Scientific Programming Exam, Monday 05, February 2018

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

Download exercises

Introduction

Part A is by Luca Bianco:

- A.1 computeGeneStats
- A.2 printSequence

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

- B.1 subsetsum

Part B.2 is a LinkedQueue exercise by David Leoni

- B.2.1 enqn
- B.2.2 deqn

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.
- **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 sciprolab-2018-02-05-exam.zip and extract it *on your desktop*. Folder content should

be something like this:

```
sciprolab-2018-02-05
    |- 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 `gene_models.tsv` is a compact representation of the gene models on the corresponding genome (the genome sequence is present in the other file, `sequences.fasta`, in fasta format). The first few lines are reported below:

```
Chr     feature start   end     ID
Chr05   gene    50988   51101   gene:MD05G1000100
Chr05   ncRNA   50988   51101   ncRNA:MD05G1000100
Chr05   gene    83210   83329   gene:MD05G1000200
Chr05   ncRNA   83210   83329   ncRNA:MD05G1000200
Chr05   gene    87650   91333   gene:MD05G1000300
Chr05   mRNA    87650   91333   mRNA:MD05G1000300
Chr05   CDS     87650   87727   CDS:MD05G1000300.8
```

as the header (first line) describes, the first column is the chromosome containing the feature (that can be gene, `ncRNA`, `exon`, `CDS`, ...), the third and fourth columns are the start and end position of the feature on the chromosome and the fifth row is the identifier of the feature `ID`. The file `sequences.fasta` stores sequence information and the corresponding sequencing quality. A mock entry is the following:

```
>Chr01
AGGCCTAGGTCTTCCAGAGTCGCTTTTTCCAGCTCCAGACCGATCTCTTCAG
AGGCCAATCGCCAGTTTACCACATACACCCAGACCGATCTCTTCAG
```

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

A.1) computeGeneStats(filename)

Implement function `computeGeneStats(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 features of type gene, printing the average gene length in the whole file. The function should also plot a bar plot of the number of genes per chromosome.

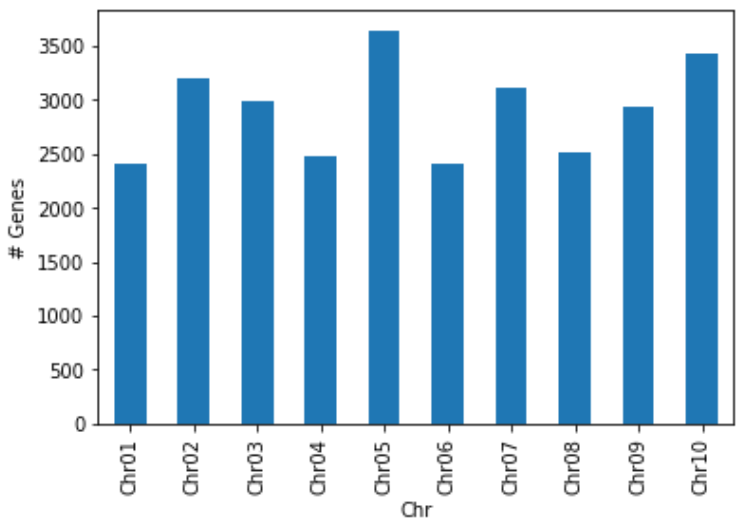
Note: The function should return the data structure containing all the data.

Calling:

```
fn = "gene_models.tsv"
seqFile = "sequences.fasta"
GenesDF = computeGeneStats(fn)
```

should give:

```
29136 genes present
Avg gene length: 2723.71
```



A.2) printSequence(geneInfo, geneID, sequenceFile)

Implement `printSequence(geneInfo, geneID, sequenceFile)`: gets the `geneInfo` data structure created by `computeGeneStats`, a gene identifier `geneID` and the filename of a fasta formatted file `sequenceFile` and if `geneID` is present in `geneInfo` it prints its length. The function must also retrieve the sequence of `geneID` (chromosome, start and end position are saved in `geneInfo`) if it is present in the fasta file `sequenceFile`. If the chromosome is not in the `sequenceFile` or `geneID` is not in `geneInfo` the function should display a corresponding message (see examples below).

Hint: you can use biophtyon to read a fasta file.

Examples

For example, assuming an entry:

```
ChrX   gene   1   10   gene:geneid1
```

in `geneInfo` and a sequence:

```
>ChrX
GATTACATAACACACTACA
```

in `sequenceFile`, calling `printSequence(geneInfo,"geneid1",sequenceFile)` should print:

```
Gene geneid1 is in ChrX and has length 9
>geneid1
GATTACATAA
```

Given `GenesDF` as returned by `computeGeneStats`, calling:

```
printSequence(GenesDF,"MD05G1027300",seqFile)
print("")
printSequence(GenesDF,"MD03G1000400",seqFile)
print("")
printSequence(GenesDF,"MD08G1000100",seqFile)
print("")
printSequence(GenesDF,"MD08G100019191",seqFile)
```

should give:

```
Gene MD05G1027300 is in Chr05 and has length 567
>MD05G1027300
GAAACAGAGATAAAAAGTATCTAAATTCACGCAAATTAACAATTTTCCCATTTCATAGAATCCAAAAGCAGATATATAATTCACCTTTCCCAGAT

Gene MD03G1000400 is in Chr03 and has length 188
>MD03G1000400
CGTATTATAGGAATTCTTCTTCCATTGAAAGGATGGATTGAAAGGATGGATGGCATGTTGTCTATACCTCTCGTCATTAGGATGTACAATAATGC

Gene MD08G1000100 is in Chr08 and has length 5383
Chr08 is not present in the sequence file.

GeneID MD08G100019191 is not present in the input file.
```

Part B

B.1) `SubsetSumTest`

By Alberto Montresor.

The problem

Let `A` be a list of non-negative (possibly repeated) integers, and `S` a target integer value. Your task is to decide whether there exists a subset of the integers in `A` whose sum is equal to the target value `S`. You need to write a function `subsetsum(A,S)` that returns `True` if there is such subset, `False` otherwise.

Suggestion: the problem is similar to the knapsack problem. You should compute a table entry for each pair `(i,s)`, where `DP[i][s]` should contain `True` if and only if it is possible to obtain `s` by summing a subset of the first `i` items.

You should express the problem based on two possibilities: - if you take the last element `A[i]`, the new target is `s-A[i]` and there are `i-1` elements left - if you don't take the last element `A[i]`, the target remain `s` and there are `i-1` elements left. If either of these cases is `True`, you should return `True`.

Pay attention to the base cases.

Example: if `A=[2,5,4,3,12]` and `S=9`, the answer is `True`; if `A=[2,5,4,3,12]` and `S=25`, the answer is `False`.

For plenty of other examples, see the test cases.

Implementation:

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

```
def subsetsum(A,S):
    """
    Let A be a list of non-negative integers, and S a target integer value.
    Return True if there exists a subset  of the integers in A whose sum is
    equal to the target value S. Otherwise, return False.

    NOTE:
      - if S = 0, always return True, even with empty array
      - numbers in A may be repeated
    """
```

Testing: `python3 -m unittest exerciseB1_test`

B.2) LinkedQueue

You are given a queue implemented as a LinkedList, with usual `_head` pointer plus additional `_tail` pointer and `_size` counter

- Data in enqueued at he right, in the tail
- Data is dequeued at the left, removing it from the head

Example, where the arrows represent _next pointers:

```
_head                          _tail
    a -> b -> c -> d -> e -> f
```

In this exercise you will implement the methods `enqn(lst)` and `deqn(n)` which respectively enqueue a python list of n elements and dequeue n elements, returning python a list of them.

Here we show an example usage, see to next points for detailed instructions.

Example:

```
q = LinkedQueue()

q.enqn(['a','b','c'])

Return nothing, queue becomes:

  _head         _tail
      a -> b -> c

q.enqn(['d'])

Return nothing, queue becomes:

  _head               _tail
      a -> b -> c -> d


q.enqn(['e','f'])

Return nothing, queue becomes:

  _head                          _tail
      a -> b -> c -> d -> e -> f

q.deqn(3)

Returns ['d', 'e', 'f'] and queue becomes:

  _head         _tail
      a -> b -> c

q.deqn(1)

Returns ['c'] and queue becomes:

  _head    _tail
      a -> b

q.deqn(5)

raises LookupError as there aren't enough elements to remove
```

B.2.1) `enqn`

Implement the method `enqn`:

```
def enqn(self, lst):
    """ Enqueues provided list of elements at the tail of the queue

        - Required complexity: O(len(lst))
        - NOTE: remember to update the _size and _tail

        Example: supposing arrows represent _next pointers:

      _head         _tail
          a -> b -> c

        Calling

        q.enqn(['d', 'e', 'f', 'g'])

        will produce the queue:

      _head                            _tail
          a -> b -> c -> d -> e -> f -> g
```

Testing: `python3 -m unittest exerciseB2_test.EnqnTest`

B.2.2) `deqn`

Implement the method `deqn`:

```
def deqn(self, n):
    """ Removes n elements from the head, and return them as a Python list,
        where the first element that was enqueued will appear at the *beginning* of
        the returned Python list.

        - if n is greater than the size of the queue, raises a LookupError.
        - required complexity: O(n)

        NOTE 1: return a list of the *DATA* in the nodes, *NOT* the nodes themselves
        NOTE 2: DO NOT try to convert the whole queue to a Python list for playing with
splices.
        NOTE 3: remember to update _size, _head and _tail when needed.


        For example, supposing arrows represent _next pointers:


      _head                            _tail
          a -> b -> c -> d -> e -> f -> g

        q.deqn(3) will return the Python list ['a', 'b', 'c']

        After the call, the queue will be like this:

      _head              _tail
          d -> e -> f -> g

    """
```

Testing: `python3 -m unittest exerciseB2_test.DeqnTest`