Quantum Computing (CSE3080)

MINI PROJECT

GROUP MEMBERS

BHAVISH R	(20201CAI0219)
SUJAY SASIKUMAR	(20201CAI0205)
HARSHITH S	(20201CAI0190)
AMAL PRAKASH	(20201CAI0204)

Shor's algorithm

Shor's algorithm is a quantum computer algorithm for finding the prime factors of an integer. It was developed in 1994 by the American mathematician Peter Shor.Although any integer number has a unique decomposition into a product of primes, finding the prime factors is believed to be a hard problem. In fact, the security of our online transactions rests on the assumption that factoring integers with a thousand or more digits is practically impossible. This assumption was challenged in 1995 when Peter Shor proposed a polynomial-time quantum algorithm for the factoring problem. Suppose we are given co-prime integers . Our goal is to compute the period of modulo , that is, the smallest positive integer such that . The basic idea is to construct a unitary operation that implements the modular multiplication function . It can be shown that eigenvalues of are closely related to the period of . Namely, each eigenvalue of has a form , where for some integer .

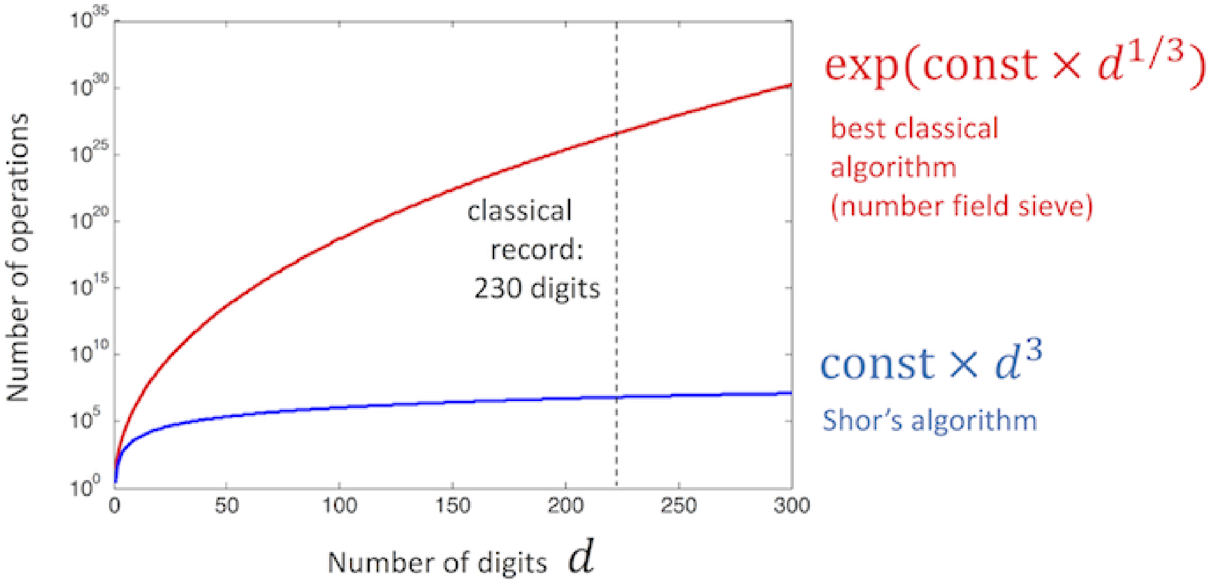
Furthermore, as we saw in the previous section on quantum phase estimation, eigenvalues of certain unitary operations can be measured efficiently using the phase estimation algorithm. Unfortunately, inferring from the measured eigenvalues of is only possible if the eigenvalues are measured *exactly* (or with an exponentially small precision). For example, factoring a 1000-digit number would require measuring the eigenvalue of with a precision of . Such accuracy cannot be achieved by a direct application of the phase estimation algorithm, as this would require too large a pointer system. Here comes the main trick: we will estimate the eigenvalue of by applying the phase estimation algorithm to a family of unitary operations with etc. We stop at with .

