QCB Midterm Module 2, December 19th, 2023

Download the data

1. Consider the file sciprog-qcb-2023-12-19-FIRSTNAME-LASTNAME-ID.zip and extract it on your desktop.

2. Rename sciprog-qcb-2023-12-19-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-2023-12-19-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

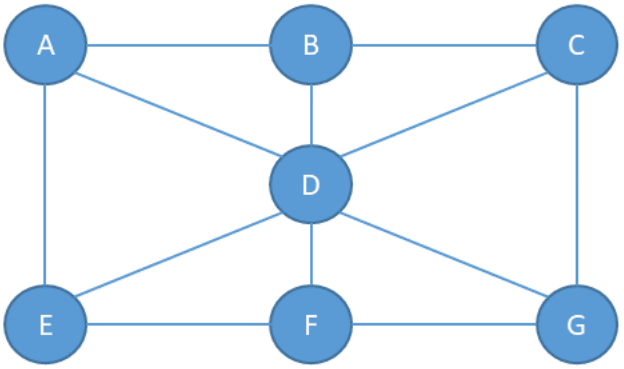
   sciprog-qcb-2023-12-19-alessandro-romanel-432432.zip

   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 [Theory]

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



Exercise 2 [Theory]

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

```
def fun(L):
    if len(L)== 1:
        return L[0]
    return max(L[0],fun(L[1:]))
```

Remember that the slicing operator creates a new list to store the sliced elements.

Exercise 3 [Lab]

Implement the *sort()* method of the class *PancakeSort* in the file exercise3.py . To test the implementation, execute the file exercise3.py, which is already equipped with a main code that tests the method by sorting the list [7, 5, 10, -11, 3, -4, 99, 1].

Imagine you have a stack of pancakes of different sizes, and you want to arrange them in order from the largest at the bottom to the smallest at the top. Here's how Pancake Sort works:

Start with your stack of unsorted pancakes. The stack represents the list of numbers that you want to sort (it is not an actual stack data structure).

1. Iterate over the stack of pancakes (list of numbers) starting from the **bottom** pancake (end of the list) and **going towards the top** (first pancake). This because at the end of each iteration the biggest pancake (number) will be moved to the bottom of the stack (last position in the list), and we want to reduce the size of the

list we work with at each iteration.

2. **At each iteration, identify the biggest pancake** (number) and get its position X.

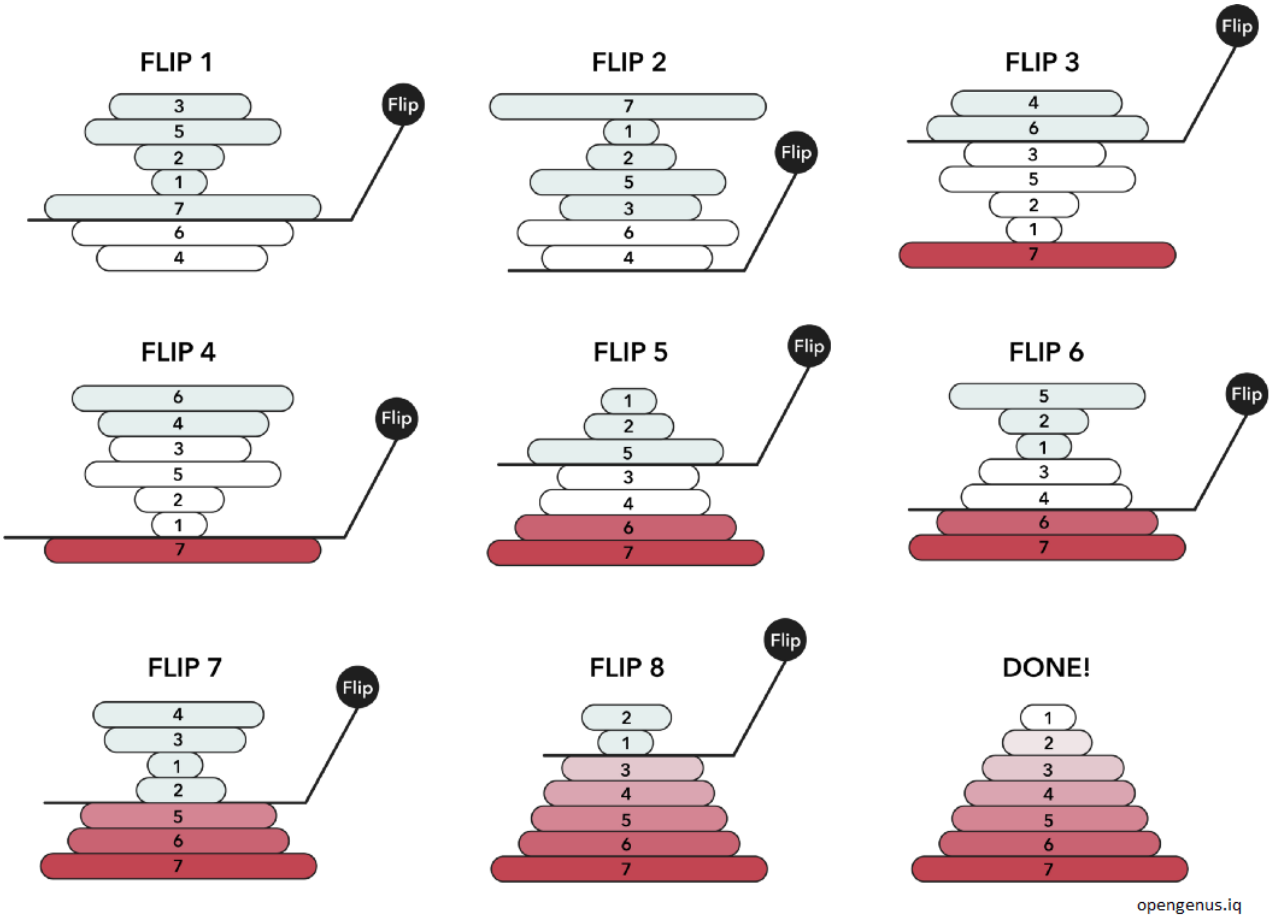
3. If the biggest pancake is not already at the top, **flip the unsorted stack** to move it there (in the first position of the list). This means that you must flip the sub-list that goes from 0 to position X.

