Scientific Programming - Exam Tuesday 21, August 2018

Part A + Part B, Quantitative Computational Biology Master, CIBIO

Download exercises

Introduction

Part A is by Luca Bianco:

- A.1 loadData
- A.2 computeNX

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

- B.1 stock

Part B.2 is about chainable lists:

- B.2.1 has_duplicates
- B.2.2 chain

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 sciprolab2-2018-08-21-exam.zip and extract it *on your desktop*. Folder content

should be like this:

```
sciprolab2-2018-08-21
    |- 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.

Part A

by Luca Bianco. Takes 1.30h

IMPORTANT: 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

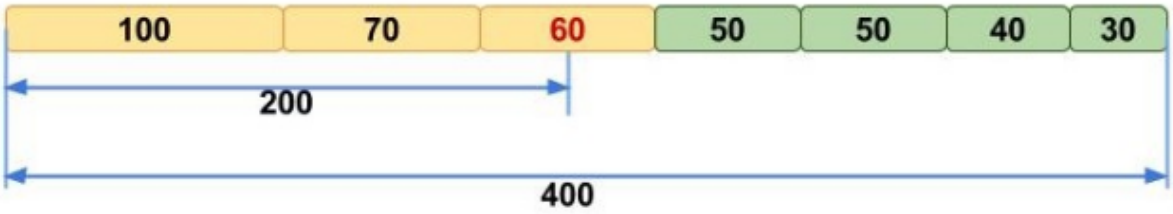
You should recall that fasta is a standard format to represent biological sequences. A couple of mock entries are the following:

```
>Contig1
AGGCCTAGGTCTTCCAGAGTCGCTTTTTCCAGCTCCAGACCGATCTCTTCAG
AGGCCAATCGCCAGTTTACCACATACACCCAGACCGATCTCTTCAG
>Contig2
TTACCACATACACCCAGACCAGAGTCGCTTTTTCCCCGATCTCTTCA
GTTTACCACATACACCCAGACCGTTTACCACATACACCCAGACCGTTTACCACATACACCCAGACC
GTACCACATACACCCAGACC
```

where the first line is the identifier of the read starting with a ">". The sequence follows the line with the identifier and can be on multiple lines.

In genome assembly, `N50` is a useful measure to give an indication of the contiguity of the assembled sequence (contig) as it is defined as the length of the smallest sequence (`N`) such that $>= 50\%$ of the total sequence is in sequences of size at least `N`. A possible way to compute the `N50` is to sort all the contig sizes decreasingly and compute the cumulative sum of the sizes starting from the biggest contig. The size that makes the cumulative sum go above the 50% of the total sum is the `N50` value. Similarly, the size that makes the cumulative sum go above the 20%, 30%, $X\%$ of the total sum is the `N20`, `N30`, `NX` value. In the example below, the `N50` is 60.

1a. Contigs, sorted according to their lengths.



1b. Calculation of N50 using sorted contigs.

Implement the following python functions:

A.1) loadData

Implement `loadData(filename)`: gets the filename of a `.fasta` file, reads it (**hint: biopython might help here**), prints the number of sequences contained, the id and size of **all** the biggest and smallest contigs (in case more than one have the same size), the mean and median size (see below) and **returns an array with the lengths of all sequences**.

Calling:

```
lens = loadData("sequences.fasta")
```

should give:

```
The file "sequences.fasta" contains 206 sequences

Longest contig(s): Contig_51 (len: 453,256)
Shortest contig(s):
Contig_54,Contig_55,Contig_84,Contig_95,Contig_102,Contig_145,Contig_150,Contig_171 (len:
133)

