Q# Tutorial

Frances Tibble and Anita Ramanan

May 23, 2018

Abstract

Q# (Q-sharp) is a domain-specific programming language used for expressing quantum algorithms. By the end of this tutorial you will have run a simple 'HelloWorld' example, demonstrated superposition and entanglement, and implemented an algorithm for quantum teleportation. This tutorial is designed to give you a grounding in Q# so that you can independently explore more advanced samples provided by the QDK - and hopefully start writing your own!

1 Installing the Quantum Development Kit (QDK)

The QDK is available for Windows, macOS, and Linux (Ubuntu). You can find the full download instructions here: `https://docs.microsoft.com/en-us/quantum/quantum-installconfig?view=qsharp-preview&tabs=tabid-vscode`

- For Windows: We recommend **Visual Studio 2017** and the **Microsoft Quantum Development Kit extension**. You can also use Visual Studio Code and the Microsoft Quantum Development Kit for Visual Studio Code extension.
- For macOS or Linux: We recommend **Visual Studio Code** and the **Microsoft Quantum Development Kit for Visual Studio Code extension**.

If you have any issues with installation please ask one of the lab helpers.

2 Cloning the Q# Tutorial Code

Once you've got your Q# Development Environment set up, you'll need to get a copy of the tutorial code on your machine. You can clone that from the following location: `https://github.com/frtibble/QuantumWorkshop`

You can use the following Git command from your favourite terminal to do this:

```
$ git clone https://github.com/frtibble/QuantumWorkshop
```

3 Writing a Quantum Program - The Basics

The Q# Programming Language
A natural model for quantum computation is to treat the quantum computer as a **coprocessor**, similar to that used for GPUs, FPGAs, and other adjunct processors. In this model, the primary control logic runs **classical** code on a classical 'host' computer. When we need to exploit quantum properties, the host program can invoke a **quantum** sub-program that runs on the adjunct processor.

Our project structure will have the following files:

- A `Driver.cs` file, which will hold the classical C# code that will 'drive' our quantum code. This can be written in C#, F# or any other classical language with support for .NET.
- An `Operation.qs` file, which will hold the quantum code itself.

3.1 Create a Project and Solution

To see how these files are used together, let's create a simple 'Hello World' project:

- For those using **Visual Studio 2017**:
  - Open up Visual Studio 2017.
  - Go to the `File` menu and select `New > Project...`.
  - In the project template explorer, under `Installed > Visual C#`, select the `Q# Application` template. Make sure you have `.NET Framework 4.6.1` selected in the list at the top of the `New Project` dialog box.
  - Give your project the name `HelloWorld`.
- For those using the command line or **Visual Studio Code**:
  - Run the following commands in your favourite command line (e.g PowerShell or Bash). The final command opens the code in Visual Studio Code.

```
$ dotnet new -i "Microsoft.Quantum.ProjectTemplates::0.2-*"
$ dotnet new console -lang Q# --output HelloWorld
$ cd HelloWorld
$ code .
```

Again, if you have any issues with installation please share your frustration with one of the lab helpers.

3.2 Add the Quantum Code

Q# Operation and Function Types
A Q# **operation** is a **quantum** subroutine that contains quantum operations.

A Q# **function** is a **classical** subroutine used within a quantum algorithm. It may contain classical code but no quantum operations. Functions may not allocate or borrow qubits, nor may they call operations. It is possible, however, to pass them operations or qubits for processing.

Together, operations and functions are known as **callables**.

To write our 'Hello World' example we're going to modify the `Operation.qs` file in our newly created project:

- You'll see it contains the definition for an operation which is currently empty. We're going to replace that definition with a new one called 'Greet', which will return a greeting using the input. The code for 'Greet' is this:

```
operation Greet (who : String) : (String)
{
    body
    {
        return $"Hello, {who}!";
    }
}
```

- Looking at the operation signature, we can see it takes a `String` as input, and returns a `String` as output (given after the colon).

Primitive Types
The `String` type is a sequence of Unicode characters that is opaque to the user once created. This type is used to report messages to a classical host.

Other primitive types include `Int`, `Double`, `Bool`, `Qubit`, `Pauli`, `Result` and `Range`. You can find more information on these in the documentation, but we'll encounter many of these soon.

