Microsoft Q# Coding Contest - Winter 2019 - Main Contest
March 1-4, 2019

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

TASKS AND SOLUTIONS

A1. Generate state 00 + 01 + 10

You are given two qubits in state $|00\rangle$. Your task is to create the following state on them:

$$\frac{1}{\sqrt{3}}\big(|00\rangle + |01\rangle + |10\rangle\big)$$

You have to implement an operation which takes an array of 2 qubits as an input and has no output. The `output` of your solution is the state in which it left the input qubits.

Solution. This was one of the easier problems in the contest, and allowed plenty of different solutions.

The straightforward solution is based on rotations and is described nicely in this StackExchange answer: start with the state $|00\rangle$, rotate the first qubit to get $\big(\frac{\sqrt{2}}{\sqrt{3}}|0\rangle + \frac{1}{\sqrt{3}}|1\rangle\big) \otimes |0\rangle$, and end with a controlled operation which applies a Hadamard gate to the second qubit if the first qubit is in $|0\rangle$ state.

Listing 1. Generate state $|00\rangle + |01\rangle + |10\rangle$ - rotation solution

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

    operation Solve (qs : Qubit[]) : Unit {
        Ry(2.0 * ArcSin(1.0 / Sqrt(3.0)), qs[0]);
        (ControlledOnInt(0, H))([qs[0]], qs[1]);
    }
}
```

Another solution doesn't require calculating rotation angles, and uses post-selection instead.

You start by generating an equal superposition of all basis states $\frac{1}{2}\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big)$ using two Hadamard gates. After that you allocate an extra qubit and do a controlled X gate with that qubit as target and the original qubits as controls to get a state $\frac{1}{2}\big(|00\rangle + |01\rangle + |10\rangle\big) \otimes |0\rangle + \frac{1}{2}|11\rangle \otimes |1\rangle$.

Now, consider what happens if you measure the last qubit? If the measurement result is 1, the first two qubits collapse to the state $|11\rangle$, which is not useful. However, if the measurement result is 0, the first two qubits collapse to the state $\frac{1}{\sqrt{3}}\big(|00\rangle + |01\rangle + |10\rangle\big)$, which is exactly the state we need!

The probability of getting the right state on the first try is 75%. You can use repeat-until-success loop to repeat the process until you get the necessary state.

Listing 2. Generate state $|00\rangle + |01\rangle + |10\rangle$ - post-selection solution

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

    operation Solve (qs : Qubit[]) : Unit {
        using (ancilla = Qubit()) {
            repeat {
                ApplyToEach(H, qs);
                Controlled X(qs, ancilla);
                let res = MResetZ(ancilla);
```

```
            }
            until (res == Zero)
            fixup {
                ResetAll(qs);
            }
        }
    }
}
```

Finally, you can use a library operation to prepare the necessary state. PrepareUniformSuperposition prepares a uniform superposition of the basis states that encode numbers 0 through $M - 1$, which is exactly the required state for $M = 3$.

Listing 3. Generate state $|00\rangle + |01\rangle + |10\rangle$ - library operation

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

    operation Solve (qs : Qubit[]) : Unit {
        PrepareUniformSuperposition(3, BigEndian(qs));
    }
}
```

A2. Generate equal superposition of four basis states

You are given N qubits in the zero state $|0...0\rangle$. You are also given four distinct bit vectors which describe four different basis states on N qubits $|\psi_i\rangle$.

Your task is to generate a state which is an equal superposition of the given basis states:

$$|S\rangle = \frac{1}{2}\big(|\psi_0\rangle + |\psi_1\rangle + |\psi_2\rangle + |\psi_3\rangle\big)$$

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

- an array of N ($2 \le N \le 16$) qubits,
- a two-dimensional array of Boolean values *bits* representing the basis states $|\psi_i\rangle$. *bits* will have exactly 4 elements, each of *bits*$[i]$ describing the basis state $|\psi_i\rangle$. Each of *bits*$[i]$ will have N elements, *bits*$[i][j]$ giving the state of qubit j in the basis state $|\psi_i\rangle$. Bit values true and false corresponding to $|1\rangle$ and $|0\rangle$ states, respectively; for example, for $N = 2$ an array `[false, true]` will represent the basis state $|01\rangle$. Each pair of *bits*$[i]$ and *bits*$[j]$ will differ in at least one position for $i \neq j$.

The operation doesn't have an output; its `output` is the state in which it leaves the qubits.

Solution. This problem builds up on the problem A3 from Q# Coding Contest - Summer 2018. In it you had to generate an equal superposition of two basis states; in this task you have to generate an equal superposition of four basis states. However, the approach remains similar: you can allocate extra qubits (one qubit if you need a superposition of two states, two qubits if you need a superposition of four states), use them to put the main qubits into the required superposition, and then uncompute the extra qubits to return them to 0 state before releasing them (you can not just measure them, since that would destroy the superposition you worked so carefully to build, and leave you with just one of the states instead).

Listing 4. Generate equal superposition of four basis states

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

