Practice 6

COMP9021, Term 3, 2019

1 Obtaining a sum from a subsequence of digits

Write a program `sum_of_digits.py` that prompts the user for two numbers, say `available_digits` and `desired_sum`, and outputs the number of ways of selecting digits from `available_digits` that sum up to `desired_sum`. For instance, if `available_digits` is `12234` and `sum` is `5` then there are 4 solutions:

- one solution is obtained by selecting 1 and both occurrences of 2 ($1 + 2 + 2 = 5$);
- one solution is obtained by selecting 1 and 4 ($1 + 4 = 5$);
- one solution is obtained by selecting the first occurrence of 2 and 3 ($2 + 3 = 5$);
- one solution is obtained by selecting the second occurrence of 2 and 3 ($2 + 3 = 5$).

Here is a possible interaction:

```
$ python3 sum_of_digits.py
Input a number that we will use as available digits: 12234
Input a number that represents the desired sum: 5
There are 4 solutions.
$ python3 sum_of_digits.py
Input a number that we will use as available digits: 11111
Input a number that represents the desired sum: 5
There is a unique solution.
$ python3 sum_of_digits.py
Input a number that we will use as available digits: 11111
Input a number that represents the desired sum: 6
There is no solution.
$ python3 sum_of_digits.py
Input a number that we will use as available digits: 1234321
Input a number that represents the desired sum: 5
There are 10 solutions.
```

2 Merging two strings into a third one

Say that two strings s_1 and s_2 can be merged into a third string s_3 if s_3 is obtained from s_1 by inserting arbitrarily in s_1 the characters in s_2, respecting their order. For instance, the two strings *ab* and *cd* can be merged into *abcd*, or *cabd*, or *cdab*, or *acbd*, or *acdb*, ..., but not into *adbc* nor into *cbda*. Write a program `merging_strings.py` that prompts the user for 3 strings and displays the output as follows:

- If no string can be obtained from the other two by merging, then the program outputs that there is no solution.
- Otherwise, the program outputs which of the strings can be obtained from the other two by merging.

Here is a possible interaction:

```
$ python3 merging_strings.py
Please input the first string: ab
Please input the second string: cd
Please input the third string: abcd
The third string can be obtained by merging the other two.
$ python3 merging_strings.py
Please input the first string: ab
Please input the second string: cdab
Please input the third string: cd
The second string can be obtained by merging the other two.
$ python3 merging_strings.py
Please input the first string: abcd
Please input the second string: cd
Please input the third string: ab
The first string can be obtained by merging the other two.
$ python3 merging_strings.py
Please input the first string: ab
Please input the second string: cd
Please input the third string: adcb
No solution
$ python3 merging_strings.py
Please input the first string: aaaaa
Please input the second string: a
Please input the third string: aaaa
The first string can be obtained by merging the other two.
$ python3 merging_strings.py
Please input the first string: aaab
Please input the second string: abcab
Please input the third string: aaabcaabb
The third string can be obtained by merging the other two.
$ python3 merging_strings.py
Please input the first string: ??got
Please input the second string: ?it?go#t##
Please input the third string: it###
The second string can be obtained by merging the other two.
```

3 The 9 puzzle (finding a solution is optional)

Dispatch the integers from `0` to `8`, with 0 possibly changed to `None`, as a list of 3 lists of size 3, to represent a 9 puzzle. For instance, let

```
[[4, 0, 8], [1, 3, 7], [5, 2, 6]]
```

or

```
[[4, None ,8], [1, 3, 7], [5, 2, 6]]
```

represent the 9 puzzle

4		8
1	3	7
5	2	6

with the 8 integers being printed on 8 tiles that are placed in a frame with one location being tile free. The aim is to slide tiles horizontally or vertically so as to eventually reach the configuration

1	2	3
4	5	6
7	8	

It can be shown that the puzzle is solvable iff the permutation of the integers 1, ..., 8, determined by reading those integers off the puzzle from top to bottom and from left to right, is even. This is clearly a necessary condition since:

- sliding a tile horizontally does not change the number of inversions;
- sliding a tile vertically changes the number of inversions by -2, 0 or 2;
- the parity of the identity is even.

