# Algorithmics
# Final Exam #4 (P4)

Undergraduate $2^{nd}$ year S4
EPITA

*May, 14th 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 (class, types, routines) is indicated in the **appendix** (last page)!
  - You can write your own functions as long as they are documented (we have to known what they do).
    In any case, the last written function should be the one which answers the question.
- ☐ Duration : 2h

---

**Exercise 1 (Restyled Floyd – *3 points*)**

1. Simply explain how to change Floyd's algorithm (view in lecture), so that it stops as soon as it detects a negative cycle.

2. In some network problems, one needs to know the point (or points) that are at least distant from all the other points of a network. We call **eccentricity** of a vertex $x$, in a directed graph with positive costs, $G =< S, A, C >$, the quantity:

$$\text{exc}(x) = \max_{y \in S}\{sdistance(x, y)\}$$

   This is the distance needed to reach $x$ from any other vertex. We call **center** of the graph $G$, a vertex of minimum eccentricity.

   Simply explain how one can use Floyd's algorithm to find the center of a graph.

**Exercise 2 (MST or not? – *2 points*)**

Check whether the spanning subgraph $T$ built by the following process is a minimum spanning tree of $G$ or not. If it is, explain why, otherwise give an example.

Let $G =< S, A, C >$ be a undirected weighted connected graph. We split $G$ into two subgraphs: $G1 =< S1, A1, C >$ and $G2 =< S2, A2, C >$ with $S = S1 \cup S2$. Let $T1$ (respectively $T2$) be a minimum spanning tree of $G1$ (respectively of $G2$). We choose a minimum cost edge $v$ among the edges having one end in $S1$ and the other one in $S2$. $T$ is then the spanning subgraph resulting of the union of the edges of $T1$, the edges of $T2$ and of $v$.

**Exercise 3 (Eat Crepes – *11 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 representation of the graph must be planar [1].

[1] For the record, this means that the edges must not cross each other!

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). And all vertices can reach the last task (last vertex 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 topological sort can be found by classifying the vertices in descending order of meeting in suffix during a depth-first search.
   Another method uses a vector of vertex in-degrees.

   Use necessarily this second method to write the function that returns a topological sort (a list of vertices) for a graph having the same specifications as above.

3. **The cook has found assistants, some tasks can thus be done simultaneously.**

   (a) The *early start* of the task $i$ is the date at which the task $i$ can start at the earliest.
   How can we compute these dates in this kind of graph?
   Fill-in the array that gives the *early start* for each task of the crepe recipe.

   (b) How much time will it take to eat our crepe?

   (c) The *late start* of the task $i$ is the latest date at which the task $i$ can start without delaying the rest of the project.
   How can we compute these dates in this kind of graph?

   (d) Write the function that computes the minimum duration to complete a project represented by a graph with the same specifications as above.
   You must use the function of the question 2.

**Exercise 4 (Prim, Quite Simply – *5 points*)**

Write the function `Prim(G)` that returns a minimum spanning tree of the connected graph $G$, using Prim's algorithm!

# Appendix

Classes `Graph` and `Heap` are supposed imported. Graphs we manage cannot be empty.

### Graphs

```
class Graph:
    def __init__(self, order, directed = False, costs = False):
        self.order = order
        self.directed = directed
        self.adjlists = []
        for i in range(order):
            self.adjlists.append([])
        if costs:
            self.costs = {}
        else:
            self.costs = None

    def addedge(self, src, dst):
        self.adjlists[src].append(dst)
        if not self.directed and dst != src:
            self.adjlists[dst].append(src)
        if cost:
            self.costs[(src, dst)] = cost
            if not self.directed:
                self.costs[(dst, src)] = cost
```

### Heap

- `Heap(`$n$`)`
  returns a new heap with size $n$
- $H$`.push(`$s$`,` $val$`)`
  add $s$ of value $val$ in $H$
- $H$`.update(`$s$`,` $newval$`)`
  if $s$ not in $H$ same as `H.push(`$s$`,` $newVal$`)`
  else updates the heap after minimization of $s$'s value with $newVal$
- `(`$val$`,` $s$`) =` $H$`.pop()`
  returns and deletes the element ($s$) of smallest value ($val$) in heap
- $H$`.isempty()`
  tests whether $H$ is empty

### 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

`List`

- `len(L)`
- `L.append(x)`
- `L.pop()` `L.pop(i)`
- `L.insert(i, x)`

### Others

- `range`.
- `min`, `max`, `abs` ...
- value `inf` or $\infty$

## 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.