Homework 2

May 2, 2020

0.1 Homework 2 - Basics of quantum computing in Q#

0.2 Preamble

The purpose of this homework is to familiarize you with programming in Q# and also give you intuition about some of the basics concepts that we encounter in quantum computation.

Each task has a certain number of points, for a total of **115pt**.

For each task you have to write some **Q# code** but also a **brief explanation** of why your solution works. You will be graded on both (70% of the points will be for the code and 30% for the written explanation). The written explanation can be either in the form of comments in your code or as a separate description in a text file (or both).

To turn in your homework, submit a zip file with your solutions on Moodle.

The deadline is: **May 4, 23:59 PST**.

Important! For tasks 1, 2A, 2B and 4, the only quantum operations you can use are the ones from Microsoft.Quantum.Intrinsic. This does not mean that your code has to consist *exclusively* of those operations. You can still use standard flow-control operations (like `for`, `while`, `if`, declaring variables etc) and create additional helper functions. The point is that you cannot use higher-level *quantum* functionalities from the language.

You are, of course, free to use any of the Q# libraries for the tests you design for your functions or for debugging purposes, but the *solutions* for tasks 1, 2, 4 should use quantum operations only from Microsoft.Quantum.Intrinsic.

0.2.1 Frequently Asked Questions

Q. If my solution produces the correct output on all the tests do I get full marks?
A. Not necessarily. The tests are there to help you check your implementation and you're free to add more if you wish or ignore them altogether. Your solution will be graded based on whether the implementation you wrote is correct and solves the problem (this is why it's useful to add the explanation of how your solution works).

Q. If I get over 100pt are the bonus points added to the next homework assignment?
A. No. Anything above 100 is truncated at 100.

Q. Do I get partial credit for an incomplete solution?
A. Yes.

Q. What if I don't write the code but just a written description of the solution?
A. You will get partial credit that's at most 30% of the total number of points for that task, since that's what's allotted for written explanations.

Q. Can we collaborate?
A. Yes, you are encouraged to discuss the homework with the other students, however your solution should be your own.

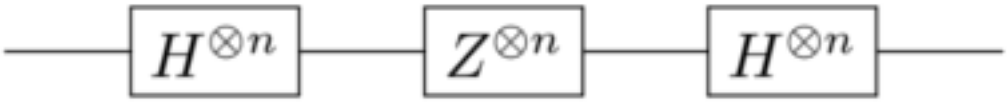
0.3 (30 pt) Task 1: Basic quantum circuits

For the following sub-tasks you can only use quantum operations from Microsoft.Quantum.Intrinsic.

This first task is meant to help you gain some familiarity with Q# and with how to implement quantum circuits.

0.3.1 (10 pt) Task 1A: Interference in a quantum circuit

Consider the following circuit on n qubits:



Recall that $\mathsf{H}^{\otimes n} = \mathsf{H} \otimes \mathsf{H} \cdots \otimes \mathsf{H}$ denotes the n-fold tensor product of the Hadamard gate, and $\mathsf{Z}^{\otimes n} = \mathsf{Z} \otimes \mathsf{Z} \cdots \otimes \mathsf{Z}$ the n-fold Z gate, respectively.

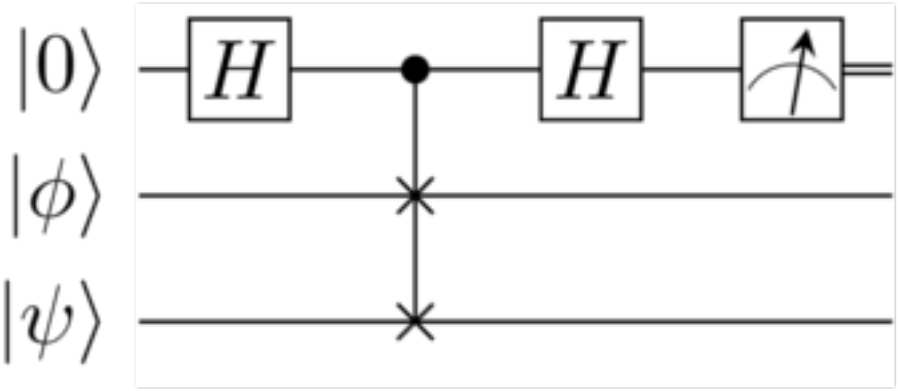
Implement the above quantum circuit in Q# by using the code template below. What is the outcome of running the circuit on $|0\rangle^{\otimes n}$ and measuring the output in the computational basis? Conclude that the circuit solves a simple n-qubit variant of the Bernstein-Vazirani problem for a particular unknown function $f_s(x) = s \cdot x \pmod 2$ with domain $\{0,1\}^n$. Feel free to verify this fact manually.