Write a program `nine_puzzle.py` with two functions:

- `validate_9_puzzle(grid)` that prints out whether or not `grid` is a valid representation of a solvable 9 puzzle;

- `solve_9_puzzle(grid)` that, assuming that `grid` is a valid representation of a solvable 9 puzzle, outputs a solution to the puzzle, with a minimal number of moves.

Here is a possible interaction:

```
$ python3
Python 3.6.3 ...
>>> from nine_puzzle import *
>>> validate_9_puzzle([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
This is an invalid or unsolvable 9 puzzle
>>> validate_9_puzzle([[1, 2, 3], [4, 1, 6], [7, 8, 0]])
This is an invalid or unsolvable 9 puzzle
>>> validate_9_puzzle([[1, 2, 3], [4, 5, 6], [7, 8]])
This is an invalid or unsolvable 9 puzzle
>>> validate_9_puzzle([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
This is a valid 9 puzzle, and it is solvable
>>> solve_9_puzzle([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
Here is a minimal solution:

  1  2  3
  4  5  6
  7  8
>>> validate_9_puzzle([[1, 2, 3], [4, 5, 6], [None, 8, 7]])
This is an invalid or unsolvable 9 puzzle
>>> validate_9_puzzle([[1, 2, 3], [4, 5, 6], [None, 7, 8]])
This is a valid 9 puzzle, and it is solvable
>>> solve_9_puzzle([[1, 2, 3], [4, 5, 6], [None, 7, 8]])
Here is a minimal solution:

  1  2  3
  4  5  6
     7  8

  1  2  3
  4  5  6
  7     8

  1  2  3
  4  5  6
  7  8
>>> validate_9_puzzle([[4, None, 8], [3, 1, 7], [5, 2, 6]])
This is an invalid or unsolvable 9 puzzle
>>> validate_9_puzzle([[4, None, 8], [1, 3, 7], [5, 2, 6]])
This is a valid 9 puzzle, and it is solvable
```

```
>>> solve_9_puzzle([[4, None, 8], [1, 3, 7], [5, 2, 6]])
Here is a minimal solution:

  4     8
  1  3  7
  5  2  6

  4  3  8
  1     7
  5  2  6

  4  3  8
     1  7
  5  2  6

  4  3  8
  5  1  7
     2  6

  4  3  8
  5  1  7
  2     6

  4  3  8
  5  1  7
  2  6

  4  3  8
  5  1
  2  6  7

  4  3
  5  1  8
  2  6  7

  4     3
  5  1  8
  2  6  7

  4  1  3
  5     8
  2  6  7

  4  1  3
  5  6  8
  2     7

  4  1  3
  5  6  8
  2  7

  4  1  3
  5  6
  2  7  8
```

```
4  1  3
5     6
2  7  8

4  1  3
   5  6
2  7  8

4  1  3
2  5  6
   7  8

4  1  3
2  5  6
7     8

4  1  3
2     6
7  5  8

4  1  3
   2  6
7  5  8

   1  3
4  2  6
7  5  8

1     3
4  2  6
7  5  8

1  2  3
4     6
7  5  8

1  2  3
4  5  6
7     8

1  2  3
4  5  6
7  8
```

4 Magic squares (Bachet, Siamese, and Lux methods are optional)

Write a program `magic_squares.py` that implements five functions: `print_square(square)`, `is_magic_square(square)`, `bachet_magic_square(n)`, `siamese_magic_square(n)`, and, finally, `lux_magic_square(n)`.

