Introduction to Programming Homework 5

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 $3^7 = 3 \cdot 3^2 \cdot (3^2)^2$.

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.

In [1]:

```
def mod_pow(a, power, n) :
    """ A faster modular power algorithm for
    integers a, power >=0, and n. Returns a**power % n. """
    assert isinstance(a,int) and isinstance(power,int) and \
           isinstance(n,int) and power >= 0
    result = 1
    double = a
    while power > 0 :
        # keep doubling until we hit a non-zero
        # coefficient in the binary expansion of
        # power and accumulate that into the result
        if power % 2 != 0 :
            result = (result * double) % n
            # we we ever get 0, we can stop
            if result == 0 : break
        double = double**2 % n
        # if our square is ever 1, we can stop
        if double == 1 : break
        # since we are taking care of the case
        # where power is odd above, we can just
        # take the integral half
        power //=2
    return result
```

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

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

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 $0 \leq r \leq s-1$, 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.

In [2]:

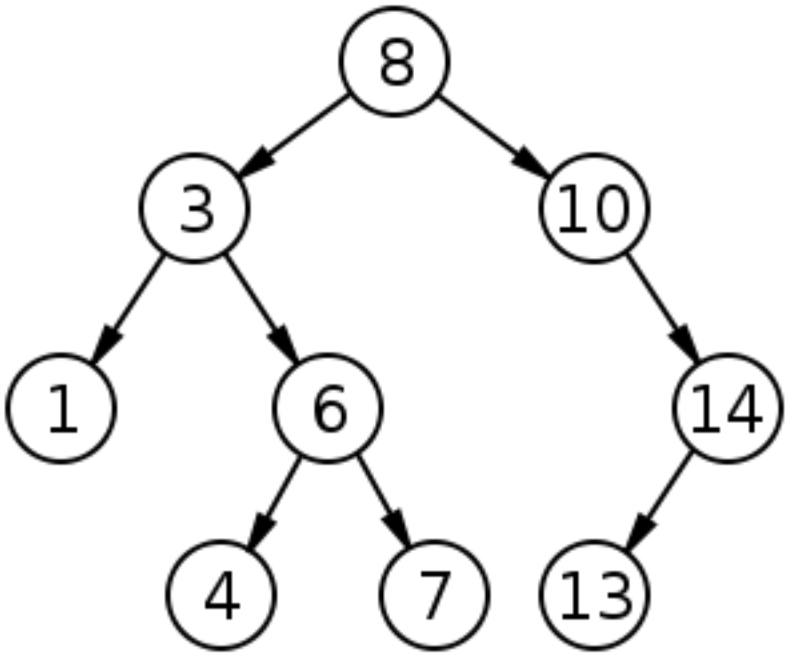
```
def decompose_even_part(n) :
    """ Returns d,s such that n = d * (2**s). Assumes
    that n is a positive integer. """
    s, d = 0, n
    while d % 2 == 0 :
        s += 1
        d //= 2
    return (d,s)

def is_miller_rabin_witness(n,x) :
    """ Tests if x is a Miller-Rabin witness for n.
    Assumes x,n are integers and n > 0. """
    d, s = decompose_even_part(n-1)
    a = pow(x,d,n)
    # inconclusive if x**d % n == 1
    if a == 1 : return False

    for _ in range(s) :
        # if we get -1 mon n, inconclusive
        if a == n - 1 :
            return False
        a = a**2 % n
        # if the square is 1 but before we were not -1, witness
        if a == 1 :
            return True

    # a is now x**(n-1) % n but it wasn't equal to 1 at the end
    # end of the last loop!
    return True

def probably_prime(n, prob) :
    """ Returns True if n is a prime with probablity greater
    than prob using the Miller-Rabin test. """
    assert isinstance(n, int) and n > 0
```

```python
    if n == 2 : return True

    if n % 2 == 0 or n == 1 : return False
    # we compute the number of tests we need knowning that
    # x chosen from {1, ..., n-1} has probablity at least 3/4
    # of being an M-R witness
    if prob >= 1. :
        num_tests = 1+(3*(n-1))//4
    else :
        # want (1/4)**num_tests <= 1 - prob
        num_tests = -math.floor(math.log2(1-prob)/2)
    tests = random.sample(range(1,n), num_tests)
    for t in tests :
        if is_miller_rabin_witness(n, t) :
            return False
    return True

def is_prime(n) :
    """ Returns True if n is a prime. Assumes
    the extended Riemann hypothesis. """
    assert isinstance(n, int) and n > 0
    if n == 2 : return True
    if n % 2 == 0 or n == 1 : return False
    # TODO : math.log only works well with n < 2**53 !!!
    test_cutoff = math.floor(2 * math.log(n)**2)
    for t in range(2, test_cutoff  + 1) :
        if is_miller_rabin_witness(n, t) :
            return False
    return True
```

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`.

