This is a Python programming assignment. You will edit your ongoing copy of `qc.py` and submit it for grading. The grader will `import qc` and then run their own testing code against it. Probably the grader will also inspect your code.

First we need to learn the *repeated squaring* algorithm. Suppose that I want to compute 5^{27}. First I compute certain powers of 5 by squaring certain other powers of 5:

$$\begin{aligned} 5^1 &= 5, \\ 5^2 &= (5^1)^2 &&= 25, \\ 5^4 &= (5^2)^2 &&= 625, \\ 5^8 &= (5^4)^2 &&= 390625, \\ 5^{16} &= (5^8)^2 &&= 152587890625. \end{aligned}$$

I know that that's enough, because meanwhile I'm expressing the exponent 27 as a sum of powers of 2:

$$27 = 16 + 8 + 4 + 2 + 1.$$

So

$$5^{27} = 5^{16} \cdot 5^8 \cdot 5^2 \cdot 5^1 = 7450580596923828125.$$

I have performed a total of seven multiplications, compared to the 26 multiplications used by the naive algorithm for computing 5^{27}. In general, computing k^ℓ requires $\mathcal{O}(\ell)$ multiplications in the naive algorithm and $\mathcal{O}(\log \ell)$ multiplications in repeated squaring. Boom.

The same idea works for powers modulo m. Just replace every multiplication with multiplication modulo m. That is, after every multiplication, divide the product by m and keep only the remainder. Then you never have to hold any number larger than m^2.

If you write the repeated squaring algorithm cleverly, then you can discover how many powers of k you need while also computing those powers and incorporating them into the answer. If you don't want to figure this out on your own, then see Mermin's Section 3.8.

A. Write a function according to the following specification. Your implementation should use the repeated squaring technique to achieve high speed.

```
def powerMod(k, l, m):
    '''Given non-negative integer k, non-negative integer l, and positive
    integer m. Computes k^l mod m. Returns an integer in {0, ..., m - 1}.'''
```

B. Write the following function.

```
def fourier(n):
    '''Returns the n-qbit quantum Fourier transform gate T.'''
```

C. Write the following function to implement the quantum core subroutine for Shor's algorithm as described in our lectures. The output is the output of the last partial measurement — the one on the input register.

```
def shor(n, f):
    '''Assumes n >= 1. Given an (n + n)-qbit gate f representing a function
    f: {0, 1}^n -> {0, 1}^n of the form f(l) = k^l % m, returns a list of
    classical one-qbit states (ket0 or ket1) corresponding to an n-bit string
    that satisfies certain mathematical properties.'''
```

D. Write a function `shorTest(n, m)`. It takes as input the integers n and m as specified in our lectures. It chooses a random k that is coprime to m (`math.gcd` might help), builds the function f that computes powers of k modulo m (using repeated squaring — `powerMod` might help), converts it to a gate F, and runs Shor's quantum core subroutine. For now, just print out the results. (In future lectures we'll see how to use the results, and we'll write a better test.)