4. **Now, flip the entire stack (list)** so that the biggest pancake is now at the bottom (end of the list).

5. Repeat the process, focusing on a smaller stack each time (excluding the pancake you've already sorted, which is now at the end).

6. Continue until all pancakes are in order (i.e. when you reach the first position in the list)

Keep in mind that with **flip** we mean to **reverse the order of the elements in the list** (or sub-list). Below there is an example of how the sorting works:

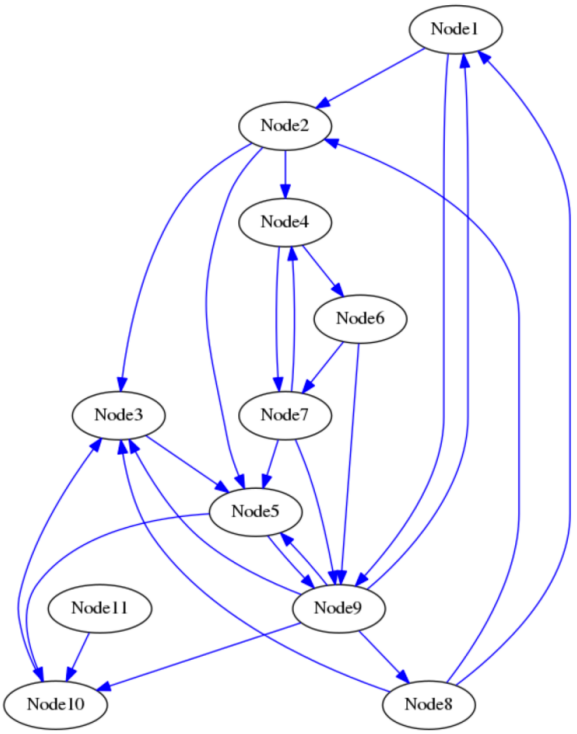


Exercise 4 [Lab]

Consider the *DiGraphAsAdjacencyMatrix* class provided in the file exercise4.py implementing a directed graph by adjacency matrix. Implement the two missing methods:

1. *getNumberOfConnectedNodes(mynode, modality)* : that returns the number of nodes that connect to the node *mynode*, if *mynode* exists. The method supports two modalities: when modality = 1, only incoming edges are considered, when modality = 2, only outgoing edges are considered. The code should print an error message and return 0 when *mynode* does not exist or when the method modality is not supported (values different from 1 and 2).

2. *getAverageNumberOfEdgesPerNode()* : that returns the average number of edges per node in the graph. To implement this method, you are allowed to use the method at point 1 if this helps. To test the implementation, execute the file exercise4.py, which already has a main code and tests the methods on the following graph:



If the two methods are properly implemented, the execution of the file will finally return:

```
PART 1:


-> modality 1 test:
        Node_3 has 4 incoming links.
        Node_7 has 2 incoming links.
        Node_11 has 0 incoming links.

-> modality 2 test:
        Node_3 has 1 outgoing links.
        Node_7 has 3 outgoing links.
        Node_11 has 1 outgoing links.

-> invalid cases:
ERROR: Node_27 does not exist
        Node_27 has 0 outgoing links.
ERROR: Node_27 does not exist
        Node_27 has 0 outgoing links.

PART 2:

Average number of edges per node in the graph: 4.545454545454546
```