# Scientific Programming Algolab - Exam Monday 11, June 2018

**Scientific Programming, Part A + Part B, Quantitative Computational Biology Master, CIBIO**

## Download exercises

## Introduction

**Part A** is by Luca Bianco:

- A.1 loadData
- A.2 createMareyPlot

**Part B.1** is more about theory with dynamic programming, and will be assessed by Alberto Montresor:

- B.1 maxRest

**Part B.2** is a supermarket queues exercise by David Leoni

- B.2.1 size
- B.2.2 enqueue
- B.2.3 dequeue

### Allowed material

There won't be any internet access. You will only be able to read:

- Scientific Programming Algolab worksheets
- Alberto Montresor slides
- Luca Bianco docs
- Python 3 documentation : html pdf
  In particular, Unittest docs
- The course book *Problem Solving with Algorithms and Data Structures using Python* html pdf

### Grading

- **Lab grade:** The grade of this lab part will range from 0 to 30. Part A is by Luca Bianco , Part B second exercise B.1) is more about theory with dynamic programming, and will be assessed by Alberto Montresor, while the remaining ones (B.2,..) are by David Leoni.
- **Correct implementations**: Correct implementations with the required complexity grant you full

grade.

- **Partial implementations**: Partial implementations *might* still give you a few points. If you just can't solve an exercise, try to solve it at least for some subcase (i.e. array of fixed size 2) commenting why you did so.
- **Bonus point**: One bonus point can be earned by writing stylish code. You got style if you:
  - do not infringe the Commandments
  - write pythonic code
  - avoid convoluted code like i.e.

```
if x > 5:
    return True
else:
    return False
```

when you could write just

```
return x > 5
```

## Valid code

**WARNING**: MAKE SURE ALL EXERCISE FILES AT LEAST COMPILE !!! 10 MINS BEFORE THE END OF THE EXAM I WILL ASK YOU TO DO A FINAL CLEAN UP OF THE CODE

**WARNING**: *ONLY* IMPLEMENTATIONS OF THE PROVIDED FUNCTION SIGNATURES WILL BE EVALUATED !!!!!!!!!

For example, if you are given to implement:

```
def f(x):
    raise Exception("TODO implement me")
```

and you ship this code:

```
def my_f(x):
    # a super fast, correct and stylish implementation

def f(x):
    raise Exception("TODO implement me")
```

We will assess only the latter one `f(x)`, and conclude it doesn't work at all :P !!!!!!!



### Helper functions

Still, you are allowed to define any extra helper function you might need. If your `f(x)` implementation calls some other function you defined like `my_f` here, it is ok:

```
# Not called by f, will get ignored:
def my_g(x):
    # bla

# Called by f, will be graded:
def my_f(y,z):
    # bla

def f(x):
    my_f(x,5)
```

## How to edit and run

To edit the files, you can use any editor of your choice, like:

- **Visual Studio Code**, *PyCharm*, *Editra*, *GEdit*

To run the tests, use **the Terminal** which can be found in *Accessories -> Terminal*

**IMPORTANT**: Pay close attention to the comments of the functions.

**WARNING**: *DON'T* modify function signatures! Just provide the implementation.

**WARNING**: DON'T change the existing test methods, just add new ones !!! You can add as many as you want.

**WARNING**: *DON'T* create other files. If you still do it, they won't be evaluated.

## Debugging

If you need to print some debugging information, you are allowed to put extra print statements in the function bodies.

**WARNING:** even if print statements are allowed, be careful with prints that might break your function!

For example, avoid stuff like this:

```
x = 0
print(1/x)
```

### What to do

1. Download sciprolab-2018-06-11-exam.zip and extract it *on your desktop*. Folder content should be like this:

```
sciprolab-2018-06-11
    |- FIRSTNAME-LASTNAME-ID
        |- exerciseA1.py
        |- exerciseB1.py
        |- exerciseB2.py
```

2. Rename `FIRSTNAME-LASTNAME-ID` folder: put your name, lastname an id number, like `john-doe-432432`

From now on, you will be editing the files in that folder. At the end of the exam, that is what will be evaluated.

