Algolab Exam

Scientific Programming Module 2
Algorithms and Data Structures

Thusday 16th, Feb 2017

Introduction

- Taking part to this exam erases any vote you had before, both lab and theory
- If you don't ship or you don't pass this lab part, you lose also the theory part.

- Log into your computer in *exam mode*, it should start Ubuntu
- To edit the files, you can use any editor of your choice: *Editra* seems easy to use, you can find it under *Applications->Programming->Editra*. Others could be *GEdit* (simpler), or *PyCharm* (more complex).

Allowed material

There won't be any internet access. You will only be able to access:

- Sciprog Algolab worksheets (index.html)
- Alberto Montresor slides (../montresor/Montresor%20sciprog/cricca.disi.unitn.it/montresor/teaching/scientific-programming/slides/index.html)
- Stefano Teso docs (../teso/disi.unitn.it/_teso/courses/sciprog/index.html)
- Python 2.7 documentation : html (../python-docs/html/index.html) pdf (../python-docs/pdf)
  - In particular, Unittest docs (../python-docs/html/library/unittest.html)
- The course book *Problem Solving with Algorithms and Data Structures using Python* html (../pythonds/index.html) pdf (../pythonds/ProblemSolvingwithAlgorithmsandDataStructures.pdf)

Grading

- The grade of this lab part will range from 0 to 30. Total grade for the module will be given by the average with the theory part of Alberto Montresor.
- Correct implementations with the required complexity grant you full grade.
- 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.
- One bonus point can be earned by writing stylish code. You got style if you:
  - do not infringe the Commandments (../algolab/index.html#Commandments)
  - write pythonic code (http://docs.python-guide.org/en/latest/writing/style)
  - avoid convoluted code like i.e.

    ```
    if x > 5:
        return True
    else:
        return False
    ```

    when you could write just

    ```
    return x > 5
    ```

> **!!!!!!!!! WARNING !!!!!!!!!**
>
> **!!!!!!!!! **ONLY** IMPLEMENTATIONS OF THE PROVIDED FUNCTION SIGNATURES WILL BE EVALUATED !!!!!!!!!**

For example, if you are given to implement:

```
def cool_fun(x):
        raise Exception("TODO implement me")
```

and you ship this code:

```
def cool_fun_non_working_trial(x):
    # do some absurdity

def cool_fun_a_perfectly_working_trial(x):
    # a super fast, correct and stylish implementation

def cool_fun(x):
    raise Exception("TODO implement me")
```

We will assess only the latter one `cool_fun(x)`, and conclude it doesn't work at all :P !!!!!!!

Still, you are allowed to define any extra helper function you might need. If your `cool_fun(x)` implementation calls some other function you defined like `my_helper` here, it is ok:

```
def my_helper(y,z):
    # do something useful

def cool_fun(x):
    my_helper(x,5)

# this will get ignored:
def some_trial(x):
    # do some absurdity
```

What to do

In /usr/local/esame (/usr/local/esame) you should find a file named `algolab-17-01-26.zip`. Download it and extract it on your desktop. The content should be like this:

```
algolab-17-01-26
    |- FIRSTNAME-LASTNAME-ID
        |- exercise1-slow.py
        |- exercise1-fast.py
        |- exercise2.py
        |- exercise3.py
```

2) Check this folder also shows under `/var/exam`.

3) 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.

4) Edit the files following the instructions in this worksheet for each exercise.

> **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.**

> **IMPORTANT: Pay close attention to the comments of the functions.**

> **IMPORTANT: 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, i.e. avoid stuff like this: `print 1/0`**

3) Every exercise should take max 25 mins. If it takes longer, leave it and try another exercise.

> **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**

1) BoolStack

You are given a class `BoolStack` that models a simple stack. This stack is similar to the `CappedStack` you already saw in class, the only differences being:

- it can only contain booleans, trying to put other type of values will raise a `ValueError`
- trying to `pop` or `peek` an empty stack will raise an `IndexError`
- there is no cap

To create a `BoolStack`, just call it:

In [6]:

```
bs = BoolStack()
print bs
```

```
BoolStack:   elements=[]
```

In [7]:

```
bs.push(True)
```

In [8]:

```
print bs
```

```
BoolStack:   elements=[True]
```

In [9]:

```
bs.push(False)
```

In [10]:

```
print bs
```

```
BoolStack:   elements=[True, False]
```

In [11]:

```
print bs.pop()
```

```
False
```

In [12]:

```
print bs
```

```
BoolStack:   elements=[True]
```

In [13]:

```
print bs.pop()
```

```
True
```

In [14]:

```
print bs
```

```
BoolStack:   elements=[]
```

1.0) test `BoolStack`

Now start editing the file `exercise1_slow.py`. To check your environment is working fine, try to run the tests for `BoolStackTest`, which contain tests for the already implemented methods pop, push, etc ...