Why does it work? The first observation is that all operations are integer powers of . Namely, if , then . This implies that the operations have the same eigenvectors. In particular, eigenvalues of the entire family can be measured simultaneously. Second, implementing is as easy as implementing - one just need to precompute the powers by the repeated squaring method. Finally, even if the eigenvalues of are measured with a poor precision (say 10%), each squaring of reduces the error in the estimated eigenvalue of by a factor of . Indeed, consider an eigenvector of with an eigenvalue . If , then the eigenvalue of is . If , then the eigenvalue of is , etc. Thus we can estimate with a constant precision (say 10%). We will see that this is enough to estimate with a precision roughly . For example, one can achieve a precision of by a sequence of fewer than measurements of with an error of 10%. Furthermore, it can be shown that estimating a few randomly picked eigenvalues with a precision less than is enough to determine the period exactly (the idea is to find the best rational approximation to the estimate of using continued fractions).

Complexity

Suppose our task is to factor an integer with decimal digits. The brute force algorithm goes through all primes up to and checks whether divides . In the worst case, this would take time roughly , which is exponential in the number of digits . A more efficient algorithm, known as the quadratic sieve, attempts to construct integers such that is a multiple of . Once such are found, one checks whether have common factors with . The quadratic sieve method has asymptotic runtime exponential in . The most efficient classical factoring algorithm, known as general number field sieve, achieves an asymptotic runtime exponential in .

The exponential runtime scaling limits applicability of the classical factoring algorithms to numbers with a few hundred digits; the world record is (which took roughly 2,000 CPU years!). In contrast, Shor's factoring algorithm has runtime *polynomial* in . The version of the algorithm described below, due to Alexey Kitaev, requires roughly qubits, and has runtime roughly .



Premise

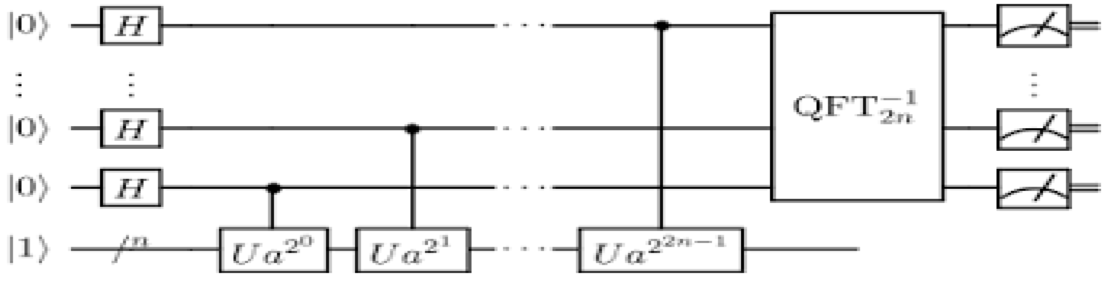
- **Measurement:** the state of a q-qubit quantum register is described by a vector of dimension 2^q, exponentially larger than the dimension of the vector required to describe q classical bits. However, there is a catch: in a classical computer we can simply read the state of the bits, whereas in a quantum computer we do not have direct, unrestricted access to the quantum state. Information on the quantum state are obtained by applying a measurement gate, after which the original quantum state is no longer available, since it collapses to a linear combination of only those basis states that are consistent with the outcome of the measurement.
- **No cloning theorem**: since measurement destroys the quantum state, it is natural to look for a way to create a copy of a quantum state, so that we can apply a measurement gate on it, leaving the original state untouched. Unfortunately, cloning is impossible. This is due to the fact that gates are unitary matrix .

Shor's algorithm consists of two parts:

1) [CLASSICAL PART] A reduction, which can be done on a classical computer, of the factoring problem to the problem of order-finding.

2) [QUANTUM PART] A quantum algorithm to solve the order-finding problem.

Structure of the algorithm



- **Algorithm:** It contains a few steps, at only step 2 the use of quantum computers is required.

    1. Choose any random number let say **r**, such that **r < N** so that they are co-primes of each other.
    2. A quantum computer is used to determine the unknown period **p** of the function **$f_{r,\ N}$ (x) = r_x** mod **N**.
    3. If **p** is an odd integer, then go back to **Step 1**. Else move to the next step.
    4. Since **p** is an even integer so, **($r_{p/2}$ – 1)($r_{p/2}$ + 1) = r_p – 1 = 0 mod N**.
    5. Now, if the value of **$r_{p/2}$ + 1 = 0 mod N**, go back to **Step 1**.
    6. If the value of **$r_{p/2}$ + 1 != 0 mod N**, Else move to the next step.
    7. Compute **d = gcd($r_{p/2}$-1, N)**.
    8. The answer required is '**d**'.

