Shor's Algorithm

Boris Varbanov and Santiago Sager

February 16, 2018

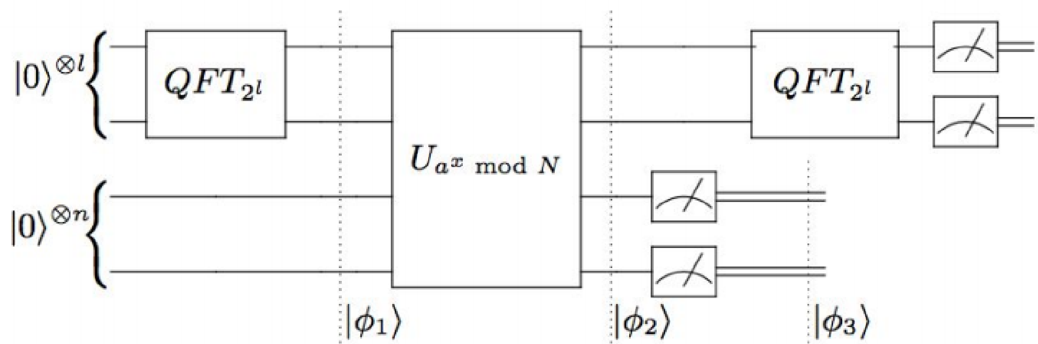
Abstract

In this project we simulate Shor's algorithm on a classical computer in order to factorize a number N. The main focus of this report will be on the description of the circuit for the modular exponentiation gate as well as the implementation, which utilzes the QX Quantum Simulator together with a multi-file Python wrapper.

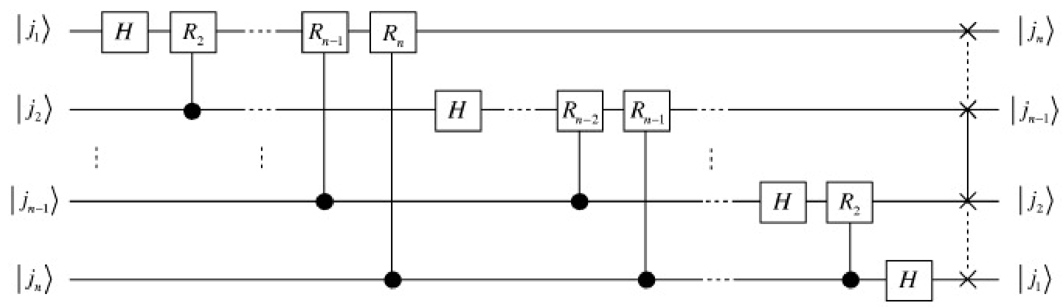
Contents

1 Introduction

Shor's algorithm has quickly become one of the most famous quantum algorithms and for a good reason. With its ability of finding the period of a function, and in particular its ability to factorize large numbers in an efficient manner, it has proven the undoubtable advantage it offers over all known classical algorithms. In this introduction we give a brief and non-detailed overview of Shor's algorithm, focusing on the gates necessary to implement it, rather than the theory behind it. The circuit itself is shown below:



Where we need the first register to have twice as many qbits as the second one. The algorithm starts with both registers set as zero, and it applies the Quantum Fourier Transform (QFT) to the first one, which in this case is equivalent to applying Hadamard gates to each of the qbits. Then we apply the modular exponentiation gate and store the value of the exponentiation in the second register, the construction of this gate is the main focus of this project and will be described in detail later on. We then measure the second register and finally, we apply again the QFT gate to the first register and we measure the outcome. The QFT can be realized using controlled phase shifts in the following way:



Where we have that the phase shift $R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{i2\pi/2^k} \end{pmatrix}$, and that the last gates are just SWAP gates.

After the last QFT we can just measure the outcome and divide it by 2^l, and one of the convergents of the fraction expansion (or one of the lowest multiples) will give us with a certain probability the period of the function. With this we know how to implement Shor's algorithm, except for the modular exponentiation gate, which will be discussed in the following section.

2 Modular exponentiation

In order to perform the modular exponentiation we will decompose it into multiplications that in turn will be decomposed into additions, and these will be performed simply using CNOT and Toffoli gates.

Mathematically (and forgetting the modular aspect for a while) we can express exponentiation as follows:

$$a^x = a^{2^n x_n + 2^{n-1} x_{n-1} + \cdots + 2^1 x_1 + 2^0 x_0} = a^{2^n x_n} a^{2^{n-1} x_{n-1}} \cdots a^{2^1 x_1} a^{2^0 x_0} \tag{1}$$

Since we will be able to choose the value of a, we can classicaly compute a^{2^i} and exponentiation becomes a matter of controlled multiplications depending on the value of the different x_i. In a very similar fashion we can write the multiplications using exclusively additions, so for example, if we call $z = a^{2^k}$ and $y = a^{2^{k-1}} \cdots a^{2^0}$ we find:

$$zy = 2^n z y_n + \cdots + 2^1 z y_1 + 2^0 z y_0 \tag{2}$$

It is useful to notice that the values $2^k z$ will be known, classically computed values and therefore we will be able to set our register to those values at will. We now can just calculate the multiplication as successive additions of those known terms controlled by the previously stored value y. With this specific choice for z and y we can easily see that we can find the exponentiation by starting with the number 1 and the number a, multiplying according to the value of x and storing the value for the next multiplication.

In order to apply the modular part we can use the following properties of modular arithmetics:

$$a + b \bmod N = (a \bmod N + b \bmod N) \bmod N \tag{3}$$

$$ab \bmod N = (a \bmod N \; b \bmod N) \bmod N \tag{4}$$

From this we can easily see that to perform the modular exponentiation we can apply the modular part in any of the intermediate steps, be it the multiplication or the addition. In particular, we will not use normal additions in our circuit but modular additions, so that the modular part will be computed at each step of the exponentiation. This has the advantage of avoiding the use of a large amount of qbits, since at each step we will ensure that the result of the addition will be smaller than N. For the same reason we will compute the classical values $2^k a^{2^i}$ in a modular fashion.

2.1 Adder

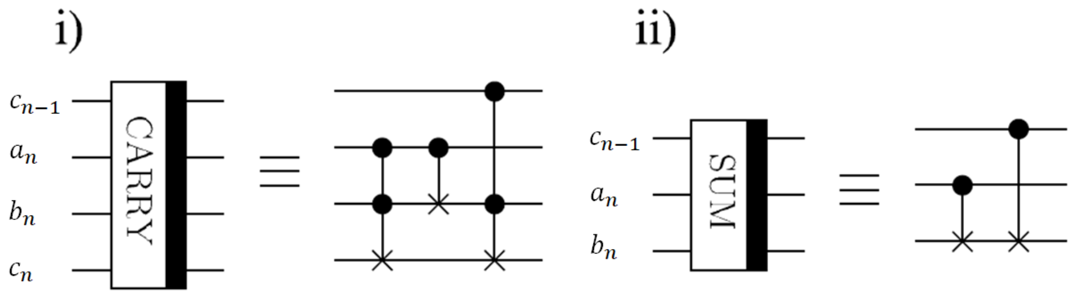
We can express the n-th bit of the number $z = a + b$ as:

