Microsoft Q# Coding Contest - Summer 2018
July 6 - 9, 2018

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

TASKS AND SOLUTIONS

A1. Generate superposition of all basis states

You are given N qubits ($1 \leq N \leq 8$) in zero state $|0...0\rangle$. Your task is to generate an equal superposition of all 2^N basis vectors on N qubits:

$$|S\rangle = \frac{1}{\sqrt{2^N}}\big(|0...0\rangle + ... + |1...1\rangle\big).$$

You have to implement an operation which takes an array of N qubits as an input and has no output. The "output" of the operation is the state in which it leaves the qubits.

Solution. The Hadamard gate H maps $|0\rangle \mapsto |+\rangle$ and $|1\rangle \mapsto |-\rangle$, where $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. This specifies the action of H on the basis $|0\rangle, |1\rangle$, which extends to an arbitrary state $\alpha|0\rangle + \beta|1\rangle$ by linearity, i.e., it maps $\alpha|0\rangle + \beta|1\rangle \mapsto \alpha|+\rangle + \beta|-\rangle$. Note that applying unitary operations $U_0, \ldots, U_{N-1}$ to separate qubits $|q_0\rangle, \ldots, |q_{N-1}\rangle$, i.e., $|q_i\rangle \mapsto U_i|q_i\rangle$, mathematically corresponds to applying the tensor product (sometimes called Kronecker product) $U_0 \otimes \ldots \otimes U_{N-1}$ which acts on the state vector of the N qubits. To get familiar with the Hadamard gate, other primitive quantum gates, and the concept of tensor products, look up resources such as Quantum logic gate @ Wikipedia or Kronecker product @ Wikipedia, consult a gentle introduction, or jump directly into the Q# docs. Specifically, if all U_i are equal to the Hadmard gate, we obtain $H^{\otimes N}$ which works out to:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \qquad H^{\otimes 2} = H \otimes H = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}, \qquad \ldots$$

This means that the first column of $H^{\otimes N}$ is the vector $\frac{1}{\sqrt{2^N}}(1, 1, \ldots, 1)^T$ (here $(\ldots)^T$ denotes transposition), or in other words $\frac{1}{\sqrt{2^N}}(|0...0\rangle + ... + |1...1\rangle)^T$. The action of a unitary U on a state vector $|q\rangle$ mathematically corresponds to the matrix vector product $|q\rangle \mapsto U|q\rangle$, i.e., upon input $|q\rangle = |0\ldots0\rangle$ which corresponds to the column vector $(1, 0, 0, \ldots, 0)^t$ of length 2^N, the result is precisely our target $\frac{1}{2^N}(|0...0\rangle + ... + |1...1\rangle)^T$. All we need to do therefore is to apply the H gate to each of the given qubits. This can be done with a simple for loop.

Listing 1. Generate superposition of all basis states

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

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

A2. Generate superposition of zero state and a basis state

You are given N qubits ($1 \le N \le 8$) in zero state $|0...0\rangle$. You are also given a bitstring *bits* which describes a non-zero basis state on N qubits $|\psi\rangle$. Your task is to generate a state which is an equal superposition of $|0...0\rangle$ and the given basis state:

$$|S\rangle = \frac{1}{\sqrt{2}}\big(|0...0\rangle + |\psi\rangle\big)$$

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

- an array of qubits qs,
- an arrays of boolean values *bits* representing the basis state $|\psi\rangle$. This array will have the same length as the array of qubits. The first element of this array $bits[0]$ will be `true`.

The operation doesn't have an output; its "output" is the state in which it leaves the qubits. An array of boolean values represents a basis state as follows: the i-th element of the array is true if the i-th qubit is in state $|1\rangle$, and false if it is in state $|0\rangle$. For example, array `[true; false]` describes 2-qubit state $|\psi\rangle = |10\rangle$, and in this case the resulting state should be $\frac{1}{\sqrt{2}}\big(|00\rangle + |10\rangle\big) = |+\rangle \otimes |0\rangle$.

Solution. The controlled NOT gate, or `CNOT` gate for short, is a useful gate to entangle qubits. Mathematically, its action is given by $|x, y\rangle \mapsto |x, x \oplus y\rangle$, where $x, y \in \{0, 1\}$ and $\oplus$ denotes the exclusive OR (XOR) operation. In other words, if $y = 0$, then the CNOT gate creates a "copy" of the basis state $|x\rangle$ as it maps $|x, 0\rangle \mapsto |x, x\rangle$. We put copy in quotes as this only works for the basis states of the computational basis and does not create a *bona fide* copy of an arbitrary quantum state $\alpha|0\rangle + \beta|1\rangle$.

Note further that a Hadamard gate, applied to the first qubit of an array of qubits that is initially in the $|0\dots0\rangle = |0\rangle^{\otimes N}$ state, will map this state to $|+\rangle|0\rangle^{\otimes(N-1)} = \frac{1}{\sqrt{2}}(|0\,0\dots0\rangle + |1\,0\dots0\rangle)$. Due to the mentioned "copying" property, applying for instance the CNOT gate between the first qubit and the third qubit of this state would now map this state to $\frac{1}{\sqrt{2}}(|0\,0\dots0\rangle + |1\,0\,1\dots0\rangle)$. Applying another CNOT from the first qubit to the fourth qubit would create $\frac{1}{\sqrt{2}}(|0\,0\dots0\rangle + |1\,0\,1\,1\dots0\rangle)$ and so on, i.e., any desired bit pattern can be created by applying the corresponding sequence of CNOT gates.

