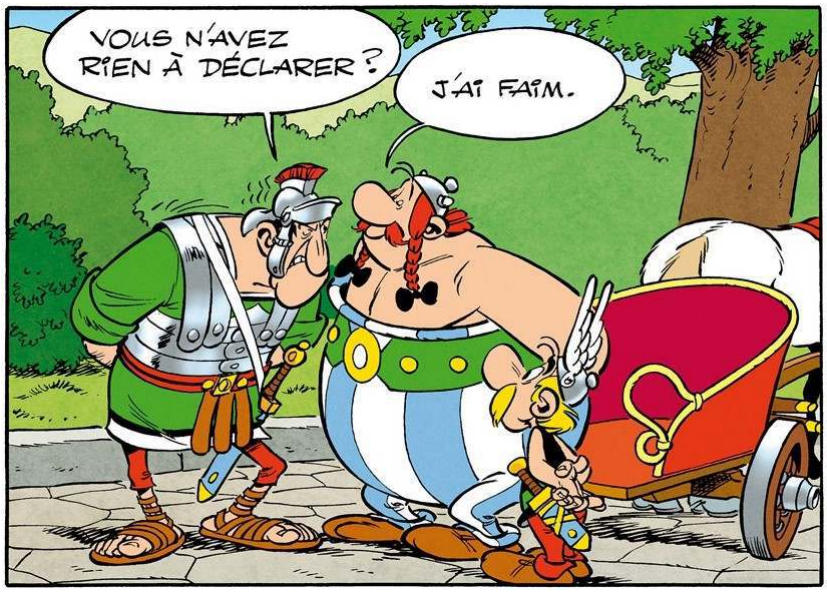
Algorithmics
Final Exam #3 (P3)

Undergraduate 2^{nd} year - S3#
EPITA

14 May 2019

Instructions (read it) :

- You must answer on **the answer sheets provided.**
  - No other sheet will be picked up. Keep your rough drafts.
  - Answer within the provided space. **Answers outside will not be marked**: Use your drafts!
  - Do not separate the sheets unless they can be re-stapled before handing in.
  - Penciled answers will not be marked.
- The presentation is negatively marked, which means that you are marked out of 20 points and the presentation points (maximum of 2) are taken off this grade.
- **Code:**
  - All code must be written in the language `Python` (no C, CAML, ALGO or anything else).
  - **Any `Python` code not indented will not be marked.**
  - All that you need (classes, types, routines) is indicated where needed!
  - You can write your own functions as long as they are documented (we have to know what they do). In any case, the last written function should be the one which answers the question.
- Duration : 2h



Exercise 1 (Spanning Forest – 2 *points*)

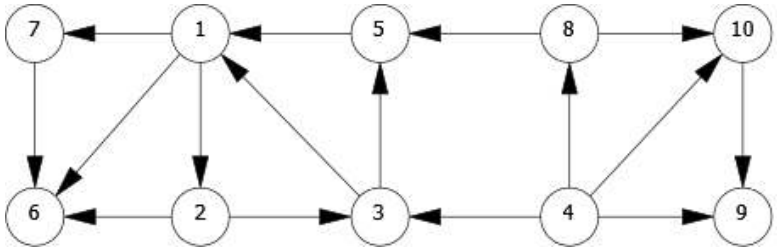


Figure 1: Graphe G_1

Build the spanning forest of the graph G_1 (figure 1) for the depth first traversal from vertex 1. Vertices are encountered in increasing order. *Add to the forest the various kinds of edges met during the traversal, with an explicit legend.*

Exercise 2 (Union-Find – 3 *points*)

Let G be the graph $< S, A >$ with vertices numbered from 0 to 13. The algorithms seen in lecture `find` (without compression) and `union` (optimized version: union by rank) give the following vector p from the list of edges A:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
p	5	8	5	8	11	-4	5	10	-6	12	8	12	-4	8

1. Give the number of vertices of each connected component of G (the order does not matter).

2. Which additional edges will be enough to make the graph connected?

3. Among the following chains, which can not exist in G?

   - $3 \rightsquigarrow 7$
   - $11 \rightsquigarrow 6$
   - $0 \rightsquigarrow 13$
   - $4 \rightsquigarrow 9$

Exercise 3 (Bipartite graph (Graphes bipartis) – *5 points*)

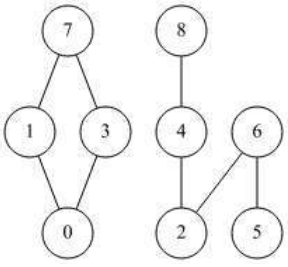


Figure 2: Graph G_2, biparti

A bipartite graph is a graph $G =< S, A >$ where vertices can be partitioned into two sets S_1 et S_2, such that $(u, v) \in A$ implies either $u \in S_1$ and $v \in S_2$, or $u \in S_2$ and $v \in S_1$. That is, no edge connects vertices in the same set.

