Instituto Politécnico Nacional
Escuela Superior de Cómputo

Evolutionary Computing
(Evolutionary Computation)

Practice No. 1: First steps in Python
(Greedy algorithms and Dynamic programming)

Student: García Medina Juan Carlos
Professor: Dr. Rosas Trigueros Jorge Luis

Date Practice: 07-Feb-2019
Date Report: 18-Feb-2019

THEORETICAL FRAMEWORK

Greedy Algorithms

An algorithm is said to be greedy if it makes the locally optimal choice at each step with the hope of eventually reaching the globally optimal solution. In some cases, greedy works–the solution is short and runs efficiently. For *many* others, however, it does not. A problem must exhibit these two properties to work:

1. It has optimal sub-structures.
   Optimal Solution to the problem contains optimal solutions to the sub-problems.

2. It has greedy property.
   If we make a choice that seems like the best at the moment and proceed to solve the remaining subproblem, we reach the optimal solution. We will never have to reconsider our previous choices. Steven Halim (2013)

Dynamic Programming

In order to solve a problem using Dynamic Programming (from now on abbreviated as DP), you must identify the relationshiops or *transitions* between current problems and their sub-problems. DP problems with small input size constraints may be already be solvable with recursive backtracking.

Top-Down versus Bottom-Up DP

Although both styles uses 'tables', the way the bottom-up DP table is filled is different to that of the top-down DP *memo* table, In the top-down DP, we used a correct 'DP table filling order' to compute the values such that the previous values needed to process the current cell have already been obtained.

The trade-offs between top-down and bottom-up DP's are listed in the table below.

Analysis	**Top-Down**	**Bottom-Up**
Pros	1. It is a natural transformation from the normal Complete Search recursion. 2. Computes the sub-problems only when necessary (sometimes this is faster).	1. Faster if many sub-problems are revisited as there is no overhead from recursive calls. 2. Can save memory space with the 'space saving trick' technique.
Cons	1. Slower if many sub-problems are revisited due to function call overhead. 2. If there are M states, an *O(M)* table size is required.	1. This style may not be intuitive like recursion. 2. If there are M states, bottom-up DP visits and filss the value of all these M states.

RESOURCES AND TOOLS

1. Computer with python3
    - The computer has an intel core i7 processor of 6th generation.
    - 8gb ram were sufficient and there were no problems of limited memory.
2. Tools
    - Matplotlib: Library used to plot the results, with high support and well written documentation.
    - Numpy: Library used to handle data and perform some operations like shuffle a list of numbers.

DISCUSSION

Coin Change

Greedy Approach

Problem description:
Given a target amount V cents and a list of denominations of n coins, i.e. we have *coinValue[i]* (in cents) for coin types $i \in [0..n\text{-}1]$, what is the minimum number of coins that we must use to represent amount V? Assuming that we have unlimited supply of coins of any type. Example: If $n = 4$, $coinValue = \{25, 10, 5, 1\}$ cents, and we want to represent $V = 42$ cents, we can use this greedy algorithm: Select the largest coin denomination which is not greater than the remaining amount, i.e. *42 - 25 = 17* $\rightarrow$ *17 -10 = 7* $\rightarrow$ *7 - 5 = 2* $\rightarrow$ *2 - 1 = 1* $\rightarrow$ *1 - 1 = 0*, a total of 5 coins. This is optimal.

The problem above has the two ingredients required for a successful greedy algorithm:

1. It has optimal sub-structures. We have seen that in our quest to represent 42 cents, we used *25 + 10 + 5 + 1 + 1*.
   This is an optimal 5-coin solution to the original problem!
   Optimal solutions to sub-problem are contained within the 5-coin solution i.e.
   a. To represent 17 cents, we can use 10 + 5 + 1 + 1 (Part of the solution for 42 cents).
   b. To represent 7 cents, we can use 5 + 1 + 1 (also part of the solution for 42 cents), etc.

2. It has the greedy property: Given every amount V, we can greedily subtract from it the largest coin denomination which is not greater than this amount V. It can be proven (not shown here for brevity) that using any other strategies will not lead to an optimal solution, at least for this set of coin denominations. However, this greedy algorithm does not work for all sets of coin denominations.

   Take for example $\{4, 3, 1\}$ cents. To make 6 cents with that set, a greedy algorithm would choose 3 coins $\{4, 1, 1\}$ instead of the optimal solution that uses 2 coins $\{3, 3\}$.

Code

The main part of the code written in python is shown below:

```
def changeCoins(coins, val):
        coins.sort()
        index = len(coins) - 1

        used_coins = dict()
        for coin in range(0, len(coins)):
                used_coins[coin] = 0
        print(coins)
        counter = 0

        while val > 0 and index >= 0:
                while index > 0 and coins[index] > val:
                        index = index - 1
                counter = counter + 1
                if coins[index] <= val:
                        used_coins.update( {index: used_coins.get(index)  + 1})
                        print(val, index)
                val = val - coins[index]
```

To see the complete code go to this page.

The argument *coins* is a list with non-ordered integers numbers which represent the

coin denominations, and the second argument *val* es the value that we cant to change. This approach attempts to take the first higher coin that is less than the value, e.g.

if *val = 20* and our list *coins = [1, 4, 5, 15, 35]* then we cannot change the *35* for the *val* as it is greater than, therefore we need to iterate until the coin with value *15* (*coins[3]* 0-based index), then *val = 20 - 15 = 5* and the next coin that can be changed is *5* so we need to change the val two times.

In Figure 1 it is shown an example with a set of coins = [1, 16, 46, 79, 85] and *val = 2795* we needed to use 13 coins of 1 cent, 1 coin of 16 cents, 1 coin of 46 cents and 32 coins of 85 cents, giving a total of 47 coins used.

