Practice 7

COMP9021, Term 3, 2019

1 Unit fractions

Let N and D be two strictly positive integers with $N < D$. The fraction N/D can be written as a sum of unit fractions, that is, there exists integers $k, d_1, \ldots, d_k \geq 1$ with $d_1 < d_2 < \ldots < d_k$ such that

$$\frac{N}{D} = \frac{1}{d_1} + \frac{1}{d_2} + \cdots + \frac{1}{d_k}.$$

There are actually infinitely many such representations. Indeed, since

$$1 = \frac{1}{2} + \frac{1}{3} + \frac{1}{6}$$

if $\frac{N}{D} = \frac{1}{d_1} + \frac{1}{d_2} + \cdots + \frac{1}{d_k}$ then also

$$\frac{N}{D} = \frac{1}{d_1} + \frac{1}{d_2} + \cdots + \frac{1}{d_{k-1}} + \frac{1}{2d_k} + \frac{1}{3d_k} + \frac{1}{6d_k}.$$

One particular representation is obtained by a method proposed by Fibonacci, in the form of a greedy algorithm. Suppose that N/D cannot be simplified, that is, N and D have no other common factor but 1. If $N = 1$ then we are done, so suppose otherwise. Let d_1 be the smallest integer such that $\frac{N}{D}$ can be written as $\frac{1}{d_1} + f_1$, with f_1 necessarily strictly positive by assumption. Looking for the smallest d_1 is what makes the algorithm greedy. Of course, d_1 is equal to $D \div N + 1$. By the choice of d_1, $\frac{1}{d_1 - 1} > \frac{N}{D}$, hence $D > N(d_1 - 1)$, hence $N > Nd_1 - D$. Since f_1 is equal to $\frac{N}{D} - \frac{1}{d_1} = \frac{Nd_1 - D}{Dd_1}$, it follows that $\frac{N}{D}$ can be written as $\frac{1}{d_1} + \frac{N_1}{D_1}$ with $N_1 < N$. If $N_1 > 1$ then the same argument allows one to greedily find $d_2 > d_1$ such that for some strictly positive integers N_2 and D_2, $\frac{N}{D}$ can be written as $\frac{1}{d_1} + \frac{1}{d_2} + \frac{N_2}{D_2}$ with $N_2 < N_1$, and if $N_2 > 1$ then the same argument allows one to greedily find $d_3 > d_2$ such that for some strictly positive integers N_3 and D_3, $\frac{N}{D}$ can be written as $\frac{1}{d_1} + \frac{1}{d_2} + \frac{1}{d_3} + \frac{N_3}{D_3}$ with $N_3 < N_2$...After a finite number of steps, we are done.

The number of summands in the sum of unit fractions given by Fibonacci's method is not always minimal: it is sometimes possible to decompose $\frac{N}{D}$ as sum of unit fractions with fewer summands. For instance, Fibonacci's method yields

$$\frac{4}{17} = \frac{1}{5} + \frac{1}{29} + \frac{1}{1233} + \frac{1}{3039345}$$

whereas $\frac{4}{17}$ can be written as a sum of 3 unit fractions, actually in 4 possible ways:

$$\frac{4}{17} = \frac{1}{5} + \frac{1}{30} + \frac{1}{510}$$
$$\frac{4}{17} = \frac{1}{5} + \frac{1}{34} + \frac{1}{170}$$
$$\frac{4}{17} = \frac{1}{6} + \frac{1}{15} + \frac{1}{510}$$
$$\frac{4}{17} = \frac{1}{6} + \frac{1}{17} + \frac{1}{102}$$

Write a program `unit_fractions.py` that implements two functions, `fibonacci_decomposition()` and `shortest_length_decompositions()`, that both take two strictly positive integers N and D as arguments, and writes N/D as, respectively:

- a sum of unit fractions following Fibonacci method, plus an integer in case $N \geq D$ (in a unique way);
- a sum of unit fractions with a minimal number of summands, plus an integer in case $N \geq D$ (in possibly many ways).

