Scientific Programming 10/02/2021

Before you start

Please write one single python script for the lab part and one text file with the answers to the theoretical questions.
IMPORTANT: Add your name and ID (matricola) on top of the .py and text files!

Theory

Please write the solution in a text file.

1. Given a list L of n elements, please compute the asymptotic computational complexity of the following function, explaining your reasoning.

```
def my_fun(L):
   T = []
   N = len(L)
   for i in range(N):
      cnt = 0
      for j in range(N):
         if  L[i] > L[j]:
            cnt += 1
      T.append(cnt)

   return T
```

2. Briefly answer the following questions: what is the Tree data structure? What is a Binary Search Tree (BST)? What can we use a BST for?

Practical part (A)

1. Implement the **ExpressionAnalyzer** class that provides a small toolkit for RNA-seq gene expression analysis. Test it with the rawcounts.all.txt file provided. The class should provide methods to:

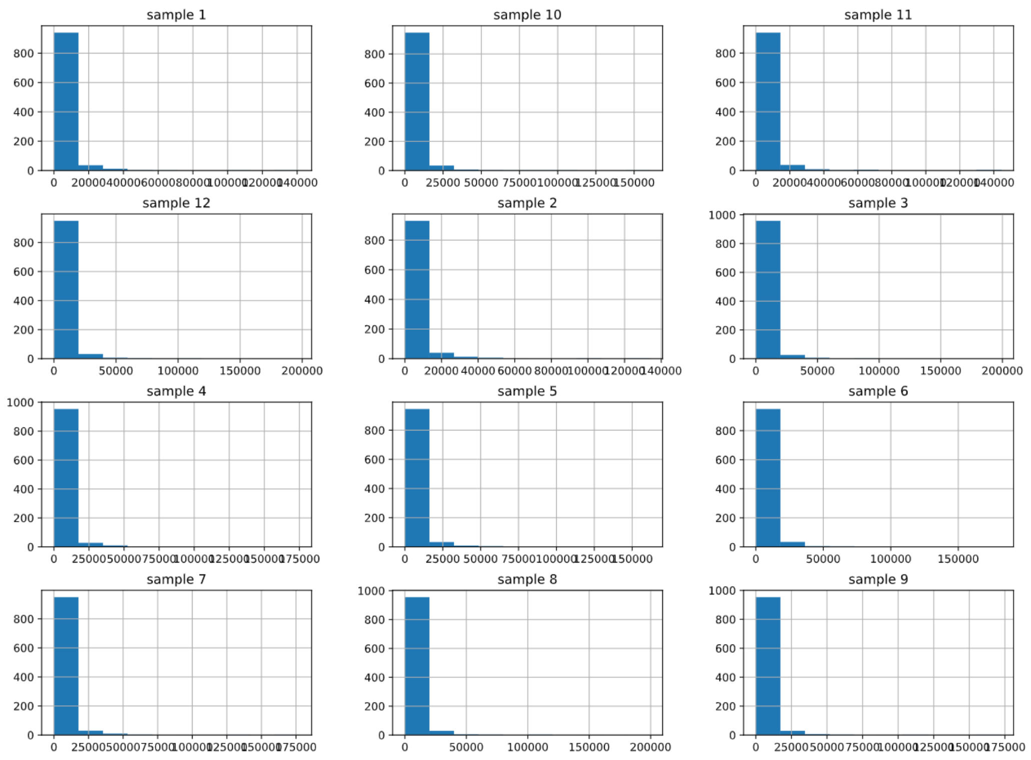
a. **Load** a tab-separated expression file (each row is a gene, each column a sample) from a user-provided path, and store the expression values, the list of genes, and the list of samples within the class (make those accessible!)

b. Compute **normalized gene expression** by obtaining the counts per million reads (**CPM**) value as:

$CPM(gene_i \text{ in } sample_j) = counts(gene_i \text{ in } sample_j) / ((\text{sum of sample counts}) / 1000000)$

Provide a method to access the CPM value for a specific gene in a specific sample.

c. Provide a **plot** function to display the distribution of read counts or CPMs according to user choice in all samples. The plot should look like the one below:

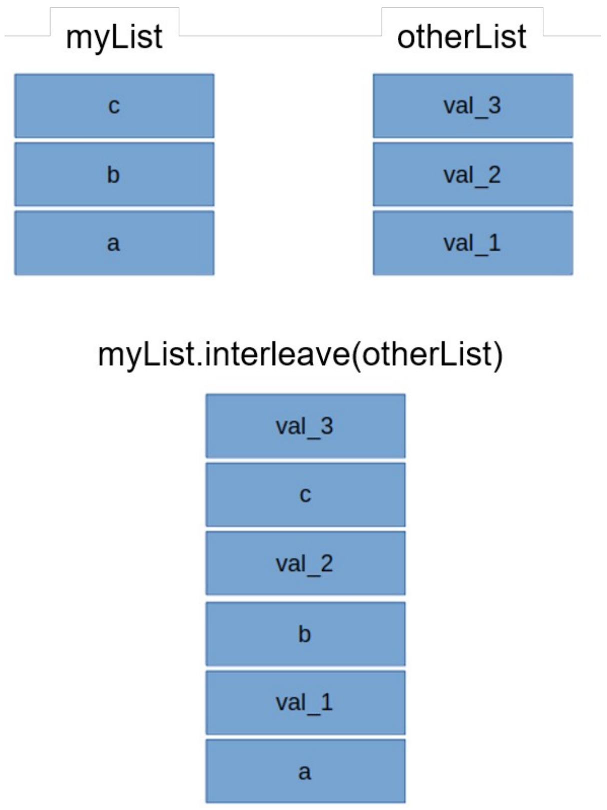


Practical part (B)

2. Recall linked lists as seen in class and in the lab. Extend the *BiLinkList* class provided in *BiLinkList.py* by creating a new class *ExtendedBLL* which implements the following methods:

a. **sortContent**(self) that returns a sorted list and prints its minimum and maximum value. The original list must not be changed at the end of this method.

b. **interleave**(self, otherBLL) that interleaves the elements of the list with the ones in otherBLL. See the following example:



A possible solution (courtesy of Massimiliano Luca)

#### PART A

```
import pandas as pd
import matplotlib.pyplot as plt
import argparse

# 1. Implement the ExpressionAnalyzer class:

class ExpressionAnalyzer():

def __init__(self, path):
# private variables where we will store the expressions,
# the genes and the samples
self._data = None
self._genes = None
self._samples = None

self._normalized_data = None

self._path = path

# functions to make the private variables accessible
```

```
def get_expressions(self):
return self._data

def get_genes(self):
return self._genes

def get_samples(self):
return self._samples

# A.1: a function to load the data and store some information
# related to the expressions, genes and samples.
def load(self):
self._data = pd.read_table(self._path)
self._genes = self._data['Gene'].to_list()
# exclude the column named 'Gene'
self._samples = self._data.columns.to_list()[1:]

# A.2: compute the CPM and provide a method to access a specific gene and sample

def get_cpm(self, gene, sample):
# if it is the first time we are looking for a cpm, we compute all the
# cpms. Alternatively, you may call the method to compute the cpm
# in the main after you initialized your ExpressionAnalyzer instance and
# you loaded the data
if self._normalized_data is None:
self._compute_cpm()

# check if we are dealing with a valid gene and a valid sample
if gene not in self.get_genes():
raise Exception(gene + ' is not a valid gene')
if sample not in self.get_samples():
raise Exception(sample + ' is not a valid sample')

return self._normalized_data[self._normalized_data['Gene'] == gene][sample].values[0]

def _compute_cpm(self):
temp = {}
# add the 'Gene' column in the normalized dataset to access it later
temp['Gene'] = self.get_genes()
for i in self.get_samples():
temp[i] = self._data[i] / (sum(self._data[i])/1000000)
self._normalized_data = pd.DataFrame(temp)

# A.3 a plot function that plot the distribution of normalized or non-normalized
# data according to the choice of the user (is_normalized).
def plot(self, is_normalized):
if is_normalized:
for i in self.get_samples():
self._normalized_data[i].hist()
plt.title(i)
plt.show()
else:
for i in self.get_samples():
self._data[i].hist()
plt.title(i)
plt.show()



if __name__ == "__main__":
parser=argparse.ArgumentParser()
parser.add_argument("--fname", type = str, help = "name of the file")
args = vars(parser.parse_args())
path=args['fname']

my_instance = ExpressionAnalyzer(path)
my_instance.load()
print(my_instance.get_cpm('TSPAN6', 'sample 1'))
my_instance.plot(is_normalized=False)
```

#### PART B

```
class Node:
def __init__(self, data):
self.__data = data
self.__prevEl = None
self.__nextEl = None

def getData(self):
return self.__data

def setData(self, newdata):
self.__data = newdata

def setNext(self, node):
self.__nextEl = node

def getNext(self):
return self.__nextEl

def setPrev(self,node):
self.__prevEl = node
def getPrev(self):
return self.__prevEl

def __str__(self):
```

```
return str(self.__data)
#for sorting
def __lt__(self, other):
return self.__data < other.__data


class BiLinkList:
def __init__(self):
self.__head = None
self.__tail = None
self.__len = 0

def __len__(self):
return self.__len
def min(self):
return self.__minEl
def max(self):
return self.__maxEl

def append(self,node):
if type(node) != Node:
raise TypeError("node is not of type Node")
else:
if self.__head == None:
self.__head = node
self.__tail = node
else:
node.setPrev(self.__tail)
self.__tail.setNext(node)
self.__tail = node

self.__len += 1

def insert(self, node, i):
# to avoid index problems, if i is out of bounds
# we insert at beginning or end
if i > self.__len:
i = self.__len #I know that it is after tail!
if i < 0:
i = 0
cnt = 0
cur_el = self.__head
while cnt < i:
cur_el = cur_el.getNext()
cnt += 1
#add node before cur_el
if cur_el == self.__head:
#add before current head
node.setNext(self.__head)
self.__head.setPrev(node)
self.__head = node
else:
if cur_el == None:
#add after tail
self.__tail.setNext(node)
node.setPrev(self.__tail)
self.__tail = node
else:
p = cur_el.getPrev()
p.setNext(node)
node.setPrev(p)
node.setNext(cur_el)
cur_el.setPrev(node)

self.__len += 1

def getAtIndex(self, i):
if i > self.__len:
return None
else:
cnt = 0
cur_el = self.__head
while cnt < self.__len:
if cnt == i:
return cur_el
else:
cnt += 1
cur_el = cur_el.getNext()

def iterator(self):
cur_el = self.__head
while cur_el != None:
yield cur_el
cur_el = cur_el.getNext()

def __str__(self):

if self.__head != None:
dta = str(self.__head)
cur_el = self.__head.getNext()
while cur_el != None:
dta += " <-> " + str(cur_el)
cur_el = cur_el.getNext()

return str(dta)
else:
return ""

def remove(self, element):
if self.__head != None:
```

```
cur_el = self.__head
while cur_el != element and cur_el != None:
cur_el = cur_el.getNext()

if cur_el != None:
p = cur_el.getPrev()
n = cur_el.getNext()

if cur_el == self.__head:
self.__head = n

if cur_el == self.__tail:
self.__tail = p

if n != None:
n.setPrev(p)
if p != None:
p.setNext(n)

self.__len -= 1


def slice(self, x, y):
m = min(x,y)
M = max(x,y)

if m > self.__len:
return None
else:
cur_el = self.__head
cnt = 0
while cnt < m:
cur_el = cur_el.getNext()
cnt += 1
nList = BiLinkList()

while cnt < M and cur_el != None:
n = Node(cur_el.getData())
cur_el = cur_el.getNext()
nList.append(n)
cnt += 1
return nList




class ExtendedBLL(BiLinkList):
```

```
def sortContent(self):
# simpler way: retrieve the values and store them in a temp list, sort it and recreate the
list
temp = []
for i in self.iterator():
temp.append(i.getData())
temp.sort()

# minimum is the first element and maximum is the last
print('Minimum ', temp[0])
print('Maximum ', temp[-1])

# create a new Extended BLL
result = ExtendedBLL()
for i in temp:
result.append(Node(i))
return(result)


def interLeave(self,otherBLL):
# goal is to modify the current list as pictured in the last page of the exam
i, j = 0, 0
while j < len(self) and i < len(otherBLL):
data = otherBLL.getAtIndex(i).getData()
self.insert(Node(data),j)
i += 1
j += 2

while i < len(otherBLL):
self.append(Node(otherBLL.getAtIndex(i).getData()))
i += 1




'''
'''
import random
MLL = BiLinkList()
for i in range(1,50,10):
n = Node(i)
MLL.append(n)
print(MLL)
for el in MLL.iterator():
print("\t{} prev:{} next:{}".format(el,
el.getPrev(),
el.getNext()))
```