Given a positive integer n, a magic square of order n is a matrix of size $n \times n$ that stores all numbers from 1 up to n^2 and such that the sum of the n rows, the sum of the n columns, and the sum of the two diagonals is constant, hence equal to $n(n^2+1)/2$. The function `print_square(square)` prints a list of lists that represents a square, and the function `is_magic_square(square)` checks whether a list of lists is a magic square. For instance:

```
>>> from magic_squares import *
>>> print_square([[2,7,6], [9,5,1], [4,3,8]])
2 7 6
9 5 1
4 3 8
>>> is_magic_square([[2,7,6], [9,5,1], [4,3,8]])
True
>>> print_square([[2,7,6], [1,5,9], [4,3,8]])
2 7 6
1 5 9
4 3 8
>>> is_magic_square([[2,7,6], [1,5,9], [4,3,8]])
False
```

Given an odd positive integer n, the Bachet method produces a magic square of order n. Taking $n = 7$ as an example, this method

- starts with a square of size $2n-1 \times 2n-1$ filled as follows:

```
 .  .  .  .  .  .  1  .  .  .  .  .  .
 .  .  .  .  .  8  .  2  .  .  .  .  .
 .  .  .  . 15  .  9  .  3  .  .  .  .
 .  .  . 22  . 16  . 10  .  4  .  .  .
 .  . 29  . 23  . 17  . 11  .  5  .  .
 . 36  . 30  . 24  . 18  . 12  .  6  .
43  . 37  . 31  . 25  . 19  . 13  .  7
 . 44  . 38  . 32  . 26  . 20  . 14  .
 .  . 45  . 39  . 33  . 27  . 21  .  .
 .  .  . 46  . 40  . 34  . 28  .  .  .
 .  .  .  . 47  . 41  . 35  .  .  .  .
 .  .  .  .  . 48  . 42  .  .  .  .  .
 .  .  .  .  .  . 49  .  .  .  .  .  .
```

- then 4 times:
  - shift the n // 2 top rows n rows below:

```
 .  .  .  .  .  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  .  .  .  .  .
 .  .  . 22  . 16  . 10  .  4  .  .  .
 .  . 29  . 23  . 17  . 11  .  5  .  .
 . 36  . 30  . 24  . 18  . 12  .  6  .
43  . 37  . 31  . 25  . 19  . 13  .  7
 . 44  . 38  . 32  1 26  . 20  . 14  .
 .  . 45  . 39  8 33  2 27  . 21  .  .
 .  .  . 46 15 40  9 34  3 28  .  .  .
 .  .  .  . 47  . 41  . 35  .  .  .  .
 .  .  .  .  . 48  . 42  .  .  .  .  .
 .  .  .  .  .  . 49  .  .  .  .  .  .
```

  - rotates clockwise by 90 degrees:

```
 .  .  .  .  .  .  . 43  .  .  .  .  .  .
 .  .  .  .  .  . 44  . 36  .  .  .  .  .
 .  .  .  .  . 45  . 37  . 29  .  .  .  .
 .  .  .  . 46  . 38  . 30  . 22  .  .  .
 .  .  . 47 15 39  . 31  . 23  .  .  .  .
 . 48  . 40  8 32  . 24  . 16  .  .  .  .
49  . 41  9 33  1 25  . 17  .  .  .  .  .
 . 42  . 34  2 26  . 18  . 10  .  .  .  .
 .  . 35  3 27  . 19  . 11  .  .  .  .  .
 .  .  . 28  . 20  . 12  .  4  .  .  .  .
 .  .  .  . 21  . 13  .  5  .  .  .  .  .
 .  .  .  .  . 14  .  6  .  .  .  .  .  .
 .  .  .  .  .  .  7  .  .  .  .  .  .  .
```