Notice that `exercise1_slow` is followed by a dot and test class name: `.BoolStackTest`

```
python -m unittest exercise1_slow.BoolStackTest
```

1.1) `true_count`, slow version

Implement the `true_count` method inside the class, **just working on this method alone**:

```
def true_count(self):
    """ Return the number of elements which are True in O(n), where n is the size of sta
ck.  """

    raise Exception("TODO IMPLEMENT ME !")
```

Testing

Once done, running this will run only the tests in `TrueCountTest` class and hopefully they will pass.

Notice that `exercise1_slow` is followed by a dot and test class name `.TrueCountTest` :

```
python -m unittest exercise1_slow.TrueCountTest
```

1.2) `true_count`, fast version

Now start editing the file `exercise1_fast.py`: inside you will find the class `FastBoolStack`. Your goal now is to implement a `true_count` method that works in `O(1)`. To make this possible, you are allowed to add any field you want in the constructor and you can also change any other method you deem necessary (like `push`) .

```
    def true_count(self):
        """ Return the number of elements which are True

            ***   MUST EXECUTE IN O(1)  ***
        """
        raise Exception("TODO IMPLEMENT ME !")
```

Testing :

> **WARNING: Since you are going to modify the whole class, make sure tests pass BOTH for `FastBoolStackTest` AND `TrueCountTest` !**

Tests for `push`, `pop`, etc:

`python -m unittest exercise1_fast.FastBoolStackTest`

Tests just for `true_count`:

`python -m unittest exercise1_fast.TrueCountTest`

2) UnorderedList

Start editing file `exercise2.py`, which contains a simplified versioned of the `UnorderedList` we saw in the labs.

2.1) `dup_first`

Implement the method `dup_first`:

```
    def dup_first(self):
        """ Modifies this list by adding a duplicate of first node right after it.

            For example, the list 'a','b','c' should become 'a','a','b','c'.
            An empty list remains unmodified.

            ** DOES NOT RETURN ANYTHING !!! **

        """

        raise Exception("TODO IMPLEMENT ME !")
```

Testing: `python -m unittest exercise2.DupFirstTest`

2.2) dup_all

Implement the method `dup_all`:

```
def dup_all(self):
        """ Modifies this list by adding a duplicate of each node right after it.

            For example, the list 'a','b','c' should become 'a','a','b','b','c','c'.
            An empty list remains unmodified.

            ** MUST PERFORM IN O(n) WHERE n is the length of the list. **

            ** DOES NOT RETURN ANYTHING !!! **
        """

        raise Exception("TODO IMPLEMENT ME !")
```

Testing: `python -m unittest exercise2.DupAllTest`

3) DiGraph

Now you are going to build some `DiGraph`, by defining functions *external* to class `DiGraph`.

> **WARNING: To build the graphs, just use the methods you find inside `DiGraph` class, like `add_vertex`, `add_edge`, etc.**

Start editing file `exercise3.py`

3.1) pie

Implement the function `pie`. Note the function is defined *outside* `DiGraph` class.

```
"""
        Returns a DiGraph with n+1 verteces, displaced like a polygon with a perimeter
        of n verteces progressively numbered from 1 to n.
        A central vertex numbered zero has outgoing edges to all other verteces.

        For n = 0, return the empty graph.
        For n = 1, return vertex zero connected to node 1, and node 1 has a self-loop.

    """
    raise Exception("TODO IMPLEMENT ME !")
```

Testing: `python -m unittest exercise3.PieTest`

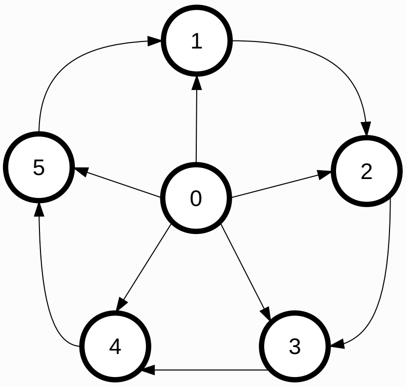
Example usage :

For *n=5*, the function creates this graph:

In [28]:

```
print pie(5)
```

```
0: [1, 2, 3, 4, 5]
1: [2]
2: [3]
3: [4]
4: [5]
5: [1]
```



Degenerate cases:

In [29]:

```
print pie(0)
```

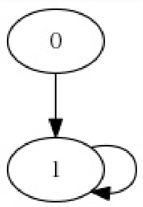
DiGraph()

In [30]:

```
print pie(1)
```

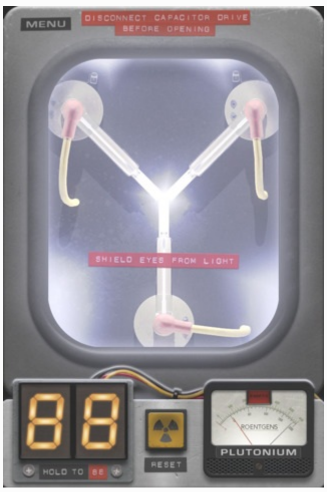
```
0: [1]
1: [1]
```

Out[31]:

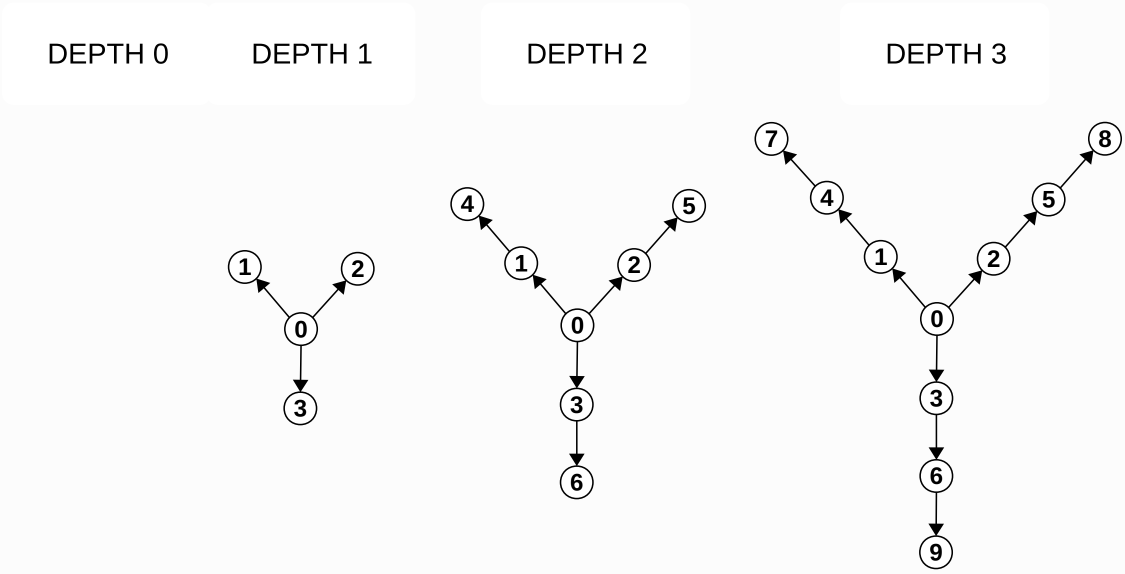


3.2) Flux Capacitor

A *Flux Capacitor* is a plutonium-powered device that enables time travelling. During the 80s it was installed on a Delorean car and successfully used to ride humans back and forth across centuries:



In this exercise you will build a Flux Capacitor model as a Y-shaped `DiGraph`, created according to a parameter `depth`. Here you see examples at different depths:



Implement the function `flux`. Note the function is defined *outside* `DiGraph` class:

```python
def flux(depth):
    """ Returns a DiGraph with 1 + (d * 3) numbered verteces displaced like a Flux Capacitor:

        - from a central node numbered 0, three branches depart
        - all edges are directed outward
        - on each branch there are 'depth' verteces.

        For example, for depth=2 we get the following graph (suppose arrows point outward):

             4         5
              \       /
               1     2
                \   /
                  0
                  |
                  3
                  |
                  6

    """
    raise Exception("TODO IMPLEMENT ME !")
```

Testing: `python -m unittest exercise3.FluxTest`

Usage examples

In [33]:

```python
print flux(0)
```

```
DiGraph()
```

In [34]:

```python
print flux(1)
```

```
0: [1, 2, 3]
1: []
2: []
3: []
```

In [35]:

```
print flux(2)
```

```
0: [1, 2, 3]
1: [4]
2: [5]
3: [6]
4: []
5: []
6: []
```

In [36]:

```
print flux(3)
```

```
0: [1, 2, 3]
1: [4]
2: [5]
3: [6]
4: [7]
5: [8]
6: [9]
7: []
8: []
9: []
```