```
    operation Solve (qs : Qubit[], bits : Bool[][]) : Unit {
        using (anc = Qubit[2]) {
            // Put the ancillas into equal superposition of 2-qubit basis states
            ApplyToEach(H, anc);

            // Set up the right pattern on the main qubits with control on ancillas
            for (i in 0 .. 3) {
                for (j in 0 .. Length(qs) - 1) {
                    if ((bits[i])[j]) {
                        (ControlledOnInt(i, X))(anc, qs[j]);
                    }
                }
            }

            // Uncompute the ancillas, using patterns on main qubits as control
            for (i in 0 .. 3) {
                if (i % 2 == 1) {
                    (ControlledOnBitString(bits[i], X))(qs, anc[0]);
                }
                if (i / 2 == 1) {
                    (ControlledOnBitString(bits[i], X))(qs, anc[1]);
                }
            }
        }
    }
}
```

B1. Distinguish three-qubit states

You are given 3 qubits which are guaranteed to be in one of the two states:

- $|\psi_0\rangle = \frac{1}{\sqrt{3}}\big(|100\rangle + \omega|010\rangle + \omega^2|001\rangle\big)$, or
- $|\psi_1\rangle = \frac{1}{\sqrt{3}}\big(|100\rangle + \omega^2|010\rangle + \omega|001\rangle\big)$, where $\omega = e^{2i\pi/3}$.

Your task is to perform necessary operations and measurements to figure out which state it was and to return 0 if it was $|\psi_0\rangle$ state or 1 if it was $|\psi_1\rangle$ state. The state of the qubits after the operations does not matter.

Solution. Let's start by considering how we would prepare one of these states, starting with the $|000\rangle$ state. You can start by preparing the W state on 3 qubits (problem A4 from Q# Coding Contest - Summer 2018), and then adjust their phases using appropriate rotations on some of the qubits (R1 gate is really indispensable for this - it is vastly more convenient than the more standard Rz rotations).

Now, back to the task at hand: how to distinguish the given states? They look very similar, and it might take some effort even to convince yourself that they are indeed orthogonal (they are). We need to find an operation which will transform them into a pair of states which can be distinguished more obviously.

It would be nice, for example, to transform one of the states into the $|000\rangle$ state, and the second one into... Actually, it doesn't matter what exactly the second state is transformed into - we know that the states $|\psi_0\rangle$ and $|\psi_1\rangle$ started orthogonal, and if the transformation we use is a unitary transformation U, it preserves inner products, so the states $U|\psi_0\rangle$ and $U|\psi_1\rangle$ are guaranteed to be orthogonal! If we choose U in such a way that $U|\psi_0\rangle = |000\rangle$, we know that $U|\psi_1\rangle$ will be some superposition of basis states, neither of which will be $|000\rangle$. Then we'll need to distinguish the $|000\rangle$ state from such superposition, which is really easy - you can measure each qubit and check if all of the measurement results are 0 or if one or more of them are 1s.

As for the transformation U, we can use an adjoint of the preparation routine we described in the first paragraph: if a routine transforms $|000\rangle$ into $|\psi_0\rangle$, its adjoint will transform $|\psi_0\rangle$ into $|000\rangle$.

Listing 5. Distinguish three-qubit states

```
namespace Solution {
```

```
    open Microsoft.Quantum.Primitive;
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Extensions.Math;
    open Microsoft.Quantum.Extensions.Convert;

    operation WState_Arbitrary_Reference (qs : Qubit[]) : Unit {

        body (...) {
            let N = Length(qs);

            if (N == 1) {
                X(qs[0]);
            } else {
                let theta = ArcSin(1.0 / Sqrt(ToDouble(N)));
                Ry(2.0 * theta, qs[0]);