  Eventually, one reads the magic square off the centre:

```
 .  .  .  .  .  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  .  .  .  .  .
 .  .  . 22 47 16 41 10 35  4  .  .  .
 .  .  .  5 23 48 17 42 11 29  .  .  .
 .  .  . 30  6 24 49 18 36 12  .  .  .
 .  .  . 13 31  7 25 43 19 37  .  .  .
 .  .  . 38 14 32  1 26 44 20  .  .  .
 .  .  . 21 39  8 33  2 27 45  .  .  .
 .  .  . 46 15 40  9 34  3 28  .  .  .
 .  .  .  .  .  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  .  .  .  .  .
```

For instance:

```
>>> print_square(bachet_magic_square(7))
22 47 16 41 10 35  4
 5 23 48 17 42 11 29
30  6 24 49 18 36 12
13 31  7 25 43 19 37
38 14 32  1 26 44 20
21 39  8 33  2 27 45
46 15 40  9 34  3 28
```

Given an odd positive integer n, the Siamese method produces a magic square of order n. This method starts with 1 put at the centre of the first row, and having placed number $k < n^2$, places number $k + 1$ by moving diagonally up and right by one cell, wrapping around when needed (as if a torus was made out of the square), unless that cell is already occupied, in which case $k + 1$ is placed below the cell where k is (with no need to wrap around). For instance:

```
>>> print_square(siamese_magic_square(7))
30 39 48  1 10 19 28
38 47  7  9 18 27 29
46  6  8 17 26 35 37
 5 14 16 25 34 36 45
13 15 24 33 42 44  4
21 23 32 41 43  3 12
22 31 40 49  2 11 20
```

Given a positive integer n of the form $4 * k + 2$ with k a strictly positive integer, the LUX method produces a magic square of order n. This method proceeds as follows.

- Consider a matrix of size $2k + 1 \times 2k + 1$ that consists of:
  - $k + 1$ rows of Ls,
  - 1 row of Us, and
  - $k - 1$ rows of Xs,

  and then exchange the U in the middle with the L above it. For instance, when $n = 10$, that matrix is:

```
L  L  L  L  L
L  L  L  L  L
L  L  U  L  L
U  U  L  U  U
X  X  X  X  X
```

- Explore all cells of this matrix as for the Siamese method, that is, starting at the cell at the centre of the first row, and then by moving diagonally up and right by one cell, wrapping around when needed (as if a torus was made out of the matrix), unless that cell has been visited already, in which case one moves down one cell (with no need to wrap around). The contents of every visited cell is then replaced by
  - if the cell contains L,

    ```
    i+4 i+1
    i+2 i+3
    ```

  - if the cell contains U,

    ```
    i+1 i+4
    i+2 i+3
    ```

  - if the cell contains X with i being the last number that has been used (starting with $i = 0$),

    ```
    i+1 i+4
    i+3 i+2
    ```

For instance:

```
>>> print_square(lux_magic_square(10))
 68  65  96  93   4   1  32  29  60  57
 66  67  94  95   2   3  30  31  58  59
 92  89  20  17  28  25  56  53  64  61
 90  91  18  19  26  27  54  55  62  63
 16  13  24  21  49  52  80  77  88  85
 14  15  22  23  50  51  78  79  86  87
 37  40  45  48  76  73  81  84   9  12
 38  39  46  47  74  75  82  83  10  11
 41  44  69  72  97 100   5   8  33  36
 43  42  71  70  99  98   7   6  35  34
```

5 Sydney temperatures (optional)

Write a program `sydney_temperatures.py` that extracts from the file `IDCJCM0037_066062.csv`, stored in the working directory, the mean min and mean max temperatures for the 12 months of the year for the years 1859 to 2016, and plots them thanks to the `matplotlib.pyplot` module, which it is convenient to import as `plt`.

