Assignment 1

COMP9021, Session 1, 2018

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, dictionaries, deques, reading from files.

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 Ed. Assignments can be submitted more than once: the last version is marked. Your assignment is due by April 15, 11:59pm.

1.3. **Assessment.** Each of the 4 exercises is worth 2.5 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.

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.

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. Triangle

Write a program, stored in a file named `triangle.py`, that performs the following task.

- The program prompts the user to input a file name. If there is no file with that name in the working directory, then the program outputs an (arbitrary) error message and exits.
- The contents of the file consists of some number of lines, each line being a sequence of integers separated by at least one space, with possibly spaces before and after the first and last number, respectively, the N^{th} line having exactly N numbers. For instance, the contents of the file `triangle_1.txt` can be displayed as follows.

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

- The program outputs:
  - the largest value than can be obtained by summing up all numbers on a path that starts from the top of the triangle, and at every level down to the penultimate level, goes down to the immediate left or right neighbour;
  - the number of paths that yield this largest value;
  - the leftmost such path, in the form of a list.

Here is a possible interaction:

```
$ cat triangle_1.txt
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
$ python3 triangle.py
Which data file do you want to use? triangle_1.txt
The largest sum is: 30
The number of paths yielding this sum is: 1
The leftmost path yielding this sum is: [7, 3, 8, 7, 5]
$ cat triangle_2.txt
     1
    2 2
   1 2 1
  2 1 1 2
 1 2 1 2 1
2 1 2 2 1 2
$ python3 triangle.py
Which data file do you want to use? triangle_2.txt
The largest sum is: 10
The number of paths yielding this sum is: 6
The leftmost path yielding this sum is: [1, 2, 1, 2, 2, 2]
```

You can assume that the contents of any test file is as expected, you do not have to check that it is as expected.

3. Fishing towns

Write a program, stored in a file named `fish.py`, that performs the following task.

- The program prompts the user to input a file name. If there is no file with that name in the working directory, then the program outputs an (arbitrary) error message and exits.
- The contents of the file consists of some number of lines, each line being a sequence of two nonnegative integers separated by at least one space, with possibly spaces before and after the first and second number, respectively, the first numbers listed from the first line to the last line forming a strictly increasing sequence. The first number represents the distance (say in kilometres) from a point on the coast to a fishing town further down the coast (so the towns are listed as if we were driving down the coast from some fixed point); the second number represents the quantity (say in kilos) of fish that has been caught during the early hours of the day by that town's fishermen. For instance, the contents of the file `coast_1.txt` can be displayed as

  $$\begin{array}{rr} 5 & 70 \\ 15 & 100 \\ 1200 & 20 \end{array}$$

  which corresponds to the case where we have 3 towns, one situated 5 km south the point, a second one situated 15 km south the point, and a third one situated 1200 km south the point, with 70, 100 and 20 kilos of fish being caught by those town's fishermen, respectively.
- The aim is to maximise the quantity of fish available in all towns (the same in all towns) by possibly transporting fish from one town to another one, but unfortunately losing 1 kilo of fish per kilometre. For instance, if one decides to send 20 kilos of fish from the second town to the first one, then the second town ends up having $100 - 20 = 80$ kilos of fish, whereas the first one ends up having $70 + 20 - (15 - 5) = 80$ kilos of fish too.
- The program outputs that maximum quantity of fish that all towns can have by possibly transporting fish in an optimal way.

Here is a possible interaction:

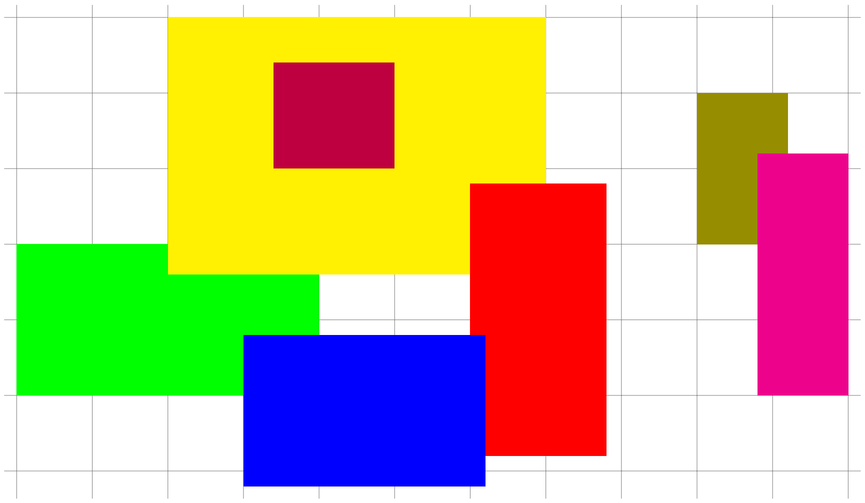
```
$ cat coast_1.txt
5 70
15 100
1200 20
$ python3 fish.py
Which data file do you want to use? coast_1.txt
The maximum quantity of fish that each town can have is 20.
$ cat coast_2.txt
20 300
40 400
340 700
360 600
$ python3 fish.py
Which data file do you want to use? coast_2.txt
The maximum quantity of fish that each town can have is 415.
```

You can assume that the contents of any test file is as expected, you do not have to check that it is as expected.