                (ControlledOnInt(0, WState_Arbitrary_Reference))(qs[0 .. 0], qs[1 .. N - 1]);
            }
        }

        adjoint auto;
        controlled auto;
        controlled adjoint auto;
    }

    operation Solve (qs : Qubit[]) : Int {
        // map the first state to 000 state and the second one to something orthogonal to it
        R1(-2.0 * PI() / 3.0, qs[1]);
        R1(-4.0 * PI() / 3.0, qs[2]);
        Adjoint WState_Arbitrary_Reference(qs);

        return MeasureInteger(LittleEndian(qs)) == 0 ? 0 | 1;
    }
}
```

B2. Not A, not B or not C?

You are given a qubit which is guaranteed to be in one of the following states:

- $|A\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big)$,
- $|B\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle + \omega|1\rangle\big)$, or
- $|C\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle + \omega^2|1\rangle\big)$, where $\omega = e^{2i\pi/3}$.

These states are not orthogonal, and thus can not be distinguished perfectly. Your task is to figure out in which state the qubit is *not*. More formally:

- If the qubit was in state $|A\rangle$, you have to return 1 or 2.
- If the qubit was in state $|B\rangle$, you have to return 0 or 2.
- If the qubit was in state $|C\rangle$, you have to return 0 or 1.
- In other words, return 0 if you're sure the qubit was *not* in state $|A\rangle$, return 1 if you're sure the qubit was *not* in state $|B\rangle$, and return 2 if you're sure the qubit was *not* in state $|C\rangle$.

Your solution will be called 1000 times, each time the state of the qubit will be chosen as $|A\rangle$, $|B\rangle$ or $|C\rangle$ with equal probability. The state of the qubit after the operations does not matter.

Solution. The task is a simple game inspired by a quantum detection problem due to Holevo [1] and Peres/Wootters [2]. In the game, a player A thinks of a number (0,1 or 2) and the opponent, player B, tries to guess any number but the one chosen by player A. Classically, if you just made a guess, you'd have to ask two questions to be right 100% of the time. If instead, player A prepares a qubit with 0, 1, or 2 encoded into three single qubit states that are at an angle of 120 degrees with respect to each other and then hands the state to the opponent, then player B can apply a Positive Operator Valued Measure (POVM) consisting of 3 states that are perpendicular to the states chosen by player A. It can be shown that this allows B to be right 100% of the time with only 1 measurement, which is something that is not achievable with a von Neumann measurement on 1 qubit. See also Peres' book [3, Chapter 9.6] for a nice description of the optimal POVM.

Next, we address how we can implement the mentioned POVM by way of a von Neumann measurement, and then how to implement said von Neumann measurement in Q#. First, we arrange the given vectors into columns of a matrix:

$$M = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & 1 \\ 1 & \omega & \omega^2 \end{pmatrix},$$

where $\omega = \exp(2\pi i/3)$ denotes a primitive 3rd root of unity. Our task will be to implement the rank 1 POVM given by vectors v_1, v_2, v_3 that live in the space of 2 qubits and are perpendicular to the columns of M. How can a vector with 4 components (such as the v_i) be orthogonal to a vector with only two components (such as the columns of M)? The answer is that we will have to "pad" the vectors in M with two additional components, both being 0. In one final (minor) complication, we need to add a fourth vector v_4 and make sure that together, $v_1 \ldots, v_4$ form an orthonormal basis so that we can implement them using a von Neumann measurement.