The Q# code does this by first applying a Hadamard gate to the first qubit and then iterating over the elements of the bit vector *bits*. A classical if statement is used to apply the CNOTs as needed.

Listing 2. Generate superposition of zero state and a basis state

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

    operation Solve (qs : Qubit[], bits : Bool[]) : ()
    {
        body
        {
            // Hadamard first qubit
            H(qs[0]);

            // iterate through the bitstring and CNOT to qubits corresponding to true bits
            for (i in 1..Length(qs)-1) {
                if (bits[i]) {
                    CNOT(qs[0], qs[i]);
                }
            }
        }
    }
}
```

A3. Generate superposition of two basis states

You are given N qubits ($1 \leq N \leq 8$) in zero state $|0...0\rangle$. You are also given two bitstrings $bits0$ and $bits1$ which describe two different basis states on N qubits $|\psi_0\rangle$ and $|\psi_1\rangle$. Your task is to generate a state which is an equal superposition of the given basis states:

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

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

- an array of qubits qs,
- two arrays of Boolean values $bits0$ and $bits1$ representing the basis states $|\psi_0\rangle$ and $|\psi_1\rangle$. These arrays will have the same length as the array of qubits. $bits0$ and $bits1$ will differ in at least one position.

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

Solution. In the previous task A2 we solved the special case of this task, in which one of the bit vectors was given by the all-zero bit-vector $bit0 = [0, \ldots, 0]$, and the first element of the the other bit vector was equal to 1, which allowed us to anchor our CNOT sequence by applying a Hadamard gate to the first qubit. In this task, we can leverage the same idea, however we will first have to find the first position at which the two inputs $bit0$ and $bit1$ differ. This is accomplished using a classical function. Read more about using Q# functions to process classical data and about functions vs operations.

Once the function `FindFirstDiff` identified the first index Δ in which $bits0$ and $bits1$ differ, we apply a Hadamard gate to that location. Now, for each position $i \neq \Delta$ there are 4 options:

- $bits0[i] = bits1[i] = 0$: do nothing.
- $bits0[i] = bits1[i] = 1$: flip the state of the qubit i.
- $bits0[i] = bits0[\Delta], bits1[i] = bits1[\Delta]$: apply CNOT with $qs[\Delta]$ as control and $qs[i]$ as target.
- $bits0[i] \neq bits0[\Delta], bits1[i] \neq bits1[\Delta]$: apply CNOT with $qs[\Delta]$ as control and $qs[i]$ as target, and then flip the state of the qubit i.

Flipping bits can be done using the Pauli X gate.

Listing 3. Generate superposition of two basis states

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

    function FindFirstDiff (bits0 : Bool[], bits1 : Bool[]) : Int
    {
        mutable firstDiff = -1;
        for (i in 0 .. Length(bits1)-1) {
            if (bits1[i] != bits0[i] && firstDiff == -1) {
                set firstDiff = i;
            }
        }
        return firstDiff;
    }

    operation Solve (qs : Qubit[], bits0 : Bool[], bits1 : Bool[]) : ()
    {
        body
        {
            // find the index of the first bit at which the bitstrings are different
            let firstDiff = FindFirstDiff(bits0, bits1);
            // Hadamard corresponding qubit to create superposition
            H(qs[firstDiff]);

            // iterate through the bitstrings again setting the final state of qubits
            for (i in 0 .. Length(qs)-1) {
                if (bits0[i] == bits1[i]) {
                    // if two bits are the same apply X or nothing
                    if (bits0[i]) {
                        X(qs[i]);
                    }
                } else {
                    // if two bits are different, set their difference using CNOT
                    if (i > firstDiff) {
                        CNOT(qs[firstDiff], qs[i]);
                        if (bits0[i] != bits0[firstDiff]) {
                            X(qs[i]);
                        }
                    }
                }
            }
        }
    }
}
```

A4. Generate W state

You are given $N = 2^k$ qubits ($0 \leq k \leq 4$) in zero state $|0...0\rangle$. Your task is to create a generalized W state on them. Generalized W state is an equal superposition of all basis states on N qubits that have Hamming weight equal to 1:

$$|W_N\rangle = \frac{1}{\sqrt{N}}\big(|100...0\rangle + |010...0\rangle + ... + |00...01\rangle\big)$$

For example, for $N = 1$, $|W_1\rangle = |1\rangle$. You have to implement an operation which takes an array of N qubits as an input and has no output. The "output" of the operation is the state in which it leaves the qubits.

Solution. We proceed by induction on k. For $k = 0$, we just need to apply a Pauli X gate to the input to create $|W_1\rangle = |1\rangle$. For general k, we first create the state $|W_{N/2}\rangle$ on the first $2^{k-1} = N/2$ qubits. This is done by recursively calling the preparation operation. Now the first $N/2$ qubits are already in a superposition where only basis states of Hamming weight 1 occur. Next, we allocate an auxiliary qubit which will serve as a flag whether we are in the block of the first $N/2$ qubits or the block of the last $N/2$ qubits. If the flag is "0", there is nothing to be done. If the flag is "1", we move the state of the first $N/2$ qubits to the state of the last $N/2$ qubits and vice versa. This can be accomplished using a controlled SWAP operation.

