Problem-Based Intro. to Computer Science (2010-1)
Greedy Change Week 7 - Lecture 1

1 Problem

image reference: `http://digitaleconomy.files.wordpress.com/2009/01/monopoly-money.jpg`

With paycheck in hand, you head to the bank to cash it in. The teller asks you what your preference is for the change. Your reply is "the fewest bills and coins possible." Moments later the teller hands you your change and you are on your way.

In order to serve your request the teller has run an *algorithm* in order to make the correct change. Do you see what the algorithm is? The teller always chooses the highest denomination amount possible, while making change, so that the minimal amount of change is used. In computer science we refer to this as a *greedy algorithm* because it always tries to stay ahead by choosing the *local optimum*.

Your task is to design a program, using the greedy algorithm, that can make change from a given amount (read from standard input). Since denomination units are different in each country, your program will also read in a series of valid denominations to use (also read from standard input). These denominations will be read, in any order, until the user enters a value of `-1`. This special value, `-1`, is called the *sentinel value* because it indicates when the loop should terminate. The sentinel value is not stored in the denomination list.

Here is an example of the program in action:

```
Enter denomination (-1 to end): 1
Enter denomination (-1 to end): 25
Enter denomination (-1 to end): 100
Enter denomination (-1 to end): 10
Enter denomination (-1 to end): 5
Enter denomination (-1 to end): -1
Change denominations are: [1, 25, 100, 10, 5]
Make change for: 967
Your change:
9 coin/s of value 100 each.
2 coin/s of value 25 each.
1 coin/s of value 10 each.
```

```
1 coin/s of value 5 each.
2 coin/s of value 1 each.
```

If the program cannot make correct change for whatever reason it should indicate so.

Finally, here is a question for you to ponder. Let's assume that correct change can physically be made. *Is there a scenario where the greedy algorithm fails to make change using the fewest coins?* In other words, is it possible that always being greedy can lead to either a non-optimal solution or no solution at all? See if you can come up with a test case that causes the program to fail to make change at all, even though it is possible.

2 Solution Design and Analysis

2.1 Representation of the Answer

The answer should be a sequence of structures, where each structure has the following parts: denomination and quantity. It is natural to represent the sequence as a Python list. It is less clear how we should represent the structures.

We have already seen a Python data structure that can represent structure defined by parts: the list. A Python tuple can also represent structure defined by parts. For example, the ordered pair $(2, 3)$ is a structure defined by parts. (What are the parts?) It can be represented by the Python tuple `(2, 3)`.

However, there are two problems when using lists or tuples to represent denomination-quantity structures. First, when accessing the parts of this structure, it feels unnatural to use numeric indices. Further, since numbers have no mnemonic value here it would be easy to forget which index corresponded to which part and introduce all sorts of program bugs. Second, we'd like to be able to distinguish between different structures with the same number of parts. While we might represent a denomination-quantity structure as a tuple, we might also want to represent a coordinate in the plane as a tuple. We don't want to run our change algorithm on a list of plane coordinates! We'd like to be able to distinguish them, and, in particular, we'd like them to be different kinds of things.

The Python `class` construct solves both of these problems. It allows us to use words rather than numbers to refer to the parts, and it declares the structure to be a new kind or class of thing.

The following Python code illustrates how to represent the denomination-quantity structure. The parts become slots in the class. The slots are specified by assigning the special `__slots__` variable a tuple of the names of the slots as strings.

```
class DenomQuant():
    """A denomination-quantity structure
       is represented as an instance of DenomQuant
    """
    __slots__ = ('denom','quant')
```

To create an *instance*, or example, it is possible to simply use the class name as a function call; however, it is preferable to write our own constructor functions that initialize the slots. Slots are initialized and accessed by writing a dot (period) and the slot name (not as a string) after the expression referring to the structure.

```
def mkDenomQuant(denom, quant):
   """Constructor:
      mkDenomQuant: Number * Number -> DenomQuant
   """
   struct = DenomQuant()
   struct.denom = denom
   struct.quant = quant
   return struct
```

The following expression evaluates to a value that represents nine coins of value 100 each.

```
mkDenomQuant(100, 9)
```

2.2 Algorithm Design

The greedy algorithm is run by the `make_change` function. It takes the money amount and the list of denominations. The main loop runs until correct change is made (the remaining amount is zero), or there are no more denominations to make change with.

Because the denominations can come in any order, we will sort the list. We then calculate the amount of change we can make for the denomination. The denomination and change amount are stored as a structure in the resulting `change_list`.

Because we always choose the next largest denomination, `change_list` is guaranteed to always be sorted in descending order of denomination units. Using the example in the problem statement, the resulting list, `change_list`, would be equal to the list generated by the expression `[mkDenomQuant(100, 9), mkDenomQuant(25, 2), mkDenomQuant(10, 1), mkDenomQuant(5, 1), mkDenomQuant(2,1)]`.

Once this list is returned, we can loop through it and print out each denomination unit and change amount.

2.3 Algorithm Analysis

The complexity of `make_change` is determined by the following factors:

- The main loop to compute the change runs a maximum of N times, where N is the number of unique denomination units. This is $O(N)$ or linear time. Each time the loop runs, one element is removed from the list.
- The cost of sorting a list is $O(N^2)$.
- The overall complexity is $O(N^2) + O(N)$, which is $O(N^2)$.

2.4 Pseudo-code

```
define make_change(amount, denom_list)
   sort(denom_list)
   change_list <- empty list

   while the amount is non-zero and there are still denoms
      pop and save the largest denom
      calculate number of coins this denom can make towards amount
      update the amount
      append mkDenomQuant(denom, number of coins) to the change_list


   if the final amount is non-zero
      we couldn't make correct change, return None
   otherwise
     return change_list to the caller

denom_list <- read the denominations until -1 is entered
amount <- read the amount to make change for
change_list <- call make_change with amount and denom_list
if change_list is a valid list
   print the change amounts in change_list
otherwise
   print a message that says correct change could not be made
```

Implementation

See `change.py`.

3 Testing

The example in the problem statement is a test case where the greedy algorithm produces an optimal result:

```
Change denominations are: [1, 25, 100, 10, 5]
Make change for: 967
Your change:
9 coin/s of value 100 each.
2 coin/s of value 25 each.
1 coin/s of value 10 each.
1 coin/s of value 5 each.
2 coin/s of value 1 each.
```

It is also possible to construct a test case where correct change cannot ever be made based on the denominations:

```
Change denominations are: [3, 7]
Make change for: 8
Can't make correct change!
```

Now to answer the question about whether the greedy algorithm can fail to produce a result. The answer is yes. Consider the following:

```
Change denominations are: [11, 9, 3]
Make change for: 18
Can't make correct change!
```

The optimal solution should be two coins of value 9 each. The greedy algorithm fails because it chooses one coin of value `11`, and then is unable to make change for the remaining amount of seven.

Concerning the answer to the question about whether the greedy algorithm can fail to produce the optimal result. The answer is yes. Consider the following:

```
Change denominations are: [7, 5, 1]
Make change for: 10
Your change:
1 coin/s of value 7 each.
3 coin/s of value 1 each.
```

The optimal solution should be two coins of value 5 each. The greedy algorithm fails because it chooses one coin of value `7`, and then must pick three `1`s to get to ten.