

## Practice 4

COMP9021, Term 3, 2019

### 1 A triangle of characters

Write a program `characters_triangle.py` that gets a strictly positive integer  $N$  as input and outputs a triangle of height  $N$ , following this kind of interaction:

```
$ python3 characters_triangle.py
Enter strictly positive number: 13
      A
     BCB
    DEFED
   GHIJIHG
  KLMNONMLK
 PQRSTUTSRQP
VWXYZABAZYXWV
CDEFGHIJIHGFEDC
KLMNOPQRSRQPONMLK
TUVWXYZABCBAZYXWVUT
DEFGHIJKLMNMLKJIHGFED
OPQRSTUVWXYZYXWVUTSRQP
ABCDEFGHIJKLMLKJIHGFEDCBA
```

Two built-in functions are useful for this exercise:

- `ord()` returns the integer that encodes the character provided as argument;
- `chr()` returns the character encoded by the integer provided as argument.

For instance:

```
>>> ord('A')
65
>>> chr(65)
'A'
```

Consecutive uppercase letters are encoded by consecutive integers. For instance:

```
>>> ord('A'), ord('B'), ord('C')
(65, 66, 67)
```

## 2 Pascal triangle

Write a program `pascal_triangle.py` that prompts the user for a number  $N$  and prints out the first  $N + 1$  lines of Pascal triangle, making sure the numbers are nicely aligned, following this kind of interaction.

```
$ python3 pascal_triangle.py
```

```
Enter a nonnegative integer: 3
```

```
  1
 1 1
1 2 1
1 3 3 1
```

```
$ python3 pascal_triangle.py
```

```
Enter a nonnegative integer: 7
```

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
```

```
$ python3 pascal_triangle.py
```

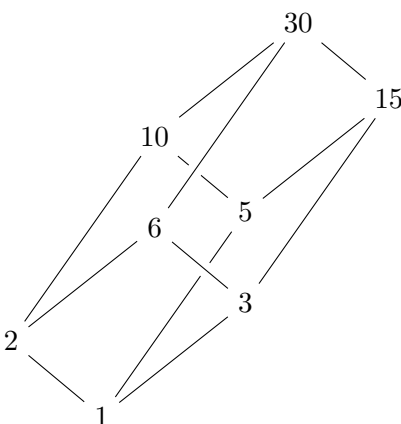
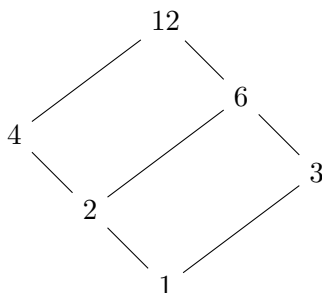
```
Enter a nonnegative integer: 11
```

```
          1
        1 1
       1 2 1
      1 3 3 1
     1 4 6 4 1
    1 5 10 10 5 1
   1 6 15 20 15 6 1
  1 7 21 35 35 21 7 1
 1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1
```

### 3 Hasse diagrams

Let a strictly positive integer  $n$  be given. Let  $D$  be the set of divisors of  $n$ . Let  $k$  be the number of prime divisors of  $n$  (the number of prime numbers in  $D$ ). The members of  $D$  can be arranged as the vertices of a solid in a  $k$ -dimensional space as illustrated below for  $n = 12$  (in which case  $D = \{1, 2, 3, 4, 6, 12\}$  and  $k = 2$ ) and for  $n = 30$  (in which case  $D = \{1, 2, 3, 5, 6, 10, 15, 30\}$  and  $k = 3$ ).

- Each of the solids' vertices is associated with two collections of nodes: those “directly below” it, and those “directly above” it. In particular, the prime divisors of  $n$  are “directly above” 1, and no vertex is below 1;  $n$  has exactly  $k$  vertices “directly below” it, and no vertex is above  $n$ . This suggests considering a dictionary whose keys are the members of  $D$  (inserted from smallest to largest), and as value for a given key  $d$ , the pair of ordered lists of members of  $D$  “directly below”  $d$  and “directly above”  $d$ , respectively.
- The solids exhibit  $k$  distinct “edge directions”, one for each prime divisor of  $n$ , defining a partition of the solids' edges. One can represent this partition as a dictionary whose keys are the prime divisors of  $n$  (inserted from smallest to largest), and as value for a given key  $p$ , the ordered list of ordered pairs of members of  $D$  that make up the endpoints of the edges whose “direction” is associated with  $p$ .



Write a Python program `hasse_diagram.py` that defines a function `make_hasse_diagram()` that returns a named tuple `HasseDiagram` with three attributes:

- `factors`, for a dictionary whose keys are the members of  $D$ , and as value for a given key  $d$ —1 excepted—, a string that represents the prime decomposition of  $d$ , using `x` for multiplication and `^` for exponentiation, displaying only exponents greater than 1;
- `vertices`, for the first dictionary previously defined;
- `edges`, for the second dictionary previously defined.

Using the `doctest` module to test `make_hasse_diagram()` and the `pprint()` function from the `pprint` module, the following behaviour would then be observed:

```
>>> HD = make_hasse_diagram(12)
>>> HD # doctest: +ELLIPSIS
HasseDiagram(factors=..., edges=..., vertices=...)
>>> HD.factors
{1: '1', 2: '2', 3: '3', 4: '2^2', 6: '2x3', 12: '2^2x3'}
>>> pprint(HD.vertices)
{1: ([], [2, 3]),
 2: ([1], [4, 6]),
 3: ([1], [6]),
 4: ([2], [12]),
 6: ([2, 3], [12]),
 12: ([4, 6], [])}
>>> HD.edges
{2: [(1, 2), (2, 4), (3, 6), (6, 12)], 3: [(1, 3), (2, 6), (4, 12)]}
>>> HD = make_hasse_diagram(30)
>>> HD # doctest: +ELLIPSIS
HasseDiagram(factors=..., edges=..., vertices=...)
>>> HD.factors
{1: '1', 2: '2', 3: '3', 5: '5', 6: '2x3', 10: '2x5', 15: '3x5', 30: '2x3x5'}
>>> pprint(HD.vertices)
{1: ([], [2, 3, 5]),
 2: ([1], [6, 10]),
 3: ([1], [6, 15]),
 5: ([1], [10, 15]),
 6: ([2, 3], [30]),
 10: ([2, 5], [30]),
 15: ([3, 5], [30]),
 30: ([6, 10, 15], [])}
>>> pprint(HD.edges)
{2: [(1, 2), (3, 6), (5, 10), (15, 30)],
 3: [(1, 3), (2, 6), (5, 15), (10, 30)],
 5: [(1, 5), (2, 10), (3, 15), (6, 30)]}
```

## 4 Encoding pairs of integers as natural numbers (optional)

Write a program `plane_encoding.py` that implements a function `encode(a, b)` and a function `decode(n)` for the one-to-one mapping from the set of pairs of integers onto the set of natural numbers, that can be graphically described as follows:

16	15	14	13	12
17	4	3	2	11
18	5	0	1	10
19	6	7	8	9
20	21	...		

That is, starting from the point  $(0,0)$  of the plane, we move to  $(1,0)$  and then spiral counterclockwise:

- `encode(0,0)` returns 0 and `decode(0)` returns  $(0,0)$
- `encode(1,0)` returns 1 and `decode(1)` returns  $(1,0)$
- `encode(1,1)` returns 2 and `decode(2)` returns  $(1,1)$
- `encode(0,1)` returns 3 and `decode(3)` returns  $(0,1)$
- `encode(-1,1)` returns 4 and `decode(4)` returns  $(-1,1)$
- `encode(-1,0)` returns 5 and `decode(5)` returns  $(-1,0)$
- `encode(-1,-1)` returns 6 and `decode(6)` returns  $(-1,-1)$
- `encode(0,-1)` returns 7 and `decode(7)` returns  $(0,-1)$
- `encode(1,-1)` returns 8 and `decode(8)` returns  $(1,-1)$
- `encode(2,-1)` returns 9 and `decode(9)` returns  $(2,-1)$
- ...

## 5 Map of CO<sub>2</sub> emissions

(optional, needs a module not installed on CSE computers)

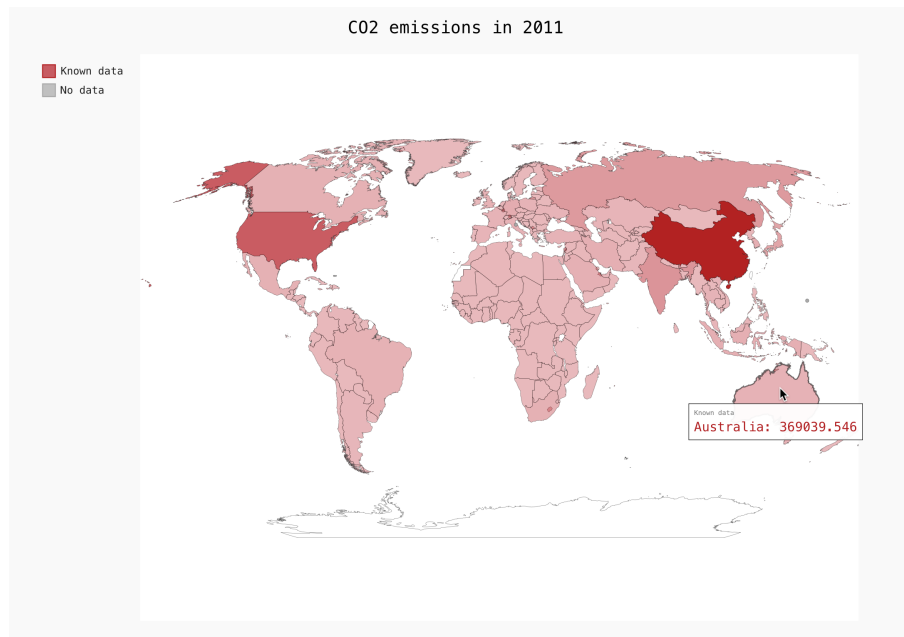
Write a program that extracts from the file `API_EN.ATM.CO2E.KT_DS2_en_csv_v2.csv`, stored in the subdirectory `API_EN` of the working directory, the country CO<sub>2</sub> emissions for the year 2011. Some data in this file are for entities different to countries, or for countries which are not values of the `COUNTRIES` dictionary of the `pygal.maps.world` module. The program will produce an output of the form

