# Microsoft Q# Coding Contest - Summer 2020

## A1. Figure out direction of CNOT

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given an operation that implements a two-qubit unitary transformation: either the CNOT gate with the first qubit as control and the second qubit as target ($CNOT_{12}$), or the CNOT gate with the second qubit as control and the first qubit as target ($CNOT_{21}$). The operation will have Adjoint and Controlled variants defined.

Your task is to perform necessary operations and measurements to figure out which unitary it was and to return 0 if it was the $CNOT_{12}$ gate or 1 if it was the $CNOT_{21}$ gate.

You are allowed to apply the given operation and its adjoint/controlled variants **exactly once**.

You have to implement an operation which takes a two-qubit operation `unitary` as an input and returns an integer. The operation `unitary` will accept an array of qubits as input, but it will fail if the array is empty or has one or more than two qubits. Your code should have the following signature:

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

    operation Solve (unitary : (Qubit[] => Unit is Adj+Ctl)) : Int {
        // your code here
    }
}
```

## A2. Distinguish I, CNOTs and SWAP

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given an operation that implements a two-qubit unitary transformation: either the identity (I ⊗ I gate), the CNOT gate (either with the first qubit as control and the second qubit as target or vice versa) or the SWAP gate. The operation will have Adjoint and Controlled variants defined.

Your task is to perform necessary operations and measurements to figure out which unitary it was and to return

- 0 if it was the I ⊗ I gate,
- 1 if it was the $CNOT_{12}$ gate,
- 2 if it was the $CNOT_{21}$ gate,
- 3 if it was the SWAP gate.

You are allowed to apply the given operation and its adjoint/controlled variants **at most twice**.

You have to implement an operation which takes a two-qubit operation `unitary` as an input and returns an integer. The operation `unitary` will accept an array of qubits as input, but it will fail if the array is empty or has one or more than two qubits. Your code should have the following signature:

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

    operation Solve (unitary : (Qubit[] => Unit is Adj+Ctl)) : Int {
        // your code here
    }
}
```

## A3. Distinguish H from X

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given an operation that implements a single-qubit unitary transformation: either the H gate or the X gate. The operation will have Adjoint and Controlled variants defined.

Your task is to perform necessary operations and measurements to figure out which unitary it was and to return 0 if it was the H gate or 1 if it was the X gate.

You are allowed to apply the given operation and its adjoint/controlled variants **at most twice**.

You have to implement an operation which takes a single-qubit operation as an input and returns an integer. Your code should have the following signature:

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

    operation Solve (unitary : (Qubit => Unit is Adj+Ctl)) : Int {
        // your code here
    }
}
```

## A4. Distinguish Rz from R1

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given an operation that implements a single-qubit unitary transformation: either the Rz gate or the R1 gate. The operation will have Adjoint and Controlled variants defined.

Your task is to perform necessary operations and measurements to figure out which unitary it was and to return 0 if it was the Rz gate or 1 if it was the R1 gate.

You are allowed to apply the given operation and its adjoint/controlled variants exactly once (you can pass to it any rotation angle that you choose).

You have to implement an operation which takes a single-qubit operation `unitary` as an input and returns an integer. The operation `unitary` will accept two parameters (similarly to Rz and R1 intrinsic gates): the first parameter is the rotation angle and the second parameter is the qubit to which the rotation is applied. Your code should have the following signature:

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

    operation Solve (unitary : ((Double, Qubit) => Unit is Adj+Ctl)) : Int {
        // your code here
    }
}
```

## A5. Distinguish Rz(θ) from Ry(θ)

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given an angle $\theta$ (in radians, $0.01\pi \leq \theta \leq 0.99\pi$) and an operation that implements a single-qubit unitary transformation: either the $Rz(\theta)$ gate or the $Ry(\theta)$ gate. The operation will have Adjoint and Controlled variants defined.

Your task is to perform necessary operations and measurements to figure out which unitary it was and to return 0 if it was the $Rz(\theta)$ gate or 1 if it was the $Ry(\theta)$ gate.

You are allowed to apply the given operation and its adjoint/controlled variants as many times as you need (within the time limit).

