# PE3: PE of 21/12/2018

---

**Master in Informatics and Computing Engineering**
**Programming Fundamentals**
**Instance: 2018/2019**

Some **important** information about this PE (Practical on computer evaluation):

- You have **90 minutes** to answer the 5 questions of the test
- **No collaboration** between students is allowed
- **It is forbidden** the presence on the table and the use of mobile phones or any other electronic devices
- The Python code that answers each question is saved in a file with **the name required in the question**
- **Before the time expires** you must *upload* a zip with the Python code of all your answers; you have **only one attempt** but you may upload the zip as many times as you wish; therefore, you should try the upload procedure at least 5 minutes before the time expires, to **guarantee you have one zip** with the answers to be graded as, otherwise, you'll have **zero**.
- You are allowed to use the **Consultation Book** and the **Standard Library**, both in PDF, but all the remaining content will be hidden

## 1. Treasure

Write a Python function `treasure(clues)` that receives a dictionary of `clues` where each key is a location and each value is a clue of what is the next location to go. Start at (0,0) and return the tuple of the final location you end up with. Note that clues may be used only once.

For example, if `clues={(0,0): (1,0), (2,1): (3,5), (1,0): (2,1)}` then you should first go to (0,0) then (1,0) then (2,1) then (3,5) and then finish because there's no clue in that location. In this case, the function returns (3,5).

Save the program in the file `treasure.py` inside the folder `PE3`.

For example:

- `treasure({(0,0): (1,0), (1,0): (2,0), (2,0): (3,0)})` returns the tuple `(3,0)`
- `treasure({(0,0): (1,0), (2,1): (3,5), (1,0): (2,1)})` returns the tuple `(3,5)`
- `treasure({(0,0): (5,6), (7,8): (6,7), (5,6): (6,7), (6,7): (7,8)})` returns the tuple `(6,7)`

## 2. Polynomials

Suppose we have a polynomial **a** represented as a list of coefficients, a[0], a[1], ..., a[n-1], where a[i] is the coefficient of xi; that is:

$$f(x) = a_0x^0 + a_1x^1 + ... + a_nx^n$$

Write a function **`evaluate(a, x)`** using list comprehensions (or *map* and *reduce*) that evaluates the value of the polynomial for a given integer **`x`**.

Save the program in the file **`evaluate.py`**

For example:

- **`evaluate([1, 2, 4], 5)`** returns the integer **`111`**
- **`evaluate([1, 2, 4], 10)`** returns the integer **`421`**
- **`evaluate([1, 2, 4, 6, 8], 2)`** returns the integer **`197`**

## 3. Recursive dot product

Write a function **`recursive_dot(l1, l2)`** that computes the inner dot product using the two lists provided — the two lists follow the same structure. For example, if **`l1=[1, [2, 3]]`** and **`l2=[4, [5, 6]]`**, then the result will be 1*4+(2*5+3*6).

Save the program in the file **`recursive_dot.py`** inside the folder **`PE3`**.

You might want to use **`type(x)`** to find the type of **`x`**.

For example:

- **`recursive_dot([1, [2, 3]], [4, [5, 6]])`** returns the integer **`32`**
- **`recursive_dot([[5, 3, 1], [2, 4]], [[4, 2, 0], [1, 3]]))`** returns the integer **`40`**
- **`recursive_dot([2], [1])`** returns the integer **`2`**

## 4. Interleave Lists

Write a Python function **`interleave(alist1, alist2)`** that, given two lists with the same structure but not necessarily the same length, **`alist1`** and **`alist2`**, which may contain other lists, returns a list with the interleaved elements of both **`alist1`** and **`alist2`**. The result list has the length of the smaller of the two lists.

Save the program in the file **`interleave.py`** inside the folder **`PE3`**.

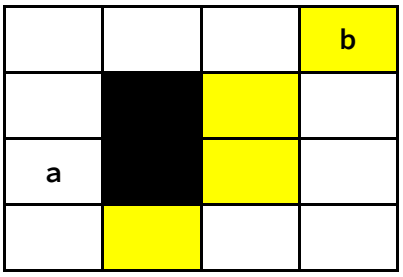
For example:

- **`interleave([1, [4,2]], [3, [7,4]])`** returns the list **`[1,3,4,7,2,4]`**
- **`interleave(['a','b','c'], [1,2,3,4,5])`** returns the list **`['a',1,'b',2,'c',3]`**
- **`interleave([], [1,2])`** returns the list **`[]`**

## 5. Minimum path

Write a function **`min_path(matrix, a, b, visited=[])`** that discovers the minimum path between **`a`** and **`b`** inside the **`matrix`** maze without going through **`visited`** twice. Positions **`a`** and **`b`** are tuples (line, column), **`matrix`** is a matrix of booleans with **`False`** indicating no obstacle and **`True`** indicating an obstacle and **`visited`** is a list of visited tuples. Valid movements include all 8 adjacent tiles.

For the maze of the following figure, a minimum path between **`a`** and **`b`**, in yellow, is **4**:




Save the program in the file **`min_path.py`** inside the folder **`PE3`**.

For example:

- ```
  mx = [
    [False]*4,
    [False, True, False, False],
    [False, True, False, False],
    [False]*4
  ]
  ```

- `min_path(mx, (2, 0), (0, 3))` returns the integer `4`
- `min_path(mx, (3, 1), (0, 1))` returns the integer `3`
- `min_path(mx, (0, 0), (3, 3))` returns the integer `4`

## The end.

*FPRO, 2018/19*
  ```