```
Leaving out Aruba
Leaving out Arab World
Leaving out American Samoa
Leaving out Antigua and Barbuda
Leaving out Bahamas, The
...
Leaving out Latin America & Caribbean (all income levels)
Leaving out Least developed countries: UN classification
Leaving out Low income
Leaving out Lower middle income
Leaving out Low & middle income
...
Leaving out Virgin Islands (U.S.)
Leaving out Vanuatu
Leaving out West Bank and Gaza
Leaving out World
Leaving out Samoa
```

to let the user know of all those entities and countries, which will be ignored. Some countries are described differently in the dictionary and in the file; these countries will not be ignored. The data will be shown interactively on a map, created as an object of class `World` of the `pygal.maps.world` module, that can be displayed in a browser by opening a file named `CO2_emissions.svg`—check out `render_to_file()`. To create the `World` object from a dictionary having as keys the keys of `COUNTRIES`, check out `add()`. The map should have—check out the `Style` class from the `pygal.style` module:

- as title for the map, `CO2 emissions in 2011`;
- one group of data with `Known data` as legend and with `#B22222` as colour, another group of data with `No data` as legend and with `#A9A9A9` as colour, both with a font size of 10pt;
- tooltips providing standard display for the first group, but with the amount of CO<sub>2</sub> emissions replaced by `?` for the second group, both with a font size of 8pt.

Here is the map with the cursor hovering over Australia, for which the CO<sub>2</sub> emissions are known.



Here is the map with the cursor hovering over Puerto Rico, for which the CO<sub>2</sub> emissions are not known.