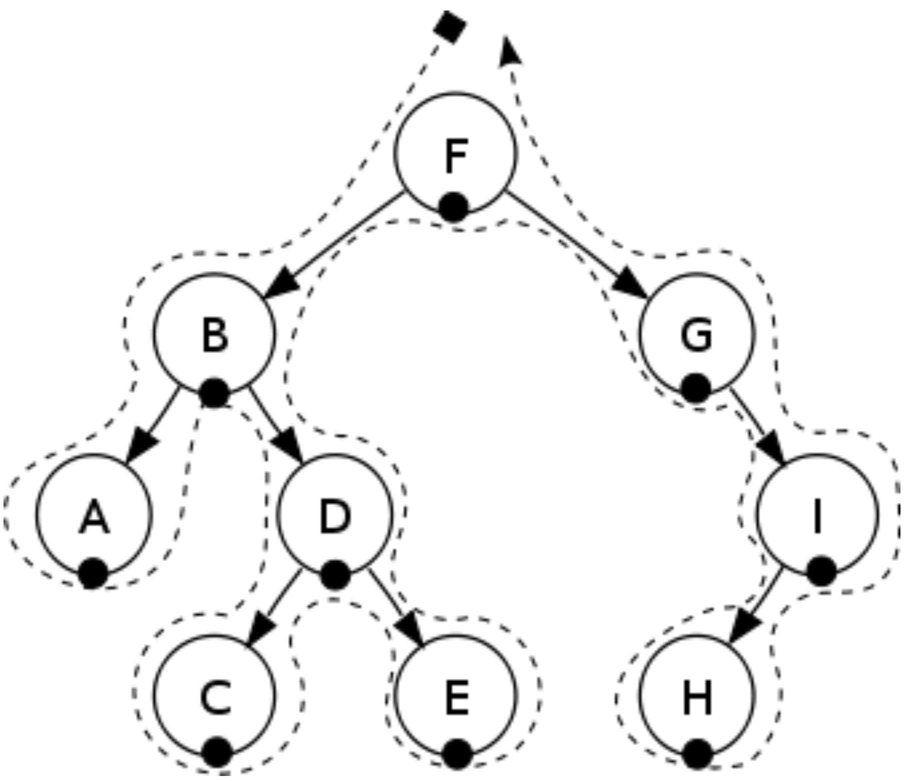
Your `__init__` method should be of type

```
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']`

In [5]:

```python
class Node:
    def __init__(self, data = None, left = None, right = None) :
        self._right = None
        self._left = None
        self._parent = None
        self.data = data
        self.left = left
        self.right = right

    @property
    def data(self) :
        return self._data

    @data.setter
    def data(self, data):
        self._data = data

    # publically, parent will be readonly!
    @property
    def parent(self):
        return self._parent

    @property
    def left(self):
        return self._left

    @left.setter
    def left(self, set_as_left) :
        if type(set_as_left) is Node or set_as_left is None :
```

```
        if type(set_as_left) is Node or set_as_left is None :
            # we must clean up the parent of the old node
            if self.left is not None :
                self.left._parent = None
            self._left = set_as_left
            # set outselves as the parent of the new node
            if self.left is not None :
                self.left._parent = self
        else :
            raise ValueError("Left child must be a Node or None")

    @property
    def right(self):
        return self._right

    @right.setter
    def right(self, set_as_right) :
        if type(set_as_right) is Node or set_as_right is None :
            # we must clean up the parent of the old node
            if self.right is not None :
                self.right._parent = None
            self._right = set_as_right
            # set outselves as the parent of the new node
            if self.right is not None :
                self.right._parent = self
        else :
            raise ValueError("Right child must be a Node or None")

    def inorder(self):
        L = []
        C = [self.data]
        R = []
        if self.left is not None:
            L = self.left.inorder()
        if self.right is not None:
            R = self.right.inorder()
        return L + C + R

def test_tree() :
    # here 'up' means go to a parent, and
    #'nl' mean "no left" for skipping the left
    # node. we keep a current node an keep setting
    # left nodes until there is one already set or 'nl'
    tree_encoding = [8,3,1,'up', 6, 4, 'up', 7,
                     'up', 'up', 'up', 10, 'nl', 14, 13]

    # using the fact that data is the first argument
    root = Node(tree_encoding[0])
    curr = root
    skip_left = False
    for d in tree_encoding[1:] :
        if d == 'up' :
            curr = curr.parent
            continue
        if d == 'nl' :
            skip_left = True
            continue
        new = Node(d)
```

```
        new = Node(d)
        if curr.left is None and not skip_left :

            curr.left = new
        else :
            curr.right = new
        curr = new
        skip_left = False

    return root
```