Assignment 20

Due by class time Tuesday, December 6

Here is the standard mathematical definition of exponentiation, for integer exponents $x \geq 0$:

$$a^x = \begin{cases} 1 & \text{if } x = 0 \\ a \times a^{x-1} & \text{if } x > 0 \end{cases}$$

According to this definition, computing a^x requires x multiplication operations. For example: $a^3 = a \times a^2 = a \times a \times a^1 = a \times a \times a \times 1$. However, we can compute a^x much more efficiently by rewriting the above definition as follows:

$$a^x = \begin{cases} 1 & \text{if } x = 0 \\ (a^{\frac{x}{2}})^2 & \text{if } x > 0 \text{ and } x \text{ is even} \\ a \times (a^{x-1}) & \text{if } x > 0 \text{ and } x \text{ is odd} \end{cases}$$

With this approach, computing a^{1000} requires only 15 multiplications, instead of 1000. In general, the number of multiplications needed by the second approach is proportional to the *logarithm* of x, which results in many fewer multiplications performed.

1. Write a recursive Python function called **power(a, x)** that computes (and returns) the value a^x using the second approach. A simple way to find out how many multiplications occur is to just put in a `print("times")` statement wherever a multiplication or squaring operation occurs in the code. For example, your output should look something like this:

```
>>> print(power(2, 100))
times
times
times
times
times
times
times
times
times
1267650600228229401496703205376
```

2. Define a new version of your exponentiation function called **powermod(a, x, M)**, which takes an extra parameter M called the *modulus*. Instead of computing a^x, your function should compute (and return) the value $a^x \bmod M$, where $p \bmod q$ is the remainder obtained when dividing p by q, which in Python can be written as `p % q`. Try to write your function so that it keeps all intermediate products within the range 0 to $M - 1$, instead of computing the full value of a^x first and then reducing it to the range 0 to $M - 1$. See pages 205-207 of the textbook for more information about modular arithmetic. Examples:

```
powermod(2, 3, 9)     => 8
powermod(2, 3, 5)     => 3
powermod(3, 3, 25)    => 2
powermod(2, 100, 17)  => 16
powermod(24, 76, 371) => 333
```

3. Read section 6.5 of *Quantum Computing for Computer Scientists* (pages 204–218).