Le graphe G_2 de la figure 2 est biparti avec, par exemple, $S_1 = \{0, 2, 5, 7, 8\}$ et $S_2 = \{1, 3, 4, 6\}$

Write a function that tests whether a graph is bipartite.

Exercise 4 (Eat Crepes – *8 points*)

Below is the recipe for the banana crepe "flambée", with for each task its duration in seconds.

The recipe	Duration (in sec.)	Ref.
Put the flour in a bowl	3	A
add two eggs,	30	B
gently add the milk and mix.	600	C
Put the rum in a pan.	3	D
Cut the bananas into thin slices,	300	E
add them to the rum.	30	F
Heat the mixture,	120	G
flame the mixture.	10	H
Cook a crepe,	10	I
Pour some rum-banana mix over the crepe.	10	J

Some details regarding the sequence of tasks:

- crepe dough and rum-banana mix can be done in parallel;
- it is only when the crepe is cooked and the mixture is ready that we can pour the mixture on the crepe and finally eat it;
- the other steps are carried out sequentially: the flour must be put before the eggs, the rum must be put before the bananas (which can be cut in advance) in the pan.

1. Model the recipe as a graph:
   - The vertices are the tasks.
   - The tasks *start* and *end* are the beginning (the command) and the end (degustation) of the project.

   The graph that represents the recipe is acyclic. Moreover, all the vertices are reachable from a given vertex (here the command of the crepe = the task *start*, vertex 0 in machine representation). In the rest of the exercise, the graph will have these specifications!

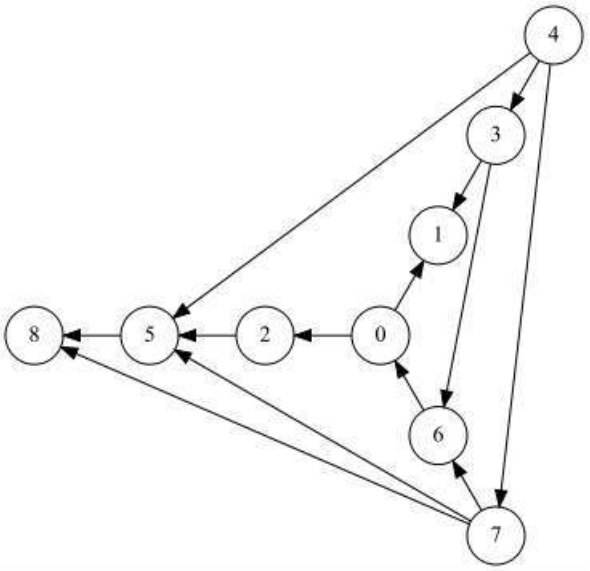
2. **The cook is alone:** the time the recipe takes is given by the sum of all durations. The cook is sharp on each task but needs help ordering them: the solution is a topological sort of the graph.
   (a) An obvious topological sort is given by the order of the recipe. Give another solution: complete the one given on the answer sheets.
   (b) **Reminder:** A topological sort of an acyclic digraph can be found by classifying the vertices in descending order of meeting in suffix during a depth-first search.
   Write the function thta returns a topologic order (a vertex list) for a digraph with the previous given specifications.
   (c) Write a function that tests whether a vertex list is a correct topological sort for a given digraph (assumed acyclic).

Exercise 5 (What does it do? – *4 points*)

Consider the following functions:

```
def build(G):
    V = [0] * G.order
    for x in range(G.order):
        for y in G.adjlists[x]:
            V[y] += 1
    return V

def what(G):
    V = build(G)
    L = []
    while len(L) < G.order:
        x = 0
        while x < G.order and V[x] != 0:
            x += 1
        V[x] = -1
        L.append(x)
        for y in G.adjlists[x]:
            V[y] -= 1
    return L
```



Graph G_3

1. Give the result of the application `build`(G_3), with G_3 the digraph in figure 5.
2. The function `what`
   (a) Give the result of the application of the function `what`G_3), with G_3 the digraph in figure 5.
   (b) What does the result list represent for the digraph in parameter?
   (c) What should be the property of the graph so that this function does not "crash"?

Appendix

Classes `Graph` and `Queue` are supposed imported.

Graphs

All exercises use the implementation with adjacency lists of graphs.
Graphs we manage cannot be empty.

```
class Graph:
    def __init__(self, order, directed = False):
        self.order = order
        self.directed = directed
        self.adjlists = []
        for i in range(order):
            self.adjlists.append([])
```

Queues

- `Queue()` returns a new queue
- q`.enqueue(`e`)` enqueues e in q
- q`.dequeue()` returns the first element of q, dequeued
- q`.isempty()` tests whether q is empty

Others

- `range`
- on lists:
  - `len(L)`
  - `L.append()`
  - `L.pop()`
  - `L.insert(index, elt)`

Your functions

You can write your own functions as long as they are **documented** (we have to know what they do).
In any case, the last written function should be the one which answers the question.