Scientific Programming Algolab - Exam Tuesday 16, January 2018

Scientific Programming Labs, lab1 + lab2, Quantitative Computational Biology Master, CIBIO

Introduction

Allowed material

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

- 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. First part is by Luca Bianco (kmer 1.1,1.2,1.3), second exercise 2.1 is more about theory with dynamic programming, and will be assessed by Alberto Montresor, while the remaining ones (GenericTree 3.1 and 3.2) 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: * **Editra editor** is easy to use, you can find it

under *Applications->Programming->Editra*. * Others could be *GEdit* (simpler), or *PyCharm* (more complex).

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 2018-01-16-exam.zip and extract it *on your desktop*. Folder content should be like this:

```
2018-01-18
    |- FIRSTNAME-LASTNAME-ID
        |- exercise1.py
        |- exercise2.py
        |- exercise3.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.

In [1]:
```
#from exercise1 import *
#from exercise1_test import *
```

1) Kmers 1h 30m

1.0) The Problem

Fasta is a format for storing sequence information and the corresponding sequencing quality. A sample entry is the following:

```
>SRR190875.100000 HWI-ST180_0167:2:1:6207:136494
AGGCCTAGGTCTTCCAGAGTCGCTTTTTCCAGCTCCAGACCGATCTCTTCAG
```

where the first line is the identifier of the read and starts with a `>`. The sequence follows the line with the identifier **(in general it can be on several lines, but you can assume it is only on one line). HINT: Biopython's ``SeqIO.parse`` can read fasta formatted files.**

Kmers are substrings of length K of a given sequence. For example, 3mers are all substrings having length 3 of a given sequence. Assuming a sequence `seq = "AATAACTAGC"`, all possible 2mers of seq are: `AA,AT,TA,AC,CT,AG,GC`. Note that the 2mers `"AA"` and `"TA"` are present twice.

Start editing `exercise1.py`, implementing the required functions as in the next points.

1.1) `computeKmers(seq, kmerLen, kmerDict)`

Implement `computeKmers(seq, kmerLen, kmerDict)` : gets a sequence seq (a string), an integer specifying the size of the kmer and fills the dictionary kmerDictwith all the possible kmers of length kmerLen present in the sequence as keys and their multiplicity (i.e. the number of times they appear in the sequence) as value. Example: given the sequence `seq="AATAACTAGC"` the dictionary of 2mers is the following:

```
{'AA': 2, 'AG': 1, 'TA': 2, 'AT': 1, 'GC': 1, 'CT': 1, 'AC': 1}.
```

Note: if kmerLen is > len(seq) an empty dictionary should be returned. Calling

```
seq = "AATTAATTAACTAGCCTTAA"
twoMers = dict()
print("sequence:" , seq)
computeKmers(seq, 2, twoMers)
print("twoMers")
fourMers = dict()
computeKmers(seq, 4, fourMers)
print("4mers")
print(fourMers)
t2Mers = dict()
computeKmers(seq, 32, t2Mers)
print("32mers")
print(t2Mers)
```

should give:

```
sequence: AATTAATTAACTAGCCTTAA
{'AT': 2, 'AC': 1, 'CT': 2, 'TA': 4, 'AA': 4, 'CC': 1, 'TT': 3, 'AG': 1, 'GC': 1}
4mers
{'CTTA': 1, 'TAAT': 1, 'TAAC': 1, 'CTAG': 1, 'TAGC': 1, 'GCCT': 1, 'CCTT': 1, 'AACT': 1,
'ATTA': 2, 'TTAA': 3, 'ACTA': 1, 'AGCC': 1, 'AATT': 2}
32mers
{}
```

1.2) `parseFasta(fileName,kmerLen)`

Implement `parseFasta(fileName,kmerLen)` reads the fasta formatted file `fileName` and returns a dictionary with the all the kmers having length kmerLen and their multiplicities present in the file. The function should print the number of sequences read from the file and the number of distinct kmers. Example: if the file fileName contains the two sequences `"ATATCACATCTG"` and `"CTGACATATAT"` (with the respective sequence headers), the output dictionary of 4mers would be:

