Dynamic programming

This homework problem is the same as the homework Recursion, but you should use dynamic programming to solve the problem. You will complete this homework without a starter code. Relying on what we learned in class about how to set up and document functions.
This Homework will give you more practice in coming up with dynamic programming solutions.

Making Change

Imagine that you just got a job working for a company that builds electronic cash registers and the software that controls them. 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 make 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 the amount is a non-negative **integer** indicating the amount of change to be made and coins are 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. If there is no possible solution, return **`math.inf`**

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]: math.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**).

A few notes and tips...
First, you need to use the **dynamic programming** strategy.
Second, you may want to use the built-in function **`min(x, y)`**, which returns the smaller of its two arguments.
Third, if the 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:
If you import the math package this way
`import math`
then you would access infinity with **`math.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(int)** 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!

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 a file named **dynamic_programming.py** This file should have at least the following functions `change()`, `giveChange()`.

Notice

- You may have other functions in **`change`** function, and you might lose points for using the helper functions in **`giveChange`** function.
- Your file should import the math library.
- Your file **should not have any commands outside of a function**.
- Your file **should not have any function calls out side of if __name__ == "__main__":**.
  Related reading of it: https://stackoverflow.com/questions/419163/what-does-if-name-main-do
- Your functions should be **documented** - this will be graded.
- Please do **not** use recursion for this homework, using recursion will cause time out error. If the autograder **fail** to generate a valid result, it is very likely your code has an infinite loop.
- **Please** test your code on your local machine and make sure it will generate a valid result in the give test case.
- Please solve the questions with the **tabulation method**. Here is an example for applying the tabulation method to calculate the Fibonacci sequence:

```
def fibonacci(n):
            if n == 0:
                return 0
            elif n == 1:
                return 1
            else:
                table = [0] * (n + 1)
                table[0] = 0
                table[1] = 1
                for i in range(2, n+1):
                    table[i] = table[i-1] + table[i-2]
                return table[n]
```

Only the solution with the **tabulation method** will be accepted. Using @cache decorator or caching the results in a dictionary will grant you **0 points**. For more information and definition on the tabulation method, you may refer to https://www.geeksforgeeks.org/tabulation-vs-memoization/.