$$z_n = a_n \oplus b_n \oplus c_{n-1}$$

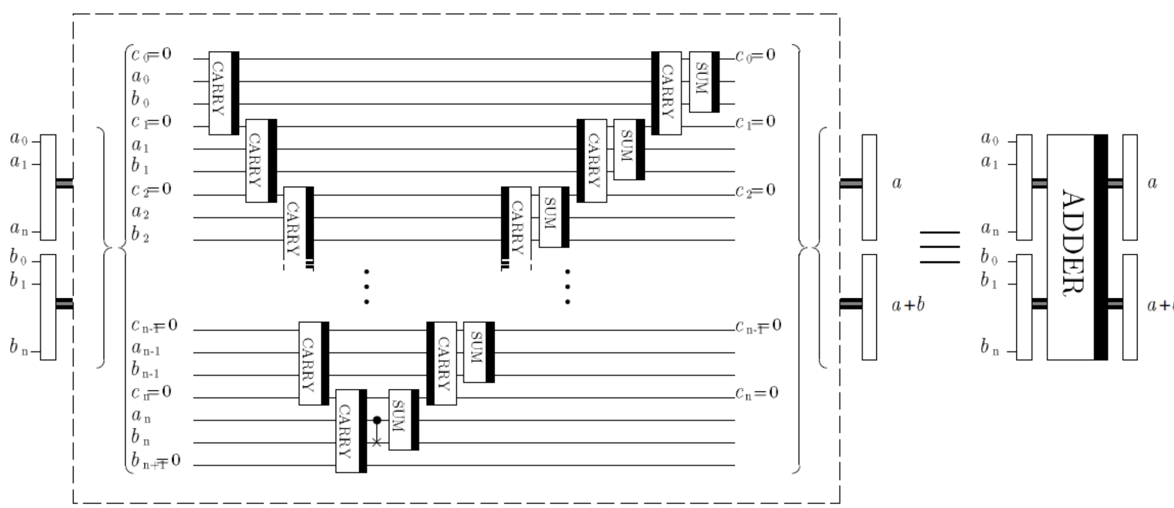
Where c_{n-1} is the carry from the previous bits and $\oplus$ represents the addition modulo 2. Now we can see that we will carry a number if the normal sum exceeds 1, which means that if two or three of the terms in the sum are 1 then $c_n = 1$ and it will be zero otherwise. We can express this as follows:

$$c_n = a_n b_n \oplus a_n c_{n-1} \oplus b_n c_{n-1}$$

We will therefore want to calculate our carry first, and then perform the addition, blocks that will be calculated using CNOTs and Toffoli gates as follows:



Where c_n denotes the carry and a_n, b_n are the digits that have been added. Here the gate Carry calculates the value of the following carry and the gate Sum performs the addition and stores it in the second register. As we expressed previously to perform the addition is necessary to calculate first the carries and then perform the sum, which can be done with the following circuit:

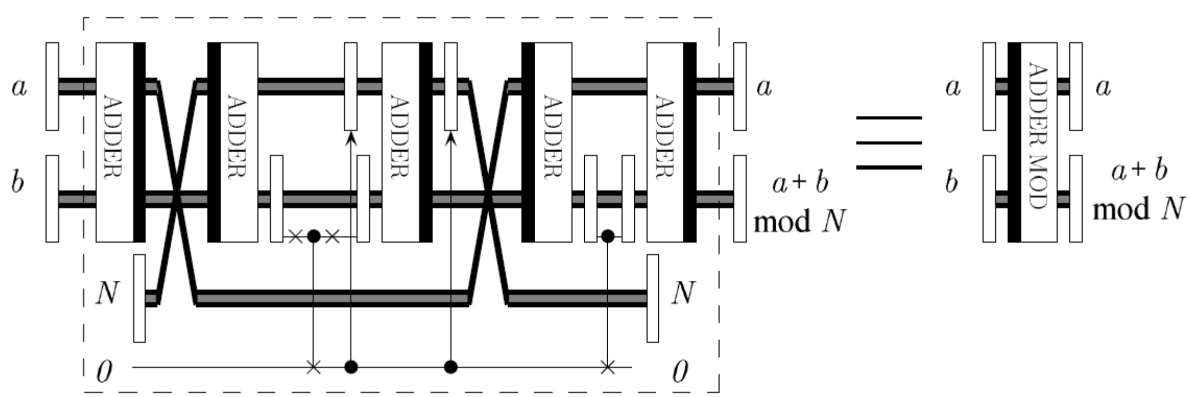


Here we not only have the carries and the sums, but we also have the inverse Carry gate expressed with a black bar on the left side. This gate is necessary not only to reset the second register to its original value (since the Carry itself modifies it) but also to reset the carries in order to perform consecutive additions. The only exception would be the last qbit b_{n-1} which is the overflow qbit of the addition. This qbit is either irrelevant or considered separately and in either case we dont want (or dont need) to reset it. As we can see, this circuit introduces $n+1$ ancilla qbits in the form of carries.

2.2 Modular adder

In order to perform the modular part we will first notice that, as in all operations in quantum computation, these operations we defined are reversible and therefore the inverse adder gate will perform a substraction. We will use the fact that $a, b < N$ to see that $a + b < 2N$ and therefore we will either need to substract N once if $a + b > N$ or do nothing if $a + b < N$.

One way to do this is to substract N so that if $a + b < N$ then the operation overflows, and we can use the overflow qbit as a control to either add N again if it overflowed or do nothing if it didnt. The circuit that performs this is shown below:



This circuit is performing the following steps:

1. We add the values a and b.
2. We substract the value N from the previous value $a + b$
3. $\begin{cases} \text{The ancilla qbit stays at 0} & \text{if overflowed} \\ \text{The ancilla qbit changes to 1} & \text{if not overflowed} \end{cases}$
4. $\begin{cases} \text{The third register stays at } N & \text{if overflowed} \\ \text{The third register gets set to 0} & \text{if not overflowed} \end{cases}$
5. $\begin{cases} \text{We add } N \text{ to } a+b-N \text{ resulting in } a+b & \text{if overflowed} \\ \text{We add 0 to } a+b-N \text{ resulting in } a+b-N & \text{if not overflowed} \end{cases}$
6. We reset the third register to N
7. $\begin{cases} \text{We substract } a\text{, giving us } b \text{ and therefore not overflowing} & \text{if overflowed} \\ \text{We substract } a\text{, giving us } b-N \text{ and therefore overflowing} & \text{if not overflowed} \end{cases}$
8. We use the last overflowed value to restore the ancilla qbit
9. $\begin{cases} \text{We add } a \text{ giving us } a+b & \text{if overflowed} \\ \text{We add } a \text{ giving us } a+b-N & \text{if not overflowed} \end{cases}$

