Introduction to Programming Homework 5

Due Thursday Oct 27 by 17h00

You will turn in your homework via e-mail (andrew.yarmola@uni.lu (mailto:andrew.yarmoal@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 (Fast modular power)

In your module `prime_tests.py`, write a function `mod_pow(a,m,n)` that computes the remainder of a^m divided by p much faster than the command `a**m % n`. Please do **not** use the built-in `pow` command or any other packages. Hint : observe that $3^7 = 3 \cdot 3^2 \cdot \left(3^2\right)^2$.

You can test your speed using the following commands in IPython.

In [7]:

```
import random
import prime_tests

a = random.sample(range(2**10,2**20),100)
m = random.sample(range(2,1000),100)
n = random.sample(range(2,1000),100)

%timeit for i in range(len(a)) : prime_tests.mod_pow(a[i], m[i], n[i])
```

```
1000 loops, best of 3: 693 µs per loop
```

You should aim for something around the `1 ms` mark. The built-in `pow` function will (most likely) always be 2-4 times faster, however.

Exercise 2 (Miller-Rabin Primality Test)

The probabilistic Miller-Rabin primality test is based on the following well known fact.

Proposition 1. If p is prime and $x^2 \equiv 1 \bmod p$, then $x \equiv \pm 1 \bmod p$

Combining this with Fermat's little theorem, Miller-Rabin observe the following corollary.

Corollary 1. Let p be an odd prime and write $p - 1 = d \cdot 2^s$ where d is odd. Then for all $0 < x < p$, $x^{p-1} \equiv 1 \bmod p$ and either

$$x^d \equiv 1 \bmod p \quad or \quad x^{d \cdot 2^r} \equiv -1 \bmod p \text{ for some } 0 \le r \le s - 1$$

Proof. The fact that $x^{p-1} \equiv x^{d \cdot 2^s} \equiv 1 \bmod p$ is Fermat's little theorem. Taking successive square roots, the conclusion follows by Proposition 1. ■

Let n be an odd positive integer and write $n - 1 = d \cdot 2^s$ where d is odd. If we want to show that n is composite, we could demonstrate a number x that fails the conclusion of Corollary 1. First, we check that $x^{n-1} \equiv 1 \bmod \mathrm{n}$ and then we consider the sequence of numbers $x^d, x^{2d}, \ldots, x^{d \cdot 2^{s-1}}$. If $x^{d \cdot 2^r} \not\equiv \pm 1 \bmod n$ but $x^{d \cdot 2^{r+1}} \equiv 1 \bmod n$ for some $0 \le r \le s - 1$, then we can conclude that n is composite.

An integer x that demonstrates that n is composite in this way is called a **Miller-Rabin witness**.

Theorem (Miller-Rabin). Let n be an odd composite positive integer. A randomly chosen x form $\{1, \ldots, n - 1\}$ has a probability of **more than** 3/4 of being a Miller-Rabin witness.

- **a.** In `prime_tests.py` write a function `is_miller_rabin_witness(n,x)` which checks whether `x` is a Miller-Rabin witness for an odd positive integer `n`. No need to validate your input for this one.

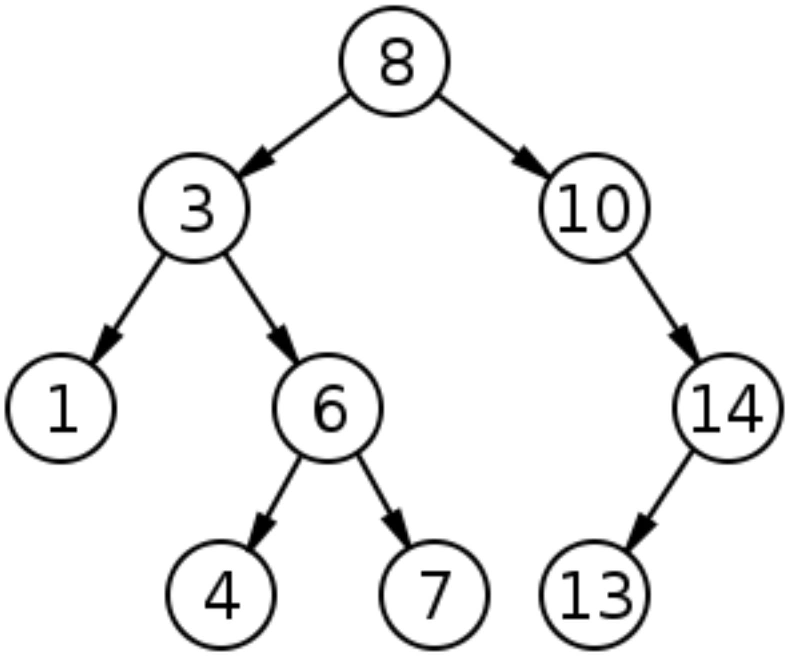
- **b.** In `prime_tests.py` write a function `probably_prime(n, prob)` which takes a number `n` and returns `True` if `n` has probability at least `prob` of being prime by the Miller-Rabin test. `False` otherwise. Use the Miller-Rabin theorem to run the test enough times to guarantee the desired probability.