Here is a possible interaction:

```
>>> from unit_fractions import *
>>> fibonacci_decomposition(1, 521)
1/521 = 1/521
>>> fibonacci_decomposition(521, 521)
521/521 = 1
>>> fibonacci_decomposition(521, 1050)
521/1050 = 1/3 + 1/7 + 1/50
>>> fibonacci_decomposition(1050, 521)
1050/521 = 2 + 1/66 + 1/4913 + 1/33787684 + 1/2854018941421956
>>> fibonacci_decomposition(6, 7)
6/7 = 1/2 + 1/3 + 1/42
>>> shortest_length_decompositions(6, 7)
6/7 = 1/2 + 1/3 + 1/42
>>> fibonacci_decomposition(8, 11)
8/11 = 1/2 + 1/5 + 1/37 + 1/4070
>>> shortest_length_decompositions(8, 11)
8/11 = 1/2 + 1/5 + 1/37 + 1/4070
8/11 = 1/2 + 1/5 + 1/38 + 1/1045
8/11 = 1/2 + 1/5 + 1/40 + 1/440
8/11 = 1/2 + 1/5 + 1/44 + 1/220
8/11 = 1/2 + 1/5 + 1/45 + 1/198
8/11 = 1/2 + 1/5 + 1/55 + 1/110
8/11 = 1/2 + 1/5 + 1/70 + 1/77
8/11 = 1/2 + 1/6 + 1/17 + 1/561
8/11 = 1/2 + 1/6 + 1/18 + 1/198
8/11 = 1/2 + 1/6 + 1/21 + 1/77
8/11 = 1/2 + 1/6 + 1/22 + 1/66
8/11 = 1/2 + 1/7 + 1/12 + 1/924
8/11 = 1/2 + 1/7 + 1/14 + 1/77
8/11 = 1/2 + 1/8 + 1/10 + 1/440
8/11 = 1/2 + 1/8 + 1/11 + 1/88
8/11 = 1/3 + 1/4 + 1/7 + 1/924
>>> fibonacci_decomposition(4, 17)
4/17 = 1/5 + 1/29 + 1/1233 + 1/3039345
>>> shortest_length_decompositions(4, 17)
```

```
4/17 = 1/5 + 1/30 + 1/510
4/17 = 1/5 + 1/34 + 1/170
4/17 = 1/6 + 1/15 + 1/510
4/17 = 1/6 + 1/17 + 1/102
```

2 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
```

3 Diophantine equations

We consider Diophantine equations of the form $ax + by = c$ with a and b both not equal to 0. We will represent such an equation as a string of the form `ax+by=c` or `ax-by=c` where `a` and `c` are nonzero integer literals (not preceded by `+` in case they are positive) and where `b` is a strictly positive integer literal (not preceded by `+`), possibly with spaces anywhere at the beginning, at the end, and around the `+`, `-` and `=` characters. The equation $ax + by = c$ has a solution iff c is a multiple of $\gcd(a, b)$. In case c is indeed a multiple of $\gcd(a, b)$, then $ax + by = c$ has has infinitely many solutions, namely, all pairs (x, y) of the form

$$\left(x_0 + \frac{\operatorname{lcm}(a,b)}{a} n, y_0 - \frac{\operatorname{lcm}(a,b)}{b} n\right) \tag{1}$$

for arbitrary integers n, where $\operatorname{lcm}(a, b)$ denotes the least common multiplier of a and b, and where (x_0, y_0) is a solution to the equation. That particular solution can be derived from the extended Euclidian algorithm, that yields not only $\gcd(a, b)$, but also a pair of Bézout coefficients, namely, two integers x and y with $ax + by = \gcd(a, b)$. To normalise the representation of the solutions, we rewrite (**??**) as

$$\left(x_0 + \frac{\operatorname{lcm}(a,b)}{|a|} n, y_0 - \operatorname{sign}(a)\frac{\operatorname{lcm}(a,b)}{b} n\right) \tag{2}$$

where $\operatorname{sign}(a)$ is 1 if a is positive and -1 if a is negative, and we impose that the pair (x_0, y_0) is such that x_0 is nonnegative and minimal.

Write a Python program `diophantine.py` that defines a function `diophantine()` that prints out whether the equation provided as argument has a solution, and in case it does, prints out the normalised representation of its solutions. The output reproduces the equation nicely formatted, that is, with a single space around the `+`, `-` and `=` characters. As for the representation of the solutions, it is also nicely formatted, omitting x_0 or y_0 when they are equal to 0, and omitting 1 as a factor of n. Using the `doctest` module to test `diophantine()`, the following behaviour would then be observed:

```
>>> diophantine('1x + 1y = 0')
1x + 1y = 0 has as solutions all pairs of the form
    (n, -n) with n an arbitrary integer.
>>> diophantine('-1x + 1y = 0')
-1x + 1y = 0 has as solutions all pairs of the form
    (n, n) with n an arbitrary integer.
>>> diophantine('1x - 1y = 0')
1x - 1y = 0 has as solutions all pairs of the form
    (n, n) with n an arbitrary integer.
>>> diophantine('-1x - 1y = 0')
-1x - 1y = 0 has as solutions all pairs of the form
    (n, -n) with n an arbitrary integer.
>>> diophantine('1x + 1y = -1')
1x + 1y = -1 has as solutions all pairs of the form
    (n, -1 - n) with n an arbitrary integer.
>>> diophantine('-1x + 1y = 1')
```

```
-1x + 1y = 1 has as solutions all pairs of the form
    (n, 1 + n) with n an arbitrary integer.
>>> diophantine('4x + 6y = 9')
4x + 6y = 9 has no solution.
>>> diophantine('4x + 6y = 10')
4x + 6y = 10 has as solutions all pairs of the form
    (1 + 3n, 1 - 2n) with n an arbitrary integer.
>>> diophantine('71x+83y=2')
71x + 83y = 2 has as solutions all pairs of the form
    (69 + 83n, -59 - 71n) with n an arbitrary integer.
>>> diophantine('  782  x  +  253  y  =  92')
782x + 253y = 92 has as solutions all pairs of the form
    (4 + 11n, -12 - 34n) with n an arbitrary integer.
>>> diophantine('-123x -456y = 78')
-123x - 456y = 78 has as solutions all pairs of the form
    (118 + 152n, -32 - 41n) with n an arbitrary integer.
>>> diophantine('-321x +654y = -87')
-321x + 654y = -87 has as solutions all pairs of the form
    (149 + 218n, 73 + 107n) with n an arbitrary integer.
```

4 The Gale Shapley algorithm (optional)

Read the AMS Feature column on the stable marriage problem and the Gale Shapley algorithm. It is explained for the same number of men and women, but it is immediately generalised to arbitrary numbers of men and women. If there are more men than women, then no woman should prefer to her own partner any of the men left without partner.

Write a program, `gale_shapley.py`, that defines a class, `GaleShapley`, with the following properties.

- When creating a `GaleShapley` object, the user is prompted to provide the preferences of n men for m women (for arbitrary positive integers n and m), and the other way around, in the form of n and m lists of permutations of $\{1,\ldots,m\}$ and $\{1,\ldots,n\}$, also represented as lists, respectively (with 1 preferred for both, the least preferred woman being referred to by m, and the least preferred man being referred to by n).
  - The user is prompted until input is as expected, with n and m determined from the first input list of lists.
  - The user is also offered to optionally input names for men and names for women; otherwise "man 1", …, "man n" and "woman 1", …, "woman m" will be used in the output.
- The class `GaleShapley` has a method, `determine_stable_matching()`, with a default argument, `men_choosing`, set to `True` by default. The method applies the Gale Shapley algorithm and outputs the matches, with proposers listed first, in lexicographic order. When `men_choosing` is set to `False`, the algorithm is applied by reversing the role of men and women, so with women proposing to men.

You might find the `literal_eval()` function from the `ast` module useful. Here is a possible interaction (the heart is the unicode character of code point `0x2661`):

```
$ python3
...
>>> from gale_shapley import *
>>> GS = GaleShapley()
Enter a list of n lists, each of which is a permutation  of 1, ..., m,
to express the preferences of n men for m women:
    [[1, 2, 3, 4, 1, 4, 3, 2], [2, 1, 3, 4], [4, 2, 3, 1]]
Your input is incorrect.
Enter a list of n lists, each of which is a permutation  of 1, ..., m,
to express the preferences of n men for m women:
    [[1, 2, 3, 4], [1, 0, 3, 2], [2, 1, 3, 4], [4, 2, 3, 1]]
Your input is incorrect.
Enter a list of n lists, each of which is a permutation  of 1, ..., m,
to express the preferences of n men for m women:
    [[1, 2, 3, 4], [1, 4, 3, 2], [2, 1, 3, 4], [4, 2, 2, 1]]
Your input is incorrect.
Enter a list of n lists, each of which is a permutation  of 1, ..., m,
```