We can for instance pick the following choice for $v_1, \ldots, v_4$, arranged into the rows of a unitary matrix:

$$U = \frac{1}{\sqrt{3}} \begin{pmatrix} 1 & -1 & 1 & 0 \\ 1 & -\omega^2 & \omega & 0 \\ 1 & -\omega & \omega^2 & 0 \\ 0 & 0 & 0 & -i\sqrt{3} \end{pmatrix}.$$

Notice that indeed applying U to input states given by column i of M (padded with two zeros to make it a vector of length 4), where $i = 0, 1, 2$ will never return the label i as the corresponding vectors are perpendicular.

We are therefore left with the problem of implementing U as a sequence of elementary quantum gates. Notice that $U \cdot \mathrm{diag}(1, -1, 1, -1)$ is equal to

$$U \cdot (\mathbf{1}_2 \otimes Z) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & \omega^2 & \omega & 0 \\ 1 & \omega & \omega^2 & 0 \\ 0 & 0 & 0 & i\sqrt{3} \end{pmatrix}.$$

Using a technique used in the Rader (also sometimes called Rader-Winograd) decomposition of the discrete Fourier transform [4], which reduces it to a cyclic convolution, we apply a 2×2 Fourier transform on the indices $i, j = 1, 2$ of this matrix (i.e. a block matrix which consists of a direct sum of blocks $\mathbf{1}_1$, H, and $\mathbf{1}_1$ which we abbreviate in short as $\mathrm{diag}(1, H, 1)$). This yields

$$\mathrm{diag}(1, H, 1) \cdot U \cdot (\mathbf{1}_2 \otimes Z) \cdot \mathrm{diag}(1, H, 1) = \begin{pmatrix} 1/\sqrt{3} & \sqrt{2/3} & 0 & 0 \\ \sqrt{2/3} & -1/\sqrt{3} & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & 0 & 0 & i \end{pmatrix}.$$

This implies that after multiplication with the diagonal operator $(S^\dagger \otimes \mathbf{1}_2)$, we are left with

$$\mathrm{diag}(1, H, 1) \cdot U \cdot (\mathbf{1}_2 \otimes Z) \cdot \mathrm{diag}(1, H, 1) \cdot (S^\dagger \otimes \mathbf{1}_2) = \begin{pmatrix} 1/\sqrt{3} & \sqrt{2/3} & 0 & 0 \\ \sqrt{2/3} & -1/\sqrt{3} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

which is a (zero-)controlled rotation R around the Y-axis by an angle given by $\arccos(\sqrt{2/3})$. Putting everything together, i.e., by applying the inverses of gates so that we find that

$$U = \mathrm{diag}(1, H, 1) \cdot R \cdot (S \otimes \mathbf{1}_2) \cdot \mathrm{diag}(1, H, 1) \cdot (\mathbf{1}_2 \otimes Z).$$

Noting finally, that to apply this sequence of unitaries to a column vector, we have to apply it in reverse when writing it as a program (as actions on vectors are left-associative). After applying a circuit identity that simplifies the resulting circuit after implementing $\mathrm{diag}(1, H, 1)$ via controlled H, we arrive at the listing shown below.

Listing 6. Not A, not B or not C?

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

    operation Solve (q : Qubit) : Int {
        mutable output = 0;
        let alpha = ArcCos(Sqrt(2.0 / 3.0));

        using (a = Qubit()) {
            Z(q);
            CNOT(a, q);
            Controlled H([q], a);
            S(a);
            X(q);
            (ControlledOnInt(0, Ry))([a], (-2.0 * alpha, q));
            CNOT(a, q);
            Controlled H([q], a);
            CNOT(a, q);

            // finally, measure in the standard basis
            let res0 = MResetZ(a);
            let res1 = M(q);

            // dispatch on the cases
            if (res0 == Zero && res1 == Zero) {
                set output = 0;
            }
            elif (res0 == One && res1 == Zero) {
                set output = 1;
            }
            elif (res0 == Zero && res1 == One) {
                set output = 2;
            }
            else {
                // this should never occur
                set output = 3;
            }
        }

        return output;
    }
}
```

