# Introduction to Programming Homework 6

## Due Friday Nov 4

You will turn in your homework via e-mail (andrew.yarmola@uni.lu). For this homework, you will work in a text editor of your choosing. See instructions from the previous homework on how to write modules.

## Exercise 1 (Generators)

Make a module called `generators.py`. You will implement the following three generators.

- **a.** (Binary Decimal Expansion) Given a ratio $m/n$ of positive integers with $m \leq n$, we can consider the binary expansion

$$\frac{m}{n} = \sum_{i=1}^{\infty} \frac{a_i}{2^i}.$$

  Write a generator function `binary_decimal(m,n)` which yields the sequence $a_i$ in order. For example,

```
import generators as gen
one_third = gen.binary_decimal(1,3)
for _ in range(5) :
  print(next(one_third))
```

  should print

```
0
1
0
1
0
```

  Be sure to raise an error on bad input.

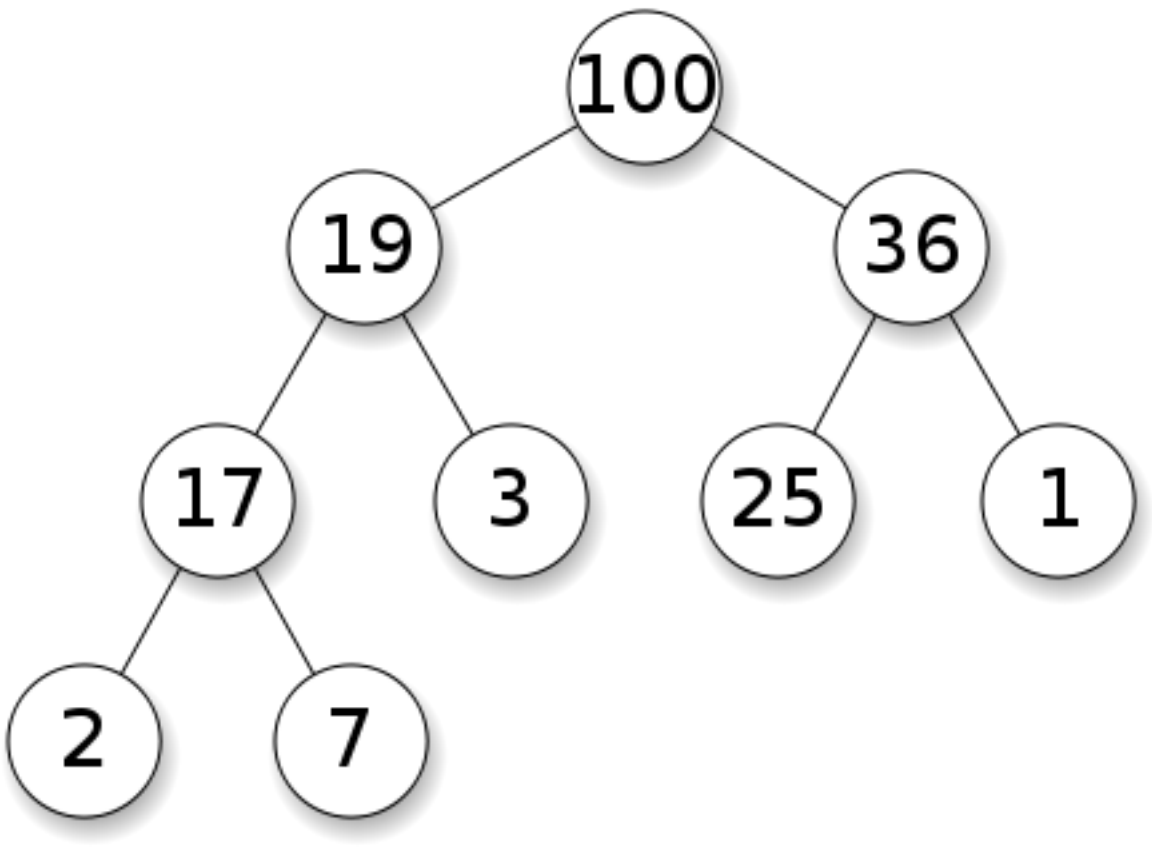
- **b.** (Biased coin from a fair one) A common problem in computer science is to take one pseudo-random number generator and use it to construct another. We can model a fair coin flip by using `random.randint(0, 1)` form the `random` module. This returns a `0` or a `1` with equal probability. **Using this fair coin**, write a function `biased_coin(m,n)` that returns `1` with probability exactly `m/n`.
  - Hint : Make use of part **a**.