Mean size: 32694.55 base pairs
Median size: 2218.00 base pairs
```

A.2) computeNX

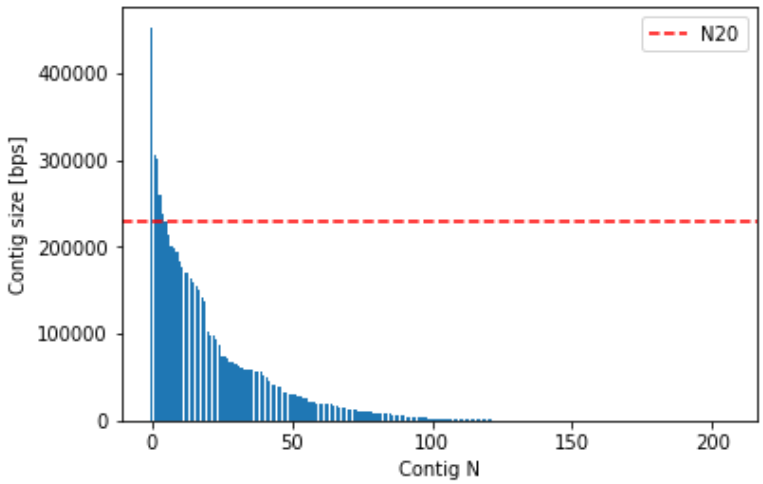
Implement `computeNX(data, X)`: gets the array of sizes (`data`) created by `loadData` and a double value (`X`) and computes the `NX` value (es. `N50` if `X` is `0.5`) providing a bar plot of all the sizes with the `NX` value highlighted with a red horizontal dashed line (**hint: you can use matplotlib's axhline function**). **Hint: if it is complicated to sort the array in descending order, you can sort it in ascending order and reverse it!**

Calling:

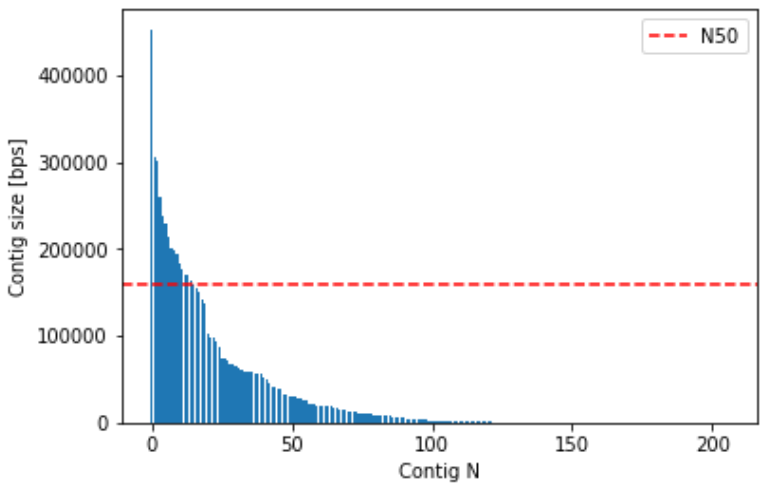
```
for x in [0.2, 0.5, 0.8, 0.9]:
    computeNX(lens,x)
```

should give something like:

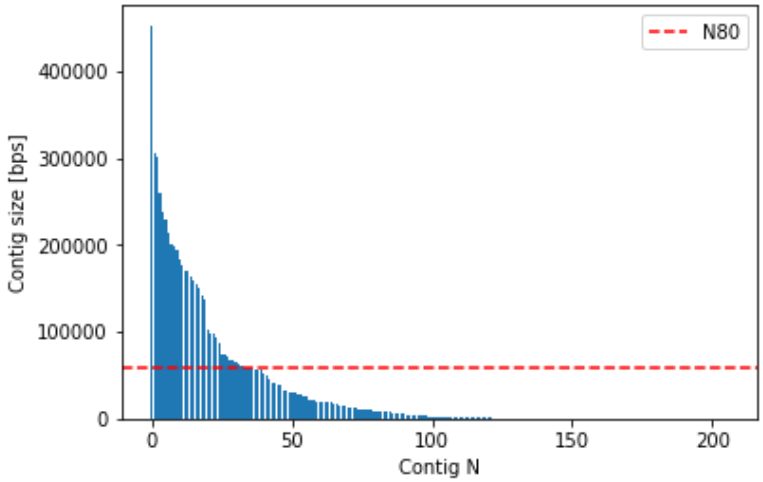
```
Total sequence size: 6,735,078 base pairs
N20: 229,794 base pairs
```



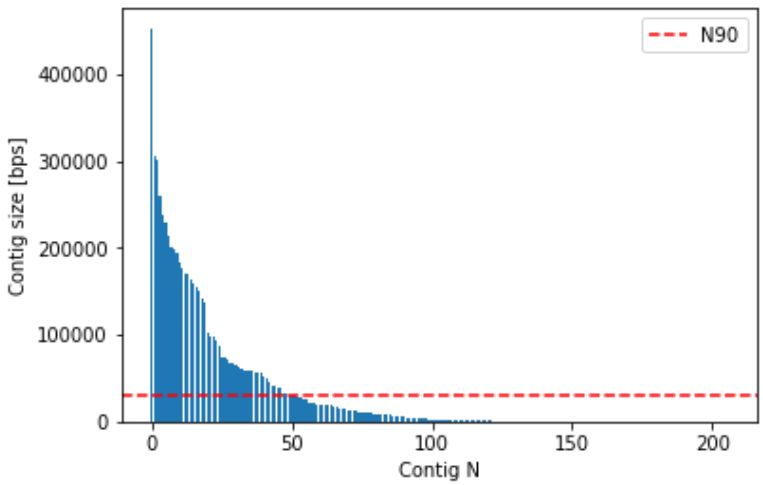

