QCB 1st exam test, January 22nd, 2024

Download the data

1. Consider the file sciprog-qcb-2024-01-22-FIRSTNAME-LASTNAME-ID.zip and extract it on your desktop.

2. Rename sciprog-qcb-2024-01-22-FIRSTNAME-LASTNAME-ID folder:

   Replace **FIRSTNAME**, **LASTNAME**, and **ID** with your first name, last name and student id number. **Failure to comply with these instructions will result in the loss of 1 point on your grade.**

   like sciprog-qcb-2024-01-22-alessandro-romanel-432432

   From now on, you will be editing the files in that folder.

3. Edit the files following the instructions.

4. At the end of the exam, **compress** the folder in a zip file

   sciprog-qcb-2024-01-22-alessandro-romanel-432432.zip

   and submit it. This is what will be evaluated. Please, include in the zip archive all the files required to execute your implementations!

NOTE: You can only use the data structures and packages provided in the exam script files. **Importing other Python packages IS NOT allowed** unless explicitly stated in the exam instructions. Using Python collections or other libraries will impact your final grade. Still, **IT IS ALLOWED** to use **built-in Python operators** as we have done during the practical classes (max, min, len, reversed, list comprehensions, etc).

Exercise 1 [FIRST MODULE]

1. Implement a small toolkit for basic RNA-seq gene expression analysis, and test it on the *rawcounts.all.txt* file provided. You must provide the following functions:

a. **loadReadCounts(file, minGeneReadCount = -1, minSampleReadCount = -1):** loads a tab-separated file containing read counts (each row is a gene, each column a sample) from a user-provided path, and extract the read count values, the list of genes, and the list of samples in an appropriate object. This object must then be returned by the function as output. If the *minGeneReadCount* parameter is set (i.e. > 0), filter to keep only genes with at least *minGeneReadCount* reads in total (sum of all samples). If *minSampleReadCount* is set (i.e. > 0), filter to keep only those samples having at least *minSampleReadCount* in total. The content of the output object must reflect the application of these two filters.

   The output of this function must be usable as the input of the following additional functions to be implemented:

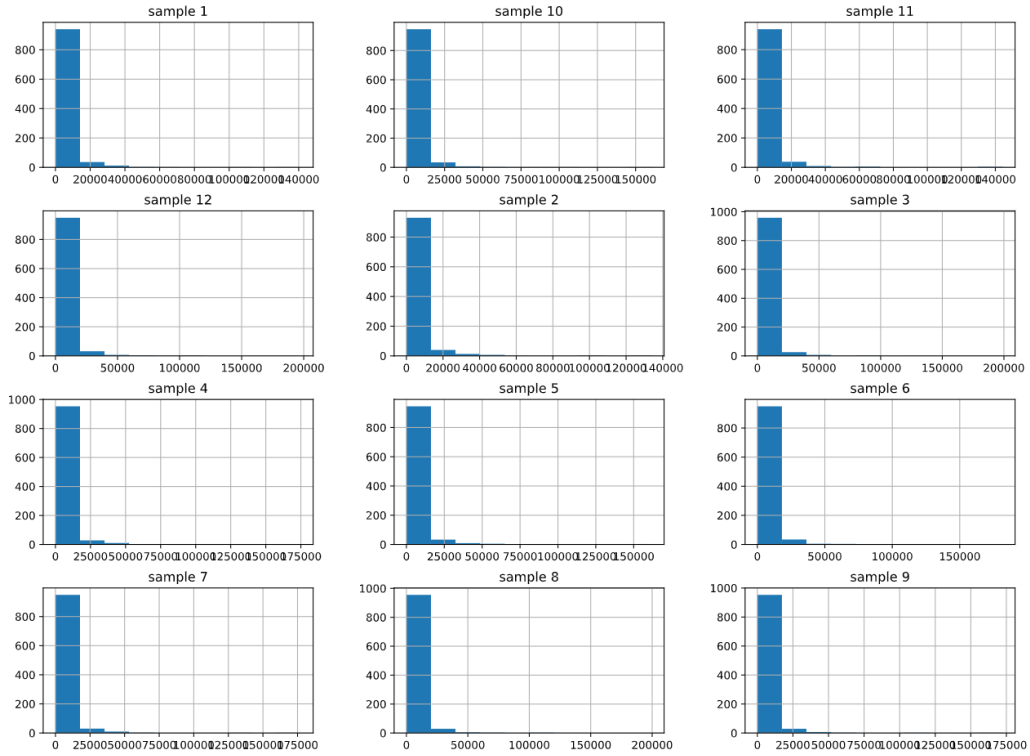
b. **getStableGenes(counts, n=10)**: identifies the *n* genes, present in the provided *counts* object, that have the lowest variance across all samples while also having non-zero counts (computed as sum of all samples). Print the genes name and their associated variance, and return their read counts in an appropriate object as output of the function.

c. **normalizeReadCounts(counts):** compute normalized gene expression values by obtaining the counts per million reads (**CPM**) values as:

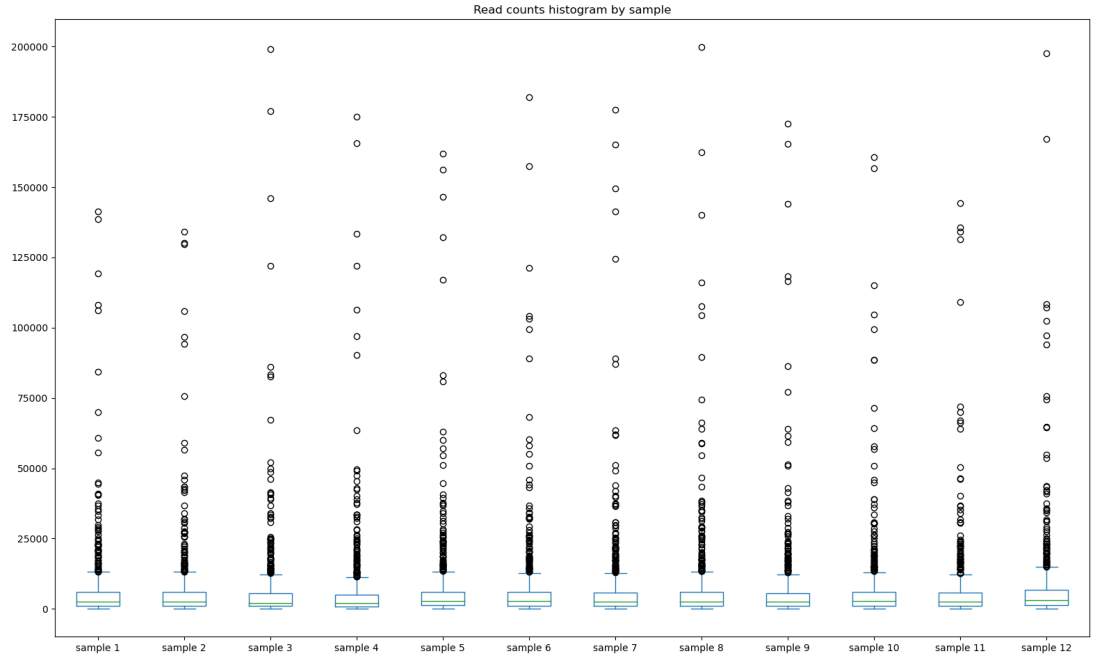
   $CPM(gene_i \text{ in } sample_j) = counts(gene_i \text{ in } sample_j) / ((\text{sum of } sample_j \text{ counts}) / 1000000)$

   Return the obtained normalized counts in an appropriate object that can be used with the *getReadCountsStats* and *plotCounts* functions.

d. **plotCounts(counts, hist=True):** function to display the distribution of read counts as histogram (if *hist* = *True*) or as boxplot (if *hist* = *False*) in all samples. In the first case, the plot should look like the one below:



If *hist* = *False* it should instead look like:



In both cases, the function should detect, and indicate in the plot title, whether the provided input data to the function consists of read counts or normalized read counts (CPMs).

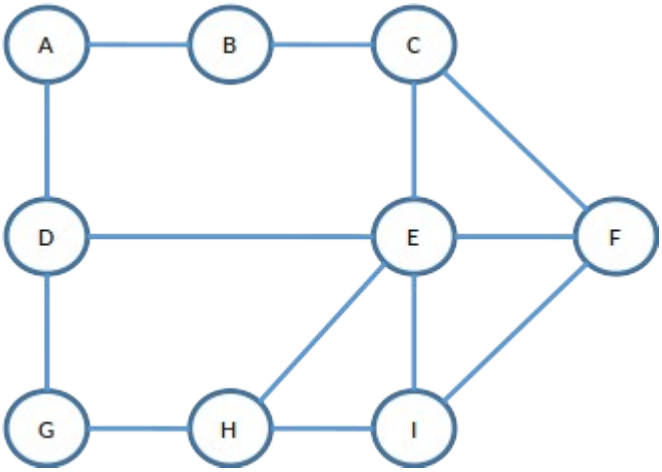
Exercise 2 [SECOND MODULE, theory]

Given a list *L* of *n* elements, a value *v*, a value *low* equal to 0 and value *high* equal to *n-1*, please compute the asymptotic computational complexity of the following function, explaining your reasoning.

```
def func(L, low, high, v):
  if high >= low:
    mid = (high + low) // 2
    if L[mid] == v:
      return mid
    elif L[mid] > v:
      return func(L, low, mid - 1, v)
    else:
      return func(L, mid + 1, high, v)
  else:
    return -1
```

Exercise 3 [SECOND MODULE, theory]

Describe the differences between the Depth-First and the Breadth-First Search algorithms for visiting graphs. Then, apply BFS to the graph below.

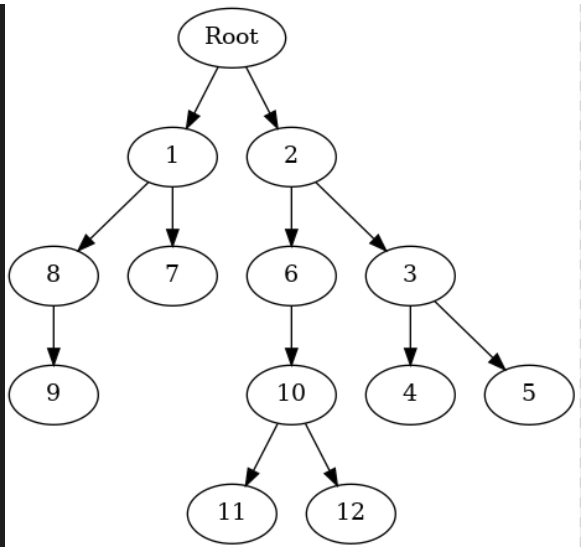


Exercise 4 [SECOND MODULE, practical]

Consider the `BinaryTree` implementation provided in the file *exercise4.py*. Complete the code where required:

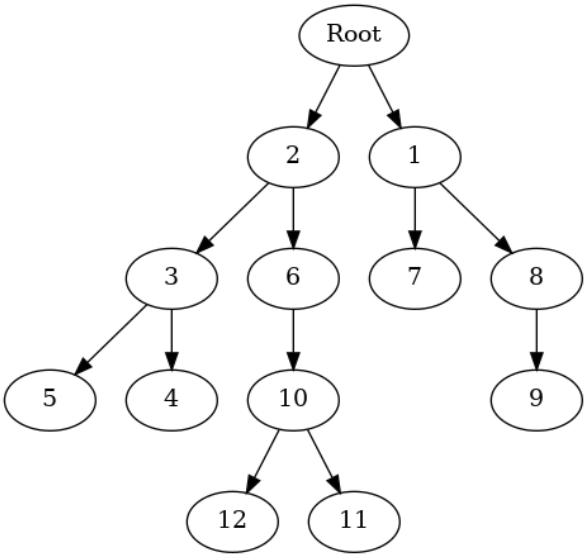
1. `mirror_binary_tree` is a function that takes as input a `BinaryTree` and mirrors the tree itself (each left and right nodes are mirrored).
Given the `BinaryTree`:

```
Root: Root
    L-- 1
        L-- 8
            R-- 9
        R-- 7
    R-- 2
        L-- 6
            R-- 10
                L-- 11
                R-- 12
        R-- 3
            L-- 4
            R-- 5
```



you should obtain:

```
Root: Root
    L-- 2
        L-- 3
            L-- 5
            R-- 4
        R-- 6
            L-- 10
                L-- 12
                R-- 11
    R-- 1
        L-- 7
        R-- 8
            L-- 9
```



2. *get_width_and_sum* a function that takes as input a *BinaryTree* and returns a list of tuples, one for each level of the tree, containing three values:

- the level number (starting with root as 0);
- the width of the level (number of nodes in that level);
- the sum of the values of the nodes of that level.