```
to express the preferences of n men for m women:
    [[1, 2, 3, 4], [1, 4, 3, 2], [2, 1, 3, 4], [4, 2, 3, 1]]
Enter a list of 4 lists, each of which is a permutation of 1, ..., 4,
to express the preferences of the women for the men:
    [[2, 1, 3], [1, 2, 3], [3, 2, 1]]
Your input is incorrect.
Enter a list of 4 lists, each of which is a permutation of 1, ..., 4,
to express the preferences of the women for the men:
    [[4, 3, 1, 2], [2, 4, 1, 3], [4, 1, 2, 3], [3, 2, 1, 4]]
Optionally input 4 names for the men.
In case you do not input 4 distinct strings,
then the men will referred to as "man 1", ..., "man 4":

Optionally input 4 names for the women.
In case you do not input 4 distinct strings,
then the women will referred to as "woman 1", ..., "woman 4":

>>> GS.determine_stable_matching()
The matches are:
man 1 ♡ woman 3
man 2 ♡ woman 4
man 3 ♡ woman 1
man 4 ♡ woman 2
>>>  GS.determine_stable_matching(False)
The matches are:
woman 1 ♡ man 3
woman 2 ♡ man 4
woman 3 ♡ man 1
woman 4 ♡ man 2
>>>  GS = GaleShapley()
Enter a list of n lists, each of which is a permutation  of 1, ..., m,
to express the preferences of n men for m women:
     [[3, 1, 4, 2], [2, 1, 4, 3], [2, 4, 1, 3], [3, 1, 2, 4]]
Enter a list of 4 lists, each of which is a permutation of 1, ..., 4,
to express the preferences of the women for the men:
     [[2, 3, 4, 1], [4, 2, 3, 1], [1, 4, 3, 2], [4, 1, 2, 3]]
Optionally input 4 names for the men.
In case you do not input 4 distinct strings,
then the men will referred to as "man 1", ..., "man 4":
     Paul John Peter Jack
Optionally input 4 names for the women.
In case you do not input 4 distinct strings,
then the women will referred to as "woman 1", ..., "woman 4":
     Lisa Gina Laura Andrea
>>>  GS.determine_stable_matching()
The matches are:
Jack ♡ Laura
John ♡ Gina
```

```
Paul ♡ Andrea
Peter ♡ Lisa
>>>  GS.determine_stable_matching(False)
The matches are:
Andrea ♡ John
Gina ♡ Paul
Laura ♡ Jack
Lisa ♡ Peter
>>> GS = GaleShapley()
Enter a list of n lists, each of which is a permutation  of 1, ..., m,
to express the preferences of n men for m women:
    [[1, 2, 3], [2, 3, 1], [1, 3, 2]]
Enter a list of 3 lists, each of which is a permutation of 1, ..., 3,
to express the preferences of the women for the men:
    [[2, 3, 1], [2, 3, 1], [1, 2, 3]]
Optionally input 3 names for the men.
In case you do not input 3 distinct strings,
then the men will referred to as "man 1", ..., "man 3":

Optionally input 3 names for the women.
In case you do not input 3 distinct strings,
then the women will referred to as "woman 1", ..., "woman 3":

>>> GS.determine_stable_matching()
The matches are:
man 1 ♡ woman 3
man 2 ♡ woman 2
man 3 ♡ woman 1
>>> GS.determine_stable_matching(False)
The matches are:
woman 1 ♡ man 3
woman 2 ♡ man 2
woman 3 ♡ man 1
>>> GS = GaleShapley()
Enter a list of n lists, each of which is a permutation  of 1, ..., m,
to express the preferences of n men for m women:
    [[2, 1], [2, 1], [1, 2]]
Enter a list of 2 lists, each of which is a permutation of 1, ..., 3,
to express the preferences of the women for the men:
    [[1, 2, 3], [2, 3, 1]]
Optionally input 3 names for the men.
In case you do not input 3 distinct strings,
then the men will referred to as "man 1", ..., "man 3":

Optionally input 2 names for the women.
In case you do not input 2 distinct strings,
then the women will referred to as "woman 1", ..., "woman 2":
```

```
>>> GS.determine_stable_matching()
The matches are:
man 1 ♡ woman 1
man 2 ♡ woman 2
>>> GS.determine_stable_matching(False)
The matches are:
woman 1 ♡ man 1
woman 2 ♡ man 2
```

5 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.py` uses 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 Notes 18, 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
```