the base cases are reached.* Keeping this in mind, consider what would happen if we removed the 10¢ coin from the second part of the split. Then, both recursive calls in Count Change would have an “all but first of coins.”

Think about what this would mean. Would we use that coin at all? We would, actually: the second recursive call accounts for one usage of this coin where it says “amount – item 1 of coins” (by the way, if you pay close attention, this shows that the 10¢ coin IS technically used when we first subtract 10 from 15 to get 5. It’s just not needed when we count to 5.). But then, the coin would be gone for good because it would be removed from the list in both recursive calls. I know this is hard to follow, but just think about it logically. If this were to happen, we could only use each coin *once*. But that wouldn’t work. For instance, one way to make change for 15¢ is to use three 5¢ coins, and that is perfectly valid. Therefore, to account for cases where we need a coin more than once, we cannot remove it from both recursive calls. If this still bothers you because it doesn’t quite match the top-level example above, just look at it this way: having the 10¢ coin there when we make change for 5¢ doesn’t hurt (we just won’t use it), and we keep it around in case we need it later.

Congratulations!

If you made it this far, I commend you. I know Count Change can be absolute torture, but keep pushing yourself. You’ll get it, and once you conquer it you’ll feel like such a boss. Good luck!