Practice 8

COMP9021, Term 3, 2019

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

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.

The face values and associated quantities should be input as a dictionary. You might find the literal_eval() function from the ast module to be useful.

A solution is output from smallest face value to largest face value. If a solution is represented as a list of pairs of the form (banknote face value, number of banknotes) ordered from smallest to largest face value, then the solutions themselves are output in lexicographical order (for sequences of pairs). All face values for a given solution are right aligned.

Here are examples of interactions:

```
$ python3 general_change.py
Input a dictionary whose keys represent banknote face values
with as value for a given key the number of banknotes
that are available for the corresponding face value:
    {2: 100, 50: 100}
Input the desired amount: 99

There is no solution.
$ python3 general_change.py
Input a dictionary whose keys represent banknote face values
with as value for a given key the number of banknotes
that are available for the corresponding face value:
    {1: 30, 20: 30, 50: 30}
Input the desired amount: 60

There is a unique solution:
$20: 3
$ python3 general_change.py
Input a dictionary whose keys represent banknote face values
with as value for a given key the number of banknotes
that are available for the corresponding face value:
    {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
$ python3 general_change.py
Input a dictionary whose keys represent banknote face values
with as value for a given key the number of banknotes
that are available for the corresponding face value:
    {1: 7, 2: 5, 3: 4, 4: 3, 5: 2}
Input the desired amount: 29

There are 4 solutions:
$1: 1
$3: 2
$4: 3
$5: 2

$2: 1
$3: 3
$4: 2
$5: 2

$2: 2
$3: 1
$4: 3
$5: 2

$3: 4
$4: 3
$5: 1
$ python3 general_change.py
Input a dictionary whose keys represent banknote face values
with as value for a given key the number of banknotes
that are available for the corresponding face value:
    {11:34, 12:34, 13: 234, 17:44, 18:54, 19: 3}
Input the desired amount: 3422

There are 8 solutions:
$11: 1
$12: 4
$13: 122
$17: 44
$18: 54
$19: 3

$11: 1
$13: 127
```

```
$17: 43
$18: 54
$19: 3

$11: 2
$12: 2
$13: 123
$17: 44
$18: 54
$19: 3

$11: 3
$13: 124
$17: 44
$18: 54
$19: 3

$12: 1
$13: 127
$17: 44
$18: 53
$19: 3

$12: 2
$13: 126
$17: 43
$18: 54
$19: 3

$12: 6
$13: 121
$17: 44
$18: 54
$19: 3

$13: 128
$17: 44
$18: 54
$19: 2
```

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

3 Fibonacci codes

Recall that the Fibonacci sequence $(F_n)_{n>=0}$ is defined by the equations: $F_0 = 0$, $F_1 = 1$ and for all $n > 0$, $F_n = F_{n+1} + F_{n-2}$

$$F_0 = 0 \quad F_1 = 1 \quad F_2 = 1 \quad F_3 = 2 \quad F_4 = 3 \quad F_5 = 5 \quad F_6 = 8 \quad F_7 = 13 \quad F_8 = 21 \quad \dots$$

It can be shown that every strictly positive integer N can be uniquely coded as a string σ of 0's and 1's ending with 1, so of the form $b_2 b_3 \dots b_k$ with $k \geq 2$ and $b_k = 1$, such that N is the sum of all F_i's, $2 \leq i \leq k$, with $b_i = 1$. For instance, $11 = 3 + 8 = F_4 + F_6$, hence 11 is coded by 00101.

Moreover:

- there are no two successive occurrences of 1 in σ;
- F_k is the largest Fibonacci number that fits in N, and if j is the largest integer in $\{2, \dots, k-1\}$ such that $b_j = 1$ then F_j is the largest Fibonacci number that fits in $N - F_k$, and if i is the largest integer in $\{2, \dots, j-1\}$ such that $b_i = 1$ then F_i is the largest Fibonacci number that fits in $N - F_k - F_j \dots$

Also, every string of 0's and 1's ending in 1 and having no two successive occurrences of 1's is a code of a strictly positive integer according to this coding scheme. For instance:

- There is only one string of 0's and 1's of length 1 ending in 1 and having no two successive occurrences of 1's; it is 1, and it codes 1.
- There is only one string of 0's and 1's of length 2 ending in 1 and having no two successive occurrences of 1's; it is 01, and it codes 2.
- The strings of 0's and 1's of length 3 ending in 1 and having no two successive occurrences of 1's are 001 and 101 and they code 3 and 4, respectively.
- The strings of 0's and 1's of length 4 ending in 1 and having no two successive occurrences of 1's are 0001, 1001 and 0101 and they code 5, 6 and 7, respectively.
- The strings of 0's and 1's of length 5 ending in 1 and having no two successive occurrences of 1's are 00001, 10001, 01001, 00101 and 10101 and they code 8, 9, 10, 11 and 12, respectively.
- ...

The *Fibonacci code* of N adds 1 at the end of σ; the resulting string then ends in two 1's, therefore marking the end of the code, and allowing one to let one string code a finite sequence of strictly positive integers. For instance, 00101100111011 codes $(11, 3, 4)$.

Write a program with two function, one that takes one argument N mean to to be a strictly positive integer and returns its Fibonacci code, and one that takes one argument σ meant to be a strict consisting 0's and 1's, returns 0 if σ cannot be a Fibonacci code, and otherwise returns the integer σ is the Fibonacci code of.