This almost creates the state $|W_k\rangle$ on N qubits, however, one issue remains: the state is now entangled with the state of the flag qubit, i.e., we have to reset the state of the flag qubit. If we were to simply measure the flag qubit in the Pauli Z basis, the overall state vector would collapse to a $|W_{N/2}\rangle$ state on one half of the qubits and $|0\rangle^{\otimes N/2}$ on the other half of the qubits, the order of states depending on whether the measured result was "Zero" or "One". Instead, we have to "uncompute" the state of the flag qubit so that it will no longer be entangled with the rest of the qubits. This can be accomplished by checking whether the Hamming weight of the second $N/2$ qubits is equal to 1, which can be done by XORing each of the second $N/2$ qubits into the flag qubit.

In the Q# code, the auxiliary qubit `here` is allocated in $|0\rangle$ state with a using statement. The controlled SWAP operation can be implemented using the controlled functor applied to the SWAP operation which itself is a primitive gate.

Listing 4. Generate W state

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

    operation Solve (qs : Qubit[]) : ()
    {
        body
        {
            let N = Length(qs);
            if (N == 1) {
                // base of recursion: |1>
                X(qs[0]);
            } else {
                let K = N / 2;
                // create W state on the first K qubits
                Solve(qs[0..K-1]);

                // the next K qubits are in |0...0> state
                // allocate ancilla in |+> state
                using (anc = Qubit[1]) {
                    let here = anc[0];
                    H(here);
                    for (i in 0..K-1) {
                        (Controlled SWAP)([here], (qs[i], qs[i+K]));
                    }
                    // unentangle here from the rest of the qubits
                    for (i in K..N-1) {
                        CNOT(qs[i], here);
                    }
                }
            }
        }
        adjoint auto;
    }
}
```

B1. Distinguish zero state and W state

You are given N qubits ($2 \leq N \leq 8$) which are guaranteed to be in one of the two states:

- $|0...0\rangle$ state, or
- $|W\rangle = \frac{1}{\sqrt{N}}\big(|100...0\rangle + |010...0\rangle + ... + |00...01\rangle\big)$ state.

Your task is to perform necessary operations and measurements to figure out which state it was and to return 0 if it was $|0..0\rangle$ state or 1 if it was the W state. The state of the qubits after the operations does not matter. You have to implement an operation which takes an array of N qubits as an input and returns an integer.

Solution. Note that the Hamming weight of each basis state in the W state is equal to 1, whereas the Hamming weight of the $|0\dots0\rangle$ state is equal to 0. To distinguish the two cases, it is therefore sufficient to measure the given state vector in the computational basis (aka measure each of the qubits in Pauli Z basis), and to count the number of "One"s that have been measured. For a background on measurements, please see this article in Q# documentation. In Q#, the measurement in the Pauli Z basis can be done using the M gate. Computing the Hamming weight can be done using a mutable counter variable.

Listing 5. Distinguish zero state and W state

```
namespace Solution {
    open Microsoft.Quantum.Primitive;
    open Microsoft.Quantum.Canon;
    operation Solve (qs : Qubit[]) : Int
    {
        body
        {
            // measure all qubits
            mutable countOnes = 0;
            for (i in 0..Length(qs)-1) {
                if (M(qs[i]) == One) {
                    set countOnes = countOnes + 1;
                }
            }
            // if there is exactly one One, it's W state, if there are no Ones, it's |0...0>
            if (countOnes == 0) {
                return 0;
            }
            return 1;
        }
    }
}
```

You can also use a shortcut: since you know that you'll get either no "One"s or exactly one "One", you can infer that the state was W as soon as you measure a "One", there's no need to measure the rest of the qubits in this case.

Listing 6. Distinguish zero state and W state

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

    operation Solve (qs : Qubit[]) : Int
    {
        body
        {
            // measure all qubits
            // if there is at least one One, it's W state, if there are no Ones, it's |0...0>
            // (and you can return as soon as get the first One)
            for (i in 0..Length(qs)-1) {
                if (M(qs[i]) == One) {
                    return 1;
                }
            }
            return 0;
        }
    }
}
```

B2. Distinguish GHZ state and W state

You are given N qubits ($2 \leq N \leq 8$) which are guaranteed to be in one of the two states:

- $|GHZ\rangle = \frac{1}{\sqrt{2}}\big(|0...0\rangle + |1...1\rangle\big)$ state, or
- $|W\rangle = \frac{1}{\sqrt{N}}\big(|100...0\rangle + |010...0\rangle + ... + |00...01\rangle\big)$ state.

Your task is to perform necessary operations and measurements to figure out which state it was and to return 0 if it was GHZ state or 1 if it was W state. The state of the qubits after the operations does not matter. You have to implement an operation which takes an array of N qubits as an input and returns an integer.

Solution. This is almost identical to the task B1, with the only difference being that the GHZ state is supported on basis states that are either the all zero or the all one vector, i.e., its Hamming weight upon measurement in the computational basis is either equal to 0 or to N. Note that as $N \neq 1$, this perfectly distinguishes this case from the case of the W states in which the Hamming weight is equal to 1. The only difference in the Q# solution is to catch the case in which the Hamming weight was N, which again can be implemented using a counter.