- **Classical part (The order finding problem):**

    This is the classical part of order finding problem. Given that **x** and **N**, such that **x<N** and **gcd(x, N) = 1**. The order of **x** is the least positive integer, y such that **x_y = 1(mod N)**.

    1. A random number **n** is picked, such that **n < N**. Compute **gcd(n, N)**.
    2. This can be done using the Euclid Algorithm.
    3. If **gcd(n, N) != 1**, then there is a non-trivial factor of N.If **(x+p) = $n_{x\ +\ p}$ mod N = $n_x$mod N = f(x)**.
    4. If r is odd, then go back to **Step 1**.
    5. If **$n_{p/2}$ = -1(mod N)**, then go back to **Step 1**.

6. The **gcd($n_{p/2}$ +/- 1, N)** is a non-trivial factor of **N**.

Quantum part:

This is the quantum order finding part. Choose a power of 2, then

Q = 2L such that N_2 <= Q <= 2N_2

And consider f = {0, 1, 2, ..., Q-1}

Where, f(y)=1/$(Q)_{1/2}$ $\sum_{x=0Q-1}$l f(x) I w_{xy} and w = exp(2π i/Q), i.e. Q_{th} root of unity.

Let’s perform Shor’s Algorithm using an example: To factor an odd integer N (let N = 17).

Choose an integer Q such that N_2 <= Q ≤ 2 N_2 (lets Q = 324).

Then, randomly choose any integer n such that gcd(n, N)=1, (let us choose the integer be n=7).

Then create two quantum registers (these registers should be entangled so that the collapse of the input registered corresponds to the collapse of the output register)

Input Register: must be containing enough qubits to represent numbers as large as Q-1.(i.e., up to 323, so we need 9 qubits).

Output Register: must be containing enough qubits to represent numbers as large as (N – 1). (i.e., up to 16, so we require 4 qubits).

Code

```
from qiskit import IBMQ

from qiskit.aqua import QuantumInstance

from qiskit.aqua.algorithms import Shor

# Enter your API token here

IBMQ.enable_account('ENTER API TOKEN HERE')

provider = IBMQ.get_provider(hub='ibm-q')

# Specifies the quantum device

backend = provider.get_backend('ibmq_qasm_simulator')

print('\n Shors Algorithm')
print('--------------------')

print('\nExecuting...\n')

# Function to run Shor's algorithm

# where 35 is the integer to be factored

factors = Shor(35)

result_dict = factors.run(QuantumInstance(

    backend, shots=1, skip_qobj_validation=False))



 # Get factors from results

result = result_dict['factors']



print(result)

print('\nPress any key to close')
```

```
    input()
```

OUTPUT

```
Shors Algorithm

- - - - - - - - - - - - - -

Executing...

[[5,7]]

Press any key to close
```

Output for the code above showing the factors 5 and 7 for N=35.

Time Complexity : O(1)

Space Complexity : O(1)

Eigenvalue & Eigenvectors

The eigenvalue and the eigenvector of a matrix $\boldsymbol{A}$ satisfy the following relationship.

$$A\mathbf{v} = \lambda\mathbf{v}$$

$$\underbrace{\begin{bmatrix}1 & -3 & 3\\ 3 & -5 & 3\\ 6 & -6 & 4\end{bmatrix}}_{\text{A}}\begin{bmatrix}1/2\\ 1/2\\ 1\end{bmatrix} = \overset{\text{eigenvalue}}{4}\underbrace{\begin{bmatrix}1/2\\ 1/2\\ 1\end{bmatrix}}_{\text{eigenvector}}$$

Phase estimation is about finding the eigenvalue of a unitary operator. Since a unitary operation preserves the norm of the state vector to 1, the corresponding eigenvalue is just an exponential complex function with real value ϕ below. ϕ is between 0 and 1 (we will use that later). Phase estimation simply means estimate the value of ϕ in the eigenvalue.

$$U\overset{\text{eigenvector}}{|\psi\rangle} = \underbrace{e^{2\pi i\phi}}_{\text{complex eigenvalue}}|\psi\rangle\,, 0 \leq \phi < 1$$

In Shor's algorithm, we want to find the period of some modulo function.