Here is a possible interaction:

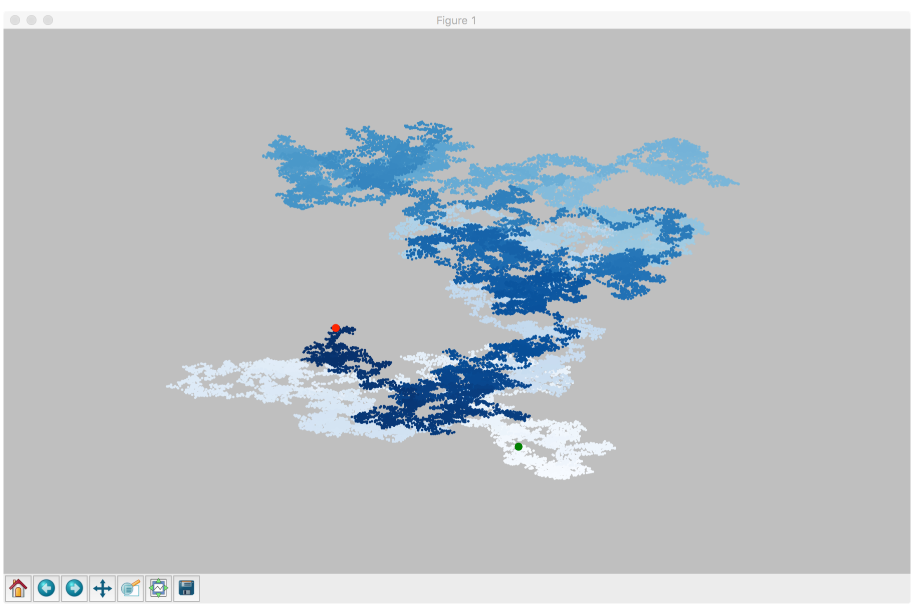
```
$ python3
...
>>> from fibonacci_codes import *
>>> encode(1)
'11'
>>> encode(2)
'011'
>>> encode(3)
'0011'
>>> encode(4)
'1011'
>>> encode(8)
'000011'
>>> encode(11)
'001011'
>>> encode(12)
'101011'
>>> encode(14)
'1000011'
>>> decode('1')
0
>>> decode('01')
0
>>> decode('100011011')
0
>>> decode('11')
1
>>> decode('011')
2
>>> decode('0011')
3
>>> decode('1011')
4
>>> decode('000011')
8
>>> decode('001011')
11
>>> decode('1000011')
14
```

4 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:



5 Markov chains (optional)

Write a program markov_chain.py that prompts the user to input two positive integers n and N, and outputs N words generated by a Markov chain where a dictionary file, named dictionary.txt, stored in the working directory, determines the probability that an n-gram (that is, a sequence of n letters) be followed by this or that character (including the "end-of-word" character). More precisely, assume that $n = 3$. Then a word $c_1 \dots c_k$ is generated as follows.

- c_1 is generated following the probability that, according to dictionary.txt, a word starts with c_1.
- c_2 is generated following the probability that, according to dictionary.txt, a word that starts with c_1 starts with c_1c_2; in case c_2 is the end of word marker then $k = 1$.
- c_3 is generated following the probability that, according to dictionary.txt, a word that starts with c_1c_2 starts with $c_1c_2c_3$; in case c_3 is the end of word marker then $k = 2$.
- c_4 is generated following the probability that, according to dictionary.txt, a word that contains $c_1c_2c_3$ contains $c_1c_2c_3c_4$; in case c_4 is the end of word marker then $k = 3$.
- c_5 is generated following the probability that, according to dictionary.txt, a word that contains $c_2c_3c_4$ contains $c_2c_3c_4c_5$; in case c_5 is the end of word marker then $k = 4$.
- c_6 is generated following the probability that, according to dictionary.txt, a word that contains $c_3c_4c_5$ contains $c_3c_4c_5c_6$; in case c_6 is the end of word marker then $k = 5$.
- ...

The program should indicate whether the word that has been generated has been invented (because it does not occur in dictionary.txt), or whether it has been rediscovered (because it does occur in dictionary.txt). Here is a possible interaction.

```
$ python3 markov_chains_for_word_generation.py
What n to use to let an n-gram determine the next character? 2
How many words do you want to generate? 10
Rediscovered ADS
Invented ENTRAMER
Invented LER
Invented EQUILIZED
Invented CIATTLY
Invented GRECOND
Rediscovered ASS
Invented WINCOT
Invented PEENIAR
Rediscovered ANTS
```

```
$ python3 markov_chains_for_word_generation.py
What n to use to let an n-gram determine the next character? 3
```

```
How many words do you want to generate? 10
Invented ROYAN
Rediscovered THING
Invented AGGREEABLE
Rediscovered RECEPTION
Invented LISHED
Invented CONTERMING
Invented TUSCUSTIVE
Invented INISM
Invented SWORTHUST
Invented BENTHANGE
$ python3 markov_chains_for_word_generation.py
What n to use to let an n-gram determine the next character? 4
How many words do you want to generate? 10
Invented REFORMEDITOR
Invented DIFFICE
Invented SEMITTERING
Invented INAPPERS
Invented PROPOLDVILLED
Invented KINGBIRDIED
Rediscovered SUBSCRIBED
Invented SCHED
Invented DEGRADIC
Rediscovered MILLION
$ python3 markov_chains_for_word_generation.py
What n to use to let an n-gram determine the next character? 5
How many words do you want to generate? 10
Rediscovered APPEARS
Rediscovered LOWS
Rediscovered SPORTS
Invented CROWDERPUFF
Invented BIRTHRIGHTNESS
Invented BREAKFASTERFUL
Rediscovered DREAMY
Rediscovered JACOB
Rediscovered BRUNHILDE
Invented REORGANISM
```