[1]:
```
operation simpleBV(qs : Qubit[]): Unit {
    // add your code in here
}
```

[1]: simpleBV

0.3.2 (20 pt) Task 1B: Estimating quantum state overlap

In this exercise, the task is to implement the following circuit known as the "SWAP Test". The input consists of two quantum states, say $|\phi\rangle$ and $|\psi\rangle$, on n qubits each and the goal is estimate $|\langle\phi|\psi\rangle|^2$. The circuit for achieving this is shown in the figure below.



In this circuit, upon measuring the top qubit, the $|0\rangle$ outcome will occur with probability $\frac{1+|\langle\phi|\psi\rangle|^2}{2}$ and the $|1\rangle$ outcome with probability $\frac{1-|\langle\phi|\psi\rangle|^2}{2}$.

Implement the circuit in Q# by adding your code into the following code template.

[23]:
```
operation SwapTest(qs: Qubit[]): Bool {
    // your code here
    return false;
}
```

[23]: `SwapTest`

The function takes as input of $2n + 1$ qubits (in the form of the register `qs`). The first qubit is the control qubit in the above circuit. The following n qubits are the $|\phi\rangle$ state and the final n are the $|\psi\rangle$ state. The function should return `true` if the top qubit was measured as $|0\rangle$ and `false` otherwise.

The operation in the middle of the circuit is a controlled-SWAP operation (also known as the Fredkin gate and denoted C-SWAP) which is based on the standard SWAP operation. As the name suggests, its action is to swap the two quantum states

$$\text{SWAP}|\phi\rangle|\psi\rangle = |\psi\rangle|\phi\rangle.$$

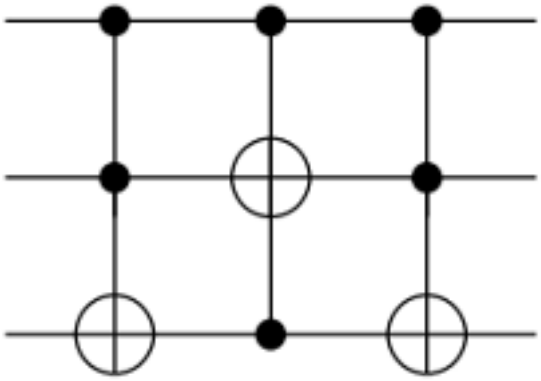
For general n-qubit states, this can be performed using the SWAP *gate,* that swaps two qubits (simply apply the SWAP gate repeatedly between pairs of qubits until all $2n$ qubits have been swapped). Here, by abuse of notation, we're writting SWAP (and C-SWAP, respectively) for the generalized version that swaps n-qubit states.

The SWAP gate is already provided in Microsoft.Quantum.Intrinsic. You will have to implement the C-SWAP gate, in order to perform the controlled-SWAP of the n-qubit states. This C-SWAP gate has an additional qubit register (the control qubit) that controls when the swap is performed. In other words, the C-SWAP performs the following opperation:

$$\text{C-SWAP}|0\rangle|\phi\rangle|\psi\rangle = |0\rangle|\phi\rangle|\psi\rangle$$

$$\mathsf{C\text{-}SWAP}|1\rangle|\phi\rangle|\psi\rangle = |1\rangle|\psi\rangle|\phi\rangle.$$

You can implement C-SWAP from *Toffoli* gates (the CCNOT gate from Microsoft.Quantum.Intrinsic), as follows:



Once you've implemented the SWAP test, test it on the following single-qubit states

$$|\phi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{1}$$

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{-i\theta}|1\rangle) \tag{2}$$

