Lab 7

COMP9021, Session 2, 2017

1 ☞ Change-making problem: greedy solution

Write a program greedy_change.py that prompts the user for an amount, and outputs the minimal number of banknotes needed to yield that amount, as well as the detail of how many banknotes of each type value are used. The available banknotes have a face value which is one of $1, $2, $5, $10, $20, $50, and $100.

Here are examples of interactions:

```
$ python3 greedy_change.py
Input the desired amount: 10

1 banknote is needed.
The detail is:
 $10: 1
$ python3 greedy_change.py
Input the desired amount: 739

12 banknotes are needed
The detail is:
$100: 7
 $20: 1
 $10: 1
  $5: 1
  $2: 2
$ python3 greedy_change.py
Input the desired amount: 35642

359 banknotes are needed
The detail is:
$100: 356
 $20: 2
  $2: 1
```

The natural solution implements a *greedy* approach: we always look for the largest possible face value to deduct from what remains of the amount.

Suppose that the available banknotes had a face value which was one of $1, $20, and $50. For an amount of $60, the greedy algorithm would not work, as it would yield one $50 banknote and ten $1 banknotes, so eleven banknotes all together, whereas we only need three $20 banknotes.

2 Change-making problem: general solution (optional)

Write a program general_change.py that prompts the user for the face values of banknotes and their associated quantities as well as for an amount, and if possible, outputs the minimal number of banknotes needed to match that amount, as well as the detail of how many banknotes of each type value are used.

Here are examples of interactions:

```
$ python3 general_change.py
Input pairs of the form 'value : number'
   to indicate that you have 'number' many banknotes of face value 'value'.
Input these pairs one per line, with a blank line to indicate end of input.

2 : 100
50: 100

Input the desired amount: 99

There is no solution.
$ python3 general_change.py
Input pairs of the form 'value : number'
   to indicate that you have 'number' many banknotes of face value 'value'.
Input these pairs one per line, with a blank line to indicate end of input.

1  : 30
20 : 30
50 : 30

Input the desired amount: 60
There is a unique solution:
$20: 3
```

```
$ python3 general_change.py
Input pairs of the form 'value : number'
   to indicate that you have 'number' many banknotes of face value 'value'.
Input these pairs one per line, with a blank line to indicate end of input.

1: 100
2: 5
3: 4
10:5
20:4
30:1

Input the desired amount: 107

There are 2 solutions:
 $1: 1
 $3: 2
$10: 1
$20: 3
$30: 1

 $2: 2
 $3: 1
$10: 1
$20: 3
$30: 1
```

The natural approach makes use of the linear programming technique exemplified in the computation of the Levenshtein distance between two words.

3 ☞ The Target puzzle

The Target puzzle is a 3×3 grid (the target) consisting of 9 distinct (uppercase) letters, from which it is possible to create one 9-letter word. The aim of the puzzle is to find words consisting of distinct letters all in the target, one of which has to be the letter at the centre of the target. Write a program `target.py` that defines a class `Target` with the following properties.

- To create a Target object, three keyword only arguments can be provided:
  - `dictionary`, meant to be the file name of a dictionary storing all valid words, with a default value for a default dictionary named `dictionary.txt`, supposed to be stored in the working directory;
  - `target`, with a default value of `None`, otherwise meant to be a 9-letter string defining a valid target (in case it is not valid, it will be ignored and a random target will be generated as if that argument had not been provided);
  - `minimal_length`, for the minimal length of words to discover, with a default value of 4.
- `__repr__()` and `__str__()` are implemented.
- It has a method `number_of_solutions()` to display the number of solutions for each word length for which a solution exists.
- It has a method `give_solutions()` to display all solutions for each word length for which a solution exists; this method has an argument, `minimal_length`, with a default value of `None`, that if provided allows one to display only solutions of that length or more.
- It has a method named `change_target()`, that takes two arguments, `to_be_replaced` and `to_replace`, both meant to be strings. The target will be modified if:
  - `to_be_replaced` and `to_replace` are different strings of the same length;
  - all letters in `to_be_replaced` are distinct and occur in the current target;
  - replacing each letter in `to_be_replaced` by the corresponding letter in `to_replace` yields a valid target.

  If those conditions are not satisfied then the method prints out a message indicating that the target was not changed. If the target was changed but consists of the same letters, and with the same letter at the centre, then the method prints out a message indicating that the solutions are not changed.

Here is a possible interaction.

```
$ python3
...
>>> from target import *
>>> target = Target()
>>> target
Target(dictionary = dictionary.txt, minimal_length = 4
>>> print(target)

       -----------
      | S | M | E |
       -----------
      | N | G | U |
       -----------
      | J | T | D |
       -----------


>>> target.number_of_solutions()
In decreasing order of length between 9 and 4:
    1 solution of length 9
    1 solution of length 8
    2 solutions of length 6
    5 solutions of length 5
    16 solutions of length 4
>>> target.give_solutions(5)
Solution of length 9:
    JUDGMENTS

Solution of length 8:
    JUDGMENT

Solutions of length 6:
    JUDGES
    SMUDGE

Solutions of length 5:
    GENUS
    GUEST
    JUDGE
    NUDGE
    STUNG
>>> target.change_target('MT', 'TT')
The target was not changed.
>>> target.change_target('JUDGMENTS', 'ABCDEFGHI')
The target was not changed.
```

```
>>> target.change_target('MT', 'TM')
The solutions are not changed.
>>> target.change_target('GM', 'MG')
>>> target.give_solutions()
Solution of length 9:
    JUDGMENTS

Solution of length 8:
    JUDGMENT

Solution of length 6:
    SMUDGE

Solutions of length 5:
    MENDS
    MENUS
    MUNDT
    MUSED
    MUTED

Solutions of length 4:
    GEMS
    GUMS
    MEND
    MENS
    MENU
    METS
    MUGS
    MUNG
    MUSE
    MUST
    MUTE
    SMUG
    SMUT
    STEM
>>> target = Target(target = 'IMRVOZATK', minimal_length = 5)
>>> print(target)

       -----------

      | I | M | R |
       -----------

      | V | O | Z |
       -----------

      | A | T | K |
       -----------
```

```
>>> target.number_of_solutions()
In decreasing order of length between 9 and 5:
    1 solution of length 9
    2 solutions of length 6
    6 solutions of length 5
>>> target.give_solutions()
Solution of length 9:
    MARKOVITZ

Solutions of length 6:
    MARKOV
    MOZART

Solutions of length 5:
    KIROV
    MAORI
    MARIO
    OZARK
    RATIO
    VOMIT
>>> target.change_target('IVAKZRMO', 'DAFNEMRS')
>>> print(target)

       -----------

      | D | R | M |

       -----------

      | A | S | E |

       -----------

      | F | T | N |

       -----------

>>> target.give_solutions(9)
Solution of length 9:
    DRAFTSMEN
```