Listing 7. Distinguish GHZ state and W state

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

    operation Solve (qs : Qubit[]) : Int
    {
        body
        {
            // measure all qubits; if there is exactly one One, it's W state,
            // if there are no Ones or all are Ones, it's GHZ
            mutable countOnes = 0;
            for (i in 0..Length(qs)-1) {
                if (M(qs[i]) == One) {
                    set countOnes = countOnes + 1;
                }
            }
            if (countOnes == 1) {
                return 1;
            }
            return 0;
        }
    }
}
```

B3. Distinguish four 2-qubit states

You are given 2 qubits which are guaranteed to be in one of the four orthogonal states:

- $|S_0\rangle = \frac{1}{2}\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big)$
- $|S_1\rangle = \frac{1}{2}\big(|00\rangle - |01\rangle + |10\rangle - |11\rangle\big)$
- $|S_2\rangle = \frac{1}{2}\big(|00\rangle + |01\rangle - |10\rangle - |11\rangle\big)$
- $|S_3\rangle = \frac{1}{2}\big(|00\rangle - |01\rangle - |10\rangle + |11\rangle\big)$

Your task is to perform necessary operations and measurements to figure out which state it was and to return the index of that state (0 for $|S_0\rangle$, 1 for $|S_1\rangle$ etc.). The state of the qubits after the operations does not matter. You have to implement an operation which takes an array of 2 qubits as an input and returns an integer.

Solution. Observe that the four potential input states, when arranged into the columns of a matrix, correspond precisely to the Hadamard matrix $H^{\otimes 2}$ defined in the solution of A1. To solve the task, we therefore apply the transformation that takes the given basis back to the computational basis, which can be accomplished by applying the inverse operation to $H^{\otimes 2}$, followed by a measurement in the Pauli Z basis. Note that $(H^{\otimes 2})^{-1} = (H^{\otimes 2})^{\dagger} = H^{\otimes 2}$ as the Hadamard gate is self-inverse.

In the Q# solution, we use a function `ResultAsInt` to convert bit-vectors to integers which is part of the "canon", a useful set of basic libraries.

Listing 8. Distinguish four 2-qubit states

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

    operation Solve (qs : Qubit[]) : Int
    {
        body
        {
            // These states are produced by H x H, applied to four basis states.
            // To measure them, apply H x H followed by basis state measurement.
            H(qs[0]);
            H(qs[1]);
            return ResultAsInt([M(qs[1]); M(qs[0])]);
        }
    }
}
```

B4. Distinguish four 2-qubit states - 2

You are given 2 qubits which are guaranteed to be in one of the four orthogonal states:

- $|S_0\rangle = \frac{1}{2}\big(\ \ |00\rangle - |01\rangle - |10\rangle - |11\rangle\big)$
- $|S_1\rangle = \frac{1}{2}\big(-|00\rangle + |01\rangle - |10\rangle - |11\rangle\big)$
- $|S_2\rangle = \frac{1}{2}\big(-|00\rangle - |01\rangle + |10\rangle - |11\rangle\big)$
- $|S_3\rangle = \frac{1}{2}\big(-|00\rangle - |01\rangle - |10\rangle + |11\rangle\big)$

Your task is to perform necessary operations and measurements to figure out which state it was and to return the index of that state (0 for $|S_0\rangle$, 1 for $|S_1\rangle$ etc.). The state of the qubits after the operations does not matter. You have to implement an operation which takes an array of 2 qubits as an input and returns an integer.

Solution. We first note that the states $|S_0\rangle, \dots, |S_3\rangle$ are mutually orthogonal. Therefore, in principle, the four cases can be perfectly distinguished by applying the operation $A^\dagger$, where A is the matrix obtained by arranging the vectors column-wise: $A = |S_0\rangle, \dots, |S_3\rangle$.

In general, applying a unitary operation requires to first decompose it into a sequence of primitive gates (or library operations), which depending on the operation can be difficult. In this case, we can solve the problem by reducing A to the Hadamard matrix $H^{\otimes 2}$: we note that A is equal to $H^{\otimes 2}$ up to permutations matrices and diagonal +1/-1 matrices that are applied to the left and the right of A. Specifically,

$$A = \mathrm{diag}(-1,1,1,1)\ (H \otimes H)\ \mathrm{diag}(-1,1,1,1)\ \pi, \qquad (1)$$

where π is the permutation $(1,2)$ (on basis states $|0\rangle, |1\rangle, |2\rangle, |3\rangle$) which corresponds to a swap of two qubits.

In Q#, we can implement the swap operation by a simple `SWAP` gate. The digaonal gate $\mathrm{diag}(-1,1,1,1)$ can be obtained from a controlled-Z gate diag(1, 1, 1, -1) by conjugating with Pauli X operations on both qubits. This is done by the operation `ApplyDiag`. Finally, note that we use partial application in the `ApplyToEach(H, _)` operation to apply the same operation (here H) to several qubits. Also, we use the `With(U,V)` combinator which implements a conjugation $U^\dagger V U$ of two operations.

Finally, a note mathematical conventions which can lead to confusion when translating mathematical descriptions of operators in to programmatic sequences of gates: in general, when implementing factorizations such as eq. (1), the order of the binding reverses: as the factors π, $\mathrm{diag}(-1,1,1,1)$, etc. in eq. (1) act by left-multiplications on (column) vectors and are therefore read from right to left, in Q# the instructions are applied in the reversed order, i.e., first π is applied, then $\mathrm{diag}(-1,1,1,1)$, etc.

Listing 9. Distinguish four 2-qubit states - 2

```
namespace Solution {
    open Microsoft.Quantum.Primitive;
    open Microsoft.Quantum.Canon;
    // Helper function to implement diag(-1, 1, 1, 1)
    operation ApplyDiag (qs : Qubit[]) : ()
    {
        body
        {
            ApplyToEach(X, qs);
            (Controlled Z)([qs[0]], qs[1]);
            ApplyToEach(X, qs);
        }
        adjoint self
    }

    operation Solve (qs : Qubit[]) : Int
    {
        body
        {
            SWAP(qs[0], qs[1]); // pi
            With(ApplyDiag, ApplyToEach(H, _), qs); // diag(..) (H \otimes H) diag(..)
            return ResultAsInt([M(qs[1]); M(qs[0])]);
        }
    }
}
```

C1. Distinguish zero state and plus state with minimum error

You are given a qubit which is guaranteed to be either in $|0\rangle$ state or in $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ state.

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

Note that these states are not orthogonal, and thus can not be distinguished perfectly. In each test your solution will be called 1000 times, and your goal is to get a correct answer at least 80% of the times. In each test $|0\rangle$ and $|+\rangle$ states will be provided with 50% probability.

You have to implement an operation which takes a qubit as an input and returns an integer.

Solution. The states $|0\rangle$ and $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ have an inner product of $\langle 0|+\rangle = \frac{1}{\sqrt{2}}$ which corresponds to an angle of $\pi/4$ (or 45° in degrees). This is shown in Figure 1 on the left.

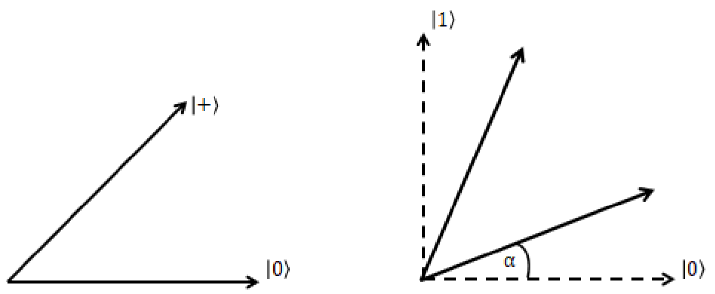

Figure 1. Shown on the left are the non-orthogonal states $|0\rangle$ and $|+\rangle$ that are at an angle $\pi/4$. The optimal single-shot measurement is shown as the dashed vectors on the right. It is obtained by rotating the given state around the y-axis by $R_y(\alpha)$, where $\alpha = \pi/8$, followed by a measurement in the computational basis.

The task is to devise a measurement that upon execution maximizes the probability to answer the correct state. This is shown in Figure 1 on the right. Let $\{E_a, E_b\}$ be a measurement with two outcomes a and b, which we identify with the answers, i.e., "$a =$ state was $|0\rangle$" and "$b =$ state was $|+\rangle$". Then we define

$$P(a|0) = \text{ probability to observe first outcome given that the state was } |0\rangle\,,$$

$$P(b|0) = \text{ probability to observe second outcome given that the state was } |0\rangle\,,$$

$$P(a|+) = \text{ probability to observe first outcome given that the state was } |+\rangle\,,$$

$$P(b|+) = \text{ probability to observe second outcome given that the state was } |+\rangle\,.$$

The task is to maximize the probability to be correct on a single shot experiment (which is the same as to minimize the probability to be wrong on a single shot). Assuming uniform prior specified in the statement, i.e., $P(+) = P(0) = 1/2$, we get

$$P_{correct} = P(0)P(a|0) + P(+)P(b|+).$$

Assuming a von Neumann measurement of the form $E_a = R_y(2\alpha)(1,0)^T = (\cos(\alpha), \sin(\alpha))^T$ and $E_b = R_y(2\alpha)(0,1)^T = (\sin(\alpha), -\cos(\alpha))^T$, we get that $P_{correct} = 1/2 + \cos^2(\alpha) + \cos(\alpha)\sin(\alpha)$. Maximizing this for α, we get max $P_{success} = 1/2(1 + 1/\sqrt(2)) = 0.8535..$, which is attained for $\alpha = \pi/8$. The definition of the R_y gate shows that $R_y(\theta) = e^{-i\theta\sigma_y/2}$ which corresponds to the rotation matrix

$$\begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix}.$$

This means that in the Q# code, rotating the input state by $\pi/8$ means to apply R_y with angle $2\pi/8 = \pi/4$.