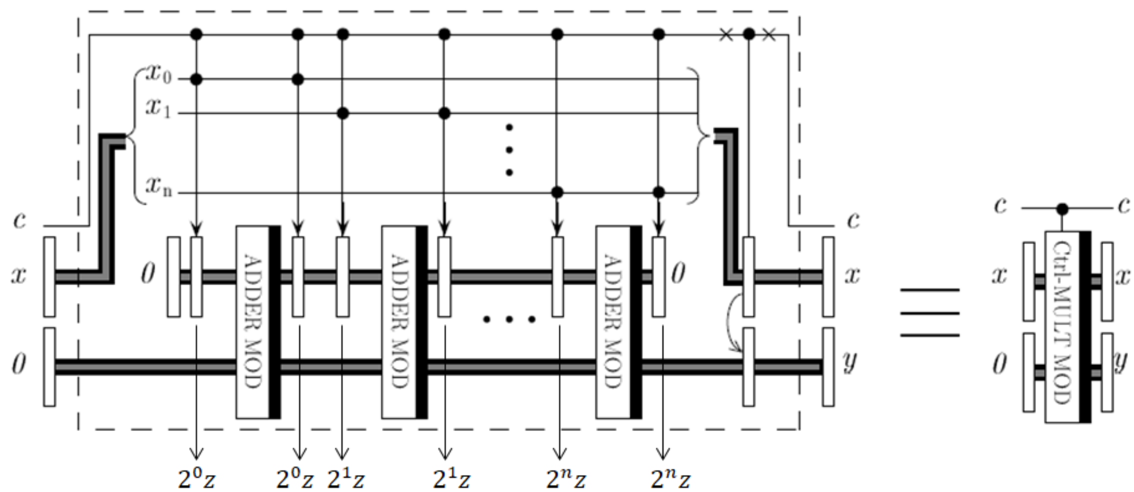
To set the third register to 0 we can just apply flip (X) gates to the qbits which are one, and these are known because the value N is known. As we can see this modular adder also resets the ancilla registers to their original values, allowing us to perform the same procedure again. The number of ancilla qbits used in this gate is $n+1$, which brings the total up to $2n+2$ ancilla qbits.

2.3 Controlled modular multiplication

The next step in constructing our modular exponentiation gate is the modular multiplication. This gate will accept as input the number x and with an ancilla set to zero, will output the values x and $y = x * z$. This means that the number z is not an input but rather an intrinsic property of the gate and so a certain modular multiplication gate has to be labeled with the number it uses to multiplicate. Let's recall how the multiplication can be performed:

$$zx = 2^n zx_n + \cdots + 2^1 zx_1 + 2^0 zx_0 \tag{5}$$

Since we now know how to perform the modular additions we can just set an extra ancilla register to either $2^i z$ or 0 according to the value of x_i , and then add it up to the previous ancilla register where the value will be stored. Additionally, for reasons that we will see later on regarding the exponentiation, we will want this gate to be controlled to either multiply normally or to just copy the number x to the ancilla register. The circuit will be as follows:



The white gates on this circuit are the "set" gates, gates that set the value of the ancilla from zero to the calculated values. Since the modular addition doesn't change the value of the first register, we can see that after each step of setting, addition and setting we have the first ancilla register at zero again and the second register has stored the value depending on the control x_i. The last part of the circuit just copies the value of x , resulting in $y = xz$ or $y = x$ depending on the control. In this case we have another ancilla register, which will have n qbits, making a total of $3n + 2$ qbits used as ancillas.

2.4 Modular exponentiation gate

We can finally implement the modular exponentiation, and in order to do it we will recall how we described the exponentiation in terms of the multiplication:

$$a^x = a^{2^n x_n} a^{2^{n-1} x_{n-1}} \cdots a^{2^1 x_1} a^{2^0 x_0} \tag{6}$$

Here we see that in order to exponentiate we only need to either multiply by a^{2^i} or by 1 depending on the control x_i. This is exactly what we designed the multiplication gate to do, however we need to consider the need to store the value and reset the ancilla gate in order to repeat these multiplications. To achieve this we will use the reversibility of the quantum gates:

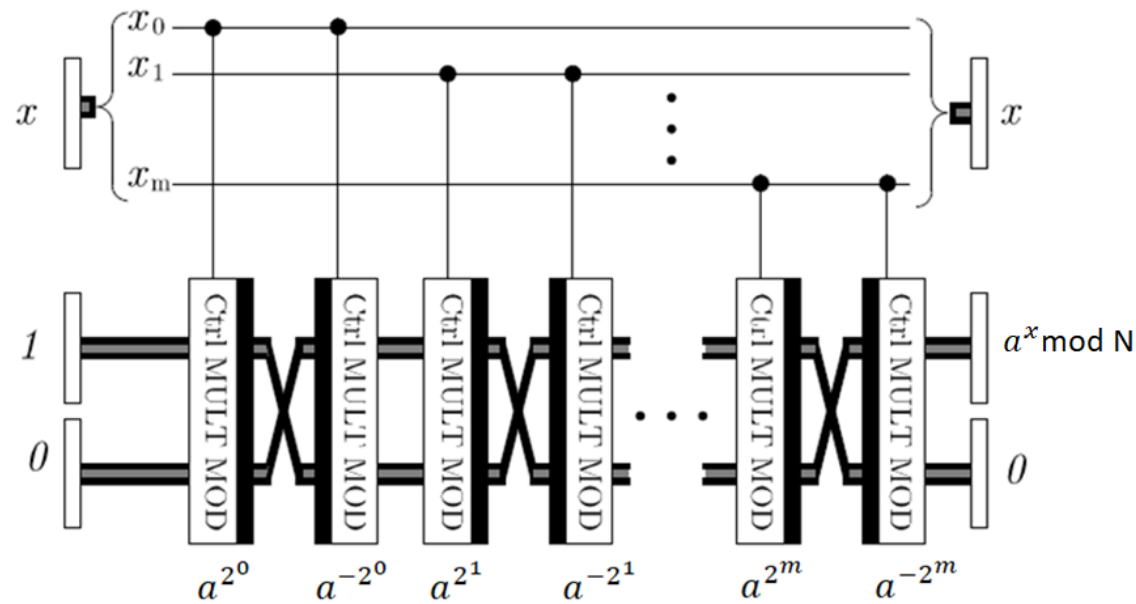
$$\text{Controlled multiplication } a = \begin{cases} x \longrightarrow x \\ 0 \longrightarrow y = xa \end{cases}$$

$$\text{Inverse controlled multiplication } a = \begin{cases} x \longrightarrow x \\ y = xa \longrightarrow 0 \end{cases}$$

This resets the second register, however that register is the one with the value that we want to store. What we can do is realize that we can re-express x as $x = ya^{-1}$, and reversing the roles of the registers we find:

$$\text{Inverse controlled multiplication } a^{-1} = \begin{cases} y \longrightarrow y \\ x = ya^{-1} \longrightarrow 0 \end{cases}$$

Where we have succesfully stored the value y and reset the other register. The only problem comes from the fact that at first glance we now have a non-integer value in the form of a^{-1}, but we can use the modular aspect of the operations to transform it into an integer. The only requirement to do this is for a^{2^i} to be coprime to N, and therefore a being coprime to N, but this is already a requirement for Shor's algorithm. Our final circuit then is:



Where the crossings show the changing in the roles of the registers and the result will either be in the second register if the number of qbits of x is odd, or in the first register if it is even. With this we have succesfully implemented the modular exponentiation gate, and counting the extra register used as ancilla (n) plus the two original registers ($2n$ and n), this brings the total number of qbits used to $7n + 2$ qbits.