C1. Alternating bits oracle

Implement a quantum oracle on N qubits which checks whether the bits in the input vector $\boldsymbol{x}$ alternate (i.e., implements the function $f(\boldsymbol{x}) = 1$ if $\boldsymbol{x}$ does not have a pair of adjacent bits in state 00 or 11).

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

- an array of N ($1 \leq N \leq 7$) 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. Note that the input register x has to remain unchanged after applying the operation. You are not allowed to use measurements in your operation.

Solution. This was another one of the easiest problems in the contest, and it could have been solved in multiple way. The easiest way was to follow the approach used in the task 1.2 of the GroversAlgorithm kata: find the states which are marked by the oracle and mark them using controlled X gates. This oracle marks only two states - the state 01010... and the state 10101..., so arranging the states of the qubits into these patterns and marking them is very straightforward.

Listing 7. Alternating bits oracle

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

    operation FlipAlternatingPositionBits (register : Qubit[], firstIndex : Int) : Unit {

        body (...) {
            // iterate over elements in every second position, starting with firstIndex
            for (i in firstIndex .. 2 .. Length(register) - 1) {
                X(register[i]);
            }
        }

        adjoint auto;
    }

    operation Solve (x : Qubit[], y : Qubit) : Unit {
        body (...) {
            // first mark the state with 1s in even positions,
            // then mark the state with 1s in odd positions
            for (firstIndex in 0..1) {
                FlipAlternatingPositionBits(x, firstIndex);
                Controlled X(x, y);
                Adjoint FlipAlternatingPositionBits(x, firstIndex);
            }
        }
        adjoint auto;
    }
}
```

C2. "Is the bit string periodic?" oracle

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

A bit string of length N is considered periodic with period P ($1 \leq P \leq N-1$) if for all $i \in [0, N-P-1]$ $x_i = x_{i+P}$. Note that P does not have to divide N evenly; for example, bit string `01010` is periodic with period $P = 2$.

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

- an array of N ($2 \leq N \leq 7$) 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. Note that the input register x has to remain unchanged after applying the operation.

Solution. Let's start by considering a simpler problem: is the bit string periodic with period P?

This problem is similar to the problem G3 of the warmup round: in that problem you had to compare the states of the qubits in symmetrical positions, and in this one you have to compare the states of the qubits at a distance P from each other.

If you can solve this sub-task, you can express the required oracle as "the bit string periodic with period 1" OR "the bit string periodic with period 2" OR ... OR "the bit string periodic with period $N-1$". Constructing this kind of expressions has been covered in the problem G2 of the warmup round.

Conceptually the solution is not very elaborate, if somewhat bulky: you have to allocate extra qubits to store the evaluation results for each period, and you need to get those results in the first place. One had to be rather careful when implementing it, though - some straightforward approaches like allocating qubits for comparing individual qubits could quickly become too slow to run in time for the larger values of N.

To fit the execution within the time limit, you had to do qubit comparison for evaluating the period in-place, using CNOTs: you would iterate over the qubits and store XOR of the states of qubits i and $i+P$ in qubit i. After that, the bit string would be periodic if all pairs of states were equal, qubits 0 through $N-P-1$ would all be in state 0. You can extract this information into the ancilla allocated for period P and uncompute the qubit states.

This approach would pass the tests, but it would take more than half of the time limit on the largest test. It can be optimized further to speed it up by another factor of 2: you don't need to consider period 1, since any string periodic with period 1 will also be periodic with period 2 (unless $N = 2$, in which case period of 2 is not valid). You don't need to consider period 2, unless $N \leq 4$, and so on.

Listing 8. "Is the bit string periodic?" oracle

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

    operation EvaluatePeriodClauses (queryRegister : Qubit[], periodAncillas : Qubit[]) : Unit {
        body (...) {
            let N = Length(queryRegister);
            for (period in 1 .. Length(periodAncillas)) {
                // Evaluate the possibility of the string having period K.
                // For each pair of equal qubits, CNOT the last one into the first one.
                for (i in 0..N-period-1) {
                    CNOT(queryRegister[i + period], queryRegister[i]);
                }

                // If all pairs are equal, the first N-K qubits should be all in state 0.
                (ControlledOnInt(0, X))(queryRegister[0..N-period-1], periodAncillas[period-1]);

                // Uncompute
                for (i in N-period-1..-1..0) {
                    CNOT(queryRegister[i + period], queryRegister[i]);
                }
            }
        }

        adjoint auto;
    }

    operation Solve (x : Qubit[], y : Qubit) : Unit {
        body (...) {
            // Try all possible periods and see whether any of them produces the necessary string
            // The result is OR on the period clauses
            let N = Length(x);
            // Valid periods are from 1 to N-1, so N-1 ancillas
            using (anc = Qubit[N - 1]) {
                EvaluatePeriodClauses(x, anc);
                (ControlledOnInt(0, X))(anc, y);
                X(y);
                Adjoint EvaluatePeriodClauses(x, anc);
```

```
            }
        }
        adjoint self;
    }
}
```

C3. "Is the number of ones divisible by 3?" oracle

Implement a quantum oracle on N qubits which checks whether the number of bits equal to 1 in the input vector $\boldsymbol{x}$ is divisible by 3 (i.e., implements the function $f(\boldsymbol{x}) = 1$ if the number of $x_i = 1$ in $\boldsymbol{x}$ is divisible by 3, and 0 otherwise).

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

- an array of N ($1 \leq N \leq 9$) 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. Note that the input register x has to remain unchanged after applying the operation.

Solution. As usual, start by considering simpler versions of the problem. To check whether the number of bits equal to 1 is divisible by 2, you can just iterate over the input qubits and do a sequence of CNOTs with each of them as controls and the output qubit as the target. The fact that CNOT does a controlled flip of the state makes sure that this is the same as taking sum of the bits modulo 2, so in the end you just flip it again to become 1 if the sum modulo 2 was 0.

You can use the same approach here, but you need to implement addition modulo 3 as an elementary operation. To do this, you allocate two extra qubits ("sum" and "carry") and figure out the rules of updating them in a way which allows to go from $|00\rangle$ to $|10\rangle$ to $|01\rangle$ to $|00\rangle$ upon each addition. You can see one possible set of rules below. If both extra qubits end up in 0 state, the number of 1s is divisible by 3.