`https://docs.microsoft.com/en-us/quantum/quantum-qr-typemodel?view=qsharp-preview#primitive-types`

Type Signatures
All Q# callables are considered to take a single value as input and return a single value as output. Both the input and output values may be tuples. Callables that have no result return the empty tuple, `()`. Callables that have no input take the empty tuple, `()`, as input.

3.3 Add the Classical Code

Now that we have a quantum routine, we can call it from the `Main` method in the `Driver.cs` file. The C# driver does four key things:

1. It uses a quantum simulator to execute and debug our quantum algorithms.
2. It computes any arguments that are required for the quantum algorithm to run.
3. It runs the quantum algorithm. Note: it executes asynchronously because execution on actual hardware will be asynchronous. When we fetch the `Result` property; this blocks execution until the task completes and returns the result synchronously.
4. It processes the result of the operation.

Here's the code that you need to add inside the `Main` method body:

```
        using (var sim = new QuantumSimulator())
         {
             var result = Greet.Run(sim, "Imperial").Result;
             System.Console.WriteLine(result);
         }
```

The Full State Quantum Simulator
The quantum simulator is exposed via the `QuantumSimulator` class. To use the simulator, simply create an instance of this class and pass it to the `Run` method of the quantum operation you want to execute along with the rest of the parameters.

3.4 Run the Code

- For those using Visual Studio 2017: Just hit `F5`, and your program should build and run!
- For command line and Visual Studio Code:

```
$ dotnet run
```

This should output 'Hello, Imperial!'. Of course, this isn't very impressive. But now we've seen how it works, we can write our first quantum application.

4 Tutorial: Challenge 1 - Creating a Bell State

To begin this series of challenges we're going to start with the simplest program possible and build it up to demonstrate quantum superposition and quantum entanglement by creating a Bell State. We will start with a qubit in a basis state $|0\rangle$, perform some operations on it and then measure the result.

Primitive Types: Qubit
The `Qubit` type represents a quantum bit or qubit. Once allocated, a qubit can then be passed to functions and operations.

In Q#, all qubits are dynamically allocated and released, rather than being fixed resources that are there for the entire lifetime of a complex algorithm. A using statement allocates a set of qubits at the start and releases those qubits at the end of the block. Any qubits allocated in this way start off in the $|0\rangle$ state.

From the perspective of a Q# program, a qubit is an entirely opaque reference to the internal structure of a target machine. A Q# program has no ability to introspect into the state of a qubit, its representation on a target machine, or even whether it is the same qubit as any other qubit available to the program. Rather, a program can call operations such as `Measure` to learn information from a qubit, and call operations such as `X` and `H` to act on the state of a qubit.

Thus, similar to how a graphics shader program accumulates a description of transformations to each vertex, a quantum program in Q# accumulates transformations to quantum states. This allows us to be entirely agnostic about what a quantum state even *is* on each target machine, which might have different interpretations depending on the machine.

4.1 Task 1: Set a Qubit State

You should have a copy of the tutorial code locally on your machine. Open the 'Bell' folder in Visual Studio or VS Code, and go to 'Operation.qs' where you'll find a commented line that shows you where to write the code for the first (and subsequent) tasks.

When we allocate a qubit it is in the $|0\rangle$ state, but we may receive a qubit in another state. For this reason we are going to begin by creating a `Set` operation which we can use to set a qubit into a known state (`Zero` or `One`). Inside the body of the `Set` operation, add the following code:

```
    let current = M(q1);
```

This code measures the qubit, **q1**, and assigns its value to the **current** variable.

Variables
Q# deals with variables in a unique way. By default, variables in Q# are immutable; their value may not be changed after they are bound. The **let** keyword is used to indicate the binding of an **immutable** variable. Operation arguments are always immutable.

If you need a variable whose value *can* change, you can declare the variable with the **mutable** keyword. A mutable variable's value may be changed using a **set** statement.

In both cases, the type of a variable is inferred by the compiler. Q# doesn't require any type annotations for variables.

