This assignment is project work due at the end of the term.

A. In qMeasurement.py, you already have a function `first`. In its doc string, change "`Assumes n == 2`" to "`Assumes n >= 1`". Improve the implementation accordingly. Recall that the unique 0-qbit state is defined in qConstants.py. Test.

B. In qMeasurement.py, do the same for the `last` function.

C. In qGates.py, implement the following function. (It might also be educational for you to contemplate how large m can be. For which m is $H^{\otimes m}$ as large as your computer's RAM? But you don't need to hand in your answer to this parenthetical question.)

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

D. 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. Although we have discussed this construction in lecture, turning the math into code is not easy. Talk to me if you can't figure it out. Also add the following test code, and use it to test your implementation for values of n and m up to 3.

```
def functionTest(n, m):
    # 2^n times, randomly pick an m-bit string.
    values = [qb.string(m, random.randrange(0, 2**m)) for k in range(2**n)]
    # Define f by using those values as a look-up table.
    def f(alpha):
        a = qb.integer(alpha)
        return values[a]
    # Build the corresponding gate F.
    ff = function(n, m, f)
    # Helper functions --- necessary because of poor planning.
    def g(gamma):
        if gamma == 0:
            return qc.ket0
        else:
            return qc.ket1
```

```
    def ketFromBitString(alpha):
        ket = g(alpha[0])
        for gamma in alpha[1:]:
            ket = tensor(ket, g(gamma))
        return ket
    # Check 2^n - 1 values somewhat randomly.
    alphaStart = qb.string(n, random.randrange(0, 2**n))
    alpha = qb.next(alphaStart)
    while alpha != alphaStart:
        # Pick a single random beta to test against this alpha.
        beta = qb.string(m, random.randrange(0, 2**m))
        # Compute |alpha> tensor |beta + f(alpha)>.
        ketCorrect = ketFromBitString(alpha + qb.addition(beta, f(alpha)))
        # Compute F * (|alpha> tensor |beta>).
        ketAlpha = ketFromBitString(alpha)
        ketBeta = ketFromBitString(beta)
        ketAlleged = application(ff, tensor(ketAlpha, ketBeta))
        # Compare.
        if not qu.equal(ketCorrect, ketAlleged, 0.000001):
            print("failed functionTest")
            print("    alpha = " + str(alpha))
            print("    beta = " + str(beta))
            print("    ketCorrect = " + str(ketCorrect))
            print("    ketAlleged = " + str(ketAlleged))
            print("    and here's F...")
            print(ff)
            return
        else:
            alpha = qb.next(alpha)
    print("passed functionTest")
```

E. In qAlgorithms.py, 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
    f : {0, 1}^n -> {0, 1} defined by mod-2 dot product with an unknown delta
```

```
    in {0, 1}^n, returns the list or tuple of n classical one-qbit states (each
    |0> or |1>) corresponding to delta.'''
```

Also add the following test code, and use it to test your implementation.

```
def bernsteinVaziraniTest(n):
    delta = qb.string(n, random.randrange(0, 2**n))
    def f(s):
        return (qb.dot(s, delta),)
    gate = qg.function(n, 1, f)
    qbits = bernsteinVazirani(n, gate)
    bits = tuple(map(qu.bitValue, qbits))
    diff = qb.addition(delta, bits)
    if diff == n * (0,):
        print("passed bernsteinVaziraniTest")
    else:
        print("failed bernsteinVaziraniTest")
        print("    delta = " + str(delta))
```