Listing 9. "Is the number of ones divisible by 3?" oracle

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

    operation AddBitsMod3 (queryRegister : Qubit[], ancillaRegister : Qubit[]) : Unit {
        body (...) {
            let sum = ancillaRegister[0];
            let carry = ancillaRegister[1];
            for (q in queryRegister) {
                // we need to implement addition mod 3:
                // bit sum carry | sum carry
                //  1   0    0   |  1    0
                //  1   1    0   |  0    1
                //  1   0    1   |  0    0
                // compute sum bit
                (ControlledOnBitString([true, false], X))([q, carry], sum);
                // bit sum carry | carry
                //  1   1    0   |   0
                //  1   0    0   |   1
                //  1   0    1   |   0
                (ControlledOnBitString([true, false], X))([q, sum], carry);
            }
        }

        adjoint auto;
    }
```

```
    operation Solve (x : Qubit[], y : Qubit) : Unit {
        body (...) {
            // Allocate two ancillas and implement a mini-adder on them:
            // add each qubit to one of the ancillas,
            // using the second one as a "carry".
            // If both qubits end up in 0 state, the number of 1s is divisible by 3.
            using (anc = Qubit[2]) {
                WithA(AddBitsMod3(x, _), (ControlledOnInt(0, X))(_, y), anc);
            }
        }
        adjoint auto;
    }
}
```

D1. Block diagonal matrix

Implement a unitary operation on N qubits which is represented by a square matrix of size 2^N which has 2x2 blocks of non-zero elements on the main diagonal and zero elements everywhere else.

For example, for $N = 3$ the matrix of the operation should have the following shape:

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

Here and in the rest of D problems `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. This was the easiest problem of the contest. The editorial for problem U2 of the warmup round covered creating simple matrices which can be represented as tensor product of other matrices, so this pattern should be immediately recognizable as a tensor product $I_{N-1} \otimes H$. Remember that the indices are given in little endian, so the Hadamard gate has to be applied to the first qubit of the array.

Listing 10. Block diagonal matrix

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

    operation Solve (qs : Qubit[]) : Unit {
        H(qs[0]);
    }
}
```

D2. Pattern of increasing blocks

Implement a unitary operation on N qubits which is represented by a square matrix of size 2^N defined as follows:

- top right and bottom left quarters are filled with zero elements,
- the top left quarter is the same pattern of size 2^{N-1} (for $N = 1$, the top left quarter is a non-zero element),

- the bottom right quarter is filled with non-zero elements.

For example, for $N = 3$ the matrix of the operation should have the following shape:

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

Solution. This task builds up on the problem U3 of the warmup round, which introduced constructing matrices which are composed of smaller quarters but can not be represented as a tensor product. It follows a very similar pattern.

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 qubit 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.

The top left block is the area where the most significant bit equals 0, and its effect on the qubits can be described as follows: if the last qubit of the input is in state $|1\rangle$, leave that qubit unchanged and apply the same pattern to the rest of the qubits. You can use recursion without unrolling it if you add a definition of controlled variant to your operation.

Listing 11. Pattern of increasing blocks

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

    operation Solve (qs : Qubit[]) : Unit {
        body (...) {
            let N = Length(qs);
            // for N = 1, we need an identity
            if (N > 1) {
                // do the bottom-right quarter
                ApplyToEachCA(Controlled H([Tail(qs)], _), Most(qs));
                // do the top-left quarter by calling the same operation recursively
                (ControlledOnInt(0, Solve))([Tail(qs)], Most(qs));
            }
        }
        adjoint auto;
        controlled auto;
        controlled adjoint auto;
    }
}
```

D3. X-wing fighter

Implement a unitary operation on N qubits which is represented by a square matrix of size 2^N which has non-zero elements on both main diagonal and anti-diagonal and zero elements everywhere else.

For example, for $N = 3$ the matrix of the operation should have the following shape:

Listing 12. X-wing fighter

Solution.

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

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