3 Implementation

3.1 Code Structure

3.1.1 Main.py

The main file checks the number of parameters given by the user, parses and checks each of them individually and then starts the factoring algorithm. It deals with the "classical" parts of the factoring algorithm and it also initializes an instance of the integrator class, passing along the user input.

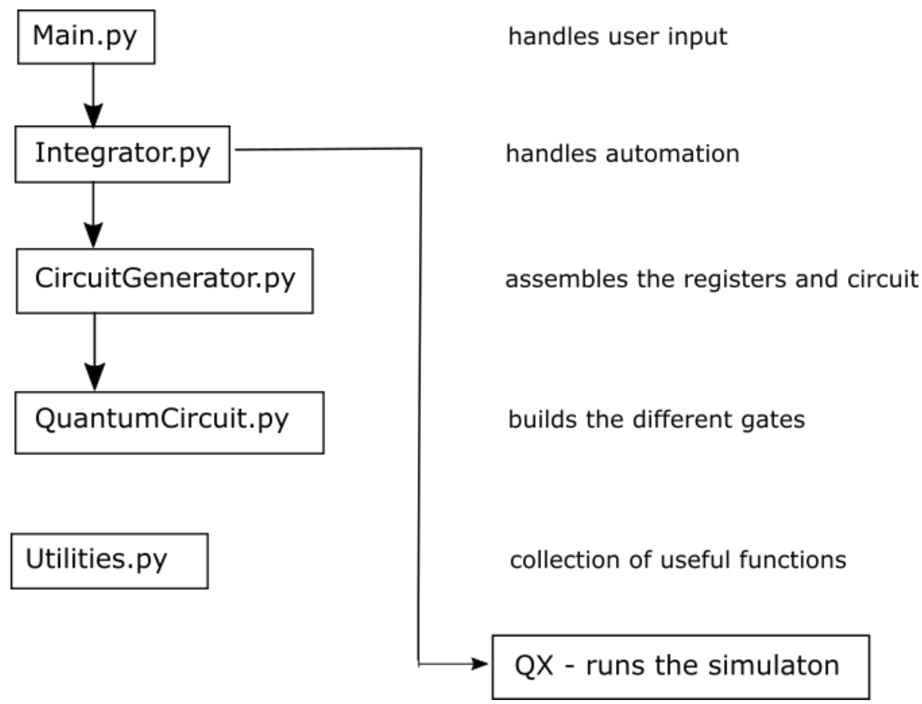


Figure 1: A brief overview of the structure of the implementation.

3.1.2 Integrator.py

The integrator class contains all the parameters of the program as well as a CircuitGenerator object. It is responsible for the initialization of CircuitGenerator, the generation of the circuit and the creation of the corresponding .rc file. It then executes the simulation with the newly written file as a parameter, reads the QX standard output and parses it to extract the measurement of the main computational register.

3.1.3 CircuitGenerator.py

The CircuitGenerator class contains the information about the quantum circuit. It is responsible for the creation of the virtual quantum registers (logical lists of unique qbits), which allow for register-wide application of quantum gates and allow for the code to be more easily understood and maintained. The CircuitGenerator then assembles the main components of the quantum circuit, by passing along the corresponding registers in the correct order. Finally this class handles the writing of the circuit data to the .rc file.

3.1.4 QuantumCircuit.py

The QuantumCircuit class contains a data string, and the definitions of all gates used through the circuit as well as the definitions of the logical registers and the operations associated with them. An application of any gate appends the corresponding QX command (translated to the QUASM language) to the data string.

3.1.5 Utilities.py

The Utilities file contains functions, which deal with file operations, methods to check for input/output, error handling functions, algorithms and arithmetic operations used throughout the creation of the circuit and the output analysis.

3.2 Requirements

- The QX Siumulator v1.0_beta executable for the simulation.
- Python 3.6.4 for the execution of the script.
- Root access if needed, for the purpose of the creation of a file.

3.3 Usage and Input Data

The project executable takes 3 command line parameters: exec_path, output_path, num_to_fac

- exec_path: a path to the location of the QX Simulator executable
- output_path: the path for the .rc output file, which will be created if it doesn't exists or if does, overwritten.

- num_to_fac: a positive integer number, larger than 1, which will be the number that the program will factorize.

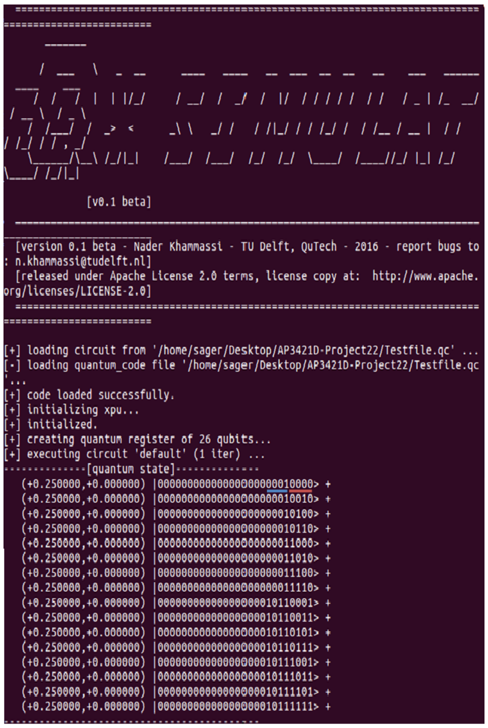
An example of how the program is called is: sudo python3 /home/.../Main.py /home/.../qx_simulator_1.0.beta_linux_x86_64 /home/.../File.qc 15

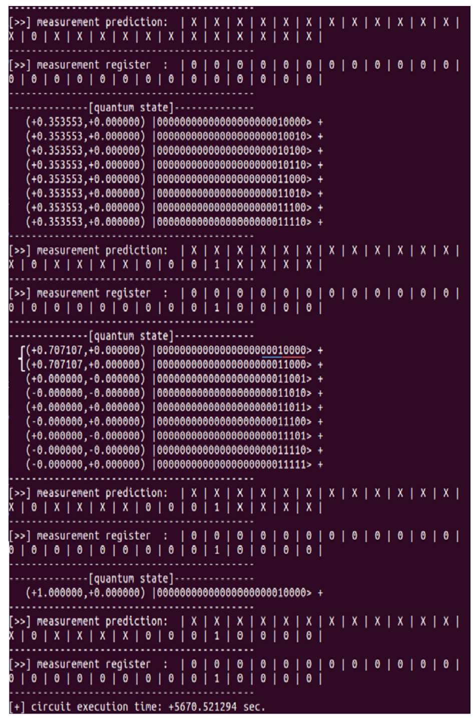
4 Results

This implementation has been done in a very general way, choosing the number to exponentiate a at random, checking if it has factors in common with N, using it in the QX simulation and then using the period to find the factors if it succeeded.

For this section, however, we will restrict this algorithm to the specific choice of $a = 11$ since this has an easy to check final state. The period in this case is $r = 2$, and as we have seen in *Fundamentals of Quantum Information* (Homework assignment 5, Exercise 1), the final state after the QFT will be a superposition of $x = 0$ and $x = 8$ with a plus or minus sign as a relative phase depending on wether we measured 1 or 11 as the exponentiated value.

