Microsoft Q# Coding Contest - Winter 2019 - Warmup Round
February 22 - 25, 2019

Mariia Mykhailova
Quantum Architectures and Computation Group, Microsoft Research, Redmond, WA, United States

TASKS AND SOLUTIONS

G1. AND oracle

Implement a quantum oracle on N qubits which implements the following function:

$$f(\boldsymbol{x}) = x_0 \wedge x_1 \wedge \dots \wedge x_{N-1}$$

You have to implement an operation which takes the following inputs:

- an array of N qubits x in arbitrary state (input register),
- a qubit y in arbitrary state (output qubit),

and performs a transformation $|x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$. The operation doesn't have an output; its `output` is the state in which it leaves the qubits.

Solution. The effect of the AND oracle can be re-worded as follows: given the input register and the output qubit, flip the state of the output qubit if and only if all qubits in the input register are in the state $|1\rangle$.

This is exactly the definition of a controlled version of the X gate: the X gate flips the state of the qubit to which it is applied, and the controlled version of an operation applies this operation if and only if all control qubits are in the state $|1\rangle$.

Q# has built-in gates for the single-controlled X (CNOT) and doubly-controlled X (CCNOT); in general you can use `Controlled` functor to create a version of an operation controlled on an arbitrary array of qubits.

Listing 1. AND oracle

```
namespace Solution {
    open Microsoft.Quantum.Primitive;
    open Microsoft.Quantum.Canon;

    operation Solve (x : Qubit[], y : Qubit) : Unit {
        body (...) {
            Controlled X (x, y);
        }
        adjoint auto;
    }
}
```

G2. OR oracle

Implement a quantum oracle on N qubits which implements the following function:

$$f(\boldsymbol{x}) = x_0 \vee x_1 \vee \dots \vee x_{N-1}$$

You have to implement an operation which takes the following inputs:

- an array of N qubits x in arbitrary state (input register),
- a qubit y in arbitrary state (output qubit),

and performs a transformation $|x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$. The operation doesn't have an output; its `output` is the state in which it leaves the qubits.

Solution. We can use De Morgan's laws to rewrite the function implemented by the OR oracle as follows:

$$f(\boldsymbol{x}) = \neg(\neg x_0 \wedge \neg x_1 \wedge \cdots \wedge \neg x_{N-1})$$

Thus, the effect of the OR oracle can be re-worded as follows: given the input register and the output qubit, flip the state of the output *unless* all qubits in the input register are in the state $|0\rangle$. This means that we can implement the operation in the following way:

1. Flip the state of the output qubit to cover majority of the cases.
2. Flip all qubits of the input register.
3. Apply Controlled X with the input register as control and the output qubit as target - this will flip the state of the output qubit again (to revert the effect of the first flip) if all qubits in the input register started in the state $|0\rangle$.
4. Flip all qubits of the input register to return them to their original state (since the oracle should not change the state of its input).

For convenience we can do steps 2-4 using ControlledOnInt library operation.

Listing 2. OR oracle

```
namespace Solution {
    open Microsoft.Quantum.Primitive;
    open Microsoft.Quantum.Canon;

    operation Solve (x : Qubit[], y : Qubit) : Unit {
        body (...) {
            X(y);
            (ControlledOnInt(0, X))(x, y);
        }
        adjoint auto;
    }
}
```

G3. Palindrome checker oracle

Implement a quantum oracle on N qubits which checks whether the vector $\boldsymbol{x}$ is a palindrome (i.e., implements the function $f(\boldsymbol{x}) = 1$ if $\boldsymbol{x}$ is a palindrome, and 0 otherwise).

You have to implement an operation which takes the following inputs:

- an array of N $(1 \le N \le 8)$ qubits x in an arbitrary state (input register),
- a qubit y in an arbitrary state (output qubit),

and performs a transformation $|x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$. The operation doesn't have an output; its `output` is the state in which it leaves the qubits.

Solution. This problem is trickier, since you can not express the condition `the bit string is a palindrome` as a single AND or OR expression. You can express it in the following way:

$$f(\boldsymbol{x}) = (x_0 = x_{N-1}) \wedge (x_1 = x_{N-2}) \wedge \cdots$$

First we need to be able to check that the states of two qubits are equal. You can rewrite the condition $x_i = x_j$ as $x_i \oplus x_j = 0$, recall that to compute XOR of two qubits you can do two CNOTs with each of the qubits as control and the target qubit as target, and finally flip the target qubit, since the condition is true if XOR is 0.

To combine the results of the individual comparisons into the value of the function, you need to:

- allocate an array of auxillary (`ancilla`) qubits,
- compare each pair of qubits and write the results into the corresponding ancilla qubit,
- calculate AND of the states of the ancilla qubits and write the result into the target qubit,
- (important!) uncompute the states of the ancilla qubits to return them to $|0\rangle$ state so that they can be released.

