COMPUTER PROGRAMMING TAKE HOME FINAL – MATH 5405 SPRING 2016

DUE: MONDAY MAY 2ND

We will implement the Elliptic Curve Digital Signature Algorithm in python using our previous work on elliptic curves. You may work together to get any previous work fully functional (ie, adding points on elliptic curves). If you can't get your previous code to work, send me an email and you can use mine if you want. You may also work together to get the hashing functions working (computing the z below).

This assignment is worth 1/6th of your Final Exam score, hence 5% of your total grade.

I created a file `ECDSA.py` and at the top put

```
import EC
import hashlib
import random
```

Remember, the ECDSA works as follows.

We fix an elliptic curve E (over a finite field) and a point Q of prime order n. Alice's private key is an integer d_A and she computes $Q_A = d_A \cdot Q$. To sign a message, she does the following.

(1) Compute e = `HASH(m)` where `m` is the message.
(2) Somehow turn `e` into a number z in [1, n-1]. The official implementation is to take the appropriate left-most number of bits.
(3) Choose a random integer $k \in [1, n-1]$.
(4) Calculate the curve point $kQ = (x_1, y_1)$.
(5) Compute $r = x_1 \bmod n$ (if $r == 0$, choose a new k).
(6) Compute $s = k^{-1}(z + rd_A) \bmod n$, if zero, choose a new k and compute r and s again.
(7) Return (r, s).

We will implement this.

1. HASH

Now need a HASH function. Fortunately several are built right into python. Indeed, see

`https://docs.python.org/2/library/hashlib.html#module-hashlib`

For instance

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update("Honesty is the first chapter in the book of wisdom.  Thomas Jefferson")
>>> s = m.digest()
>>> s
>>> len(s)
>>> s = m.hexdigest()
>>> s
>>> int(s, 16)
```

Note here s is a string of exactly 32 characters. It turned that quote of Jefferson into another string of random (non-readable) characters. The final `int(s,16)` command turns the number (written in hexidecimal) into an integer

You now need to turn this into a number between 1 and $n-1$. Figure out how you want to do this. You could just turn your string `s` into a number, and then take it modulo n (or maybe $n-2$?) The point is . Here is how my function began. Remember, you can work with groups on this part.

```
def HashMessage(message, n):
    h = hashlib.sha256()
    h.update(message)
    s = h.hexdigest()
    val = int(s, 16)
    return ...
```

2. Signature Implementation

Now write a ECDSA function. Here's how my function started

```
def sign(m, Q, n, d, Elist, char):
    #m is the message,
    #d is the private key
    #Q is my point,
    #n = ord(Q) is the prime number I already computed elsewhere
    #(or someone computed for me).
    #Elist is my elliptic curve,
    #char is the characteristic
    z = HashMessage(m, n)
    myRand = random.SystemRandom() # a cryptographically secure random number generator
    r = 0
    while(r == 0):
        k = myRand.randint(1,n-1)
        kQ = EC.multPt(k,Q,Elist,char) #this is my function that computes k*Q,
            #you might have called your function something different
        ...
        r = x1%n
    ...
    return [r,s]
```

Note every signature you create with this algorithm will be different, even for the same message `m`. This is because of the randomly chosen `k`.

Your turned in work... must include a working signature function.

3. Signature Verification

Alice will publicly share $R = d_A \cdot Q$, Q, $n = \text{ord}(Q)$, the elliptic curve and characteristic, since that is her public key. Remember, the expectation is that it is hard for someone to figure out what d_A is knowing only R and Q (it's the discrete log problem for elliptic curves). Alice will also share the message `m`, and her signature `[r,s]` which her function returned.

To verify the signature, Bob does the following

(1) Compute e = `HASH(m)` where `m` is the message.
(2) Somehow turn `e` into a number z in [1, n-1]. The official implementation is to take the appropriate left-most number of bits. You must use the same implementation as Alice does though!
(3) Compute $w = s^{-1} \bmod n$
(4) Compute $u1 = zw \bmod n$ and $u2 = rw \bmod n$.

(5) Compute $T = (x1, x2) = u1 \cdot Q + u2 \cdot R$.
(6) Verify that $r \equiv_n x1$.

I started my implementation as follows.

```
def validate(m, R, Q, n, Elist, char, signature):
    #m,Q,n,Elist,char as as above
    #R is d*Q, this is Alice's public key
    #signature is the list [r,s] above
    r = signature[0]
    s = signature[1]
    z = HashMessage(m, n)
    ...
    return ( (T[0])%n == r%n)
```

Your turned in work... must include a working validation function.

4. Check your own work.

The following elliptic curve and point Q are the ones used in bitcoin.

- The characteristic is

$$p = 115792089237316195423570985008687907853269984665640564039457584007908834671663$$

- The elliptic curve is defined by

$$y^2 = x^3 + 7$$

- The point $Q = (x, y)$ is defined as

$$x = 55066263022277343669578718895168534326250603453777594175500187360389116729240$$
$$y = 32670510020758816978083085130507043184471273380659243275938904335757337482424$$

- The order of Q is

$$n = 115792089237316195423570985008687907852837564279074904382605163141518161494337$$

Verify that Q is on the curve (to make sure your elliptic curve functions are working).

Finally, send a message to yourself and validate it, using the elliptic curve above to make sure your signature functions are working. Remember, you will have to choose your own random d (your private key) and you will have to compute $d \cdot Q$ (your public key). Include a copy of this output in your turned in assignment.

Your turned in work... must include a copy of this output showing that your signature validation scheme works.