```
{'ATAT': 3, 'TCTG': 1, 'TCAC': 1, 'TGAC': 1, 'CATA': 1, 'GACA': 1, 'TATA': 1, 'TATC': 1,
'CATC': 1, 'ATCT': 1, 'CACA': 1, 'ATCA': 1, 'CTGA': 1, 'ACAT': 2}
```

Given the input files shortSample.fasta and contigs82.fasta (that is included in your working folder), calling:

```
myFile = "shortSample.fasta"
kmers = parseFasta(myFile,4)
print(kmers)
```

should give:

```
Read 2 sequences.
Total number of 4mers: 14
{'GACA': 1, 'ATCA': 1, 'TCTG': 1, 'TATC': 1, 'CTGA': 1, 'TATA': 1, 'ATCT': 1, 'ATAT': 3,
'CACA': 1, 'ACAT': 2, 'TGAC': 1, 'TCAC': 1, 'CATA': 1, 'CATC': 1}
```

while

```
myFile = "test_reads_75k.fasta"
kmers = parseFasta(myFile,17)
```

should print:

```
Read 75000 sequences.
Total number of 17mers: 4575380
```

1.3) `plotKmerSpectrum(kmers)`

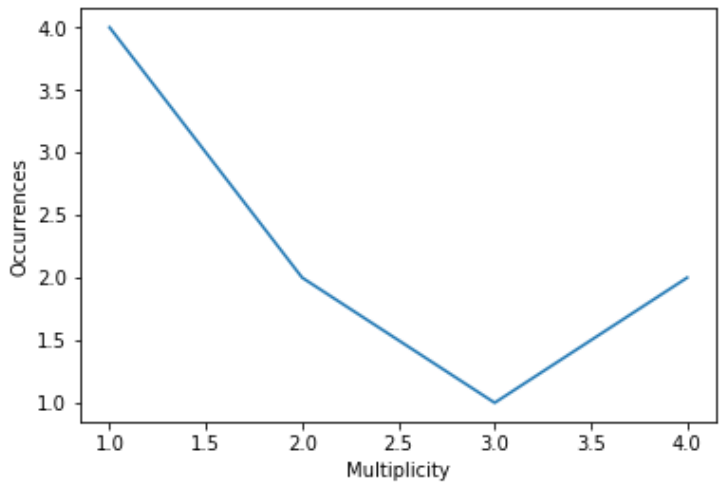
Implement `plotKmerSpectrum(kmers)`: given a dictionary kmers created as above (i.e. keys are kmer sequences and values are the number of occurrences of the kmer sequence) plots the kmer spectrum of all the kmers present in the dictionary. The kmer spectrum is a histogram of the multiplicities of all kmers. For example, given the dictionary

```
{'AC': 1, 'AG': 1, 'AA': 4, 'AT': 2, 'CT': 2, 'CC': 1, 'TA': 4, 'TT': 3, 'GC': 1}
```

a dictionary representing the kmer spectrum of these kmers is:

```
{1: 4, 2: 2, 3: 1, 4: 2}
```

which basically means that there are 4 2mers with multiplicity 1, 2 with multiplicity 2, 1 with multiplicity 3 and 2 with multiplicity 4. The plot for such a kmer spectrum should be:



Calling

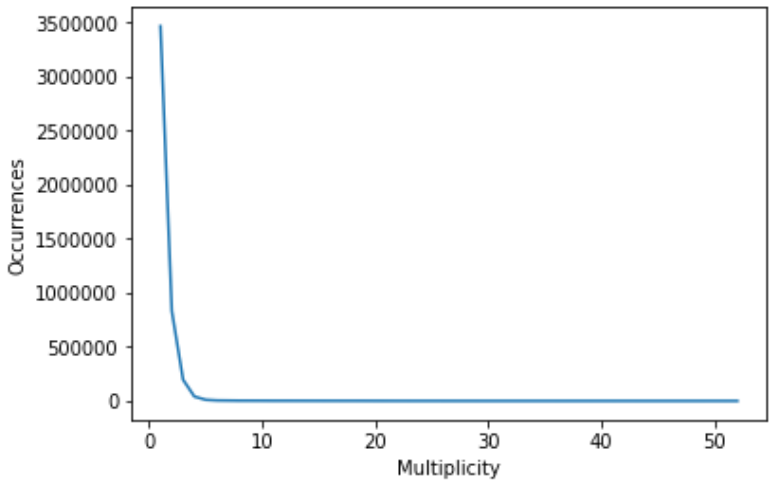
```

```