- **c.** (Shuffle generator) Write a generator `shuffle_gen(start, stop)` that returns random numbers between `start` and `stop` without repetition. Make sure this works for very large integers.
    - Hint: See the modern version of the Fisher-Yates shuffle at https://en.wikipedia.org/wiki/Fisher–Yates_shuffle#Modern_method but instead of *swapping* numbers into a list you should use a better data structure.

## Exercise 2 (Heap)

Create a module called `heap.py`.

A binary **heap** is a binary tree with the property that for every node `x` one has `x.data >= x.left.data` and `x.data >= x.right.data`. For example, this image represents a heap




In this exercise, we will create a `Heap` class that will manage a tree with the heap property. **Instead** of using the `Node` class from the previous homework, the `Heap` will **store all data in a `list` in level-order** (i.e. we read off each level of the tree from left to right). For example, the data in the tree above will stored as the list

```
data_list = [100, 19, 36, 17, 3, 25, 1, 2, 7]
```

You can see that the node at index `i` of `data_list` has the left child at index `2*i+1` and the right child at index `2*i+2`. Similarly, you can use the index in the list to find a node's parent.

You will notice that with this structure, the maximum of the data in the tree is always the first element of the list. Our goal will be to write methods that allow us to insert new data into the heap and to remove the largest value from the heap all while preserving the heap property.

- **a.** Create a class called `Heap`. Inside, you will store a private `self._data_list` attribute. The init method should simply have type `__init__(self)` where you initialize the data list. In part **b**, we will be inserting elements to the end of `self._data_list` and then using the following methods to force the heap property using the following methods :
  - `def _bubble_down(self, i)`:
    - start with the node at index `i`
    - if the data at index `i` is larger than that of its children, stop.
    - if not, switch the data at index `i` with either child that has greater value.
    - continue doing this process to the child until you stop.
  - `def _bubble_up(self, i)`:
    - start with the node at index `i`
    - if the data at index `i` is smaller than that of its parent, stop.
    - if not, switch the data at index `i` with that of its parent.
    - continue doing this process to the parent until you stop.

**Do not use recursion for the above methods!**

- **b.** We will now us the `_bubble_down` and `_bubble_up` methods to insert elements into the heap and delete the largest value. Implement the following methods :
  - `def insert(self, value)`:
    - add `value` at the end of `self._data_list`
    - run `self._bubble_up` to move the newly inserted element into a position that doesn't violate the heap property.
  - `def pop_max(self)`:
    - record the first value of `self._data_list`
    - replace the first value of `self._data_list` with the last value
    - use `self._bubble_down` to push this replaced value down until it doesn't violate the heap property.
    - return the recorded first value as the maximum in the heap.

Observe that you can now sort a list of data by inserting elements one by one into a heap and then using `.pop_max()` to read off the sorted list from largest to smallest.

- **c.** Create a global function called `sort_all_lines(path_to_file)` which will read a file at `path_to_file`. Each line of `path_to_file` will be a **comma separated list** of integers. **Use the heap class you built above** to sort each line from greatest to smallest. Write you data to a file named as follows : if `path_to_file = 'dir1/dir2/filename.extension` save your file to `'dir1/dir2/filename_sorted.extension'`. For example, if file `'test_data.csv'` contains

  ```
  4,6,1,47,241,352,7,0,-140,352
  35,2601,362,1350305,-9352,38351
  ```

  you code should write a file called `test_data_sorted.csv` as

  ```
  352, 352, 241, 47, 7, 6, 4, 1, 0, -140
  1350305, 38351, 2601, 362, 35, -9352
  ```