Computer Science 1
Greedy Change Lecture Week 7

9/29/2013

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. The teller asks you what is your preference for the change. Your reply is "the fewest bills and coins possible." Moments later the teller hands you the change and you are on your way.

In order to serve your request, the teller ran an *algorithm* to make the correct change. Do you see the algorithm? While making change, the teller always chooses the highest denomination amount possible so that the minimal number of coins and bills is returned. In computer science, we refer to this kind of algorithm as a *greedy algorithm* because it always operates by choosing the *local optimum*.

Your task is to design a program using a greedy algorithm that makes change for a given amount (read from *standard input*). Since denomination units may be different in each country, your program must also read a series of valid denominations to use (also read from standard input). The program will read these denominations in any order until the user enters a value of `-1`. This special value, `-1`, is called *a sentinel value* because it indicates when the (input) 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 print a message stating that it cannot make change for the amount.

Now, here is a question for you to ponder. Let's assume that making correct change is physically possible. *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 create a test case that causes the program to fail to make change, even though it is possible to make the correct change.

2 Problem Analysis and Solution Design

2.1 Representation of the Answer

The answer for the example could be crudely expressed in English as follows:

```
9 hundreds, 2 twenty-fives, 1 ten, 1 five and 2 ones.
```

Notice that each is a pair of values; we need a structure for each piece of the answer.

The answer must 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, but it is less clear how we should represent the structures.

We have already seen a Python data structure that can represent a 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; We can write a tuple assignment statement, `aPoint = (2, 3)`, to create a Python tuple. What are this tuple's parts, and how do we access them?

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. Since numbers have no mnemonic value, it is easy to forget which index corresponds to which part, and forgetting would introduce all sorts of program errors. Second, we would like to be able to distinguish between different structures that have 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 we would like points and denomination-quantity structures to be different *types*.

The Python `class` construct solves both of these problems. A class declares a **type**, a new *kind* of structure or object. Using a class for a structure with parts allows us to use named words rather than numbers to refer to the parts.

The following example Python code shows how to use a class declaration to represent a denomination-quantity structure. The parts become slots in a class. The slots are specified

by assigning a *tuple of strings* specifying the names of the slots to the special `__slots__` variable inside the class.

```
class DenomQuant():
    """
    A DenomQuant represents an instance of DenomQuant.
    """
    __slots__ = ('denom','quant')
```

To create an *instance* of the class, it is possible to use the class name as a function call, but this does not initialize the slots. Instead we can write a constructor, or builder, function to initialize the slots with values. Slots are initialized and accessed by writing a period (.) and the slot name (without quotations) after the expression or variable referring to the structure.

```
def mkDenomQuant( denom, quant ):
    """
    mkDenomQuant: Number * Number -> DenomQuant
    mkDenomQuant creates a DenomQuant with a denomination and
    a quantity.
    """
    struct = DenomQuant()     # create the instance,
    struct.denom = denom      # initialize the parts, and
    struct.quant = quant
    return struct             # return the instance.
```

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

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

2.2 Algorithm Design

The `make_change` function implements the greedy algorithm. It takes the money amount and the list of denominations. The `make_change` has a loop that runs until it makes correct change (the remaining amount is zero), or there are no more denominations remaining.

Because users can enter denominations in any order, we must sort the list. We then calculate the amount of change we can make using each denomination. As we use each denomination, a list named `change_list` stores the denomination and quantity structures.

Because we always choose the next largest denomination, the `change_list` will 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 this expression:

```
 [mkDenomQuant( 100, 9 ),
  mkDenomQuant( 25, 2 ),
  mkDenomQuant( 10, 1 ),
  mkDenomQuant( 5, 1 ),
  mkDenomQuant( 2, 1 )]
```

After `make_change` returns this list, we loop through it and print the number of coins and each denomination unit used.

2.3 Algorithm

```
Function make_change( amount, denom_list ) :
   call sort on the denom_list
   change_list <- make an empty list

   while the amount > zero and there are still denoms in denom_list :

      denom <- pop the largest denom out of the denom_list
      calculate number_of_coins denom can make towards the amount
      update the amount
      append mkDenomQuant( denom, number_of_coins ) to the change_list

   if the amount is non-zero :
      return None because the function could not make correct change
   otherwise
     return change_list to the caller

Procedure main program :
   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`.

2.4 Complexity Analysis

The complexity of the `make_change` program may be determined by the considering the following major operations:

- The cost of sorting a list, which we have learned is $O(N^2)$.
- The loop in `make_change` runs a maximum of N times, where N is the number of unique denomination units. Each time the loop runs, it removes one element from a list and appends a new DenomQuant structure to a list. Removal from the end of a list and `list.append` can each execute in $O(1)$, constant time. This loop, in which all steps are $O(1)$, runs in $O(1 \times N)$, linear time.
- The cost of printing the results: a simple loop through the `change_list` has complexity of $O(N)$.

The overall complexity of the program is the sum of its costs. The cost of `make_change` is the sorting cost, $O(N^2)$, plus the cost of its loop, $O(N)$. We also add the cost of printing the results, whose complexity is $O(N)$. Before simplification, we have $O(N^2 + 2 \times N)$; then we remove constants from the expression and keep only the term with the largest exponent. Simplification leaves $O(N^2)$, which is *quadratic time complexity*.

3 Testing

The example in the problem statement is a test case in which 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 find a test case in which 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 change when making change is actually possible. 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 is unable to make change for the remaining amount of `7`.

Here is another test case in which the greedy algorithm can produce a result that is *not an optimal result*. 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 `10`.