This is Python project work, due at the end of the term.

Before you start coding, you might want to do this study question (but not hand it in): Write out the one-qbit QFT gate T, as a 2×2 matrix. Write out the two-qbit T, as a 4×4 matrix.

A. In qGates.py, write the following function.

```
def fourier(n):
    '''Assumes n >= 1. Returns the n-qbit quantum Fourier transform gate T.'''
```

B. In qGates.py, paste the following function, and use it to test your `fourier` function. This is not a great test, because I don't know how to write a great test without giving away some version of the `fourier` implementation. If you like, add more cases.

```
def fourierTest(n):
    if n == 1:
        # Explicitly check the answer.
        t = fourier(1)
        if qu.equal(t, qc.h, 0.000001):
            print("passed fourierTest")
        else:
            print("failed fourierTest")
            print("    got T = ...")
            print(t)
    else:
        t = fourier(n)
        # Check the first row and column.
        const = pow(2, -n / 2) + 0j
        for j in range(2**n):
            if not qu.equal(t[0, j], const, 0.000001):
                print("failed fourierTest first part")
                print("    t = ")
                print(t)
                return
        for i in range(2**n):
            if not qu.equal(t[i, 0], const, 0.000001):
                print("failed fourierTest first part")
```

```
            print("    t = ")
            print(t)
            return
    print("passed fourierTest first part")
    # Check that T is unitary.
    tStar = numpy.conj(numpy.transpose(t))
    tStarT = numpy.matmul(tStar, t)
    id = numpy.identity(2**n, dtype=qc.one.dtype)
    if qu.equal(tStarT, id, 0.000001):
        print("passed fourierTest second part")
    else:
        print("failed fourierTest second part")
        print("    T^* T = ...")
        print(tStarT)
```

C. In qAlgorithms.py, 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 second 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 or tuple
    of n classical one-qbit states (|0> or |1>) corresponding to the output of
    Shor's quantum circuit.'''
```

D. In qAlgorithms.py, write a function `shorTest(n, m)`. It takes as input the integers n and m as specified in our lectures. It may assume that $2^n \geq m^2$, because that's required by Shor's algorithm. It may also assume that $n \geq 4$, because handling the low-n cases is overly tedious. Then it does these steps:

1. Chooses a random k that is coprime to m (`math.gcd` should help).
2. Builds the function f that computes powers of k modulo m (`qu.powerMod` should help).
3. Runs Shor's quantum core subroutine on the corresponding gate F.
4. Interprets the output as an integer $b \in \{0, \ldots, 2^n - 1\}$.
5. Prints b. (Later we will improve this step, to make `shorTest` a real test.)

I recommend that you run the test at least once, just to make sure that it runs. For example, try `shorTest(4, 3)`, `shorTest(4, 4)`, and `shorTest(5, 5)`. Last time I tried `shorTest(8, 15)`, the operating system stopped Python after five minutes of painful churning.