Beneath that line of code, we are going to use an **if** statement to see if our qubit is in the state we want. If it is, we leave it alone, otherwise, we flip it with the `X` gate. How can we translate that into code? Like this:

```
            if (current == desired)
            {
                // Do nothing
            } else {
                X(q1);
            }
```

This can be written more succinctly, so you should add this code to the **body**, underneath the measurement:

```
            if (current != desired)
            {
                X(q1);
            }
```

The complete file should now look like:

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

    operation Set (desired: Result, q1: Qubit) : ()
    {
        body
        {
            let current = M(q1);
            if (desired != current)
            {
                X(q1);
            }
        }
    }
}
```

Great!

4.2 Task 2: Create a Looping Counter

When we create a Bell State we create an equal superposition of `Zero` and `One`. However, when we perform a measurement, the qubit will collapse to just one of these states. If we would like to verify that we *are* creating a Bell State, we can do so statistically, by creating a `BellTest` that performs this process multiple times, and maintains a count of how many times we've observed a `One` and how many times we've observed a `Zero`. To begin with, we'll create a looping counter that simply sets a qubit in some state and then measures it - so there should be no surprises here.

Note: To begin with, we'll give a full description of the problem, and a full solution. As we progress, the solutions will become less complete so that it is more of a challenge.

To complete the code body of `BellTest` you should:

- Declare a **mutable** variable with the name 'numOnes' with the initial value '0'. This is what we will use to tally how many `Ones` we have seen.
- Add a `using` statement, which allocates an array of qubits for use in our block of code. The array should contain 1 `Qubit` and be allocated to a variable named 'qubits'. The syntax for that looks like this:

```
using (qubits = Qubit[1])
     {
          //
          // More code will go here
          //
     }
```

- Inside the `using` statement, you should add a `for` loop which loops from **1** to **count**. This will enable us to repeat the process of creating a Bell State and measuring it multiple times. The syntax for that looks like this:

```
        for (test in 1..count)
        {
             //
             // More code will go here
             //
        }
```

- Inside the `for` loop you should use the `Set` operation we defined earlier to set the first qubit in our array to be the **initial** value. We can access the first qubit in the array with `qubits[0]`. This sets our qubit to a known value.

- Beneath that, you should measure the first qubit in the array, and assign that to an immutable 'result' variable (recall we use the `let` keyword to assign immutable variables).
- Beneath this we're going to add an `if` statement to test if the result is `One`. If it is, we will increment the value of 'numOnes'. Recall that we use the `set` statement to alter the value of mutable variables).
- We are now finished with the qubit we have been using. It is good practice to reset the qubit back to a known state. Using the `Set` operation we defined earlier, set the first qubit in the array to `Zero`. This line of code goes outside the `for` loop, but within the `using` statement.
- Finally, outside the `using` statement, but within the `body` you should add a line of code that returns a tuple, the first in the tuple should be the number of `Zero`s, and the second in the tuple should be the number of `One`s.

Compare your solution to the one provided below:

```
operation BellTest (count : Int, initial: Result) : (Int,Int)
{
    body
    {
        mutable numOnes = 0;
        using (qubits = Qubit[1])
        {
            for (test in 1..count)
            {
                Set (initial, qubits[0]);

                let result = M (qubits[0]);

                // Count the number of ones we saw:
                if (result == One)
                {
                    set numOnes = numOnes + 1;
                }
            }
            Set(Zero, qubits[0]);
        }
        // Return number of times we saw a |0> and number of times we
             saw a |1>
        return (count-numOnes, numOnes);
    }
}
```

Now switch to the `Driver.cs` file in your development environment. We are going to write the classical code which will do the main tasks we discussed, use a quantum simulator, compute any arguments we need, run the algorithm, and process the result.

- First, you're going to add a `using` statement inside the `Main` method, to create the quantum simulator that we will use - remember we saw that syntax in the 'HelloWorld' project.

- Inside the `using` statement we're going to create an array of `Result` values. These are the **initial** values we'll use in `BellTest`. The code for that is this:

```
            // Try initial values
            Result[] initials = new Result[] { Result.Zero,
                Result.One };
```

- Underneath that line we'll have a `foreach` statement, which will loop through each `Result` in **initials**. The syntax for that looks like this:

```
        foreach (Result initial in initials)
        {
             //
             // More code will go here
             //
        }
```

- Inside the `foreach` statement, we're going to add a line that runs the `BellTest` using the quantum simulator, a count of '1000' and the initial `Result` as parameters, and assigns it to a 'result' variable. If you're not sure how to do this, remind yourself of how we achieved this in the 'HelloWorld' example with the simulator and string parameters.
- The `Result` in the previous line is a tuple. We need to deconstruct the tuple to get the two fields.
- Finally, we should write these out to the console.

The full code for the `Driver.cs` file is this:

```
using Microsoft.Quantum.Simulation.Core;
using Microsoft.Quantum.Simulation.Simulators;

namespace Quantum.Bell
{
    class Driver
    {
        static void Main(string[] args)
        {
            using (var sim = new QuantumSimulator())
            {
                // Try initial values
                Result[] initials = new Result[] { Result.Zero,
                    Result.One };
                foreach (Result initial in initials)
                {
                    var res = BellTest.Run(sim, 1000, initial).Result;
                    var (numZeros, numOnes) = res;
                    System.Console.WriteLine(
                    $"Init:{initial,-4} 0s={numZeros,-4}
                        1s={numOnes,-4}");
                }
            }
            System.Console.WriteLine("Press any key to continue...");
            System.Console.ReadKey();
        }
    }
}
```

Now you can run the code! You should get the following output:

```
Init:Zero 0s=1000 1s=0
Init:One 0s=0   1s=1000
Press any key to continue...
```

4.3 Task 3: Create a Superposition

Currently our code just sets our qubit to a desired value, measures it, and returns the result. In this next task we're going to modify our code so that we set our qubit to a desired value, put it in a superposition, measure it, and return the result.

- Return to the `BellTest` code and add a line of code that applies a Hadamard operation, `H()`, to the first qubit in the array.

If you run your code again you should get the following output:

```
Init:Zero 0s=484 1s=516
Init:One 0s=522 1s=478
Press any key to continue...
```

Every time we measure, we ask for a classical value, but the qubit is in a continuous state halfway between 0 and 1, so we get (statistically) 0 half the time and 1 half the time. This is known as **superposition** and gives us our first real view into a quantum state.

4.4 Task 4: Create Entanglement

Now we'll make the promised Bell State and show off **entanglement**.

- The first thing we'll need to do is allocate 2 qubits instead of 1 in `BellTest`.
- Just as before, we're going to set the first qubit (at index 0) to be the **initial** value.
- Beneath that, we're going to set the second qubit (at index 1) to be `Zero`.
- Next line, we're going to keep our Hadamard applied to the first qubit as before - that line stays the same.

- Below that, we're going to use a `CNOT` operation with the first qubit as the control, and the second qubit as the target. You can find more information on `CNOT` and how to use it in the documentation (along with the other operations that have already been mentioned, such as `M` and `X`): `https://docs.microsoft.com/en-us/qsharp/api/prelude/microsoft.quantum.primitive.cnot?view=qsharp-preview`)
- Then we're going to keep the line that measures the first qubit and assigns the value to the result variable.
- Finally, we also need to reset the second qubit before releasing it (this could also be done with a `for` loop). We'll add a line after qubit 0 is reset to do this.

If we run this again, we'll get exactly the same 50-50 result we got before. However, what we're really interested in is how the second qubit reacts to the first being measured. We're going to capture this with another statistic.

- Go to `BellTest` once again. Below the 'numOnes' declaration, add a **mutable** variable 'agree' with the value '0'.
- Inside the `for` loop, after we've made the measurement on the first qubit, we're going to add an `if` statement. The condition for the `if` statement will be, 'does a measurement on the *second* qubit equal the result?'. If it *does* we want to increment 'agree' by 1.
- Finally, we're going to modify what we return. Currently we return two things: the number of times we saw a $|0\rangle$ and the number of times we saw a $|1\rangle$. We're going to add a third to this, the number of times the value of the second qubit was the same as the first qubit after measurement, i.e. our 'agree' variable.
- One more thing - since we're now returning three `Int`s instead of two, we'll need to change the type signature of `BellTest` to reflect this.

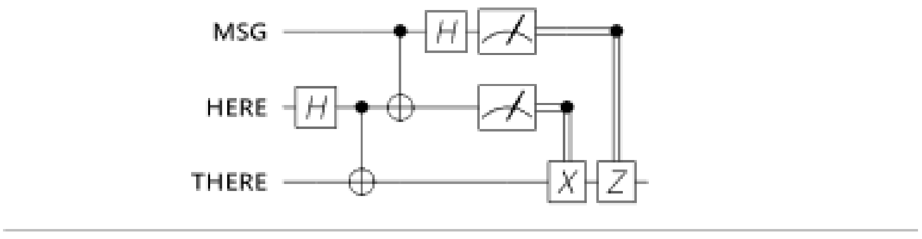
There is now a new return value (`agree`) that will keep track of every time the measurement from the first qubit matches the measurement of the second qubit. Of course, we also have to update the `Driver.cs` file accordingly:

- Where we desconstruct the `Result` to extract the 'numZeros' and 'numOnes', we're going to add 'agree' to this as well.
- You should also update the output to the console, so that 'agree' is included in the output also.

Now when we run the code, we get something pretty amazing:

```
Init:Zero 0s=499 1s=501 agree=1000
Init:One 0s=490 1s=510 agree=1000
```

Our statistics for the first qubit haven't changed (50-50 chance of a 0 or a 1), but now when we measure the second qubit, it is **always** the same as what we measured for the first qubit. Our `CNOT` has entangled the two qubits so that their states are now correlated, regardless of how far apart they are. If



```
Teleportation(msg : Qubit, there : Qubit) : ()
```

Figure 1: A text-book quantum circuit that implements the teleportation, including the quantum part, the measurements, and the classically-controlled correction operations.

you reversed the measurements (did the second qubit before the first), the same thing would happen. The first measurement would be random and the second would be in lock step with whatever was discovered for the first.

5 Tutorial: Challenge 2 - Quantum Teleportation

Quantum teleportation is a technique for sending an unknown quantum state (which we'll refer to as the '**message**') from a qubit in one location to a qubit in another location (we'll refer to these qubits as '**here**' and '**there**', respectively). It can be achieved using the quantum circuit in Figure 1.

The tasks in the following challenge are broken down into 'theory' and 'code'. The theory isn't necessary to complete the tasks, but it may be useful if you want to understand what they are for!

5.1 Task 1: Create an Entangled State

Theory
We can represent our **message** as a vector using Dirac notation:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

The **message** qubit's state is unknown to us as we do not know the values of α and β.

In order to send the message we need for the qubit **here** to be entangled with the qubit **there**. This is achieved by applying a Hadamard gate, followed by a CNOT gate. Let's look at the math behind these gate operations. We will begin with the qubits **here** and **there** both in the $|0\rangle$ state. After entangling these qubits, they are in the state:

$$|\phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

First you need to open the 'QuantumTeleportation' folder in Visual Studio or VS Code. Here you'll find 'Operation.qs'. In this task you need to add to the **body** of the `Teleport` operation.

- The first thing you should do is write a `using` statement that allocates 1 `Qubit` to 'register', just as we did in `BellTest`.
- Next you are going to assign the first qubit in that register to an immutable variable called 'here'.
- We can then create the entangled pair between 'here' and 'there'. The first thing you want to do is apply a `Hadamard` to 'here' (you can see this in the circuit diagram).
- Next you want to apply a `CNOT` with 'here' as the control, and 'there' as the target.

5.2 Task 2: Send the Message

Theory
To send the **message** we first apply a CNOT gate with the **message** qubit and **here** qubit as inputs (the **message** qubit being the control and the **here** qubit being the target qubit, in this instance). This input state can be written:

$$|\psi\rangle|\phi^+\rangle = (\alpha|0\rangle + \beta|1\rangle)(\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle))$$

This expands to:

$$|\psi\rangle|\phi^+\rangle = \frac{\alpha}{\sqrt{2}}|000\rangle + \frac{\alpha}{\sqrt{2}}|011\rangle + \frac{\beta}{\sqrt{2}}|100\rangle + \frac{\beta}{\sqrt{2}}|111\rangle$$

As a reminder, the CNOT gate flips the target qubit when the control qubit is 1. So for example, an input of $|000\rangle$ will result in no change as the first qubit (the control) is 0. However, take a case where the first qubit is 1 - for example an input of $|100\rangle$. In this instance, the output is $|110\rangle$ as the second qubit (the target) is flipped by the CNOT gate. Let's now consider our output once the CNOT gate has acted on our input above. The result is:

$$\frac{\alpha}{\sqrt{2}}|000\rangle + \frac{\alpha}{\sqrt{2}}|011\rangle + \frac{\beta}{\sqrt{2}}|110\rangle + \frac{\beta}{\sqrt{2}}|101\rangle$$

The next step to send the **message** is to apply a Hadamard gate to the **message** qubit (that's the first qubit of each term). As a reminder, the Hadamard gate does the following:

Input	Output
$\|0\rangle$	$\frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$
$\|1\rangle$	$\frac{1}{\sqrt{2}}(\|0\rangle - \|1\rangle)$

If we apply the Hadamard gate to the first qubit of each term of our output above, we get the following result:

