Assignment 1

COMP9021, Session 2, 2015

1. General matters

1.1. Aims. The purpose of the assignment is to:

- let you design solutions to simple problems;
- let you implement these solutions in the form of short python programs;
- practice the use of arithmetic computations, tests, repetitions, lists, tuples, dictionnaries.

1.2. Submission. Your programs will be stored in a number of files, with one file per exercise, of the appropriate name. After you have developed and tested your programs, upload your files using WebCMS. Assignments can be submitted more than once: the last version is marked. Your assignment is due by August 30, 11:59pm.

1.3. Assessment. Each of the 5 exercises is worth 2 marks. For all exercises, the automarking script will let each of your programs run for 30 seconds. Still you should not take advantage of this and strive for a solution that gives an immediate output for "reasonable" inputs (if any).

Late assignments will be penalised: the mark for a late submission will be the minimum of the awarded mark and 10 minus the number of full and partial days that have elapsed from the due date.

Your program should not make any assumption on the actual solution, except obvious ones on the ranges of some values. Hard coded solutions (that is, programs that more or less just print out the solution) will not receive any mark.

The outputs of your programs should be **exactly** as indicated.

1.4. Reminder on plagiarism policy. You are permitted, indeed encouraged, to discuss ways to solve the assignment with other people. Such discussions must be in terms of algorithms, not code. But you must implement the solution on your own. Submissions are routinely scanned for similarities that occur when students copy and modify other people's work, or work very closely together on a single implementation. Severe penalties apply.

2. An equation

Write a python program, named `equation.py`, that solves the equation

$$abc * de * fg = a * bc * defg$$

where each of the letters a to g stands for a **nonzero** digit, and where different letters possibly stand for the same digit. For instance, if we make $a = 2$, $b = 7$, $c = 5$, $d = 6$, $e = 6$, $f = 5$, and $g = 5$, then the equation reads

$$275 * 66 * 55 = 2 * 75 * 6655$$

which happens to be true, making this specific assignment of digits to the letters a to g a particular solution.

The output of your program should be of the form:

```
There are X solutions:
___ * __ * __ = _ * __ * ____
...
___ * __ * __ = _ * __ * ____
```

of course replacing `X` by the appropriate number and then displaying all solutions as shown, of course replacing every occurrence of `_` by an appropriate digit, one solution per line, in lexicographic order (so from the lowest values of abc to the largest values of abc, and for a given value of abc from the lowest values of de to the largest values of de, etc.).

3. A multiple of 2004

Write a python program, named `multiple_2004.py`, that finds two strictly positive integers A and B so that $A2004B$ (the digits in A followed by 2004 followed by the digits in B) is the **smallest** multiple of 2004 possible.

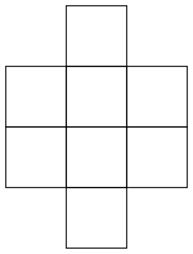
The output of your program should be of the form:

```
A = _, B = _, N = _ is the solution.
```

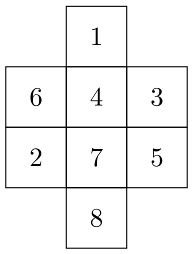
of course replacing each occurrence of `_` by the appropriate number.

4. A grid of numbers

Write a python program, named `adjacency_grid.py`, that arranges the numbers 1 to 8 in the grid below such that adjacent numbers are not in adjacent cells (neither horizontally nor vertically nor diagonally).



For instance,



is **not** a solution because of 3 being adjacent to 4, and 4 being adjacent to 5, and 6 being adjacent to 7, and 7 being adjacent to 8.

The output of your program should be of the form

```
    _
_   _   _
_   _   _
    _

    _
_   _   _
_   _   _
    _

...
```

of course replacing each occurrence of _ by the appropriate digit, consecutive solutions being separated by a blank line (but with no blank line before of after the last solution), in lexicographic order (reading the values in the grid from top to bottom and from left to right).

You might find it useful to make use of the following statement:

```
from itertools import permutations
```

5. A division

Write a python program, named `division.py`, that solves the division below where *a* as well as every star stands for a digit. Different stars can stand for the same digit. All occurrences of *a* stand for the same digit, and no star stands for that digit. Of course, no number starts with 0.

```
* * * * a * * | * a *
              |--------
* * a a       | * * a *
-------       |
  * * * a     |
    * * a     |
    -----     |
    * * * *   |
    * a * *   |
    -------   |
      * * * * |
      * * * * |
      ------- |
            0 |
```

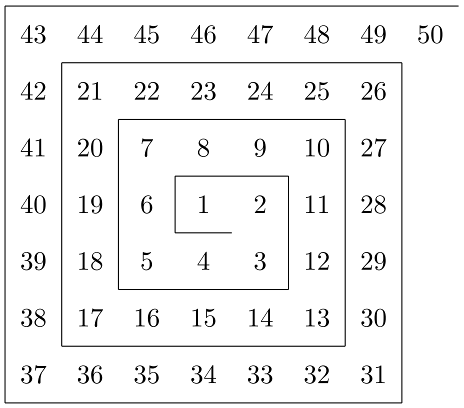
There is a actually a unique solution. The output of your program should be of the form

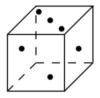
```
_______ / ___ = ____ is the solution.
```

of course replacing each occurrence of `_` by the appropriate digit.

6. A pivoting die

Consider the board below, which can be imagined as being infinite, so only a small part is represented. A die is placed on cell number 1 of the board in the position indicated: 3 at the top, 2 at the front (towards cell number 4 of the board), and 1 on the right (towards cell number 2 of the board). Recall that 4 is opposite to 3, 5 is opposite to 2, and 6 is opposite to 1. The die can be moved from cell 1 to cell 2, then to cell 3, then to cell 4..., each time pivoting by 90 degrees in the appropriate direction (to the right, forwards, to the left, or backwards). For instance, after it has been moved from cell 1 to cell 2, 2 is still at the front but 6 is at the top and 3 is on the right.





Write a python program, named `pivoting_die.py`, that prompts the user for a natural number N at least equal to 1, and outputs the numbers at the top, the front and the right after the die has been moved to cell N.

Here is a possible interaction.

```
$ python pivoting_die.py
Enter the desired goal cell number: A
Incorrect value, try again
Enter the desired goal cell number:
Incorrect value, try again
Enter the desired goal cell number: -1
Incorrect value, try again
Enter the desired goal cell number: 0
Incorrect value, try again
Enter the desired goal cell number: 1
On cell 1, 3 is at the top, 2 at the front, and 1 on the right.
$ python pivoting_die.py
Enter the desired goal cell number: 29
On cell 29, 3 is at the top, 2 at the front, and 1 on the right.
$ python pivoting_die.py
Enter the desired goal cell number: 2006
On cell 2006, 4 is at the top, 1 at the front, and 2 on the right.
```