**Shor’s Algorithm**

Shor’s algorithm is one of the key algorithms discovered for quantum computing. It allows us to factor integers in a much more efficient manner compared to classical techniques. Prime factorisation can be used to crack cryptographic keys, and with a much more efficient factorisation technique being known we may need to reassess our current cryptographic means.

The ‘quantum part’ of the algorithm finds the period of a function and we can use this period to our advantage when factoring a number.

**Properties:**

Postconditions

* Checks that the c\_amod15 function generates a circuit that contains CSwap, CCX, and CX gates
* Checks that the c\_amod15 function throws an exception if a non-coprime integer is entered
* Checks that the qft\_dagger function generates a circuit that contains H, Swap, and Cphase gates
* Checks that qpe\_amod15 function returns a phase between 0 and 1
* Checks that qpe\_amod15 function throws an exception if a non-coprime integer is entered
* Checks that find\_factor function returns an array containing 3 or 5 (or both) upon entering a coprime integer
* Checks that find\_factor function throws an exception upon entering a non-coprime integer

Metamorphic properties

* Checks that a longer circuit is generated if we use a larger power to generate the modular exponentiation circuit
* Checks that an equal length circuit is generated if we use a, equal power to generate the modular exponentiation circuit
* Checks that a longer circuit is generated if we use a larger power to generate the qft dagger circuit
* Checks that an equal length circuit is generated if we use a, equal power to generate the qft dagger circuit

**Side-by-side implementation examples**

|  |  |  |
| --- | --- | --- |
| Qiskit | CirQ | Q# |
|  |  |  |
| Define a modular exponentiation circuit:  We define a function that takes in a coprime integer: a (with the number we are factoring)  The power parameter is altered when reapplying this gate on the different controlled qubits. | Define a modular exponentiation circuit:  For cirq, we use a class to define a custom gate.  We take this approach to make it easier to apply control qubits to this circuit | We cannot define a modular exponentiation *circuit* in Q#, but we can apply the same gate operations on the qubits that *would be* in the circuit.  Note the “Unit is Ctl” after the operation definition, this tells Q# that we plan to make this operation “controllable”, by other qubits. |
|  |  |  |
| Check that the ‘a’ value is coprime, otherwise throw an exception.  Create a quantum circuit of 4 qubit length.  Apply Swap and X gates to create our custom gate. | Check that the ‘a’ value is coprime, otherwise throw an exception.  Create a quantum circuit of 4 qubit length.  Apply Swap and X gates to create our custom gate. | Check that the ‘a’ value is coprime, otherwise throw an exception.  Create a quantum circuit of 4 qubit length.  Apply Swap and X gates to create our custom gate. |
|  |  |  |
| Convert the function that defines this circuit into a custom gate.  Give the gate a name.  Make the gate controllable by another qubit. | We need to define the functions for the gate class in CirQ.  We tell cirq how many qubits need to be passed into the gate.  We define private variables in the gate class (the same parameters passed into the other functions, ‘a’ and ‘power’).  We also name the circuit. | As already highlighted before, when defining the operation “Unit is Ctl” tells Q# that the operation is controllable. |
|  |  |  |
| Define QFT dagger, this function has been seen before in QPE, so I will not go in too much detail.  We only specify the amount of qubits for the circuit, and the swap/rotation/h gates are applied where needed. | We also include the circuit object and the qubit objects to apply the gates.  (technically we do not need the qubit objects as we can get those from the circuit using the all\_qubits() method) | We only need the length and qubit object. |
|  |  |  |
| Define a function to apply quantum phase estimation on the modular exponentiation circuit (as well as qft dagger to get a result in computational basis) | 🡨 | 🡨 |
|  |  |  |
| Make the main quantum circuit, place the output qubits in superposition. Place the final qubit in the circuit in the |1> position.  Apply the unitary (modular exponentiation circuit) multiple times, like QPE, with a control in each of the output qubits in superposition. | Make the main quantum circuit, place the output qubits in superposition. Place the final qubit in the circuit in the |1> position.  On the final statement in the screenshot, we use controlled\_by(), to tell cirq what qubits control the custom aMod15Gate.  (This is the reason we took the approach of defining it as a separate gate class) | Make the main quantum circuit, place the output qubits in superposition. Place the final qubit in the circuit in the |1> position.  On the final statement in the screenshot, we use (controlled c\_amod15), to tell Q# what qubits control the c\_amod15 operation (This is available because of “Unit is Ctl in the operation definition”) |
|  |  |  |
| Append the qft dagger circuit to the same qubits we placed in superposition.  Measure the same qubits that are in superposition. | Append the qft dagger circuit to the same qubits we placed in superposition.  Measure the same qubits that are in superposition. | Append the qft dagger circuit to the same qubits we placed in superposition.  Measure the same qubits that are in superposition (for loop to apply the M). |
|  |  |  |
| Setup the simulator  Execute the circuit  Measure the result  Calculate the phase from the readings | Setup the simulator  Execute the circuit  Measure the result  Calculate the phase from the readings | No other setup is required, execution and measurement are performed at the [M] stage in the previous screenshot.  Phase calculation is performed in the host program |
|  |  |  |
| Now, the algorithm has a roughly greater than 50 chance of finding a useful result, so we need to repeat the algorithm if we fail (which is why it is placed in a loop)  Using the phase we got from the qpe\_amod15() function we can get the denominator | 🡨 | 🡨  As an extra note: this is done in the host C# program.  The measurement results collected into a string containing the binary representation of the quantum (Q#) output. |
|  |  |  |
|  |  | Calculating phase is more involved in the host C# program (in fact we don’t need the phase, we are only looking for the simplified denominator = r).  We make a function to get the *simplified* denominator of the phase from the numerator and denominator. |
|  |  |  |
| If phase is 0, the output is not useful, so we must run the algorithm again.  After finding the period of the function, we can try and calculate the factors.  If the guessed factors are not 1 and the number we are factoring. | If phase is 0, the output is not useful, so we must run the algorithm again.  After finding the period of the function, we can try and calculate the factors.  If the guessed factors are not 1 and the number we are factoring. | If phase is 0, the output is not useful, so we must run the algorithm again.  After finding the period of the function, we can try and calculate the factors.  If the guessed factors are not 1 and the number we are factoring. |