3. Edit the files following the instructions in this worksheet for each exercise. Every exercise should take max 25 mins. If it takes longer, leave it and try another exercise.

# Part A

by Luca Bianco.

> Please write only in exerciseA.py, don't create other files! **IMPORTANT**: Add your name and ID (matricola) on top of the .py file!

### The Problem

The `.tsv` file `map_info.tsv` is tab separated value file representing genetic and physical information on the chromosomes of a genome, based on some single nucleotide polymorphisms (SNPs). The first few lines are reported below:

```
MarkerID    GeneticChr  GeneticPos   PhysicalChr  PhysicalPos
SNPid1      LG1     12.141      Chr2        40943390
SNPid2      LG1     12.141      Chr2        40943797
SNPid3      LG10        46.511      Chr13       16915715
SNPid4      LG7     13.209      Chr7        7242600
SNPid5      LG1     20.956      Chr2        37244585
```

as the header (first line) describes, the first column is the identifier of the SNP linking the genetic map to the physical position, the second is the genetically identified chromosome (linkage group), the third is the genetic position within that linkage group (in centiMorgan, that is 1 recombination per 100 individuals), the fourth and fifth are the physical position of the SNP (chromosome and position within the chromosome).



Implement the following python functions:

## A.1) loadData(filename)

Implement function `loadData(filename)`: gets the filename of a `.tsv` file as explained above, stores its content in a suitable data structure of your choice (**hint: pandas might help here**), counts (and prints) the number of SNPs in the file, the number of distinct linkage groups (i.e. values in the GeneticChr column), the number of distinct chromosomes (i.e. values in the PhysicalChr column) and prints the average number of SNPs for each linkage group and for each chromosome.

**Note: The function should return the data structure containing all the data.**

Calling:

```
dF = loadData("map_info.tsv")
```

should give:

```
The file "map_info.tsv" contains 883 SNPs

Data has information for the following 15 linkage groups:
         LG1,LG3,LG10,LG6,LG4,LG2,LG13,LG14,LG8,LG9,LG7,LG11,LG5,LG16,LG12

LG      SNPcount
LG1     114
LG3     114
LG10    108
LG6     97
LG4     82
LG2     79
LG13    74
LG14    73
LG8     62
LG9     46
LG7     28
LG11    2
LG5     2
LG16    1
LG12    1

Data has information for the following 11 chromosomes:
         Chr11,Chr2,Chr13,Chr10,Chr8,Chr6,Chr14,Chr9,Chr12,Chr4,Chr7

Chr     SNPcount
Chr11   116
Chr2    116
Chr13   110
Chr10   97
Chr8    81
Chr6    79
Chr14   78
Chr9    73
Chr12   60
Chr4    47
Chr7    26
```

## A.2) createMareyPlot(data, ChrID)

Implement function `createMareyPlot(data, ChrID)`: gets the data structure created by loadData and a physical chromosome ID and produces a scatter plot with the physical position on the X coordinate and the geneticPosition on the Y coordinate, coloring dots depending on the geneticChr they belong to (see examples below). If `ChrID` is not in data, then a message “No information on