We'll focus on the last step in more detail, since it contains an important subtlety. You can not simply measure the ancilla qubits to reset them before release, since this can affect the state of the other qubits. Consider, for example, the calculation for input/output in state $(|00\rangle_x + |01\rangle_x) \otimes |0\rangle_y$. We need to end up in the state $|00\rangle_x |1\rangle_y + |01\rangle_x |0\rangle_y$ Here are the intermediary states of the system after each step:

- allocate the ancilla: $(|00\rangle_x + |01\rangle_x) \otimes |0\rangle_a |0\rangle_y$
- compare input qubits: $(|00\rangle_x |1\rangle_a + |01\rangle_x |0\rangle_a) \otimes |0\rangle_y$
- calculate the value of the function based on the state of the ancilla: $|00\rangle_x |1\rangle_a |1\rangle_y + |01\rangle_x |0\rangle_a |0\rangle_y$
- if we measure the ancilla qubit now, the state of the system will collapse to either $|00\rangle_x |1\rangle_y$ or $|01\rangle_x |0\rangle_y$, depending on the outcome of the measurement, instead of the superposition state we need. However, if we perform adjoint of the step 2 here, **undoing** the computation we did to get the state of the ancilla, the state of the system will become $(|00\rangle_x |1\rangle_y + |01\rangle_x |0\rangle_y) \otimes |0\rangle_a$, and we'll be able to release the ancilla qubit without damaging the state of other qubits.

The pattern of `apply transformation U - apply transformation V - apply adjoint of transformation U` is extremely widespread in quantum computing, so there is a library operation With which implements it.

Listing 3. Palindrome checker oracle

```
namespace Solution {
    open Microsoft.Quantum.Primitive;
    open Microsoft.Quantum.Canon;

    operation EvaluateEqualityClauses (x : Qubit[], ancillaRegister : Qubit[]) : Unit {
        body (...) {
            let N = Length(x);
            for (i in 0 .. N / 2 - 1) {
                // Compute XOR of x[i] and x[N - 1 - i] into anc[i]
                CNOT(x[i], ancillaRegister[i]);
                CNOT(x[N - 1 - i], ancillaRegister[i]);
                // Negate it (XOR has to equal 0 for equality to be 1)
                X(ancillaRegister[i]);
            }
        }

        adjoint auto;
    }

    operation Solve (x : Qubit[], y : Qubit) : Unit {
        body (...) {
            let N = Length(x);
            using (anc = Qubit[N / 2]) {
                WithA(EvaluateEqualityClauses(x, _), Controlled X(_, y), anc);
            }
        }
        adjoint auto;
    }
}
```

U1. Anti-diagonal unitary

Implement a unitary operation on N qubits which is represented by an anti-diagonal matrix (a square matrix of size 2^N which has non-zero elements on the diagonal that runs from the top right corner to the bottom left corner and zero elements everywhere else).

For example, for $N = 2$ the matrix of the operation should have the following shape:

```
...X
..X.
.X..
X...
```

Here `X` denotes a `non-zero` element of the matrix (a complex number which has the square of the absolute value greater than or equal to 10^{-5}), and `.` denotes a `zero` element of the matrix (a complex number which has the square of the absolute value less than 10^{-5}).

Solution. Let's consider applying U_1 to the basis state $|0...0\rangle$: the result is described by the first column of the matrix you are given. Only the last row of that column contains a non-zero element, i.e., $U_1|0...0\rangle = \alpha_{0...0}|1...1\rangle$.

The operation you need to implement (let's denote it as U_1) is a unitary, thus it preserves the inner product of the vectors it is applied to, and in particular it preserves the norm of the vectors it is applied to. The norm of the basis vector $|0...0\rangle$ is 1, so the norm of $U_1|0...0\rangle$ also has to be 1, thus $|\alpha_{0...0}| = 1$.

We can apply the same logic to the rest of the entries in the matrix, and realize that the absolute value of each non-zero element of the matrix has to be 1. For simplicity we can choose all these elements to equal 1; here is the example for $N = 2$:

$$U_1 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Now, we need to actually implement the operation which corresponds to this unitary matrix. We can again consider its effect on each basis state, starting with the case $N = 2$:

$$\begin{aligned} U_1|00\rangle &= |11\rangle \\ U_1|10\rangle &= |01\rangle \\ U_1|01\rangle &= |10\rangle \\ U_1|11\rangle &= |00\rangle \end{aligned}$$

This is exactly the result of flipping each qubit independently, which can be done by applying an X gate on each qubit.