```
    operation Solve (qs : Qubit[]) : Unit {
        ApplyToEach(CNOT(qs[0], _), qs[1 .. Length(qs) - 1]);
        H(qs[0]);
        ApplyToEach(CNOT(qs[0], _), qs[1 .. Length(qs) - 1]);
    }
}
```

D4. TIE fighter

Implement a unitary operation on N qubits which is represented by a square matrix of size 2^N which has non-zero elements in the following positions:

- the central 2x2 sub-matrix,
- the diagonals of the top right and bottom left square sub-matrices of size $2^{N-1} - 1$ that do not overlap with the central 2x2 sub-matrix,
- the anti-diagonals of the top left and bottom right square sub-matrices of size $2^{N-1} - 1$ that do not overlap with the central 2x2 sub-matrix.

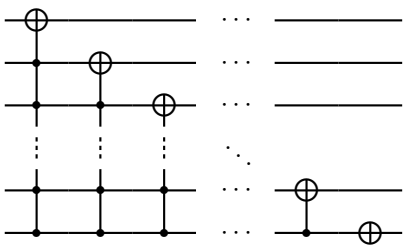
For example, for $N = 3$ the matrix of the operation should have the following shape:

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

Solution. We note first that one possible solution to this problem consists of a matrix

$$M = \frac{1}{\sqrt{2}} \begin{pmatrix} \sigma\Pi_X & \sigma \\ \Pi_X\sigma\Pi_X & -\Pi_X\sigma \end{pmatrix},$$

where $\sigma = \sum_{i=0}^{2^n-1} |i-1\rangle\langle i|$ is a decrementer by 1 (indices are computed modulo 2^n), and $\Pi_X = X^{\otimes n}$. This gives us everything we need in order to find a unitary matrix with the target pattern: We build our matrix M as a Hadamard transform ($H \otimes \mathbf{1}_2{}^{\otimes n-1}$) on the most significant qubit, followed by a right multiplication of $\mathbf{1}_2 \otimes \sigma$. Next, notice that an incrementer can for instance be implemented via the following circuit, see [5]:



Finally, note that fixing the circuit so that we get the pattern of M can be accomplished by applying controlled $X^{\otimes n-1}$ operators, which can be done by a cascade of CNOT gates. Overall, we obtain the listing shown below.

Listing 13. TIE fighter

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

    operation Decrement (qs : Qubit[]) : Unit {
        X(qs[0]);
        for (i in 1..Length(qs)-1) {
            (Controlled X)(qs[0..i-1], qs[i]);
        }
    }

    operation Reflect (qs : Qubit[]) : Unit {
        body (...) {
            ApplyToEachC(X, qs);
        }
        controlled auto;
    }

    operation Solve (qs : Qubit[]) : Unit {
        let n = Length(qs);
        X(qs[n-1]);
        (Controlled Reflect)([qs[n-1]], qs[0..(n-2)]);
        X(qs[n-1]);
        Decrement(qs[0..(n-2)]);
        H(qs[n-1]);
        (Controlled Reflect)([qs[n-1]], qs[0..(n-2)]);
    }
}
```

D5. Creeper

Implement a unitary operation on 3 qubits which is represented by a square matrix of size 8 with the following pattern:

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

Solution. To find a unitary matrix with this pattern, we first note that a matrix with the block structure

can easily be obtained from applying a Hadamard gate on the least significant qubit, followed by H_0 which is a (negatively) controlled $H \otimes H$ gate on the most significant qubit. The remaining task is to find a suitable permutation that permutes the blocks into the right positions.

Noting that a controlled NOT from the middle qubit to the most significant qubit corresponds to a matrix with pattern we almost get the right pattern by simply applying $\mathrm{CNOT}(1,0)H_0(\mathbf{1}_2 \otimes \mathbf{1}_2 \otimes H)\mathrm{CNOT}(1,0)$. The remaining issue is to re-map the entries in the middle block, which can be accomplished using a cyclic rotation of the corresponding block.

The cyclic rotation in turn is implemented like the incrementer in Problem D4 and is applied conditionally on the most significant qubit being 1. Overall, we get the following solution:

```
XXXX....
XXXX....
XXXX....
XXXX....
....XX..
....XX..
......XX
......XX
```

Listing 14. Creeper

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

    operation Solve (qs : Qubit[]) : Unit {
        CNOT(qs[1], qs[2]);
        CNOT(qs[2], qs[0]);
        CCNOT(qs[0], qs[2], qs[1]);
        X(qs[2]);
        (Controlled H)([qs[2]], qs[1]);
        X(qs[2]);
        H(qs[0]);
        CNOT(qs[1], qs[2]);
    }
}
```

D6. Hessenberg matrix

Implement a unitary operation on N qubits which is represented by an upper Hessenberg matrix (a square matrix of size 2^N which has all zero elements below the first subdiagonal and all non-zero elements on the first subdiagonal and above it).

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