Find the period r of this function

$$\underline{\frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}|j\rangle|x^j \mod N\rangle}$$

e.g. $\frac{1}{\sqrt{2^t}}[|0\rangle|1\rangle + |1\rangle|7\rangle + |2\rangle|4\rangle + |3\rangle|13\rangle + |4\rangle|1\rangle + |5\rangle|7\rangle + |6\rangle|4\rangle + \cdots]$

But to do it efficiently, we establish a relationship between the period of a function and the phase value of the eigenvalue. So solving the phase helps us to find the period.

Approximate as finding r in phase estimation

$$\frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}|j\rangle|x^j \mod N\rangle \rightarrow \frac{1}{\sqrt{r2^t}}\sum_{s=0}^{r-1}\sum_{j=0}^{2^t-1}e^{2\pi isj/\underline{r}}|j\rangle|u_s\rangle$$

eigenvector for U where $U|y\rangle = |xy \mod N\rangle$

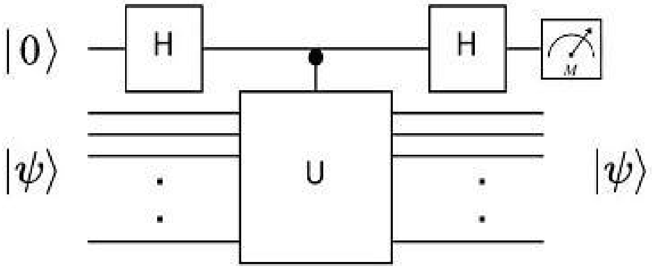
It will take a while to establish this relationship. So, we will focus on phase estimation for now.

Phase estimation

If we know U and $|\psi\rangle$ below, we can estimate ϕ.

$$\overset{\text{Known}}{U|\psi\rangle} = e^{2\pi i\overset{\text{Estimate}}{\phi}}\,|\psi\rangle$$

One way to do it is to use a controlled-U gate below with the eigenvector $|\psi\rangle$as the input to the target bits.



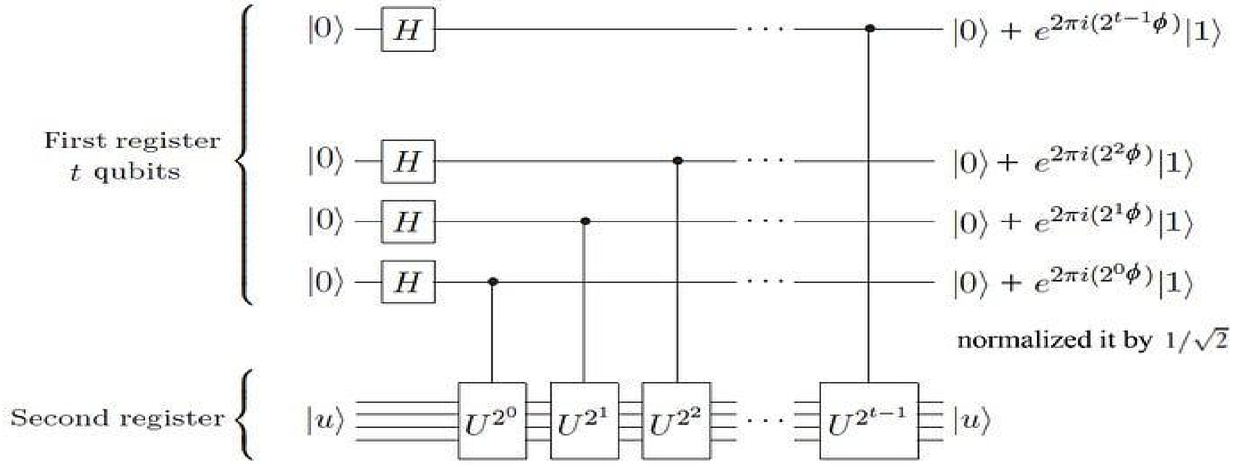
The expected measurement for the control bit above will be:

$$\langle\hat{Z}\rangle = \langle\psi|(\hat{U} + \hat{U}^\dagger)|\psi\rangle/2 = \frac{1}{2}\left(e^{2\pi i\phi} + e^{-2\pi i\phi}\right) = \cos(2\pi\phi).$$