You can also think of it as representing the matrix U_1 as a tensor product: $U_1 = X \otimes X$, where $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. As the size of the matrix grows, it still can be represented as a tensor product of the smaller matrix of this shape and an X gate.

Listing 4. Anti-diagonal unitary

```
namespace Solution {
    open Microsoft.Quantum.Primitive;
    open Microsoft.Quantum.Canon;

    operation Solve (qs : Qubit[]) : Unit {
        ApplyToEach(X, qs);
    }
}
```

U2. Chessboard unitary

Implement a unitary operation on N qubits which is represented by a square matrix of size 2^N in which zero and non-zero elements form a chessboard pattern with alternating 2x2 squares (top left square formed by non-zero elements). For example, for $N = 3$ the matrix of the operation should have the following shape:

```
XX..XX..
XX..XX..
..XX..XX
..XX..XX
XX..XX..
XX..XX..
..XX..XX
..XX..XX
```

Solution. If you look at the UnitaryPatterns kata, you will recognize this task as a something between tasks 3 (a pattern of two large diagonal blocks) and 4 (a chessboard pattern of 1x1 squares). Both of them are formed using only Hadamard gates, so it makes sense to try and see if the pattern in this task can be expressed this way as well.

Consider the case of $N = 2$; the pattern you need to implement in this task is exactly the same as the pattern of two large diagonal blocks, which you can obtain by applying an H gate to the first qubit and doing nothing to the second qubit. To scale this pattern as you add qubits, you need to apply H gates to the newly added qubits; a little experimentation shows that in the end you need to apply an H gate to each qubit except the second one.

More formally, you can think about this task in terms of tensor products. If you multiply H by a matrix A, you effectively repeat the pattern of that matrix 4 times (once in each of the quarters of the result); if you multiply a matrix A by H, you replace each zero element of A with a block of 4 zeroes, and each non-zero element with a block of non-zeroes. Check it manually or using a tensor product calculator like this one.

The smallest block you need to obtain to repeat it is

$$I \otimes H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

Now you can repeat it as many times as you need by multiplying H gates by it:

$$H \otimes (I \otimes H) = \begin{bmatrix} I \otimes H & I \otimes H \\ I \otimes H & -I \otimes H \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 \\ 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 \\ 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & -1 & -1 \\ 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 \end{bmatrix}$$

In the end the tensor product for the pattern of N qubits is $H \otimes ... \otimes H \otimes I \otimes H$. Remember that the pattern is given using little endian indices, so to translate the tensor product of the matrices to the gates applied to the qubits you need reverse the order of the gates: the first gate in the tensor product is applied to the last qubit, the second gate - to the second-to-last qubit etc.

Listing 5. Chessboard unitary

```
namespace Solution {
    open Microsoft.Quantum.Primitive;

    operation Solve (qs : Qubit[]) : Unit {
        H(qs[0]);
        for (i in 2 .. Length(qs) - 1) {
            H(qs[i]);
        }
    }
}
```

U3. Block unitary

Implement a unitary operation on N qubits which is represented by the following square matrix of size 2^N:

- top right and bottom left quarters are filled with zero elements,
- top left quarter is an anti-diagonal matrix of size 2^{N-1},
- bottom right quarter is filled with non-zero elements.

For example, for $N = 2$ the matrix of the operation should have the following shape:

```
.X..
X...
..XX
..XX
```

Solution. This matrix can not be represented as a tensor product, but it clearly involves two patterns from previous tasks - the top left block is the pattern from task U1, obtained as a tensor product of X gates, and the bottom right block is a tensor product of Hadamard gates. But how to combine them?

The top left block is the area where the most significant bit of both input and output indices equals 0. (Remember that with little endian encoding of the indices this bit is stored in the last qubit.) This means that the left half of the matrix can be described as follows: if the last qubit of the input is in state $|0\rangle$, leave that qubit unchanged and apply an X gate to each of the rest of the qubits. This can be done using zero-controlled version of the transformation, i.e., by flipping the state of the last qubit, applying controlled transformation and flipping the state of the last qubit again.

Similarly, the bottom right block is the area where the most significant bit equals 1, and its effect on the qubits can be described as follows: if the last quibt of the input is in state $|1\rangle$, leave that qubit unchanged and apply a Hadamard gate to each of the rest of the qubits.

Listing 6. Block unitary

```
namespace Solution {
    open Microsoft.Quantum.Primitive;
    open Microsoft.Quantum.Canon;

    operation Solve (qs : Qubit[]) : Unit {
        ApplyToEach(Controlled H([Tail(qs)], _), Most(qs));
        ApplyToEach((ControlledOnInt(0, X))([Tail(qs)], _), Most(qs));
    }
}
```