- **c.** Assuming the extended Riemann hypothesis, Miller proved that every composite number `n` has a Miller-Rabin witness in the set $\{2, \ldots, \min(n - 1, \lfloor 2(\log n)^2 \rfloor)\}$. Use the `math` module to write a function `is_prime(n)` that uses this test to conclude whether `n` is prime or not. Here, $\log$ is the base e logarithm.
  - Remark : The Miller-Rabin Theorem tells us that if we find $(n - 1)/4$ **non-witnesses** in $\{1, \ldots, n - 1\}$, then we can conclude that n is prime. Notice that for `n > 241` one has $\lfloor 2(\log n)^2 \rfloor) < (n - 1)/4$, so this test is more efficient for large n.

Exercise 3 (Trees)

Create a module `binary_tree.py`

In the exercise, your job will be to create a class that represents the fundamental building block of a binary tree. A binary tree is a structure that looks like this :



In the above diagram, every circle is called a **node** of the tree. Each node has some data stored inside (a number in the above case). Most nodes also have a left and right child node. Nodes that do not have children are called **terminal** nodes. The top node in the diagram above is called the **root** node. As you can see, the **root** node does not have a **parent** node.

- **a.** Create a class called `Node`. It should have readable properties `.parent`, `.left`, `.right`, and `.data`. The `.parent`, `.left` and `.right` properties should again be `Node` instances or `None`. To simulate a node not having children (or a parent) we can set those properties to `None`.

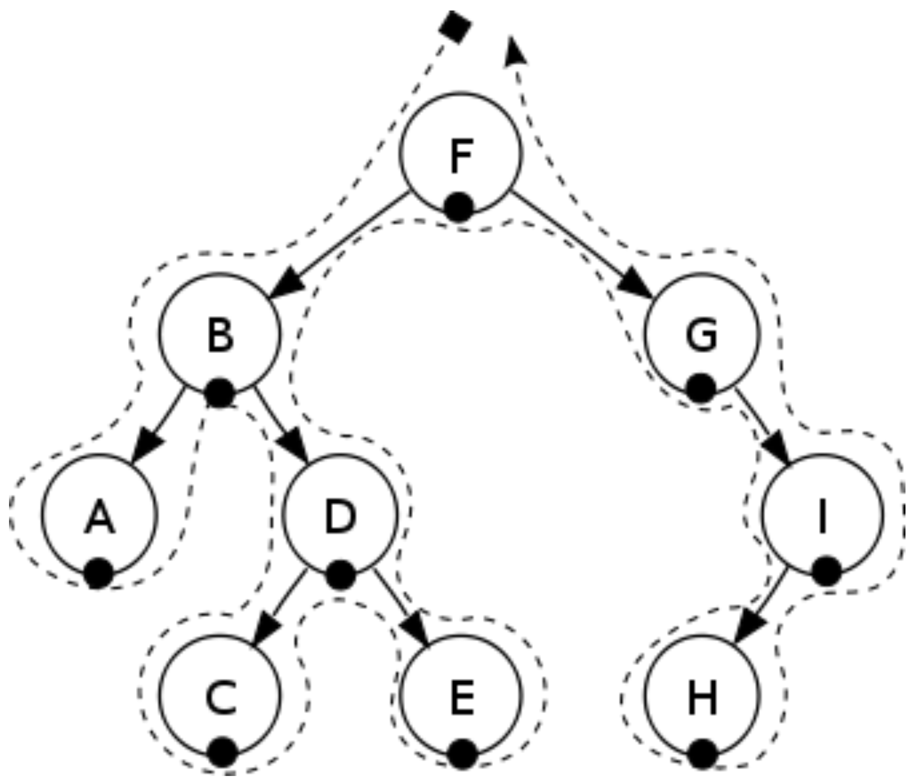
Your `__init__` method should be of type

```
def __init__(self, data = None, left = None, right = None)
```

Write setters for `.data, .left, .right`. Make `.parent` a read only property and manage it internally. When writing the setter for `.left` and `.right`, be sure to check that the object you are setting are instances of class `Node` or that they are `None`. Be sure to set the parent node internally when setting the children nodes. Also, be sure to clear the parent property internally when removing or replacing a child node.

- **b.** A binary tree can be represented by its starting root node. In fact, given any node, you can read off the (sub)tree below it by looking at its children. Write a module global function called `test_tree` which returns the root node of the binary tree in the above picture.

- **c.** Frequently, it is useful to read the data of the tree is a specific order. Create a **recursive** instance method called `.inorder` which returns a list containing the data of the tree in the following order :



So, if `my_tree` is the tree in the above image, `my_tree.inorder() = ['A','B',...,'H','I']`