Our results for this simulation are the following:





Here the rightmost 4 qbits represent the x register, the next 4 qbits to the left are the exponentiated register and the leftmost number represents the amplitude coefficient.

The first (left) figure represents the state after applying the modular exponentiation gate, and as we can see all of the states are in an equal superposition where the even values of x have an associated value of $a^x \text{mod } N = 1$ and the odd values have $a^x \text{mod } N = 11$.

The second (right) figure represents first the state after measuring the exponentiated register, choosing at random in this case the value 1. The second display shows how after the QFT the only nonzero values are the ones that have associated $x = 0$ and $x = 8$, giving us the correct result and therefore proving the success on the implementation of Shor's algorithm.

5 Conclusion and further improvements

We have succesfully created a fully working implementation of Shor's algorithm for an arbitrary number N, however this implementation is severely limited by our computational power. There are two main ways to improve this implementation in order to factorize larger numbers.

Firstly, we could improve the software used to simulate this algorithm. In this case some of the qbits are not being used for all of the operations, since each of the ancilla registers acts only when their respective operation is being computed and have an otherwise constant known value when they aren't part of the operation, hence increasing the number of qbits used in some of the steps. As an example, when performing the modular addition the first step is a normal addition in which the ancilla containing N does not contribute. Changing this however cannot be easily done in QX, since we would need to store the values of the different amplitudes and create different states with those amplitudes.

Secondly, we could improve the theoretical implementation of the whole circuit. There are examples, which show that Shor's algorithm applied to factorizing can be done using only $2n + 3$ qbits [4], reducing the first register to 1 qbit and the modular exponentiation gate using additions in the Fourier space. Furthermore, for the special case of $N = 15$ it can be shown that the total number of qbits necessary drops to just 7, factorization that has been realized experimentally [5].

References

[1] Vlatko Vedral, Adriano Barenco and Artur Ekert. *Quantum Networks for Elementary Arithmetic Operations*. 10.1103/PhysRevA.54.147.

[2] David Elkouss and Leonardo DiCarlo. *Lecture notes and lecture slides for Fundamentals of Quantum Information*. Delft University of Technology.

[3] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge Universityy Press.

[4] Stéphane Beauregard *Circuit for Shor's algorithm using 2n+3 qubits*. Quantum Information and Computation, Vol. 3, No. 2 (2003) pp. 175-185

[5] Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni, Mark H. Sherwood and Isaac L. Chuang. *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance*. Nature, Vol 414, 20/27 December 2001

A Code

Main.py

```
import Integrator
import sys
import Utilities


def main():
    if (len(sys.argv)) != 4:
        print("Invalid number of arguements: arguements expected: exec_path, output_file_path, number_to_fac. Please refer to the documentaton for more detail on the usage.")
        sys.exit() # checks whether the correct number of arguements have been given to the function

    exec_path = sys.argv[1]
    if Utilities.check_exec_path(exec_path) is False:
        print("Please enter a valid QX simulator executable path")
        sys.exit()

    output_file = sys.argv[2]
    if Utilities.check_path_existence(output_file) is False:
        print("Please enter a valid path to an .rc file or to an existing directory")
        sys.exit()

    num_to_fac = Utilities.parseInteger(sys.argv[3])
    if Utilities.check_input_num(num_to_fac) is False:
        print("Invalid integer input, please enter a positive number great or equal to 2")
        sys.exit()

    try:
        # Start by doing the easy to perform checks for factoring numbers (check if N is even or a power of a prime)
        factor = Utilities.check_num_to_fac(num_to_fac)
```

```
        if factor != 0:
            print("A factor of ", num_to_fac, " has been found and it is: ", factor)
            sys.exit()
        # Pick a random power coeffient between 2 and the factor to be factorized
        pwr_coeff = Utilities.get_pwr_coeff(num_to_fac)
        # Check if that coeffient is already a factor
        factor = Utilities.get_gcd_ext(pwr_coeff, num_to_fac)[0]
        if factor > 1:
            print("A factor of ", num_to_fac, " has been found and it is: ", factor)
            sys.exit()
        # Initialize the integrator and pass on the parameters of the program
        integrator_instance = Integrator.Integrator(exec_path, output_file, num_to_fac, pwr_coeff)
        # Create the circuit file
        integrator_instance.startGeneration()
        # Run the QX simulation and obtain the measurement outcome
        measurement = integrator_instance.run_Circuit()
        # Get the period by the method of fraction expansion
        period = Utilities.get_period(measurement/(2 ** (Utilities.get_primary_reg_len(num_to_fac))), num_to_fac, pwr_coeff)
        # Calculate the factor of the number
        factor = Utilities.get_gcd_ext(pwr_coeff ** (period) + 1, num_to_fac)[0]
        print("A factor of ", num_to_fac, " has been found and it is: ", factor)
    except Exception as excpt:
        print(excpt)
        sys.exit()


if __name__ == "__main__":
    main()
```

Integrator.py

```python
import CircuitGenerator
import Utilities
import subprocess


class Integrator:
    """
    The integrator class handles the initization of the circuit generator and the creating of the .rc file
    as well as the running of the simulation and the parsing of results
    """
    def __init__(self, execFilePath, outputFilePath, num_to_fac, pwr_coeff):
        self._execFilePath = execFilePath
        self._outputFilePath = outputFilePath
        self._num_to_factor = num_to_fac
        self._pwr_coeff = pwr_coeff
        self._circuit_generator = CircuitGenerator.CircuitGenerator() # Constructor

    def startGeneration(self):
        # Generates the cirucit data
        self._circuit_generator.generate_circuit(self._num_to_factor, self._pwr_coeff, Utilities.get_primary_reg_len(self._num_to_factor), Utilities.get_secondary_reg_len(self._num_to_factor))
        try:
            with open(self._outputFilePath, ’w’) as _file:
                # saves the circuit data to the open file
                self._circuit_generator.save(_file)
        except Exception as excpt:
            print(excpt) # Runs the circuit generator and calls for the generator to save the .rc file content to the .rc file

    def run_Circuit(self):
        # Creates the QX simulation subprocess
        process = subprocess.Popen([self._execFilePath, self._outputFilePath], stdout=subprocess.PIPE)
```

```
        # Collects the output data of QX
        stdout, stderr = process.communicate()
        # Return the parsed measurement of the first register(as
            ↪ an integer)
        return Utilities.process_output(stdout,
            ↪ self._num_to_factor) # Runs the QX simulator as a
            ↪ background process after which it collects and
            ↪ prases the output of QX
```

