HW 7
Due: Friday, November 1

§3.13/21, 33, 56b, 58, 61; §3.14/12, 14

Also, implement the algorithm of 56b as a program `expmod(y, x, n)`. Check that it works by evaluating `expmod(13, 56728, 28139)`.

Hint for 56b: let x_k be the number with binary expansion $b_{w-k+1}b_{w-k+2}\dots b_w$. Show that after k iterations of the loop, $b \equiv y^{x_k} \pmod{n}$ and $x = x_k$.

3.13.56b Here's the code:

```
def expmod(y, x, n):
    """Compute y^x mod n."""
    a, b, c = x, 1, y
    while a != 0:
        if a%2 == 0:
            a, c = a//2, (c*c)%n
        else:
            a, b = a-1, (b*c)%n
    return b
```

Notice that at each iteration, a is getting smaller, so eventually $a = 0$. Thus the algorithm eventually terminates.

For the proof, first observe that if the value of a is odd, then in the loop the new value of a is even (`a=a-1`). Thus we could rewrite the code as follows:

```
def expmod(y, x, n):
    """Compute y^x mod n."""
    a, b, c = x, 1, y
    while a != 0:
        if a%2 != 0:
            b = (b*c)%n
        a, c = a//2, (c*c)%n
    return b
```

With this new code, let a_k, d_k, c_k be the values of a, b, c after k iterations. (I use d_k instead of b_k since b_k is already taken in the binary expansion of x.) Clearly $c_k = y^{2^k} \pmod{n}$. Since taking the quotient on division by 2 is the same as dropping the last bit, we have that a_k has binary expansion

$$b_1 b_2 \dots b_{w-k}.$$

Finally, I will show by induction that $d_k \equiv y^{(b_{w-k+1}\dots b_w)_2} \pmod{n}$. When $k = 0$, the exponent is 0, and indeed $d_0 = 1$. Thus the base case holds.

Suppose $d_k \equiv y^{(b_{w-k+1}\dots b_w)_2} \pmod{n}$. On the $k+1$st iteration, there are 2 cases: either $b_{w-k} = 0$ or 1. If $b_{w-k} = 1$, since $a_k = (b_1 b_2 \dots b_{w-k})$, then

a_k is odd. Thus

$$d_{k+1} = d_k \cdot c_{k-1} \equiv y^{(b_{w-k+1} \ldots b_w)_2} \cdot y^{2^{k-1}} \pmod{n} = y^{(1 b_{w-k+1} \ldots b_w)_2}.$$

Since $b_{w-k} = 1$, the inductive claim follows. If $b_{w-k} = 0$, then

$$d_{k+1} = d_k \equiv y^{(b_{w-k+1} \ldots b_w)_2} \pmod{)} n = y^{(0 b_{w-k+1} \ldots b_w)_2},$$

and the claim follows again.

Finally, the output occurs when $a_k = 0$; by our formula for a_k, this occurs when $k = w$. From our formula for d_k, the algorithm is correct.

3.13.58 For (a), there are at most 4 square roots of x, so she expects to get m on average after 4 iterations.

For (b), there is no efficient method for Oscar to compute *any* square roots without knowing p and q.

For (c), Eve enters 1 several times. Note that both 1 and $-1 \equiv n-1$ are square roots of 1, but there are 2 others. Eve inputs 1 until she gets one of the *other* square roots of 1; say she obtains $m \not\equiv \pm 1 \pmod{n}$. Then she computes $\gcd(m-1, n)$ to get one of the prime factors.

The reason this works is as follows. By Sun Tzu's Theorem, $m^2 \equiv 1 \pmod{p}$ and $m^2 \equiv 1 \pmod{q}$, but we cannot have

$$m \equiv 1 \pmod{p}, m \equiv 1 \pmod{q},$$

for in this case we would have $m \equiv 1 \pmod{n}$. Similarly, we cannot have

$$m \equiv -1 \pmod{p}, m \equiv -1 \pmod{q}.$$

Therefore she has obtained

$$m \equiv 1 \pmod{p}, m \equiv -1 \pmod{q}$$

or vice versa; without loss of generality she gets the above. Then $p \mid (m-1)$ while $q \nmid (m-1)$, and hence $\gcd(m-1, n) = p$, enabling her to factor n.

3.14.12 As 34807 is a prime $\equiv 3 \pmod{4}$ we may use the proposition in §3.9. We have $(34807 + 1)/4 = 8702$, and using our Python code,

```
expmod(26055,8702,34807)
```

yields 33573. We have to check that this works with

```
expmod(33573,2,34807)
```

which indeed equals 26055.