HW 6 - More Recursion

Making Change

At this point in the semester you should be able to attempt a HW without starter code.
This Homework will give you more practice on coming up with recursive solutions which will help you work with Grapsh/Networks and Trees.

Imagine that you just got a job working for a company that builds electronic cash registers and the software that controls them. The cash registers are used all around the world, in countries with many different coin systems.
Next, imagine that we are given a list of the coin types in a given country. For example, in the U.S. the coin types are:

```
[1, 5, 10, 25, 50, 100]
```

and in Europe they are:

```
[1, 2, 5, 10, 20, 50, 100, 200]
```

But in the Kingdom of Shmorbodia, the coin types are:

```
[1, 7, 24, 42]
```

In general, the coin system could be anything, except that there is **always a 1-unit coin (penny)**. In these examples, we gave the denominations from smallest to largest, but that's not always necessarily the case. There's nothing about use-it-or-lose-it that depends on the order of the coins.
Here's the problem: given an amount of money and a list of coin types, we would like to find the **smallest number of coins** that makes up that amount of money. For example, in the U.S. system, if we want to make 48 cents, we give out 1 quarter, 2 dimes, and 3 pennies. That solution uses 6 coins, which is the best we can do for this case. Making 48 cents in the Shmorbodian system, however, is different. Giving out a 42-cent coin—albeit tempting—will force us to give the remaining balance with 6 pennies, using a total of 7 coins. We could do better by simply giving two 24-cent coins.
Your first task is to write a function called change(amount, coins), where amount is a non-negative integer indicating the amount of change to be made and coins is a list of coin values. The function should return a non-negative integer indicating the minimum number of coins required to make up the given amount.
Here is an example of this function in action:

```
In [1]: change(48, [1, 5, 10, 25, 50])
Out[1]: 6


In [2]: change(48, [1, 7, 24, 42])
Out[2]: 2


In [3]: change(35, [1, 3, 16, 30, 50])
Out[3]: 3


In [4]: change(6, [4, 5, 9])
Out[4]: inf
```

In the last case, the function returns the special number "inf", meaning infinity, to indicate that change can't be made for that amount (because there is no 1-unit coin). See below for more on this case.

A few notes and tips...

Not surprisingly, the secret to all happiness is to use the *use-it-or-lose-it* recursion strategy.

Second, you may want to use the built-in function min(x, y), which returns the smaller of its two arguments.

Third, in the event that change is confronted with a problem for which there is no solution, returning an infinite value is an appropriate way to indicate that there is no number of coins that would work. This happens, for example, when we are asked to make change for some positive amount of money but there are no coins in the list. What is infinity in Python? In class, we saw one way to do this: First, import the `math` package:

```
from math import *
```

Now, you have access to the value `inf`. Alternatively, if you import the math package this way...

```
import math
```

...then you would access infinity with `math.inf`. Finally, if you don't import the math package at all, you can still get infinity in this slightly weird way: float('inf').

Giving Change

Just knowing the minimum number of coins is not as useful as getting the actual list of coins. Next, write another version of the change function called giveChange, which takes the same arguments as change but returns a list whose first member is the minimum number of coins and whose second member is a list of the coins in that optimal solution. We sometimes call such a list a "care package" because it packages up all the information you need to know about a given solution, including the information needed to use that solution in developing a more complex one!
Here's an example:

```
In [1]: giveChange(48, [1, 5, 10, 25, 50])
Out[1]: [6, [25, 10, 10, 1, 1, 1]]

In [2]: giveChange(48, [1, 7, 24, 42])
Out[2]: [2, [24, 24]]

In [3]: giveChange(35, [1, 3, 16, 30, 50])
Out[3]: [3, [16, 16, 3]]

In [4]: giveChange(6, [4, 5, 9])
Out[4]: [inf, []]
```

The order in which the coin values are presented in the original list doesn't matter, and similarly, the order in which your solution reports the coins to use is also unimportant: In other words the solution `[3, [16, 16, 3]]` is the same to us as `[3, [3, 16, 16] ]` or `[3, [16, 3, 16]]`. After all, all of these solutions use the same three coins!
Here are a few observations to keep in mind. First, while it may be tempting to have `giveChange` call your `change` function, this is actually not so helpful! Instead, write an all-new function called `giveChange` that calls itself recursively but doesn't call any other function for help! In particular, `giveChange` should NOT call the `change` function. However, your `giveChange` function will be structured very similarly to your `change` function.
Note that `giveChange` will always return a list of the form [numberOfCoins, listOfCoins]. So, if your original `change` function returned 0, for example, then your new `giveChange` function would probably return `[0, []]` instead to indicate that there are zero coins of change and the list of coins is the empty list!

Now, use your `change` function as a guide to write your `giveChange` function, but keep in mind that every time `change` would have returned a number (a number of coins), your new `giveChange` function will return a list of the form [numberOfCoins, listOfCoins]. You will have to "take apart" that list so that you can work with its individual components, and then put it back together to get your final result.

Submit

Submit your file as `hw6.py