Listing 10. Distinguish zero state and plus state with minimum error

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

    operation Solve (q : Qubit) : Int
    {
        body
        {
            // Rotate the input state by Pi/8 means to apply Ry with angle 2*Pi/8.
            Ry(0.25*PI(), q);
            if (M(q) == Zero) {
                return 0;
            }
            return 1;
        }
    }
}
```

C2. Distinguish zero state and plus state without errors

You are given a qubit which is guaranteed to be either in $|0\rangle$ state or in $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ state.

Your task is to perform necessary operations and measurements to figure out which state it was and to return 0 if it was a $|0\rangle$ state, 1 if it was $|+\rangle$ state or -1 if you can not decide, i.e., an "inconclusive" result. The state of the qubit after the operations does not matter.

Note that these states are not orthogonal, and thus can not be distinguished perfectly. In each test your solution will be called 10000 times, and your goals are:

- never give 0 or 1 answer incorrectly (i.e., never return 0 if input state was $|+\rangle$ and never return 1 if input state was $|0\rangle$),
- give -1 answer at most 80% of the times,
- correctly identify $|0\rangle$ state at least 10% of the times,
- correctly identify $|+\rangle$ state at least 10% of the times.

In each test $|0\rangle$ and $|+\rangle$ states will be provided with 50% probability.
You have to implement an operation which takes a qubit as an input and returns an integer.

Solution. A simple strategy that gives an inconclusive result with probability 0.75 and never errs in case it yields a conclusive result can be obtained from randomizing the choice of measurement basis between the computational basis (std) and the Hadamard basis (had). Observe that when measured in the standard basis, the state $|0\rangle$ will always lead to the outcome "0", whereas the state $|+\rangle$ will lead to outcomes "0" and "1" with probability 1/2. This means that upon measuring "1" we can with certainty conclude that the state was $|+\rangle$. We respond inconclusive "-1" if we measured "0". A similar argument applies to the scenario where we measure in the Hadamard basis, where $|0\rangle$ can lead to both outcomes, whereas $|+\rangle$ always leads to "0". Then upon measuring "1" we can with certainty conclude that the state was $|0\rangle$. In this case again, we respond inconclusive "-1" if we measured "0". This leads to the following scenarios (shown are the conditional probabilities of the above scenarios and resulting answers):

state	basis	P(output "0" | state, basis)	P(output "1" | state, basis)	P(output "−1" | state, basis)
$\|0\rangle$	std	0	0	1
$\|+\rangle$	std	0	1/2	1/2
$\|0\rangle$	had	1/2	0	1/2
$\|+\rangle$	had	0	0	1

As the priors for choosing the state and the basis are both uniform, the probability for an inconclusive result is given by $\mathrm{P}(\text{output "}-1\text{"} \mid \text{state, basis}) \cdot \mathrm{P}(\text{state, basis}) = 1 \cdot 1/4 + 1/2 \cdot 1/4 + 1/2 \cdot 1/4 + 1 \cdot 1/4 = 3/4$, and the probabilities for outputting the other two cases are 1/8 each. This means we can implement the simple strategy in Q# by first performing a coin toss to determine whether we are measuring in std/had and then dispatching the various cases using if/else statements.

It should be noted that the presented strategy is not the best possible. Indeed, there is a *quantum* strategy which yields an inconclusive result with probability $1/\sqrt{(}2) = 0.7071...$ which is better than the above strategy.

Listing 11. Distinguish zero state and plus state without errors

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

    operation Solve (q : Qubit) : Int
    {
        body
        {
            mutable output = 0;
            let basis = RandomInt(2);
            // randomize over std and had

            if (basis == 0) {
                // use standard basis
                let result = M(q);
                if (result == One) {
                    // this can only arise if the state was |+>
                    set output = 1;
                }
                else {
```

```
                set output = -1;
            }
        }
        else {
            // use Hadamard basis
            H(q);
            let result = M(q);
            if (result == One) {
                // this can only arise if the state was |0>
                set output = 0;
            }
            else {
                set output = -1;
            }
        }
        return output;
    }
  }
}
```

D1. Oracle for $f(x) = b \cdot x \mod 2$

Implement a quantum oracle on N qubits which implements the following function: $f(\boldsymbol{x}) = \boldsymbol{b} \cdot \boldsymbol{x} \mod 2 = \sum_{k=0}^{N-1} b_k x_k \mod 2$, where $\boldsymbol{b} \in \{0,1\}^N$ (a vector of N integers, each of which can be 0 or 1). For an explanation on how this type of quantum oracles works, see Introduction to quantum oracles. You have to implement an operation which takes the following inputs:

- an array of N qubits x in arbitrary state (input register), $1 \leq N \leq 8$,
- a qubit y in arbitrary state (output qubit),
- an array of N integers b, representing the vector $\boldsymbol{b}$. Each element of b will be 0 or 1.

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

Solution. Recall that the CNOT gate between qubit x_i of the x-register as control and qubit y as target corresponds to the operation that maps $|x_i\rangle|y\rangle \mapsto |x_i\rangle|y \oplus x_i\rangle$ for any value of x_i that corresponds to a computational basis vector. Therefore if we apply a CNOT gate between x_i and y if and only if $b_i = 1$, we'll get a transformation $|x_i\rangle|y\rangle \mapsto |x_i\rangle|y \oplus b_i x_i\rangle$. Applying this logic for all indices $i = 0, ..., N-1$ gives the desired result.