4 The n-queens puzzle (optional, advanced)

This is a well known puzzle: place n chess queens on an $n \times n$ chessboard so that no queen is attacked by any other queen (that is, no two queens are on the same row, or on the same column, or on the same diagonal). There are numerous solutions to this puzzle that illustrate all kinds of programming techniques. You will find lots of material, lots of solutions on the web. You can of course start with the wikipedia page: http://en.wikipedia.org/wiki/Eight_queens_puzzle. You should try and solve this puzzle in any way you like.

One set of technique consists in generating permutations of the list $[0, 1, ..., n-1]$, a permutation $[k_0, k_1, ..., k_{n-1}]$ requesting to place the queen of the first row in the $(k_0 + 1)$-st column, the queen of the second row in the $(k_1 + 1)$-st column, etc. For instance, with $n = 8$ (the standard chessboard size), the permutation $[4, 6, 1, 5, 2, 0, 3, 7]$ gives rise to the solution:

```
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1
```

The program `cryptarithm_v1.py`, written by Raymond Hettinger as part of ActiveState Code Recipes, demonstrates the use of the `permutations()` function from the `itertools` module. With `cryptarithm_v2.py`, we have a variant with an implementation of Heap's algorithm to generate permutations and a technique to 'skip' some of them. We could do the same here. For instance, starting with $[0, 1, 2, 3, 4, 5, 6, 7]$, we find out that the queen on the penultimate row is attacked by the queen on the last row, and skip all permutations of $[0, 1, 2, 3, 4, 5, 6, 7]$ that end in $[6, 7]$. If you have acquired a good understanding of the description of Heap's algorithm given in `Permutations.pdf` and of `cryptarithm_v2.py`, then try and solve the n-queens puzzle generating permutations and skipping some using Heap's algorithm; this is the solution I will provide. Doing so will bring your understanding of recursion to new levels, but it is not an easy problem, only attempt it if you want to challenge yourself...

Here is a possible interaction. It is interesting to print out the number of permutations being tested.

```
$ python3
...
>>> from queen_puzzle import *
>>> puzzle = QueenPuzzle(8)
>>> puzzle.print_nb_of_tested_permutations()
3544
>>> puzzle.print_nb_of_solutions()
92
>>> puzzle.print_solution(0)
 0 0 0 0 1 0 0 0
 0 0 0 0 0 0 1 0
 0 1 0 0 0 0 0 0
 0 0 0 0 0 1 0 0
 0 0 1 0 0 0 0 0
 1 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 0 1
>>> puzzle.print_solution(45)
 0 0 0 0 0 1 0 0
 1 0 0 0 0 0 0 0
 0 0 0 0 1 0 0 0
 0 1 0 0 0 0 0 0
 0 0 0 0 0 0 0 1
 0 0 1 0 0 0 0 0
 0 0 0 0 0 0 1 0
 0 0 0 1 0 0 0 0
>>> puzzle = QueenPuzzle(11)
>>> puzzle.print_nb_of_tested_permutations()
382112
>>> puzzle.print_nb_of_solutions()
2680
>>> puzzle.print_solution(1346)
 0 0 0 0 0 1 0 0 0 0 0
 0 0 0 0 0 0 0 1 0 0 0
 0 0 1 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 1
 0 0 0 0 0 0 1 0 0 0 0
 0 1 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 1 0
 1 0 0 0 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 1 0 0
 0 0 0 0 1 0 0 0 0 0 0
```

5 Random walk (optional)

Write a program `random_walk.py` that creates the picture of a random walk for a default of 50,001 points, starting from the point $(0, 0)$ and randomly choosing at every step to move horizontally and vertically by at most 4 units, west or east and north or south, respectively—it is allowed to move only horizontally or only vertically, but not to stay in place. The picture is drawn thanks to the `matplotlib.pyplot` module, which it is convenient to import as `plt`.

- The picture is 5 inches wide and 3 inches high—check out `plt.figure()`, passing as argument the system's resolution (in dots per inch) for best results.
- Check out `plt.scatter()`:
  - we want the points to be printed out with a size of 1 point^2, with no edges, and use the `plt.cm.Blues` colormap, the first points being the lightest, the last points being the darkest, which we obtain by letting the colour of the $(i+1)^{\text{st}}$ point be determined by i itself;
  - we want to print out the first point in green, the last point in red, with a size of 10 point^2, with no edges.
- We do not want to display axis lines and labels—check out `plt.axis()`.

To display the figure, check out `plt.show()`. Here is one possible such picture: