DS 1st exam test, January 19th, 2023

Download the data

1. Consider the file sciprog-ds-2023-01-19-FIRSTNAME-LASTNAME-ID.zip and extract it on your desktop.

2. Rename sciprog-ds-2022-01-19-FIRSTNAME-LASTNAME-ID folder:

   put your name, lastname an id number

   like sciprog-ds-2023-01-19-luca-marchetti-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-ds-2023-01-19-luca-marchetti-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!

Exercise 1 [FIRST MODULE]

Calls.csv is a dataset of calls between students, the dataset contains the timestamp (in seconds) representing when the call happened. Then it has the id of the caller and the id of the callee, finally, it contains the duration of the call. (if duration equals -1 then the callee did not answer the phone)

1) Load the dataset **calls.csv** stored in the DATA folder.

2) A duration of -1 implies that the callee did not answer the phone. Search for the couple that has the maximum number of missing calls. Build a function that takes in inputs the calls, and searches for the maximum number of missing calls between two users. The function MUST return the id of the two users:

```
def max_nb_missing_calls(calls):
        …
        …
        return user_A,user_B
```

3) Build a function that takes in input all the calls, the id of the caller, the id of the callee and a min_dur. You have to count how many times the caller called the callee that has a duration greater or equal to the minimum duration. The function has to return the number of calls. **NOTE**: the default value for min_dur must be set to 10

```
def count_calls(calls,caller,callee,min_dur):
        …
        …
        return number_of_calls
```

4) Build a function that normalizes the durations from 0 to 1. i.e. the shortest call became equal to duration 0, while the maximum duration became equal to 1.

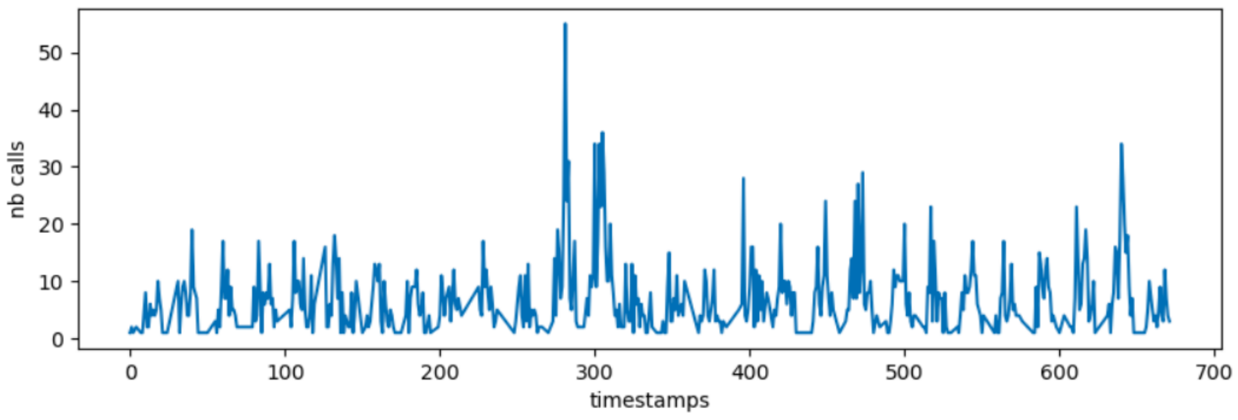
Formally: $X_{SCALED} = \frac{X - X_{min}}{X_{max} - X_{min}}$.

NOTE: do not include missed calls (-1) in the normalization!
The output of this function should be a list of scaled durations (from 0 to 1) but with -1 for missed calls.

```
def normalize_durations(calls):
        …
        …
        return list_of_durations
```

5) The column timestamp is given in seconds, you have to discretize it! Each interaction appearing in 3600 seconds has to be discretized into 1. Then you have to plot the time series. (to count the number of interactions happening in the discretized series, you can use np.unique(array,return_counts=True).
NOTE: i) set the size of the figure equal to 3,10 [plt.figure(figsize=(10,3))]. ii) set the x label with “timestamps” iii) set the y label with “nb calls”.

You should obtain a figure like the one bellow:



Exercise 2 [SECOND MODULE, theory]

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

```
def fun2(L, low, high, v):
          if high >= low:
               m = (high + low) // 2
               if L[m] == v:
                    return mid
               elif L[m] > v:
                    return fun2(L, low, m - 1, v)
               else:
                    return fun2(L, m + 1, high, v)
          else:
               return -1

def fun(L,v):
          n = len(L)
          return fun2(L,0,n,v)
```

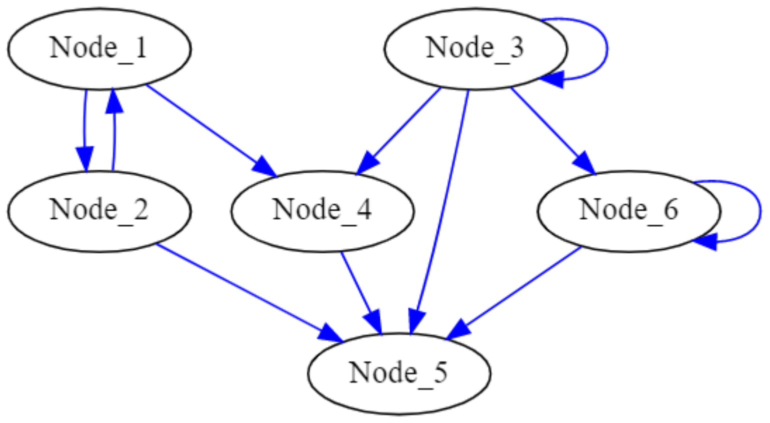
Exercise 3 [SECOND MODULE, theory]

Consider a hash table with separate chaining (implemented as mono-directional linked list) to handle collisions. The keys 14, 16, 4, 5, 35 and 18 are inserted into an initially empty hash table of length 7 using the hash function h(k) = k%7 (modulo operator). What is the resulting hash table?

Exercise 4 [SECOND MODULE, practical]

Consider the *DiGraphAsAdjacencyMatrix* class provided in the file **exercise4.py** implementing a directed **graph** by adjacency **matrix**.

The graph structure is shown below:



Please **check carefully how the class is implemented** since it might slightly differ from the one you have seen during the practical class.

Implement the two missing methods:

1. *checkSelfEdge(self, node):*
   This method checks whether a node has a "self" edge, meaning an edge pointing to itself (e.g. node 3 and 6). This method returns a boolean value.

   HINT: Remember to make sure that the node you are checking for a self edge is actually in the list of nodes.

2. *getTopAvgWeights_incoming(self):*
   This method first calculates the average of the weights of the incoming edges for **each** node. Second, it finds the node/nodes with the highest average weights and returns it/them (i.e. this method returns a list).

   HINT: Please remember that a "0" in the matrix means no edge and hence you do not want to include zeros in the calculation of the average. Also keep it mind that you might have a tie, meaning two or more nodes with the same average.