$$\frac{\alpha}{\sqrt{2}}(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle))|00\rangle + \frac{\alpha}{\sqrt{2}}(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle))|11\rangle +$$

$$\frac{\beta}{\sqrt{2}}(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle))|10\rangle + \frac{\beta}{\sqrt{2}}(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle))|01\rangle$$

Note that each term has two $\frac{1}{\sqrt{2}}$ factors. We can multiply these out giving the following result:

$$\frac{\alpha}{2}(|0\rangle + |1\rangle)|00\rangle + \frac{\alpha}{2}(|0\rangle + |1\rangle)|11\rangle + \frac{\beta}{2}(|0\rangle - |1\rangle)|10\rangle + \frac{\beta}{2}(|0\rangle - |1\rangle)|01\rangle$$

The $\frac{1}{2}$ factor is common to each term so we can now take it outside the brackets:

$$\frac{1}{2}\big[\alpha(|0\rangle + |1\rangle)|00\rangle + \alpha(|0\rangle + |1\rangle)|11\rangle + \beta(|0\rangle - |1\rangle)|10\rangle + \beta(|0\rangle - |1\rangle)|01\rangle\big]$$

We can then multiply out the brackets for each term giving:

$$\frac{1}{2}\big[\alpha|000\rangle + \alpha|100\rangle + \alpha|011\rangle + \alpha|111\rangle + \beta|010\rangle - \beta|110\rangle + \beta|001\rangle - \beta|101\rangle\big]$$

In this task you are going to add to the **body** of the `Teleport` operation, beneath the code in Task 1.

- To send the message you need to use a `CNOT` gate with ‘msg’ as the control and ‘here’ as the target.
- You then need to apply a `Hadamard` gate to ‘msg’.

5.3 Task 3: Measuring the result

Theory
Due to **here** and **there** being entangled, any measurement on **here** will affect the state of **there**. If we measure the first and second qubit (**message** and here) we can learn what state there is in, due to this property of entanglement.

- If we measure and get a result 00, the superposition collapses, leaving only terms consistent with this result. That's $\alpha|000\rangle + \beta|001\rangle$. This can be refactored to $|00\rangle(\alpha|0\rangle + \beta|1\rangle)$. Therefore if we measure the first and second qubit to be 00, we know that the third qubit, **there**, is in the state $(\alpha|0\rangle + \beta|1\rangle)$.

- If we measure and get a result 01, the superposition collapses, leaving only terms consistent with this result. That's $\alpha|011\rangle + \beta|010\rangle$. This can be refactored to $|01\rangle(\alpha|1\rangle + \beta|0\rangle)$. Therefore if we measure the first and second qubit to be 01, we know that the third qubit, **there**, is in the state $(\alpha|1\rangle + \beta|0\rangle)$.

- If we measure and get a result 10, the superposition collapses, leaving only terms consistent with this result. That's $\alpha|100\rangle - \beta|101\rangle$. This can be refactored to $|10\rangle(\alpha|0\rangle - \beta|1\rangle)$. Therefore if we measure the first and second qubit to be 10, we know that the third qubit, **there**, is in the state $(\alpha|0\rangle - \beta|1\rangle)$.

- If we measure and get a result 11, the superposition collapses, leaving only terms consistent with this result. That's $\alpha|111\rangle - \beta|110\rangle$. This can be refactored to $|11\rangle(\alpha|1\rangle - \beta|0\rangle)$. Therefore if we measure the first and second qubit to be 11, we know that the third qubit, **there**, is in the state $(\alpha|1\rangle - \beta|0\rangle)$.

In this task you are going to add to the **body** of the `Teleport` operation, beneath the code in Task 2.

- We are going to perform a measurement first on 'msg'. You need to write an `if` statement. The condition of the `if` statement will be: measure 'msg' and sees if it is equal to `One`. For now, leave the body code empty (we'll fill this in in the next task). Hint: you can check for equality using '=='.

- We're also going to add an `if` statement whose condition is: measure 'here' and see if it is equal to `One`. Again, leave the body code empty until the next task.

5.4 Task 4: Interpreting the result

Theory
As a reminder, the original message we wished to send was:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

We need to get the **there** qubit into this state, so that the received state is the one that was intended.

- If we measured and got a result of 00, then the third qubit, **there**, is in the state $(\alpha|0\rangle + \beta|1\rangle)$. As this is the intended **message**, no alteration is required.