```
XXXX
XXXX
.XXX
..XX
```

Solution. Our strategy will be to implement a matrix with Hessenberg structure as a product of several unitary matrices that differ from the identity only in a 2×2 subblock.

Let $s_1, s_2 \in \{0, \ldots, 2^n - 1\}$ denote a pair of indices and consider operations of the form

$$T(s_1, s_2) \quad = \quad \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & * & & & * & \leftarrow s_1 & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & * & & & * & \leftarrow s_2 & \\ & & & & & & & 1 & \\ & & & & & & & & \ddots \\ & & & & & & & & & 1 \end{pmatrix}, \quad \begin{matrix} & & s_1 & & s_2 & & \\ & & \downarrow & & \downarrow & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \end{matrix}$$

where the "$*$" stand for non-zero entries. Any such operation can be mapped to a multiply-controlled gate by first permuting the rows and columns of the matrix. Specifically, we can map s_1 to the bit string $11\ldots10$ and s_2 to the bit

string $11\ldots11$, after which the resulting operation can be implemented by a multiply controlled gate of the submatrix given by the "$*$" entries. This strategy has been described also in [6].

After implementing $T(s_1, s_2)$ in this fashion, we now utilize the fact that a unitary that has the form

$$T(0,1) \cdot T(1,2) \cdot T(2,3) \cdot \ldots \cdot T(2^n - 2, 2^n - 1) \tag{1}$$

will have Hessenberg form, provided that all "$*$" in the original 2×2 submatrix are non-zero. We therefore arrive at the solution shown below which instantiates this idea where the submatrix is given by the 2×2 Hadamard gate. Note that as usual, when traversing products such as the one in eq. (1), the order in the corresponding program has to be reversed as we are acting on *column* vectors (on which the action is left-associative).

Listing 15. Hessenberg matrix

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

    // Helper function for Embedding_Perm: finds first location where bit strings differ.
    function FirstDiff (bits1 : Bool[], bits2 : Bool[]) : Int {
        for (i in 0 .. Length(bits1)-1) {
            if (bits1[i] != bits2[i]) {
                return i;
            }
        }
        return -1;
    }

    // Helper function for Embed_2x2_Operator: performs a Clifford to implement a base change
    // that maps basis states index1 to 111...10 and index2 to 111...11
    //(in big endian notation, i.e., LSB in qs[n-1])
    operation Embedding_Perm (index1 : Int, index2 : Int, qs : Qubit[]) : Unit {
        body (...) {
            let n = Length(qs);
            let bits1 = BoolArrFromPositiveInt(index1, n);
            let bits2 = BoolArrFromPositiveInt(index2, n);
            // find the index of the first bit at which the bit strings are different
            let diff = FirstDiff(bits1, bits2);

            // we care only about 2 inputs: basis state of bits1 and bits2

            // make sure that the state corresponding to bits1 has qs[diff] set to 0
            if (bits1[diff]) {
                X(qs[diff]);
            }

            // iterate through the bit strings again, setting the final state of qubits
            for (i in 0..n-1) {
                if (bits1[i] == bits2[i]) {
                    // if two bits are the same, set both to 1 using X or nothing
                    if (not bits1[i]) {
                        X(qs[i]);
                    }
                } else {
                    // if two bits are different, set both to 1 using CNOT
                    if (i > diff) {
                        if (not bits1[i]) {
                            X(qs[diff]);
                            CNOT(qs[diff], qs[i]);
                            X(qs[diff]);
                        }
```

```
                        if (not bits2[i]) {
                            CNOT(qs[diff], qs[i]);
                        }
                    }
                }
            }

            // move the differing bit to the last qubit
            if (diff < n-1) {
                SWAP(qs[n-1], qs[diff]);
            }
        }
        adjoint auto;
    }

    // Helper function: apply the 2x2 unitary operator at the sub-matrix given by indices
    operation Embed_2x2_Operator (U : (Qubit => Unit : Controlled),
                                  index1 : Int, index2 : Int, qs : Qubit[]) : Unit {
        Embedding_Perm(index1, index2, qs);
        (Controlled U)(Most(qs), Tail(qs));
        (Adjoint Embedding_Perm)(index1, index2, qs);
    }

    operation Solve (qs : Qubit[]) : Unit {
        let n = Length(qs);
        for (i in 2^n-2..-1..0) {
            Embed_2x2_Operator(H, i, i+1, qs);
        }
    }
}
```


[1] A. Holevo. Information-theoretical aspects of quantum measurement. Problems of Information Transmission, vol. 9, no. 2, pp. 110–118 (1973)

[2] A. Peres and W. K. Wootters. Optimal detection of quantum information. Phys. Rev. Lett., vol. 66, pp. 1119-1122, Mar. 1991.

[3] A. Peres. Quantum Theory: Concepts and Methods. Kluwer Academic Publishers, 2002.

[4] C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. Proc. IEEE 56, 1107–1108 (1968).

[5] G. Alber, Th. Beth, M. Horodecki, P. Horodecki, R. Horodecki, M. Roetteler, H. Weinfurter, and A. Zeilinger. *Quantum Information: An Introduction to Basic Theoretical Concepts and Experiments*, volume 173 of *Springer Texts in Modern Physics*, Chapter *Quantum algorithms: Applicable Algebra and Quantum Physics*. Springer, 2001.

[6] A. Barenco, et al. Elementary gates for quantum computation. Physical Review A, 52(5):3457–3467, 1995.