Listing 12. Oracle for $f(x) = b \cdot x \mod 2$

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

    operation Solve (x : Qubit[], y : Qubit, b : Int[]) : ()
    {
        body
        {
            for (i in 0..Length(x)-1) {
                if (b[i] == 1) {
                    CNOT(x[i], y);
                }
            }
        }
    }
}
```

D2. Oracle for $f(x) = b \cdot x + (1-b) \cdot (1-x) \mod 2$

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

$$f(\boldsymbol{x}) = (\boldsymbol{b} \cdot \boldsymbol{x} + (\boldsymbol{1} - \boldsymbol{b}) \cdot (\boldsymbol{1} - \boldsymbol{x})) \mod 2 = \; = \sum_{k=0}^{N-1} (b_k x_k + (1 - b_k) \cdot (1 - x_k)) \mod 2$$

Here $\boldsymbol{b} \in \{0,1\}^N$ (a vector of N integers, each of which can be 0 or 1), and $\boldsymbol{1}$ is a vector of N 1s. For an explanation on how this type of quantum oracles works, see Introduction to quantum oracles. You have to implement an operation which takes the following inputs:

- an array of N qubits x in arbitrary state (input register), $1 \leq N \leq 8$,
- a qubit y in arbitrary state (output qubit),
- an array of N integers b, representing the vector $\boldsymbol{b}$. Each element of b will be 0 or 1.

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

Solution. Note that the binary complement $\overline{x}$ of a Boolean variable x is obtained by $\overline{x} = 1 - x$ if we cast the Boolean variable as "false"$= 0$ and "true"$= 1$. We therefore can compute the function $f(x) = b \cdot x + (1-b) \cdot (1-x) \mod 2$ by first computing the function $b \cdot x$ as in task D1 for the values of b that are equal to "1", which is then followed by computing the binary complement of variable x_i and copying this into y for the values of b that are equal to "0". The binary complement of a variable is computed using Pauli X which implements the bit-flip; note that we have to "uncompute" the bit-flip operation as the oracle is required to implement $|x\rangle |y\rangle \mapsto |x\rangle |y \oplus f(x)\rangle$, i.e., it needs to leave the input state $|x\rangle$ unchanged.

Listing 13. Oracle for $f(x) = b \cdot x + (1-b) \cdot (1-x) \mod 2$

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

    operation Solve (x : Qubit[], y : Qubit, b : Int[]) : ()
    {
        body
        {
            for (i in 0..Length(x)-1) {
                if (b[i] == 1) {
                    CNOT(x[i], y);
                } else {
                    // do a 0-controlled NOT
                    X(x[i]);
                    CNOT(x[i], y);
                    X(x[i]);
                }
            }
        }
    }
}
```

D3. Oracle for majority function

Implement a quantum oracle on 3 qubits which implements a majority function. Majority function on 3-bit vectors is defined as follows: $f(\boldsymbol{x}) = 1$ if vector $\boldsymbol{x}$ has two or three 1s, and 0 if it has zero or one 1s. For an explanation on how this type of quantum oracles works, see Introduction to quantum oracles. You have to implement an operation which takes the following inputs:

- an array of 3 qubits x in arbitrary state (input register),

- a qubit y in arbitrary state (output qubit).

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

Solution. Boolean function $f(x_0, x_1, x_2) = MAJ(x_0, x_1, x_2)$ can be written as $f(x_0, x_1, x_2) = (x_0 \wedge x_1) \oplus (x_0 \wedge x_2) \oplus (x_1 \wedge x_2)$: if two of the inputs equal 1, exactly one term equals 1, and if all three inputs equal 1, all terms equal 1 and their XOR equals 1 as well.

The AND function can be implemented using a so-called Toffoli gate (aka CCNOT gate) which maps $|x, y, z\rangle \mapsto |x, y, z \oplus xy\rangle$. Therefore, we can implement the function as shown in the Q# listing below. Note that there are better ways for implementing the majority function that require only 1 Toffoli gate and several CNOT gates.