Note that the $|\phi\rangle$ state is simply a $|+\rangle$ state, which you can prepare by acting with H on $|0\rangle$. The $|\psi\rangle$ state is a "rotated" $|+\rangle$. You can prepare it using the R_Z rotation gate, that's provided in Microsoft.Quantum.Intrinsic. If we denote the probability that the test succeeds in this case as p_0, convince yourself that p_0 is given by

$$p_0 = \frac{1}{2} + \frac{1}{2}\cos^2(\theta/2).$$

You can therefore estimate $\cos(\theta)$ to very high precision by simply running the circuit a couple of times and keeping track of the number of times you see the outcome 0. If you run the circuit N times, then you can approximate p_0 as the fraction $p_0 \approx \frac{\#(\text{outcomes}=0)}{N}$, and hence $|\langle\phi|\psi\rangle|^2 = \cos^2(\theta/2) \approx 2\frac{\#(\text{outcomes}=0)}{N} - 1$. Using the fact that $cos^2(\theta/2) = \frac{1}{2}(1 + cos(\theta))$, we have that $cos(\theta) \approx 4\frac{\#(\text{outcomes}=0)}{N} - 3$. You can use the following rule of thumb: in order to get an additive ϵ approximation, i.e. within $\cos(\theta) \pm \epsilon$, you should have roughly $N = O\left(\frac{1}{\epsilon^2}\right)$ many repetitions. Here, a value of $\epsilon = 0.01$ should be sufficient.

Using a simple loop, run the circuit for every $\theta \in [0, 2\pi]$ in increments of 0.1 and compute the value of $4\frac{\#(\text{outcomes}=0)}{N} - 3$ in each step. Afterwards, plot the values against the values of θ.

0.4 (35 pt) Task 2: Preparing entangled states

For the following sub-tasks, except 2C, you can only use quantum operations from Microsoft.Quantum.Intrinsic.

In this task you need to prepare a number of generic entangled states.

0.4.1 (5 pt) Task 2A: n Bell pairs

Prepare n copies of the $|\phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ Bell state, starting from $|0\rangle^{\otimes 2n}$, by completing the following code block:

[19]:
```
operation createBell(n: Int) : Unit {
    using (qs = Qubit[2 * n]) {
        // your code here
    }
}
```

[19]: `createBell`

0.4.2 (10 pt) Task 2B: Poor man's cat state

Prepare the n-qubit poor man's cat state (see this paper for its origin), $|CAT\rangle = \frac{1}{\sqrt{2}}(|x\rangle + |\bar{x}\rangle)$, starting from $|0\rangle^{\otimes n}$ and the reference string $x \in \{0,1\}^n$ (given as an array of `Bool`), by completing the following code block:

[18]:
```
operation createCat(x: Bool[]) : Unit {
    let n = Length(x);
    using (qs = Qubit[n]) {
        //  your code here
    }
}
```

[18]: `createCat`

Here $\bar{x}$ denotes the binary complement of x. In other words, $\bar{x}$ is obtained by negating all the bits in x. Alternatively, $\bar{x} = 1^n \oplus x$. As an example of a poor man's cat state on 3 qubits, with $x = 101$, we have:

$$\frac{1}{\sqrt{2}}(|101\rangle + |010\rangle)$$

Note that this is the same as the state we would have created for $x = 010$.

0.4.3 (20 pt) Task 2C: Ham2 state

The restriction to Microsoft.Quantum.Intrinsic doesn't apply here. The full power of the language is available to you.

Prepare the n-qubit state (for $n > 1$), $|HAM2\rangle = \frac{1}{\sqrt{N}} \sum_{x \in \{0,1\}^n, |x|=2} |x\rangle$, with $N = \binom{n}{2}$, by completing the following code block:

[17]:
```
operation createHam2(n: Int) : Unit {
    // your code here
}
```

[17]:
```
createHam2
```

This state is an equal superposition of all n-bit strings having Hamming weight 2. As an example, the 3-qubit version is:

$$\frac{1}{\sqrt{3}}(|011\rangle + |101\rangle + |110\rangle)$$

You can use as many qubits as you like for this task (in other words, you're not restricted to just n qubits).

It might be useful to look at the preparation of the analogous version of the state with strings of Hamming weight 1. That state is known as the $|W\rangle$ state, and you can see a Q# implementation of it here.