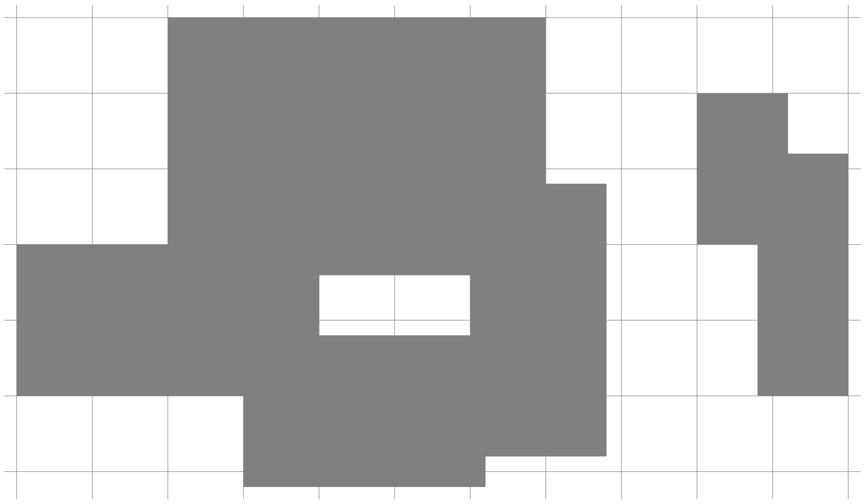
4. Overlapping photographs

Write a program, stored in a file named `perimeter.py`, that performs the following task.

- Prompts the user to input the name of text file assumed to be stored in the working directory. We assume that if the name is valid then the file consists of lines all containing 4 integers separated by whitespace, of the form x_1 y_1 x_2 y_2 where (x_1, y_1) and (x_2, y_2) are meant to represent the coordinates of two opposite corners of a rectangle. With the provided file `frames_1.txt`, the rectangles can be represented as follows, using from top bottom the colours green, yellow, red, blue, purple, olive, and magenta.



  We assume that all rectangles are distinct and either properly overlap or are disjoint (they do not touch each other by some of their sides or some of their corners).
- Computes and outputs the perimeter of the boundary, so with the sample file `perimeter.py`, the sum of the lengths of the (external or internal) sides of the following picture.



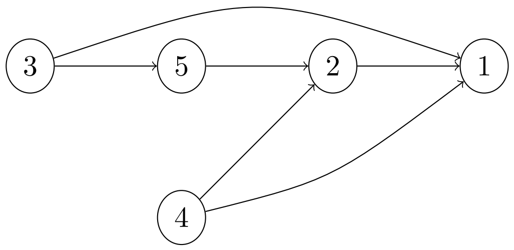
Here is a sample run of the program with the two provided sample files.

```
$ python3 perimeter.py
Which data file do you want to use? frames_1.txt
The perimeter is: 228
$ python3 perimeter.py
Which data file do you want to use? frames_2.txt
The perimeter is: 9090
```

5. Partial orders

Write a program, stored in a file named `nonredundant.py`, that performs the following task.

- The program prompts the user to input a file name. If there is no file with that name in the working directory, then the program outputs an (arbitrary) error message and exits.
- The contents of the file consists of lines with text of the form `R(m,n)` where `m` and `n` are integers (that just represent labels), with possibly spaces before and after the opening and closing parentheses, respectively. It represents a partial order relation R (so an asymmetric and transitive binary relation). For instance, the contents of the file `partial_order_1.txt` can be represented as:



  It can be seen that two facts are redundant:
  - the fact $R(3,1)$, which follows from the facts $R(3,5)$, $R(5,2)$ and $R(2,1)$;
  - the fact $R(4,1)$, which follows from the facts $R(4,2)$ and $R(2,1)$.
- The program outputs the facts that are nonredundant, respecting the order in which they are listed in the file.

Here is a possible interaction:

```
$ cat partial_order_1.txt
R(3,5)
R(4,2)
R(5,2)
R(2,1)
R(3,1)
R(4,1)
$ python3 nonredundant.py
Which data file do you want to use? partial_order_1.txt
The nonredundant facts are:
R(3,5)
R(4,2)
R(5,2)
R(2,1)
```

```
$ cat partial_order_2.txt
R(3,5)
R(5,2)
R(2,6)
R(2,1)
R(3,6)
R(6,1)
R(4,2)
R(4,1)
$ python3 nonredundant.py
Which data file do you want to use? partial_order_2.txt
The nonredundant facts are:
R(3,5)
R(5,2)
R(2,6)
R(6,1)
R(4,2)
$ cat partial_order_3.txt
R(1,2)
R(2,3)
R(3,4)
R(1,3)
R(2,4)
$ python3 nonredundant.py
Which data file do you want to use? partial_order_3.txt
The nonredundant facts are:
R(1,2)
R(2,3)
R(3,4)
```

You can assume that the contents of any test file is as expected, you do not have to check that it is as expected.