You have to implement an operation which takes a floating-point number and a single-qubit operation as an input and returns an integer. Your code should have the following signature:

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

    operation Solve (theta : Double, unitary : (Qubit => Unit is Adj+Ctl)) : Int {
        // your code here
    }
}
```

# A6. Distinguish four Pauli gates

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given an operation that implements a single-qubit unitary transformation: either the identity (I gate) or one of the Pauli gates (X, Y or Z gate). The operation will have Adjoint and Controlled variants defined.

Your task is to perform necessary operations and measurements to figure out which unitary it was and to return

- 0 if it was the I gate,
- 1 if it was the X gate,
- 2 if it was the Y gate,
- 3 if it was the Z gate.

You are allowed to apply the given operation and its adjoint/controlled variants **exactly once**.

You have to implement an operation which takes a single-qubit operation as an input and returns an integer. Your code should have the following signature:

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

    operation Solve (unitary : (Qubit => Unit is Adj+Ctl)) : Int {
        // your code here
    }
}
```

# A7. Distinguish Y, XZ, -Y and -XZ

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given an operation that implements a single-qubit unitary transformation: either the Y gate (possibly with an extra global phase of $-1$) or the sequence of Pauli Z and Pauli X gates (the Z gate applied first and the X gate applied second; possibly with an extra global phase of $-1$). The operation will have Adjoint and Controlled variants defined.

Your task is to perform necessary operations and measurements to figure out which unitary it was and to return

- 0 if it was the Y gate,
- 1 if it was the $-$XZ gate,
- 2 if it was the $-$Y gate,
- 3 if it was the XZ gate.

You are allowed to apply the given operation and its adjoint/controlled variants **at most three times**.

You have to implement an operation which takes a single-qubit operation as an input and returns an integer. Your code should have the following signature:

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

    operation Solve (unitary : (Qubit => Unit is Adj+Ctl)) : Int {
        // your code here
    }
}
```

# B1. "Is the bit string balanced?" oracle

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Implement a quantum oracle on $N$ qubits which checks whether the input bit string is balanced, i.e., whether it has exactly $\frac{N}{2}$ zeros and $\frac{N}{2}$ ones in it.

Your operation should take the following inputs:

- an array of $N \leq 10$ qubits "inputs" in an arbitrary state. $N$ will be an even number.
- a qubit "output" in an arbitrary state.

Your operation should perform a unitary transformation on those qubits that can be described by its effect on the basis states: if "inputs" is in the basis state $|x\rangle$ and "output" is in the basis state $|y\rangle$, the result of applying the operation should be $|x\rangle|y \oplus f(x)\rangle$, where $f(x) = 1$ if the bit string $x$ has the same number of zeros and ones in it, and $0$ otherwise.

For example, if the qubits passed to your operation are in the state $\frac{1}{\sqrt{2}}(|01\rangle + |00\rangle)_x \otimes |0\rangle_y$, the state of the system after applying the operation should be $\frac{1}{\sqrt{2}}(|01\rangle_x \otimes |1\rangle_y + |00\rangle_x|0\rangle_y)$.

Your code should have the following signature (note that your operation should have Adjoint and Controlled variants defined for it; `is Adj+Ctl` in the operation signature will generate them automatically based on your code):

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

    operation Solve (inputs : Qubit[], output : Qubit) : Unit is Adj+Ctl {
        // your code here
    }
}
```

Your code is not allowed to use measurements or arbitrary rotation gates. This operation can be implemented using just the X gate and its controlled variants (possibly with multiple qubits as controls).

## B2. "Is the number divisible by 3?" oracle

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Implement a quantum oracle on $N$ qubits which checks whether the input bit string is a little-endian notation of a number that is divisible by 3.

Your operation should take the following inputs:

- an array of $N \leq 8$ qubits "inputs" in an arbitrary state.
- a qubit "output" in an arbitrary state.

Your operation should perform a unitary transformation on those qubits that can be described by its effect on the basis states: if "inputs" is in the basis state $|x\rangle$ and "output" is in the basis state $|y\rangle$, the result of applying the operation should be $|x\rangle|y \oplus f(x)\rangle$, where $f(x) = 1$ if the integer represented by the bit string $x$ is divisible by $3$, and $0$ otherwise.

For example, if the qubits passed to your operation are in the state $\frac{1}{\sqrt{2}}(|110\rangle + |001\rangle)_x \otimes |0\rangle_y = \frac{1}{\sqrt{2}}(|3\rangle + |4\rangle)_x \otimes |0\rangle_y$, the state of the system after applying the operation should be $\frac{1}{\sqrt{2}}(|3\rangle_x \otimes |1\rangle_y + |4\rangle_x|0\rangle_y) = \frac{1}{\sqrt{2}}(|110\rangle_x \otimes |1\rangle_y + |001\rangle_x|0\rangle_y)$.