0.5 (15 pt) Task 3: Elitzur-Vaidman bomb detector

For this task you will implement a version of the Elitzur-Vaidman bomb detector. In the original bomb detector the problem was this: you are given a bomb which is either live or a dud and you have to decide which it is. The live bomb will detonate if even a single photon touches it. How then can we tell whether we have a live bomb without detonating it? Elitzur and Vaidman proposed a quantum solution to solve this problem and this is what we'll try to emulate.

The specific version of the bomb detector that we're considering is the one from this paper. The setup is the following. You are given an oracle to a two-qubit gate which is promised to be either the CNOT gate (bomb) or the identity gate (dud) and you have to determine which it is. A natural approach is to just initialize them as $|10\rangle$, apply the oracle and then measure the second qubit. If the second qubit is $|0\rangle$, the oracle must be the identity gate and if we measure a $|1\rangle$, it must be the CNOT gate.

There are two constraints however. The first is that *you are only allowed to act on the first qubit* (the control qubit). The second is that whenever the oracle is applied on the qubits, the second one (the one that might be flipped) is measured immediately in the computational basis. If the second qubit is ever measured in the $|1\rangle$ state, the "bomb" explodes. If the qubit is not measured in the $|1\rangle$ state, you can continue applying operations (including the oracle) to the state. However, each application of the oralce is immediately followed by a computational-basis measurement of the second qubit.

Implement your CNOT detector by completing the `mySolution` function in the following code block:

[2]:
```
    // This is the function you need to implement
    // It returns two Boolean values
    // - The first is whether it has determined (with high probability) whether it's a bomb or not
    //   In other words, it should be true if the detection was performed and false otherwise;
    // - The second is the decision of whether it is a bomb or not
    //   In other words, it should be true if it's a bomb and false otherwise.
    operation mySolution(N: Int, counter: Int, q: Qubit): (Bool, Bool) {
        // your solution here
        return (false, false);
    }

    operation bombDetector(oracle: ((Qubit, Qubit) => Unit)): Bool {
        let N = 100; // number of iterations; You are allowed to change this value

        // you are not allowed to change the code below
        mutable counter = 0; // counts number of iterations
        mutable detection = false; // checks if detection was made
        mutable isBomb = false; // will be the output of the function
        using (qs = Qubit[2]) {
            repeat {
                set counter += 1; // increment counter
                set (detection, isBomb) = mySolution(N, counter, qs[0]); // run detection code
                oracle(qs[0], qs[1]); // apply oracle
                let res = M(qs[1]); // measure second qubit
                if (res == One) { // if outcome is |1> bomb explodes
                    Message("Boom!");
                    set counter = N; // end loop
                    set isBomb = true; // found a bomb, but it's too late...
                }
            } until (counter == N or detection == true); // end when counter ends or detection worked
            ResetAll(qs);
        }

        return isBomb;
    }
```

[2]:
```
bombDetector, mySolution
```

The code above works as follows: `bombDetector` is performing a loop of up to `N` iterations (you are allowed to modify the value of `N`). Inside this loop, your solution code is run first. The function,

`mySolution`, receives as arguments the total number of iterations, `N`, the current iteration, `counter`, as well as the first qubit in the register `qs` (`qs[0]`). The function returns two boolean values, indicating whether a detection has been performed and what the result of that detection is (for this latter value, it should be `true` if a bomb has been detected and `false` otherwise). The oracle is then applied to the two qubits and the second qubit is measured in the computational basis. If a $|1\rangle$ is detected, the loop terminates and "Boom!" is printed, indicating that the bomb has exploded. Otherwise, the code continues to run until either the maximum number of iterations has been reached, or `mySolution` has arrived at a detection. The result of that detection is the output of the function.

You need to implement your solution so that, when `oracle` is a CNOT (the bomb case), the probability of triggerring it is less than 1% and your accuracy for detecting whether the oracle is CNOT or identity is at least 99%. In other words, `bombDetector` must detect correctly with very high probability and have a very low probability of causing a detonation.

For this task, you need to implement the `mySolution` function. You can define auxillary functions (and even use additional qubits in `mySolution`, though you won't need to), however you are not allowed to edit the `bombDetector` function except for the value of `N`, that determines the number of iterations in the loop.

0.6 (15 pt) Task 4: Quantum adder

For this task you can only use quantum operations from Microsoft.Quantum.Intrinsic.