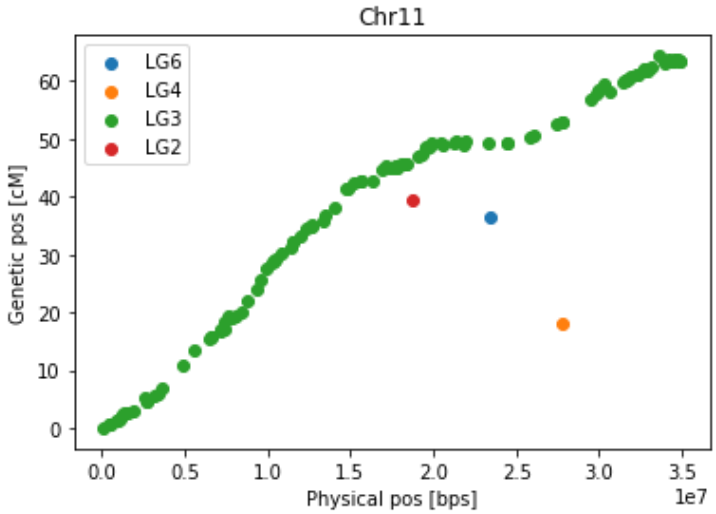
ChrID" should be returned.

Calling:

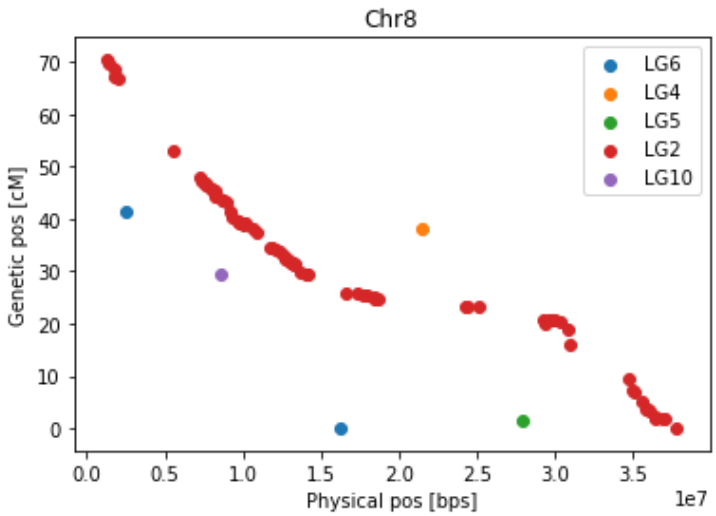
```
createMareyPlot(dF, "Chr11")
createMareyPlot(dF, "Chr8")
createMareyPlot(dF, "Chr0")
```

should give something like (colors might change):

```
MareyPlot for Chr11:
```




```
MareyPlot for Chr8:
```




```
No information on Chr0
```



# Part B

## B.1) MaxRest

By Alberto Montresor.

### The problem

You are given a list `L` containing `n` coin values, where `L[i]` is the value of the `i-th` coin in cents. For example,

```
L = [2,1,5,2,2,5]
```

means that you have one coin with value 1, three coins with value 2, two coins with value 5.

Your task is to write an algorithm that takes a list and an integer `R` has input, and returns the maximum number of coins that could be used to give a rest of exactly `L` cents. For example, consider the value `R=7` and the previous list `L`; you can obtain `7` by:

- 5+2 (2 coins)
- 2+2+2+1 (4 coins)

The algorithm should return 4. For other examples, see test cases.

In order to write the algorithm, use dynamic programming/memoization and compute a table `PD[i][r]` that contains the maximum amount of coins that can be used to give a rest of `r` considering the first `i` coins.

**IMPORTANT**: dynamic programming exercises are the most difficult ones, we know it. If not all test pass (or not even one..) don't despair, you might still get a good grade!

### Implementation

Start editing the file `exerciseB1.py` which contains the function:

```
def maxRest(L,R):
    """ Takes

            L: a list of coins where each element represents the value of the coin
               -  elements are integer > 0
               - list can be empty
            R: an integer rest R
               - may be negative, zero, or positive.

        Return:   the maximum number of coins to add so the sum is R.
                  If it is impossible to find coins such that their sum equals R,
                  return minus infinity
    """
```

**Testing:** `python3 -m unittest exerciseB1.MaxRestTest`

## B.2) Supermarket queues

By David Leoni

In this exercises, you will try to model a supermarket containing several cash queues.

