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.

A. Near the `swap` and `cnot` gates, add the three-qbit gate `toffoli`.

B. Implement the following function.

```
def power(stateOrGate, m):
    '''Given an n-qbit gate or state and m >= 1, returns the mth tensor power,
    which is an (n * m)-qbit gate or state. Assumes n >= 1. For the sake of
    time and memory, m should be small.'''
```

(Although it's not part of your official assignment, it might also be educational for you to contemplate how large m can be in, for example, `qc.power(qc.h, m)`. For which m is $H^{\otimes m}$ as large as your computer's RAM?)

C. In the doc string of your function `function`, replace

```
Assumes that n = m = 1.
```

with

```
Assumes that n, m >= 1.
```

Then update your implementation to match this new specification. That is, your function should be able to convert an arbitrary $f : \{0,1\}^n \to \{0,1\}^m$ to its corresponding $(n+m)$-qbit gate F. (By the way, problems E and G below will stress-test your implementation.)

D. Implement the circuit of Bernstein and Vazirani (1992) in the following function.

```
def bernsteinVazirani(n, f):
    '''Given n >= 1 and an (n + 1)-qbit gate f representing a function
    {0, 1}^n -> {0, 1} defined by mod-2 dot product with an unknown w in
    {0, 1}^n, returns the list or tuple of n classical one-qbit states
    (ket0 or ket1) corresponding to w.'''
```

E. Write a function `bernsteinVaziraniTest`, which takes as input an integer `n`, randomly generates an n-qbit example of the Bernstein-Vazirani problem, runs `bernsteinVazirani` on it, and prints out diagnostic information to convince the user that `bernsteinVazirani` works.

By the way, at some point you might need to test whether a given one-qbit state equals `ket0` or `ket1`. And you might not know how to do that with `numpy` arrays. Try mimicking this code:

```
if (myState == ket0).all():
    print("myState equals ket0")
else:
    print("myState does not equal ket0")
```

F. Implement the circuit of Simon (1994) in the following function.

```
def simon(n, f):
    '''The inputs are an integer n >= 1 and an (n + n - 1)-qbit gate f
    representing a function {0, 1}^n -> {0, 1}^(n - 1) hiding an n-bit string w
    as in the Simon (1994) problem. Returns a list of n classical one-qbit
    states (ket0 or ket1) corresponding to a uniformly random bit string gamma
    that is perpendicular to w.'''
```

G. Write a function `simonTest`, which takes as input an integer `n`, randomly generates an n-qbit example of the Simon problem, runs `simon` repeatedly, does (at least some of) the enveloping linear algebra, and prints out diagnostic information to convince the user that (most of) Simon's algorithm has been correctly executed. (The function `reduction` exists to help you. Let me know if you find bugs in `reduction`.)