- The picture is 5 inches wide and 3.5 inches high—check out `plt.figure()`, passing as argument the system's resolution (in dots per inch) for best results. It has as title, with a font size of 10 points, `Mean min and max temperatures in Sydney`—check out `plt.title()`. The grid should be displayed—check out `plt.grid()`.
- Denoting by min_temp the largest integer smaller than or equal to the smallest mean min temperature and by max_temp the smallest integer greater than or equal to the largest mean max temperature, the plotting area is a rectangle whose lower left corner has coordinates $(0.5, \text{min\_temp} - 1)$ and whose upper right corner has coordinates $(12.5, \text{max\_temp} + 1)$—check out `plt.axis()`.
- The mean min and mean max temperatures for the i^{th} month, $i \in \{1, \ldots, 12\}$, are displayed at points of x-coordinate i, while the averages of the mean min and mean max temperatures for December and January are displayed both at points of x-coordinate 0.5 and at points of x-coordinate 12.5, on blue curves for the mean min temperatures, on red curves for the mean max temperatures—check out `plt.plot()`.
- The area between the curves for the mean min and mean max temperatures is coloured in grey with, to create an appearance of transparency, a value of 0.5 on the alpha channel—check out `plt.fill_between()`.
- The x-axis labels at coordinates 1 to 12 the names of the months of the year, conveniently retrieved from the `month_name` list of the `calendar` module, with a font size of 8 points, and displayed slanted—check out `plt.xticks()` and `autofmt_xdate()`.
- The y-axis labels the temperatures between min_temp and max_temp, in steps of 0.5, with a font size of 4 points—check out `plt.yticks()`.

To display the figure, check out `plt.show()`. Here is the expected output:

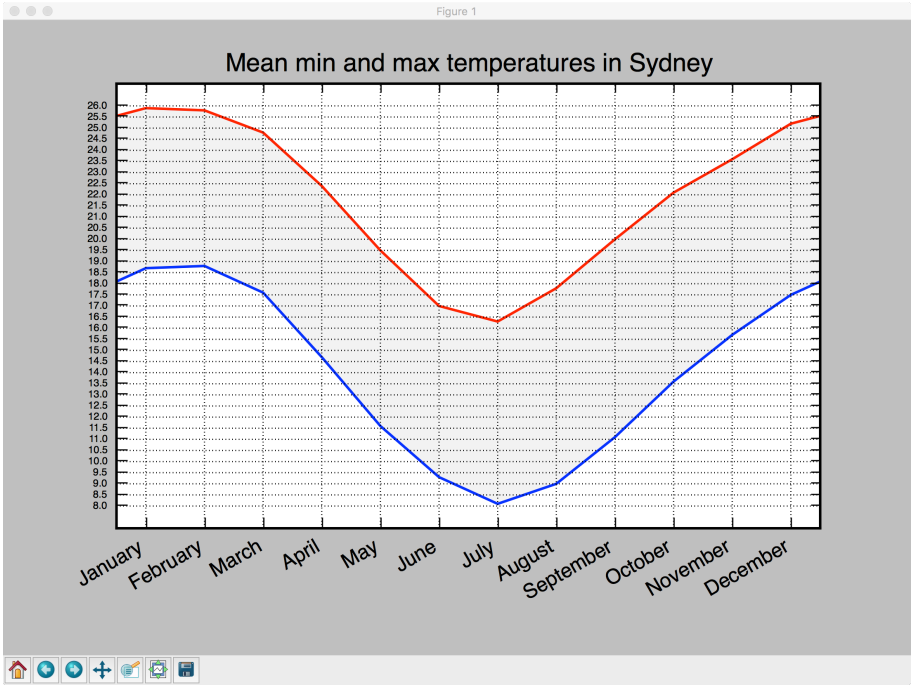
Figure 1
Mean min and max temperatures in Sydney
26.0
25.5
25.0
24.5
24.0
23.5
23.0
22.5
22.0
21.5
21.0
20.5
20.0
19.5
19.0
18.5
18.0
17.5
17.0
16.5
16.0
15.5
15.0
14.5
14.0
13.5
13.0
12.5
12.0
11.5
11.0
10.5
10.0
9.5
9.0
8.5
8.0
January
February
March
April
May
June
July
August
September
October
November
December