### CashQueue

#### WARNING: DO *NOT* MODIFY CashQueue CLASS

For us, a `CashQueue` is a simple queue of clients represented as strings. A `CashQueue` supports the `enqueue`, `dequeue`, `size` and `is_empty` operations:

- Clients are enqueued at the right, in the tail
- Clients are dequeued from the left, removing them from the head

For example:

```
q = CashQueue()

q.is_empty()      # True

q.enqueue('a')    #  a
q.enqueue('b')    #  a,b
q.enqueue('c')    #  a,b,c

q.size()          # 3

q.dequeue()   #       returns:  a
              # queue becomes:  [b,c]

q.dequeue()   #       returns:  b
              # queue becomes:  [c]

q.dequeue()   #       returns:  c
              # queue becomes:  []

q.dequeue()   # raises LookupError as there aren't enough elements to remove
```

### Supermarket

A `Supermarket` contains several cash queues. It is possible to initialize a `Supermarket` by providing queues as simple python lists, where the first clients arrived are on the left, and the last clients are on the right.

For example, by calling:

```
s = Supermarket([
    ['a','b','c'],     # <------ clients arrive from right
    ['d'],
    ['f','g']
])
```

internally three `CashQueue` objects are created. Looking at the first queue with clients `['a','b','c']`, `a` at the head arrived first and `c` at the tail arrived last

```
>>> print(s)

Supermarket
0 CashQueue: ['a', 'b', 'c']
1 CashQueue: ['d']
2 CashQueue: ['f', 'g']
```

Note a supermarket must have at least one queue, which may be empty:

```
s = Supermarket( [[]] )

>>> print(s)

Supermarket
0 CashQueue: []
```

### Supermarket as a queue

Our `Supermarket` should maximize the number of served clients (we assume each clients is served in an equal amount of time). To do so, the whole supermarket itself can be seen as a particular kind of queue, which allows the `enqueue` and `dequeue` operations described as follows:

- by calling `supermarket.enqueue(client)` a client gets enqueued in the shortest `CashQueue`.
- by calling `supermarket.dequeue()`, all clients which are at the heads of non-empty `CashQueue`s are dequeued all at once, and their list is returned (this simulates parallelism).

### Implementation

Now start editing `exerciseB2.py` implementing methods in the following points.

## B.2.1) Supermarket size

Implement `Supermarket.size` :

```
def size(self):
    """ Return the total number of clients present in all cash queues.
    """
```

**Testing**: `python3 -m unittest exerciseB2_test.SizeTest`

## B.2.2) Supermarket dequeue

Implement `Supermarket.dequeue` :

```
def dequeue(self):
    """ Dequeue all the clients which are at the heads of non-empty cash queues,
        and return a list of such clients.

        - clients are returned in the same order as found in the queues
        - if supermarket is empty, an empty list is returned

        For example, suppose we have following supermarket:

        0  ['a','b','c']
        1  []
        2  ['d','e']
        3  ['f']


        A call to deque() will return ['a','d','f']
        and the supermarket will now look like this:

        0  ['b','c']
        1  []
        2  ['e']
        3  []
     """
```

**Testing:** `python3 -m unittest exerciseB2_test.DequeueTest`

## B.2.3) Supermarket enqueue

Implement `Supermarket.enqueue` :

```
def enqueue(self, client):
    """ Enqueue provided client in the cash queue with minimal length.

        If more than one minimal length cash queue is available, the one
        with smallest index is chosen.

        For example:

        If we have supermarket

        0  ['a','b','c']
        1  ['d','e','f','g']
        2  ['h','i']
        3  ['m','n']

        since queues 2 and 3 have both minimal length 2, supermarket.enqueue('z')
        will enqueue the client on queue 2:

        0  ['a','b','c']
        1  ['d','e','f','g']
        2  ['h','i','z']
        3  ['m','n']
    """
```

**Testing:** `python3 -m unittest exerciseB2_test.EnqueueTest`