Your code should have the following signature (note that your operation should have Adjoint and Controlled variants defined for it; `is Adj+Ctl` in the operation signature will generate them automatically based on your code):

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

    operation Solve (inputs : Qubit[], output : Qubit) : Unit is Adj+Ctl {
        // your code here
    }
}
```

Your code is not allowed to use measurements or arbitrary rotation gates. This operation can be implemented using just the X gate and its controlled variants (possibly with multiple qubits as controls).

## C1. Prepare superposition of basis states with 0s

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given $N$ qubits in the state $|0...0\rangle$. Your task is to prepare an equal superposition of all basis states that have one or more $0$ in them.

For example, for $N = 2$ the required state would be $\frac{1}{\sqrt{3}}\big(|00\rangle + |01\rangle + |10\rangle\big)$.

You are not allowed to use any gates except the Pauli gates (X, Y and Z), the Hadamard gate and the controlled versions of those

(you are allowed to use multiple qubits as controls in the controlled versions of gates). However, you are allowed to use measurements.

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

Your code should have the following signature:

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

    operation Solve (qs : Qubit[]) : Unit {
        // your code here
    }
}
```

## C2. Prepare superposition of basis states with the same parity

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given $N$ qubits in the state $|0...0\rangle$, and an integer $parity \in \{0, 1\}$. Your task is to prepare an equal superposition of all basis states that have the given parity of the number of $1$s in their binary notation, i.e., the basis states that have an *even* number of $1$s if $parity = 0$ or the basis states that have an *odd* number of $1$s if $parity = 1$.

For example, for $N = 2$ the required state would be

- $\frac{1}{\sqrt{2}}\big(|00\rangle + |11\rangle\big)$ if $parity = 0$.
- $\frac{1}{\sqrt{2}}\big(|01\rangle + |10\rangle\big)$ if $parity = 1$.

You are not allowed to use any gates except the Pauli gates (X, Y and Z), the Hadamard gate and the controlled versions of those (you are allowed to use multiple qubits as controls in the controlled versions of gates). However, you are allowed to use measurements.

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

Your code should have the following signature:

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

    operation Solve (qs : Qubit[], parity : Int) : Unit {
        // your code here
    }
}
```

## D1. Quantum Classification - Dataset 3

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given a training dataset, in which each entry is a features vector (an array of 2 real numbers) and a label 0 or 1 indicating the class to which this vector belongs.

Your goal is to use this dataset to train a quantum classification model that will accurately classify a validation dataset - a different dataset generated using the same data distribution as the training one. The error rate of classifying the validation dataset using your model (the percentage of incorrectly classified samples) should be less than 5%.

- The quantum classification library that will use your model to classify the data is documented here.
- This tutorial has an end-to-end example of training a model using this library as a Python notebook.
- The warmup round editorial discusses solving easier problems features in the warmup round.
- **You can find the exact implementation of the testing harness for the D problems of this round, including the preprocessing methods, here.**
- You can find examples of training a model and using it for classification here.

### Input

Your code will not be given any inputs. Instead, you should use the provided dataset file to train your model.

The training dataset is represented as a JSON file and consists of two arrays, "Features" and "Labels". Each array has exactly 400 elements. Each element of the "Features" array is an array with 2 elements, each of them a floating-point number. Each element of the "Labels" array is the label of the class to which the corresponding element of the "Features" array belongs, 0 or 1.

### Output

Your code should return the description of the model you'd like to use in the following format:

- The model is described using a tuple `((Int, Double[]), ControlledRotation[], (Double[], Double))`.
- The first element of the tuple describes the classical preprocessing you perform on the data before encoding it into the quantum classifier.
- The second element of the tuple describes circuit geometry of the model as an array of controlled rotation gates.
- The third element of the tuple describes numeric parameters of the model and is a tuple of an array of rotation angles used by the gates and the bias used to decide the class of the model.

Your code should have the following signature:

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

    operation Solve () : ((Int, Double[]), ControlledRotation[], (Double[], Double)) {
        // your code here
    }
}
```

**Classical preprocessing**

This step allows you to add new features to the data before encoding it in the quantum state and feeding it into the classifier circuit. To do this, you need to pick one of the available preprocessing methods and return a tuple of its index and its parameters. The parameters of all methods are `Double[]`.

- **Method 1: padding.** The resulting data is a concatenation of the parameters and the features.
- **Method 2: tensor product.** The resulting data is an array of pairwise products of the elements of the parameters and the features.
- **Method 3: fanout.** The resulting data is a tensor product of the parameters, the features and the features (so that you have access to all pairwise products of features).
- **Method 4: split fanout.** The resulting data is tensor product of (concatenation of the left halves of parameters and features) and (concatenation of the right halves).
- **Default method: no preprocessing.** The features remain unchanged, the parameters are ignored. This method is used when any index other than 1-4 is returned.

After the preprocessing step the resulting data is encoded in the quantum state using amplitudes encoding: element $j$ of the data is encoded in the amplitude of basis state $|j\rangle$. If the length of the data array is not a power of 2, it is right-padded with $0$s to the nearest power of two; the number of qubits used for encoding is the exponent of that power.

### Note

Note that majority of the data analysis is going to happen "offline" before you submit the solution. The solution has to contain only the description of the trained model, not the training code itself - if you attempt to train the model "online" in your submitted code during the evaluation process, it will very likely time out.

Training your model offline is likely to involve:

- Defining the circuit structure that your model will use.
- Generating several parameter seed vectors - the values from which training the model will start.
- Selecting appropriate hyperparameters of the training process (learning rate, batch size, tolerance, maximal number of iterations etc.)
- Training a number of classification models (one per each seed vector and hyperparameter combination)
- Selecting the best trained model and submitting it.

## D2. Quantum Classification - Dataset 4

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

This problem is identical to the problem D1 in every aspect except the training dataset. Please refer to that problem for the full problem statement.

## D3. Quantum Classification - Dataset 5

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

This problem is identical to the problem D1 in every aspect except the training dataset. Please refer to that problem for the full problem statement.

## D4. Quantum Classification - Dataset 6

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

This problem is identical to the problem D1 in every aspect except the training dataset. Please refer to that problem for the full problem statement.

## D5. Quantum Classification - Dataset 7

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

This problem is identical to the problem D1 in every aspect except the training dataset. Please refer to that problem for the full problem statement.

## E1. Power of quantum Fourier transform

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Implement an operation that is equivalent to the operation $\text{QFT}^P$, where QFT is the quantum Fourier transform.

Your operation should take the following inputs:

- an integer $P$ ($2 \le P \le 2.1 \cdot 10^6$).
- a register of type `LittleEndian` - a wrapper type for an array of qubits that encodes an unsigned integer in little-endian format, with the least significant bit written first (corresponding to the array element with index 0). (If you need to, you can convert it to an array type using unwrap operator: `let qubitArray = inputRegister!;`)

The "output" of your solution is the state in which it left the input qubits.

Your code should have the following signature (note that your operation should have Adjoint and Controlled variants defined for it; `is Adj+Ctl` in the operation signature will generate them automatically based on your code):

```
namespace Solution {
    open Microsoft.Quantum.Arithmetic;
    open Microsoft.Quantum.Intrinsic;

    operation Solve (p : Int, inputRegister : LittleEndian) : Unit is Adj+Ctl {
        // your code here
    }
}
```

You can learn more about QFT in this kata. You are allowed to take advantage of library operations, including `QFTLE` which implements the necessary transform in the first power.

## E2. Root of quantum Fourier transform

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Implement an operation that is equivalent to the operation $\text{QFT}^{1/P}$, where QFT is the quantum Fourier transform. In other words, your operation, applied $P$ times, should have the same effect as applying QFT. You can implement the required transformation up to a global phase.

Your operation should take the following inputs:

- an integer $P$ ($2 \le P \le 8$).
- a register of type `LittleEndian` - a wrapper type for an array of qubits that encodes an unsigned integer in little-endian format, with the least significant bit written first (corresponding to the array element with index 0). (If you need to, you can convert it to an array type using unwrap operator: `let qubitArray = inputRegister!;`.) The register will contain at most 7 qubits.

The "output" of your solution is the state in which it left the input qubits.

Your code should have the following signature (note that your operation should have Adjoint and Controlled variants defined for it; `is Adj+Ctl` in the operation signature will generate them automatically based on your code):

```
namespace Solution {
    open Microsoft.Quantum.Arithmetic;
    open Microsoft.Quantum.Intrinsic;

    operation Solve (p : Int, inputRegister : LittleEndian) : Unit is Adj+Ctl {
        // your code here
    }
}
```

You can learn more about QFT in this kata. You are allowed to take advantage of library operations, including `QFTLE`.

---