Listing 14. Oracle for majority function

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

    operation Solve (x : Qubit[], y : Qubit) : ()
    {
        body
        {
            CCNOT(x[0], x[1], y);
            CCNOT(x[0], x[2], y);
            CCNOT(x[1], x[2], y);
        }
    }
}
```

E1. Bernstein-Vazirani algorithm

You are given a quantum oracle - an operation on $N+1$ qubits which implements a function $f : \{0,1\}^N \to \{0,1\}$. You are guaranteed that the function f implemented by the oracle is scalar product function (oracle from problem D1):

$$f(\boldsymbol{x}) = \boldsymbol{b} \cdot \boldsymbol{x} \mod 2 = \sum_{k=0}^{N-1} b_k x_k \mod 2$$

Here $\boldsymbol{b} \in \{0,1\}^N$ (an array of N integers, each of which can be 0 or 1). Your task is to reconstruct the array $\boldsymbol{b}$. Your code is allowed to call the given oracle only once. You have to implement an operation which takes the following inputs:

- an integer N - the number of qubits in the oracle input ($1 \leq N \leq 8$),
- an oracle Uf, implemented as an operation with signature ((Qubit[], Qubit) => ()), i.e., an operation which takes as input an array of qubits and an output qubit and has no output.

The return of your operation is an array of integers of length N, each of them 0 or 1.

Solution. The Bernstein-Vazirani is a well-known quantum algorithm, first described in this paper. For a Q# implementation, see the example ParityViaFourierSampling at Microsoft/Quantum GitHub repository.

The algorithm is based on the following sequence of steps: (1) first, create a register of N qubits in the uniform state. This can be accomplished by applying $H^{\otimes N} |0\rangle^{\otimes N}$. Next, (2) create a qubit in state $|-\rangle$. This can be accomplished by applying H to a qubit in state $|-\rangle$. Next, (3) compute the oracle corresponding to the input function Uf into the N qubit register x and the target register which holds the $|-\rangle$ state. This is sometimes called "phase-kickback" in the quantum computing literature. The overall effect of this is to compute the map $|x\rangle |-\rangle \mapsto (-1)^{Uf(x)} |x\rangle |-\rangle$, i.e., the oracle Uf is computed into the phase. The qubit $|-\rangle$ factors off and can henceforth be ignored. Next, observe that the column vectors $\sum_x (-1)^{Uf(x)} |x\rangle$ corresponding to the various functions $f(x) = b \cdot x$ for all N-bit vectors b are mutually orthogonal and correspond precisely to the columns of the Hadamard transform $H^{\otimes N}$. Hence, by (4) applying the inverse to that operation (which again is equal to $H^{\otimes N}$), the state is mapped to the basis vector $|b_0, \ldots, b_{N-1}\rangle$. Upon measuring this vector in the standard basis, we obtain the desired result.

Listing 15. Bernstein-Vazirani algorithm

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

    operation Solve (N : Int, Uf : ((Qubit[], Qubit) => ())) : Int[]
    {
        body
        {
            mutable r = new Int[N];

            // allocate N+1 qubits
            using (qs = Qubit[N+1]) {
                // split allocated qubits into input register and answer register
                let x = qs[0..N-1];
                let y = qs[N];

                // prepare qubits in the right state
                ApplyToEach(H, x);
                X(y);
                H(y);

                // apply oracle
                Uf(x, y);

                // apply Hadamard to each qubit of the input register
                ApplyToEach(H, x);

                // measure all qubits of the input register;
                // the result of each measurement is converted to a Bool
                for (i in 0..N-1) {
                    if (M(x[i]) != Zero) {
                        set r[i] = 1;
                    }
                }

                // before releasing the qubits make sure they are all in |0> state
                ResetAll(qs);
            }
            return r;
        }
    }
}
```

E2. Another array reconstruction algorithm

You are given a quantum oracle - an operation on $N + 1$ qubits which implements a function $f : \{0, 1\}^N \to \{0, 1\}$. You are guaranteed that the function f implemented by the oracle can be represented in the following form (oracle from problem D2):

$$f(\boldsymbol{x}) = (\boldsymbol{b} \cdot \boldsymbol{x} + (\boldsymbol{1} - \boldsymbol{b}) \cdot (\boldsymbol{1} - \boldsymbol{x})) \mod 2 = = \sum_{k=0}^{N-1} (b_k x_k + (1 - b_k) \cdot (1 - x_k)) \mod 2$$

Here $\boldsymbol{b} \in \{0, 1\}^N$ (a vector of N integers, each of which can be 0 or 1), and $\boldsymbol{1}$ is a vector of N 1s. Your task is to reconstruct the array $\boldsymbol{b}$ which could produce the given oracle. Your code is allowed to call the given oracle only once. You have to implement an operation which takes the following inputs:

- an integer N - the number of qubits in the oracle input ($1 \le N \le 8$),
- an oracle Uf, implemented as an operation with signature ((Qubit[], Qubit) => ()), i.e., an operation which takes as input an array of qubits and an output qubit and has no output.

The return of your operation is an array of integers of length N, each of them 0 or 1. Note that in this problem we're comparing the oracle generated by your return to the oracle Uf, instead of comparing your return to the (hidden) value of $\boldsymbol{b}$ used to generate Uf. This means that any kind of incorrect return results in "Runtime Error" verdict, as well as actual runtime errors like releasing qubits in non-zero state.

Solution. Note that the function $f(x)$ defined in the task can be simplified as $f(x) = b \cdot x + (1-b) \cdot (1-x) \mod 2 = \sum_i b_i \oplus \sum_i x_i$ if the number of bits N is even and to $\sum_i b_i \oplus \sum_i x_i \oplus 1$ if the number of bits N is odd. Further note that half of all 2^N bit-strings are valid answers for r (the ones that have the same parity as r). To determine which half contains the vector r, we only have to apply the oracle to an input of all zeroes, and measure the qubit y to figure out the parity of vector r.

Listing 16. Another array reconstruction algorithm

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

    operation Solve (N : Int, Uf : ((Qubit[], Qubit) => ())) : Int[]
    {
        body
        {
            mutable r = new Int[N];

            // allocate N+1 qubits
            using (qs = Qubit[N+1]) {
                // split allocated qubits into input register and answer register
                let x = qs[0..N-1];
                let y = qs[N];

                // apply oracle to qubits in all 0 state
                Uf(x, y);

                // remove the N from the expression
                if (N % 2 == 1) {
                    X(y);
                }

                // now y = sum of r

                // measure the output register
                let m = M(y);
                if (m == One) {
                    // adjust parity of bit vector r
                    set r[0] = 1;
                }

                // before releasing the qubits make sure they are all in |0> state
                ResetAll(qs);
            }
            return r;
        }
    }
}
```