CircuitGenerator.py

```
import Utilities
import QuantumCircuit


class CircuitGenerator:
    """
    The CircuitGenerator class holds a quantum circuit object and
        ↪ it itializes the circuit, creates the needed registers
    and applies the main components of Shor's algorithm in the
        ↪ correct order on the corresponding registers (which
    are passes are parameters). Finally it has the method to save
        ↪ the gate sequence string to a file.
    """
    def __init__(self):
        self._circuit = QuantumCircuit.QuantumCircuit() #
            ↪ Constructor

    def generate_circuit(self, mod_num, pwr_coeff,
        ↪ primary_reg_len, secondary_reg_len):
        # Initilize all qubits
        self._circuit.init_quibts(primary_reg_len + 5 *
            ↪ secondary_reg_len + 2)
        # Define the two main computational as well as the
            ↪ anscilla registers
        self._circuit.define_register(0, primary_reg_len)
        self._circuit.define_register(primary_reg_len,
            ↪ primary_reg_len + secondary_reg_len)
        self._circuit.define_register(primary_reg_len +
            ↪ secondary_reg_len, primary_reg_len +
```

```
 2*secondary_reg_len)
        self._circuit.define_register(primary_reg_len + 2*secondary_reg_len, primary_reg_len + 3*secondary_reg_len)
        self._circuit.define_register(primary_reg_len + 3*secondary_reg_len, primary_reg_len + 4*secondary_reg_len)
        self._circuit.define_register(primary_reg_len + 4*secondary_reg_len, primary_reg_len + 5*secondary_reg_len)
        # Create a final register for the addition control qubit
        self._circuit.define_register(primary_reg_len + 5*secondary_reg_len, primary_reg_len + 5*secondary_reg_len+1)
        # Add the overflow anscilla qubit to the ansicall register used in the addition gate
        self._circuit.append_to_register(3, primary_reg_len + 5*secondary_reg_len+1)
        # Create the initial superposiiton
        self._circuit.apply_entanglement(self._circuit.get_register(0))
        # Apply the exponential modular gate between the first and the second computational registers
        self._circuit.apply_exponential_mod_gate(self._circuit.get_register(0), self._circuit.get_register(1), self._circuit.get_register(2), self._circuit.get_register(3), self._circuit.get_register(4), self._circuit.get_register(5), self._circuit.get_register(6)[0], mod_num, pwr_coeff)
        # Measure the second computational register
        self._circuit.measure_register(self._circuit.get_register(1))
        # Apply the quantum Fourier transform to the first computational register
        self._circuit.apply_QFT(self._circuit.get_register(0))
        # Measure the first computational register
        self._circuit.measure_register(self._circuit.get_register(0))
        # Display the measured state
        self._circuit.display_qubits()

```

```python
    def save(self, file):
        Utilities.saveString(file, self._circuit._circut_data) # Saves the circuit data (sequence of gates) to a given file
```

