Making Change [30 points; individual or pair]

Copied from:
https://www.cs.hmc.edu/twiki/bin/view/CS5/MakingChange2011 on 3/22/2017

You may wish to wait until after Thursday's lecture to embark on this problem.

In class, we discussed the Longest Common Subsequence Problem and the use-it-or-lose-it paradigm. In this problem, you will solve a different problem using the same powerful use-it-or-lose-it strategy. There is no Knapsack problem on this week's homework, but you might want to review the knapsack solution from class and think about how to extend that solution to return the actual list of items (as Geoff discussed just briefly and in general terms at the end of Thursday's lecture).

Imagine that you just got a job working for Cash Register Advanced Products. (The public-relations division has suggested that the company avoid using an acronym for the company name.) The company 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]
```

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, the denominations were given 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 **least 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?
Using `float('inf')` will do the trick; the result behaves like infinity should. Here are some examples:

```
In [1]: float('inf') > 42
Out[1]: True

In [2]: 42 + float('inf')
Out[2]: inf                          <-- It's still infinity!

In [3]: min(42, float('inf'))
Out[3]: 42
```

Finally, note that `with` is a Python keyword, so don't try creating variables named `with` and `without`. We suggest `useIt` and `loseIt` instead.

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. 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 really 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!

This problem may seem challenging at first, but keep in mind that once you establish that `giveChange` will **always** return a list of the form `[numberOfCoins, listOfCoins]`, you can modify your `change` function relatively modestly to get the `giveChange` function. First, your base cases must observe the convention

and return a list of the form `[numberOfCoins, listOfCoins]`. Then, when you call `giveChange` recursively, remember that it is returning a list of this form.

Your function will need to pick that list apart to get at the number of coins and the list of coins in that solution. Finally, after deciding whether the use-it solution or the lose-it one is better, you can prepare your list of the form `[numberOfCoins, listOfCoins]` and return that list.

Submit

Submit your file as `hw1pr3.py`.