```
myFile = "test_reads_75k.fasta"
kmers = parseFasta(myFile,17)
plotKmerSpectrum(kmers)
```

should produce:

```
Read 75000 sequences.
Total number of 17mers: 4575380
```



2.1) Minimum Squares Sum (30 min)

Every natural number can be expressed as the sum of one or more squares. For example, `4=2^2`, `12=2^2+2^2+2^2` or `12=3^2+1^2+1^2`, `13=2^2+3^2`.

Write an algorithm that given an input value `n`, computes the minimum number of squares that need to be summed together to obtain `n`.

In the examples above, all the proposed sums are minimal: for examples, 13 can be expressed as `2^2+3^2`, but also as `2^2+2^2+2^2+1^2`. So the output for the value 13 should be equal to 2 (two squares).

The problem can be solved by writing a recursive formula `PD(n)` representing the number of squares needed to express `n` and then by transforming it in Python through dynamic programming or memoization.

Clearly, the bases cases are either 0 and 1, which needs 0 and 1 squares to be expressed.

The recursive case is as follows: in order to compute the number of squares for `n`, you need to try all the numbers `k` between 1 and `int(math.sqrt(n))` (both included), and subtract `k^2` from `n`. You recursively compute the number of squares needed to obtain `(n-k^2)`, and you add 1 because of `k^2`. You take the minimum between all these values.

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!

Start editing `exercise2.py`, implementing the function `squares`:

```
def squares(S):
    """ Given an integer n, return the minimum number of squares that
        need to be summed together to obtain n.
    """
    raise Exception("TODO IMPLEMENT ME !!!")
```

Testing

Once done, running this will run only the tests in `SquaresTest` class and hopefully they will pass.

Notice that ``exercise1`` is followed by a dot and test class name ``.SqauresTest`` :

```
python -m unittest exercise1.SquaresTest
```

3) GenericTree leaves (30 min)

3.1) `count_leaves`

Implement the method `count_leaves`:

```
def count_leaves(self):
    """  Return the number of leaves in the tree (= nodes with no children at the bottom)

        - Root node can never be considered a leaf
        - Must execute in O(n) where n is the number of nodes of the tree.
        - It is acceptable for this method to be implemented in a recursive way.
    """
```

Testing: `python -m unittest exercise3.CountLeavesTest`

Examples:

This tree has 3 leaves:

```
a
├b
│└c <-- leaf
└d
 └e
  ├f <-- leaf
  └g <-- leaf
```

Root is never a leaf, so this tree has zero leaves:

```
a
```

3.2) `detach_leaves`

Implement the method `detach_leaves`:

```
def detach_leaves(self):
    """ Detaches all the leaves from the tree (= nodes with no children at the bottom)

        - Root node can never be considered a leaf
        - Must execute in O(n) where n is the number of nodes of the tree.
        - In order to solve it, feel free to use the provided detach_child and
detach_sibling methods
        - It is acceptable for this method to be implemented in a recursive way.

    """
```

Testing: `python -m unittest exercise3.DetachLeavesTest`

This tree has 3 leaves:

```
a
├b
│└c <-- to remove
└d
 └e
  ├f <-- to remove
  └g <-- to remove
```

detaches leaves transforms it into this:

```
a
├b
└d
 └e
```

Root is never a leaf, so this tree would be left unchanged:

```
a
```