- If we measured and got a result of 01, then the third qubit, **there**, is in the state $(\alpha|1\rangle + \beta|0\rangle)$. This differs from the intended **message**, however applying a NOT gate gives us the desired state $(\alpha|0\rangle + \beta|1\rangle)$.
- If we measured and got a result of 10, then the third qubit, **there**, is in the state $(\alpha|0\rangle - \beta|1\rangle)$. This differs from the intended **message**, however applying a Z gate gives us the desired state $(\alpha|0\rangle + \beta|1\rangle)$.
- If we measured and got a result of 11, then the third qubit, **there**, is in the state $(\alpha|1\rangle - \beta|0\rangle)$. This differs from the intended **message**, however applying a NOT gate followed by a Z gate gives us the desired state $(\alpha|0\rangle + \beta|1\rangle)$.

To summarize, if we measure and the first qubit is 1, a Z gate is applied. If we measure and the second qubit is 1, a NOT gate is applied.

In this task you are going to add the body of the `if` statements you wrote in Task 3.

- If we measure the first qubit ‘msg’ and the result is `One`, we need to apply a `Z` gate, which we can do using `Z()`. You can read more about this operation here: `https://docs.microsoft.com/en-us/qsharp/api/prelude/microsoft.quantum.primitive.z?view=qsharp-preview`
- If we measure the second qubit ‘here’ and the result is `One`, we need to apply a `NOT` gate, whcih we can do using `X()`. You can read more about this operation here: `https://docs.microsoft.com/en-us/qsharp/api/prelude/microsoft.quantum.primitive.x?view=qsharp-preview`
- Finally you want to `Reset` the ‘here’ qubit.

5.5 Task 5: Using the Teleport operation

Now that we’ve created an operation that can ‘teleport’ one qubit to another, we’re going to use it to send a qubit state.

- To do that, we’re going to fill out the ‘TeleportArbitraryState’ operation. You can see from the type definition it takes a unitary operation as input and returns nothing.

- Inside the body you should allocate two qubits to a register.
- Using the 'let' keyword, assign the first qubit (at index 0) to 'msg'
- Using the 'let' keyword, assign the second qubit (at index 1) to 'there'
- Apply the unitary gate, u, to 'msg' to prepare state on the message qubit
- Teleport the message, msg, to 'there' using the `TeleportMessage` operation
- Apply the inverse of the unitary to the target qubit. You can do that by writing: `(Adjoint u)(there);`
- Reset 'msg' and 'there'. You can do this using the `ApplyToEach` operation, documented here: `https://docs.microsoft.com/en-us/qsharp/api/canon/microsoft.quantum.canon.applytoeach?view=qsharp-preview`. You should pass it the `Reset` operation, and the array of qubits on which to apply it to ('register').
- You can now run this!

Functors
A functor in Q# defines a new operation from another operation. For example, the operation that results from applying the `Adjoint` functor to the `Y` operation is written as `(Adjoint Y)`. The new operation may then be invoked like any other operation.

Operation types are specified by the their signature and the list of functors they support. For example, the `TeleportArbitraryState` operation has type (Qubit =>() : Adjoint), which indicates that it supports the `Adjoint` functor. An operation type that does not support any functors is specified by its signature alone, with no trailing '`:`'.

Adjoint
In quantum computing, the adjoint of an operation is the complex conjugate transpose of the operation. For operations that implement a unitary operator, the adjoint is the inverse of the operation.

6 Tutorial: Challenge 3 - Simple Algorithms

This isn't really a challenge as such. It is an invitation to explore the quantum samples which you can find here:

`https://github.com/Microsoft/Quantum/tree/master/Samples`
Here's you'll find implementations of Shor's algorithm for factoring integers, Grover's algorithm for searching, and more.

7 Tutorial: Testing

Testing is an important part of quantum development with Q#. The Quantum Development Kit comes with two different simulators (full state and trace) which are used for different types of test and debugging scenarios.

You can find a tutorial on testing using both these simulators in the Testing folder here:
`https://github.com/frtibble/QuantumWorkshop/`

Please let the lab helpers know if you are starting the testing tutorial and they will provide further explanation and a walkthrough of the sample code provided.