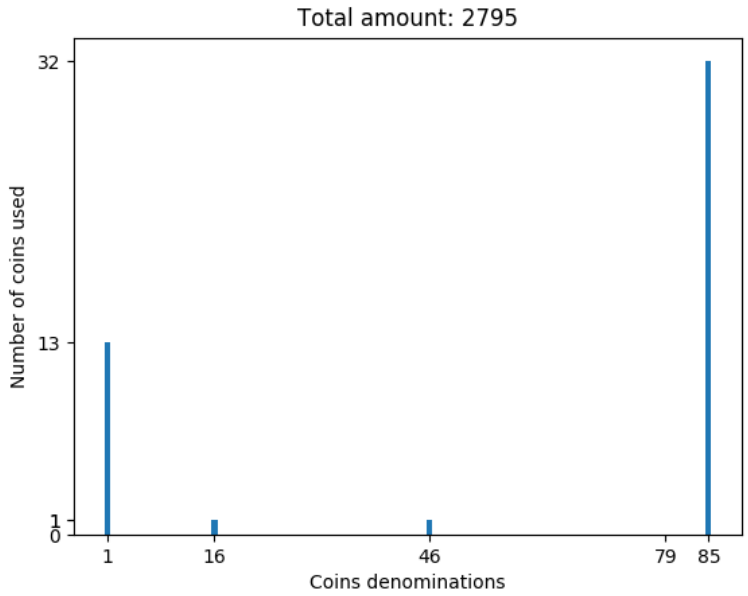

Figure 1: Plot greedy coin change

DP Approach

Code

The main part of the code written in python is shown below:

```
def changeCoins(coins, value):
        coins.sort()
        memo = np.full(value + 1, np.Inf)

        memo[0] = 0

        for i in range(1, value + 1):
                memo[i] = np.Inf

        for i in range(1, value + 1):
                for j in range(0, len(coins)):
```

```
                        current = memo[i - coins[j]]

                        if current != np.Inf and current + 1  < memo[i]:
                                memo[i] = current + 1
    return int(memo[value])
```

To see the complete code go to this page

Knapsack

Problem: Given n items, each with its own value V_i, and weight $W_i \in$ [0..n-1], and a maximum knapsack size S, compute the maximum value of the items that we can carry, if we can either, ignore or take a particular item (hence the term 0-1 for ignore/take).

Example: $n = 4$, W $= \{10, 4, 6, 12\}$, $S = 12$.
If we select item 0 with weight 10 and value 100, we cannot take any other item. Not optimal.
If we select item 3 with weight 12 and value 10, we cannot take any other item. Not optimal.
If we select item 1 and 2 with total weight 10 and total value 120, this is the maximum.

Greedy Approach

A greedy algorithm for the fractional knapsack problem. Note that we sort by v/w without modifying v or w so that we can output the indices of the actual items in the knapsack at the end.

Code:

```
def KnapsackFrac(v, w, W):
  order = bubblesortByRatio(v, w)
  weight = 0.0
  value = 0.0
  knapsack = []
  n = len(v)
  index = 0
  while (weight < W) and (index < n):
    if weight + w[order[index]] <= W:
      knapsack.append((order[index], 1.0))
      weight = weight + w[order[index]]
      value = value + v[order[index]]
    else:
      fraction = (W - weight) / w[order[index]]
      knapsack.append((order[index], fraction))
      weight = W
      value = value + v[order[index]] * fraction
    index = index + 1
  return (knapsack, value)
```

To see the complete code go to this page

DP Approach

Solution: Use these complete Search recurrences `val(id, remW)` where `id` is the index of the current item to be considered and `remW` is the remaining wight left in the knapsack:

1. `val(id, 0) = 0` i.e if `remW= 0`, we cannot take anything else.
2. `val(n, remW) = 0` i.e if `id = n`, we have considered all items.
3. if `W[id] > remW` we have no choice but to ignore this item
   `val(id, remW) = val(id + 1, remW)`
4. if `W[id]` $\leq$ `remW`, we have to choices, ignore or take this item; we take the maximum `val(id, remW) = max(val(id + 1, remW), V[id] + val(id + 1, remW - W[id])`

The answer can be found by calling `value(0, S)`. Note the opverlapping sub-problems in this 0-1 Knapsack problem. Example: After taking item 0 and ignoring item 1-2, we arrive at state `(3, 2)` –at the third item `id = 3` with two units of weight left `(remW = 2)`. After ignoring item 0 and taking item 1-2, we also arrive at the state `(3, 2)`. Although there are overlapping sub-problems, there are only *O(nS* possuble distinct states (as `id` can vary between `[0..n-1]`) and `remW` can vary between `[0..S]`! We can compute each of these states in *O(1)* , thus the overall time complexity of this `DP` solution is *O(nS)*.

CONCLUSION AND RECOMMENDATIONS

At the end of this practice I was able to identify which kind of problems can be candidates to be solved using a greedy solution or DP, but considering these points:

- Greedy algorithms can have a low space and time complexity, however it is good to try finding counterexamples due to that not all the problems (including specific restrictions) satisfy the conditions to give an optimal answer as we could see with the coin change problem.
- DP problems offer right solutions with just a little bit greater space and time complexity than greedy but (at least for me) it is harder to find the way to solve a problem with this paradigm, specially with `Bottom-up` DP.

  Doing more research about DP problems, I found that to solve a problem using DP you can try first to use backtracking and then use `Top-down` DP, after this the `Bottom-up` could be more easy to see. I also realised that to master these paradagims you have to solve a lot of problems.

REFERENCES

Steven Halim, F. H., 2013, Competitive programming 3: Lulu.