QuantumCircuit.py

```python
import Utilities
import math


class QuantumCircuit:
    """
    The QuantumCircuit file holds the sequence of gate operations and the defined registers.
    It also contains the definition of the signle qubit gates, as well as the more complex gates.
    """
    def __init__(self):
        self._registers = []
        self._circut_data = "" # Constructor

    def define_register(self, first_qubit_label, last_qubit_label):
        qubits = []
        for i in range(first_qubit_label, last_qubit_label):
            qubits.append(i)
        self._registers.append(qubits) # Defines a logical register contains the qubits between the first qubit and the last

    def get_registers_number(self):
        return len(self._registers) # Returns the total number of register

    def get_register_size(self, register_label):
        return len(self._registers[register_label]) # Returns the length of a specified register

    def get_register(self, register_label):
```

```python
        return self._registers[register_label] # Return the register with the specified index

    def append_to_register(self, register_label, qubit_label):
        for i in range(0, self.get_registers_number()):
            if qubit_label in self._registers[i]:
                print("Qubit is already a part of the register, cannot append")
                return False
        self._registers[register_label].append(qubit_label)
        return True # Attempts to append a qubit of a specific index to a specific register

    def init_quibts(self, num_qubts):
        self._circut_data += "qubits %d\n" % num_qubts # Initalizes the qubits

    def apply_hadamard(self, qubit_label):
        self._circut_data += "h q%d\n" % qubit_label # Applies a hadamard gate

    def display_qubits(self):
        self._circut_data += "display\n" # Displays the states of all the qubits

    def measure_qubit(self, qubit_label):
        self._circut_data += "measure q%d\n" % qubit_label # Measures a qubit

    def measure(self):
        self._circut_data += "measure\n" # Measures all qubits

    def apply_cr(self, qubit_label_1, qubit_label_2):
        self._circut_data += "cr q%d,q%d\n" % (qubit_label_1, qubit_label_2) # Applies the phase shift gate used in QFT

    def apply_cnot(self, qubit_label_1, qubit_label_2):
        self._circut_data += "cnot q%d,q%d\n" % (qubit_label_1, qubit_label_2) # Applies a CNOT gate
```

```python

    def apply_x(self, qubit_label):
        self._circut_data += "x q%d\n" % (qubit_label) # Applies a X gate

    def apply_toffoli(self, qubit_label_1, qubit_label_2, qubit_label_3):
        # Applies a toffli gate
        self._circut_data += "toffoli q%d,q%d,q%d\n" % (qubit_label_1, qubit_label_2, qubit_label_3)

    def initialize_register_to(self, register, int_num):
        # Initializes a register to the binary representation of an integer (the register must be in 0)
        _int_str = Utilities.get_binary_form(int_num, len(register))
        for index, bit in enumerate(reversed((_int_str))):
            if bit == 1:
                self.apply_x(register[index])

    def set_register_to(self, qubit, register, int_num, ctrl_qubit=None):
        # Sets a register to the binary representation of an integer, controlled by either 1 or 2 qubits
        _int_str = Utilities.get_binary_form(int_num, len(register))
        for index, bit in enumerate(reversed((_int_str))):
            if bit == 1:
                if ctrl_qubit is None:
                    self.apply_cnot(qubit, register[index])
                else:
                    self.apply_toffoli(ctrl_qubit, qubit, register[index])

    def copy_register(self, register_1, register_2, ctrl_qubit):
        # Copies to contents of a register to another based on a control qubit
        for index, qubit in enumerate(register_1):
                self.apply_toffoli(ctrl_qubit, qubit, register_2[index])
```

```python

    def apply_swap(self, qubit_label_1, qubit_label_2):
        # Swaps two qubits
        self._circut_data += "swap q%d,q%d\n" % (qubit_label_1, qubit_label_2)

    def swap_registers(self, register_1, register_2):
        # Swaps two registers
        for qubit1, qubit2 in zip(register_1, register_2):
            self.apply_swap(qubit1, qubit2)

    def measure_register(self, register):
        # Measures a register
        for qubit in register:
            self.measure_qubit(qubit)

    def apply_carry_gate(self, qubit_label_1, qubit_label_2, qubit_label_3, qubit_label_4):
        # Applies the carry gate
        self.apply_toffoli(qubit_label_2, qubit_label_3, qubit_label_4)
        self.apply_cnot(qubit_label_2, qubit_label_3)
        self.apply_toffoli(qubit_label_1, qubit_label_3, qubit_label_4)

    def apply_sum_gate(self, qubit_label_1, qubit_label_2, qubit_label_3):
        # Applies the sum gate
        self.apply_cnot(qubit_label_2, qubit_label_3)
        self.apply_cnot(qubit_label_1, qubit_label_3)

    def apply_carry_gate_reversed(self, qubit_label_1, qubit_label_2, qubit_label_3, qubit_label_4):
        # Applies the reverse carry gate (same as the carry but in a reversed order)
        self.apply_toffoli(qubit_label_1, qubit_label_3, qubit_label_4)
        self.apply_cnot(qubit_label_2, qubit_label_3)
        self.apply_toffoli(qubit_label_2, qubit_label_3, qubit_label_4)
```

```

    def apply_sum_gate_reveresed(self, qubit_label_1, qubit_label_2, qubit_label_3):
        # Applies the reverse sum gate (same as the sum gate, but in a reversed order)
        self.apply_cnot(qubit_label_1, qubit_label_3)
        self.apply_cnot(qubit_label_2, qubit_label_3)

    def apply_adder_gate(self, comp_register_1, comp_register_2, anscilla_register_1):
        # Applies the adder_gate
        for i in range(0, len(comp_register_1)):
            self.apply_carry_gate(anscilla_register_1[i], comp_register_1[i], comp_register_2[i], anscilla_register_1[i+1])
        self.apply_cnot(comp_register_1[-1], comp_register_2[-1])
        self.apply_sum_gate(anscilla_register_1[-2], comp_register_1[-1], comp_register_2[-1])
        for i in range(len(comp_register_1)-1, 0, -1):
            self.apply_carry_gate_reversed(anscilla_register_1[i-1], comp_register_1[i-1], comp_register_2[i-1], anscilla_register_1[i])
            self.apply_sum_gate(anscilla_register_1[i-1], comp_register_1[i-1], comp_register_2[i-1])

    def apply_adder_gate_reveresed(self, comp_register_1, comp_register_2, anscilla_register_1):
        # Applies the reversed adder gate
        for i in range(0, len(comp_register_1)-1):
            self.apply_sum_gate_reveresed(anscilla_register_1[i], comp_register_1[i], comp_register_2[i])
            self.apply_carry_gate(anscilla_register_1[i], comp_register_1[i], comp_register_2[i], anscilla_register_1[i+1])
        self.apply_sum_gate_reveresed(anscilla_register_1[-2], comp_register_1[-1], comp_register_2[-1])
        self.apply_cnot(comp_register_1[-1], comp_register_2[-1])
        for i in range(len(comp_register_1), 0, -1):
            self.apply_carry_gate_reversed(anscilla_register_1[i-1], comp_register_1[i-1], comp_register_2[i-1],
```

```
            ↪ anscilla_register_1[i])

    def apply_adder_mod_gate(self, comp_register_1,
        ↪ comp_register_2, anscilla_register_1,
        ↪ anscilla_register_2, temp_qubit, int_num):
        # Applies the addition modular gate
        self.initialize_register_to(anscilla_register_2, int_num)
        self.apply_adder_gate(comp_register_1, comp_register_2,
            ↪ anscilla_register_1)
        self.swap_registers(comp_register_1, anscilla_register_2)
        self.apply_adder_gate_reveresed(comp_register_1,
            ↪ comp_register_2, anscilla_register_1)
        self.apply_x(anscilla_register_1[-1])
        self.apply_cnot(anscilla_register_1[-1], temp_qubit)
        self.apply_x(anscilla_register_1[-1])
        self.set_register_to(temp_qubit, comp_register_1, int_num)
        self.apply_adder_gate(comp_register_1, comp_register_2,
            ↪ anscilla_register_1)
        self.set_register_to(temp_qubit, comp_register_1, int_num)
        self.swap_registers(comp_register_1, anscilla_register_2)
        self.apply_adder_gate_reveresed(comp_register_1,
            ↪ comp_register_2, anscilla_register_1)
        self.apply_cnot(anscilla_register_1[-1], temp_qubit)
        self.apply_adder_gate(comp_register_1, comp_register_2,
            ↪ anscilla_register_1)
        self.initialize_register_to(anscilla_register_2, int_num)

    def apply_adder_mod_gate_reversed(self, comp_register_1,
        ↪ comp_register_2, anscilla_register_1,
        ↪ anscilla_register_2, temp_qubit, int_num):
        # Applies the reversed addition modular gate
        self.initialize_register_to(anscilla_register_2, int_num)
        self.apply_adder_gate_reveresed(comp_register_1,
            ↪ comp_register_2, anscilla_register_1)
        self.apply_cnot(anscilla_register_1[-1], temp_qubit)
        self.apply_adder_gate(comp_register_1, comp_register_2,
            ↪ anscilla_register_1)
        self.swap_registers(comp_register_1, anscilla_register_2)
        self.set_register_to(temp_qubit, comp_register_1, int_num)
        self.apply_adder_gate_reveresed(comp_register_1,
```

```
            comp_register_2, anscilla_register_1)
        self.set_register_to(temp_qubit, comp_register_1, int_num)
        self.apply_x(anscilla_register_1[-1])
        self.apply_cnot(anscilla_register_1[-1], temp_qubit)
        self.apply_x(anscilla_register_1[-1])
        self.apply_adder_gate(comp_register_1, comp_register_2, anscilla_register_1)
        self.swap_registers(comp_register_1, anscilla_register_2)
        self.apply_adder_gate_reveresed(comp_register_1, comp_register_2, anscilla_register_1)
        self.initialize_register_to(anscilla_register_2, int_num)

    def apply_ctrl_mult_mod_gate(self, comp_register_1, comp_register_2, anscilla_register_1, anscilla_register_2, anscilla_register_3, temp_qubit, ctrl_qubit, mod_num, int_mult_coeff):
        # Applies the controled multiplication modular gate
        for index, qubit in enumerate(comp_register_1):
            mult_num = Utilities.get_mod(((2 ** index) * int_mult_coeff), mod_num)
            self.set_register_to(qubit, anscilla_register_1, mult_num, ctrl_qubit)
            self.apply_adder_mod_gate(anscilla_register_1, comp_register_2, anscilla_register_2, anscilla_register_3, temp_qubit, mod_num)
            self.set_register_to(qubit, anscilla_register_1, mult_num, ctrl_qubit)
        self.apply_x(ctrl_qubit)
        self.copy_register(comp_register_1, comp_register_2, ctrl_qubit)
        self.apply_x(ctrl_qubit)

    def apply_ctrl_mult_mod_gate_reversed(self, comp_register_1, comp_register_2, anscilla_register_1, anscilla_register_2, anscilla_register_3, temp_qubit, ctrl_qubit, mod_num, mult_coeff):
        # Applies the reversed controled multiplication modular gate
        self.apply_x(ctrl_qubit)
        self.copy_register(comp_register_1, comp_register_2,
```

```python
ctrl_qubit)
        self.apply_x(ctrl_qubit)
        for index, qubit in reversed(list(enumerate(comp_register_1))):
            mult_num = Utilities.get_mod(((2 ** index) * mult_coeff), mod_num)
            self.set_register_to(qubit, anscilla_register_1, mult_num, ctrl_qubit)
            self.apply_adder_mod_gate_reversed(anscilla_register_1, comp_register_2, anscilla_register_2, anscilla_register_3, temp_qubit, mod_num)
            self.set_register_to(qubit, anscilla_register_1, mult_num, ctrl_qubit)

    def apply_exponential_mod_gate(self, comp_register_1, comp_register_2, anscilla_register_1, anscilla_register_2, anscilla_register_3, anscilla_register_4, temp_qubit, mod_num, pwr_coeff):
        # Applies the exponential modular gate
        self.initialize_register_to(comp_register_2, 1)
        for index, qubit in enumerate(comp_register_1):
            mult_coeff = Utilities.get_mod((pwr_coeff ** (2 ** index)), mod_num)
            self.apply_ctrl_mult_mod_gate(comp_register_2, anscilla_register_4, anscilla_register_1, anscilla_register_2, anscilla_register_3, temp_qubit, qubit, mod_num, mult_coeff)
            self.swap_registers(comp_register_2, anscilla_register_4)
            mult_coeff = Utilities.get_mod_inverse((pwr_coeff ** (2 ** index)), mod_num)
            self.apply_ctrl_mult_mod_gate_reversed(comp_register_2, anscilla_register_4, anscilla_register_1, anscilla_register_2, anscilla_register_3, temp_qubit, qubit, mod_num, mult_coeff)

    def apply_entanglement(self, register):
        # Applies Hadamard gates to all quibts of a register to create a superposiiton
        for qubit in register:
```

```python
            self.apply_hadamard(qubit)

    def apply_QFT(self, register):
        # Applies the Quantum Fourier Transform
        for i in range(0, len(register)):
            self.apply_hadamard(register[i])
            for j in range(i+1, len(register)):
                self.apply_cr(register[i], register[j])
        for i in range(0, math.floor(len(register)/2.0)):
            self.apply_swap(register[i], register[-1-i])
```