```
Total sequence size: 6,735,078 base pairs
N50: 159,971 base pairs
```




```
Total sequence size: 6,735,078 base pairs
N80: 58,495 base pairs
```




```
Total sequence size: 6,735,078 base pairs
N90: 29,226 base pairs
```



Part B

Takes 1.30h

B.1) stock

By Alberto Montresor.

Using a sophisticated Deep Learning algorithm, you are now able to predict the performance of the stock market (mercato azionario) over the next few days. You focus on the share price (prezzo delle azioni) of a single company. Every day, you can (1) buy a *single* share at the daily price, or (2) sell *all* the shares you have at the daily price, or (3) do nothing.

Write an algorithm `stock(V)` that takes a list containing `n` share prices for the next `n` days and returns the maximum profit that you may obtain by buying and selling shares during these days. Suppose you have enough money to buy single shares whenever you want.

Examples:

- `V = [1,7,3,6]`, the best actions are buy on day 0, sell on day 1, buy on day 2, sell on day 3, for a maximum profit `(7-1)+(6-3) = 9`
- `V = [7,1,2,5,3,1]`, the best actions are buy on days 1 and 2, sell on day 3, do nothing the other days, for a total profit `(5*2)-1-2=7`

Suggestion:

The best approach is to use dynamic programming or memoization. Let `DP[i][k]` be the maximum profit that can be obtained considering the days ranging from `i` to `n`, and having bought a total number of shares equal to `k` before the `i-th` day. The base case is when `i==n-1`, i.e. we are on the last day: the only action you can take is sell. The solution will be contained in in `DP[0][0]`.

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 stock(V):
    """ Take a list of positive integers as stock quotes, and return an integer.
        For more info see exam text and tests.
    """
```

Testing: `python3 -m unittest exerciseB1_test`

B.2)

You will be doing exercises about chainable lists, using plain old Python lists.

Start editing the file `exerciseB2.py` and read the following.

B.2.1) has_duplicates

Implement the function `has_duplicates`

```
def has_duplicates(external_list):
    """
        Returns True if internal lists inside external_list contain duplicates,
        False otherwise. For more info see exam and tests.

        INPUT: a list of list of strings, possibily containing repetitions, like:

            [
                ['ab', 'c', 'de'],
                ['v', 'a'],
                ['c', 'de', 'b']
            ]

        OUTPUT: Boolean  (in the example above it would be True)

    """
```

- **MUST RUN IN** $O(m * n)$, where m is the number of internal lists and n is the length of the longest internal list (just to calculate complexity think about the scenario where all lists have equal size)
- **HINT**: Given the above constraint, whenever you find an item, you cannot start another for loop to check if the item exists elsewhere - that would cost around $O(m^2 * n)$. Instead, you need to keep track of found items with some other data structure of your choice, which must allow fast read and writes.

Testing: `python3 -m unittest exerciseB2_test.TestHasDuplicates`

B.2.2) chain

Implement the function `chain`:

```
def chain(external_list):
    """
        Takes a list of list of strings and return a list containing all the strings
        from external_list in sequence, joined by the ending and starting strings
        of the internal lists. For more info see exam and tests.

        INPUT: a list of list of strings , like:

                [
                    ['ab', 'c', 'de'],
                    ['gh', 'i'],
                    ['de', 'f', 'gh']
                ]

        OUTPUT: a list of strings, like   ['ab', 'c', 'de', 'f', 'gh', 'i']
```

It is assumed that

- `external_list` always contains at least one internal list
- internal lists always contain at least two strings
- no string is duplicated among all internal lists

Output sequence is constructed as follows:

- it starts will all the items from the first internal list
- successive items are taken from an internal list which starts with a string equal to the previous taken internal list last string
- sequence must not contain repetitions (so joint strings are taken only once).
- *all* internal lists must be used. If this is not possible (because there are no joint strings), raise ValueError

Be careful that:

- **MUST BE WRITTEN WITH STANDARD PYTHON FUNCTIONS** (stuff like BioPython is *not* allowed ...)
- **MUST RUN IN** $O(m * n)$, where m is the number of internal lists and n is the length of the longest internal list (just to calculate complexity think about the scenario where all lists have equal size)

- **HINT**: Given the above constraint, whenever you find a string, you cannot start another for loop to check if the string exists elsewhere (that would likely introduce a quadratic m^2 factor) Instead, you need to first keep track of both starting strings and the list they are contained within using some other data structure of your choice, which must allow fast read and writes.
- if possible avoid slicing (which doubles memory usage) and use `itertools.islice` instead

Testing: `python3 -m unittest exerciseB2_test.TestChain`