In this task you need to implement a quantum circuit that adds two registers and puts the result in a third register. On basis states, the operation should perform the following

$$|x\rangle|y\rangle|z\rangle \to |x\rangle|y\rangle|z \oplus (x+y)\rangle$$

where $|x\rangle$ and $|y\rangle$ are n-bit registers, $|z\rangle$ is an $(n+1)$-bit register and $x+y$ denotes addition *with carry* of x and y. As an example, consider

$$|011\rangle|101\rangle|000\rangle \to |011\rangle|101\rangle|1000\rangle$$

As we can see, the left-most bit is the most significant bit and the right-most bit is the least significant bit. Complete this task by filling in the code-block below:

[3]:
```
operation binaryAdder(x: Qubit[], y: Qubit[], z: Qubit[]): Unit {
    // your code here
}
```

[3]:
```
binaryAdder
```

For a reference, see: https://arxiv.org/pdf/quant-ph/0008033.pdf. Note that this paper implements a ripple carry adder, that maps $|x\rangle|y\rangle|b\rangle$ to $|x\rangle|x+y\rangle|c \oplus b\rangle$, where $|b\rangle$ is one qubit and c is the carry bit of the addition between x and y. The quantum adder that you need to implement acts on $3n+1$ qubits in total.

0.7 (20 pt) Task 5: Quantum dice

In this task, you will implement a version of the "weird dice" example that was discussed in Lecture 1.

We start with two registers of 3 qubits each, denoted $|x\rangle$ and $|y\rangle$. To make things easier, we will assume each die takes values between 0 and 7 (rather than the traditional 1 to 6). Each die will start off in the $|000\rangle$ state. Next, perform these operations:

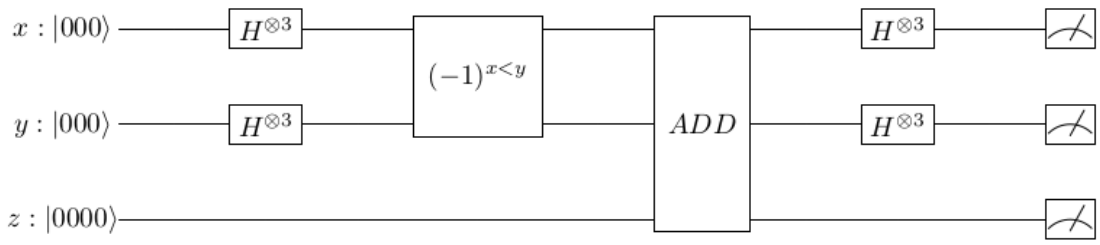
1. Place each die in an equal superposition of all 8 possible values by initializing each of the two registers as $|000\rangle$ and Hadamarding them.
2. Perform the operation that we'll refer to as `phaseOnAscend`. The effect of this operation is to add a phase of (-1) to the state if $x < y$ (the registers are in strictly ascending order) and otherwise it does nothing. In other words,

$$phaseOnAscend\ |x\rangle|y\rangle = (-1)^{[x<y]}|x\rangle|y\rangle$$

To do this, you might find this operation useful.

3. Add the values of the two register into a third register $|z\rangle$, of 4 qubits (initialized as $|0000\rangle$). For this step, you can use either the quantum adder from Task 4 or one of the adders already implemented in Q# (see the documentation).
4. Hadamard the $|x\rangle$ and $|y\rangle$ registers to recombine the amplitudes (this will cause cancellation resulting from the phases added by `phaseOnAscend`).
5. Measure all registers in the computational basis. We say that the result in the $|z\rangle$ register is *accepted* if and only if the top registers are both in the $|000\rangle$ state. Otherwise, we ignore the z outcome.

Schematically, the circuit for this procedure should look like this:



Repeat this procedure a number of times and create a histogram of the *accepted* z outcomes.

There is no code block for this task. You are free to implement this however you find it most convenient. You can either add your solution into this Jupyter notebook, or add it as a separate file in the archive. Either way, you should also have the histogram plot from the statistics of measuring $|z\rangle$.

If your implementation takes too long with these 10 qubits (as well as any additional ancilla you may be using), you can also have each die be supported on just 2 qubits and the $|z\rangle$ register be on 3 qubits, for a total of 7 qubits.