Utilities.py

```python
import os
import math
from random import randint
import sys
from fractions import gcd
from fractions import Fraction


class InvalidPeriod(Exception):
    """Raised when an invalid period is found by the algorithm"""
    def __init__(self):
        Exception.__init__(self, "an invalid period was found,
            ↪ please run the program again")
    pass  # Error class for invaid periods


def parseInteger(str):
    try:
        value = int(str)
        return value
    except Exception as excpt:
        print(excpt)
        sys.exit() # Parses a string to an integer


def saveString(file, content):
    if file.closed:
```

```python
        print("Error opening the file")
        return False
    try:
        file.write(content)
        return True
    except Exception as excpt:
        print(excpt) # Writes a string to an open file


def is_exe(file_path):
    return os.path.isfile(file_path) and os.access(file_path,
        ↪ os.X_OK)


def check_exec_path(exec_path):
    file_path, file_name = os.path.split(exec_path)
    if file_path:
        if is_exe(exec_path):
            return True
    else:
        for path in os.environ["PATH"].split(os.pathsep):
            exe_file = os.path.join(path, exec_path)
            if is_exe(exe_file):
                return True
    return False


def check_path_existence(output_file):
    if os.path.exists(output_file) or
        ↪ os.path.isdir(os.path.dirname(output_file)):
        return True
    return False # Checks whether the .rc file or its directory
        ↪ exist


def check_input_num(input_num):
    return not (input_num <= 1)


def get_binary_form(int_num, str_len=1):
```

```
        return [int(x) for x in format(int_num, 'b').zfill(str_len)] # Returns the binary representation of a number


def get_gcd_ext(int_num_1, int_num_2):
    if int_num_1 == 0:
        return (int_num_2, 0, 1)
    else:
        g, y, x = get_gcd_ext(int_num_2 % int_num_1, int_num_1)
        return (g, x - (int_num_2 // int_num_1) * y, y) # Get the greatest common denom and other goodies


def get_mod_inverse(int_num, mod_num):
    g, x, y = get_gcd_ext(int_num, mod_num)
    if g != 1:
        raise Exception("The modular inverse of ", int_num, " and ", mod_num, "does not exist.")
    else:
        return x % mod_num # Returns the modular inverse.


def get_mod(int_num, mod_num):
    return int_num % mod_num # Returns the modular of a number


def get_witness(num):
    tmp_num_1, tmp_num_2 = 0, num-1
    while tmp_num_2 % 2 == 0:
        tmp_num_1, tmp_num_2 = tmp_num_1+1, int(tmp_num_2/2)
    for i in range(5):
        rand_num = randint(2, num-1)
        p_num = pow(rand_num, tmp_num_2, num)
        if p_num == 1 or p_num == num-1:
            continue
        for r in range(1, tmp_num_1):
            p_num = (p_num * p_num) % num
            if p_num == 1:
                return rand_num
```

```
            if p_num == num-1:
                break
        else:
            return rand_num
    return 0 # Returns a witness used for checking for prime power


def check_power(num, tmp_num):
    count = 0
    while num > 1 and num % tmp_num == 0:
        num, count = num / tmp_num, count + 1
    if num == 1:
        return tmp_num
    else:
        return 0 # Checks the powers of a prime


def check_num_to_fac(num):
    if num % 2 == 0:
        return 2
    tmp_num = num
    while True:
        witness = get_witness(tmp_num)
        if witness == 0:
            return check_power(num, tmp_num)
        cur_num = gcd(pow(witness, tmp_num, num)-witness, tmp_num)
        if cur_num == 1 or cur_num == tmp_num:
            return 0
        tmp_num = cur_num # Checks if a number is even or a prime power and return the corresponding factor


def process_output(output, _num_to_factor):
    measurement = output.split(b'\n')[-4]
    measurement = measurement.replace(b'|', b'')
    measurement = measurement.replace(b' ', b'')
    return int(measurement[-_num_to_factor:], 2) # Processes the QX output and returns the measurement outcome of the first register.

```

```python

def get_pwr_coeff(num_to_factor):
    return randint(2, num_to_factor) # Picks a random integer between 2 and N-1


def get_primary_reg_len(num_to_factor):
    return math.floor(math.log(2*(num_to_factor ** 2), 2)) # Calculates the length of the primary computational register


def get_secondary_reg_len(num_to_factor):
    return math.ceil(math.log(num_to_factor, 2)) # Calculates the length of the second computational register and the anscilla registers


def calc_period(float_num, mod_num, mult_coeff):
    frac = Fraction(float_num).limit_denominator()
    for i in range(1, 6):
        if get_mod(mult_coeff ** (frac.denominator*i), mod_num) is 1:
            return frac.denominator*i
    return 0 # Checks if a number or a multiple of that number (up to a factor of 5) is a period to a power coeffient mod N


def frac_expansion(float_num):
    inv_frac = (1 / float_num)
    return inv_frac - math.floor(inv_frac) # Performs a fraction expansion


def check_period_validity(int_num, mod_num, mult_coeff):
    if int_num % 2 != 0 or (mult_coeff ** (int_num)) + 1 % mod_num == 0:
        raise InvalidPeriod # Checks if the found period is valid

```

```python

def reduce_period(int_num, mod_num, mult_coeff):
    period = int_num
    while (True):
        new_period = int(period / 2)
        if get_mod(mult_coeff ** (new_period), mod_num) is 1:
            period = new_period
        else:
            break
    return period # For an even period it reduces the period to the minumum possible value


def get_period(float_num, mod_num, mult_coeff):
    frac = float_num
    period = calc_period(frac, mod_num, mult_coeff)
    if period is not 0:
        check_period_validity(period, mod_num, mult_coeff)
        return reduce_period(period, mod_num, mult_coeff)
    runs = 0
    while (period == 0 and runs < 10):
        frac = frac_expansion(frac)
        period = calc_period(frac, mod_num, mult_coeff)
        runs += 1
    if period is not 0:
        check_period_validity(period, mod_num, mult_coeff)
        reduce_period(period, mod_num, mult_coeff)
    return period # Calculates the period by the method of fraction expansion
```