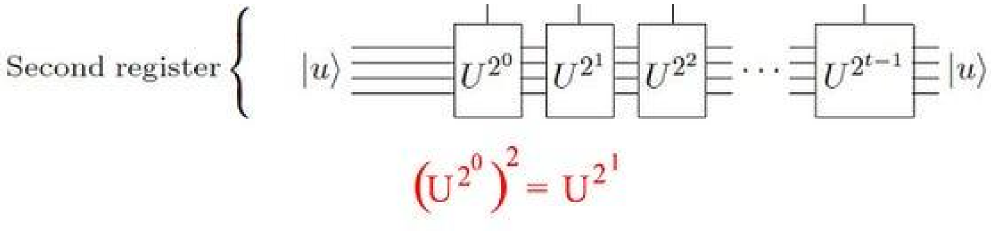
Therefore, by repeating the calculations many times, we can use the measurements to estimate ϕ. However, to compute this value with an accuracy up to m bits, the number of re-calculations is in the order of

$$O(2^{2m})$$

This is bad because we bring back the exponential complexity again. To reduce the number of measurements, the circuit is redesigned to:


First register
t qubits
Second register
$|0\rangle$
H
$|u\rangle$
U^{2^0}
U^{2^1}
U^{2^2}
$U^{2^{t-1}}$
$|0\rangle + e^{2\pi i(2^{t-1}\phi)}|1\rangle$
$|0\rangle + e^{2\pi i(2^2\phi)}|1\rangle$
$|0\rangle + e^{2\pi i(2^1\phi)}|1\rangle$
$|0\rangle + e^{2\pi i(2^0\phi)}|1\rangle$
normalized it by $1/\sqrt{2}$

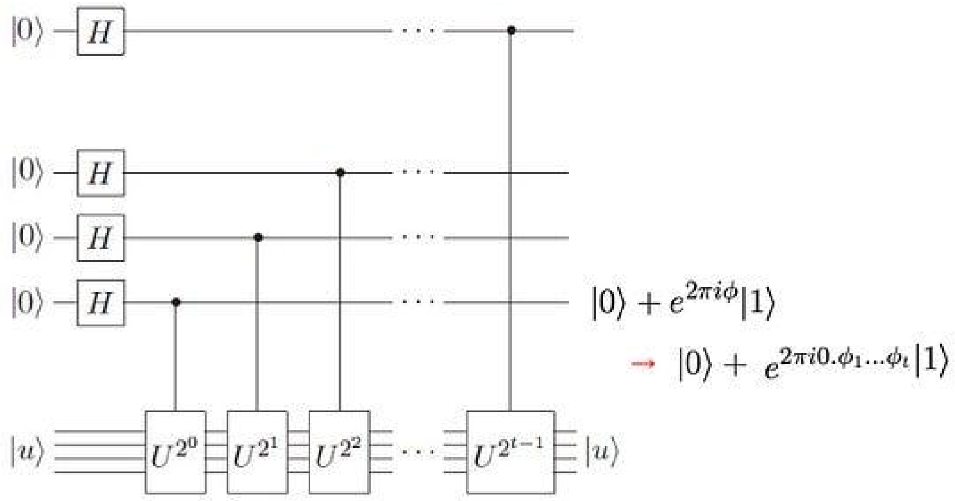
For each step, the operator below is the square of the previous one. This sounds complicated and turns out to be simple for modulo arithmetics. But we will wait until the next article to discuss the implementation.



We use the eigenvector as the input to the target bits (note, we change our previous notation from |ψ⟩ to |u⟩ here). As mentioned before, ϕ is between 0 to 1. Therefore, we can represent it with:

$$\phi = 0.\phi_1\phi_2\ldots\phi_t$$

(e.g. ϕ = 0.10111...) We can rewrite the lowest bit for the control as:

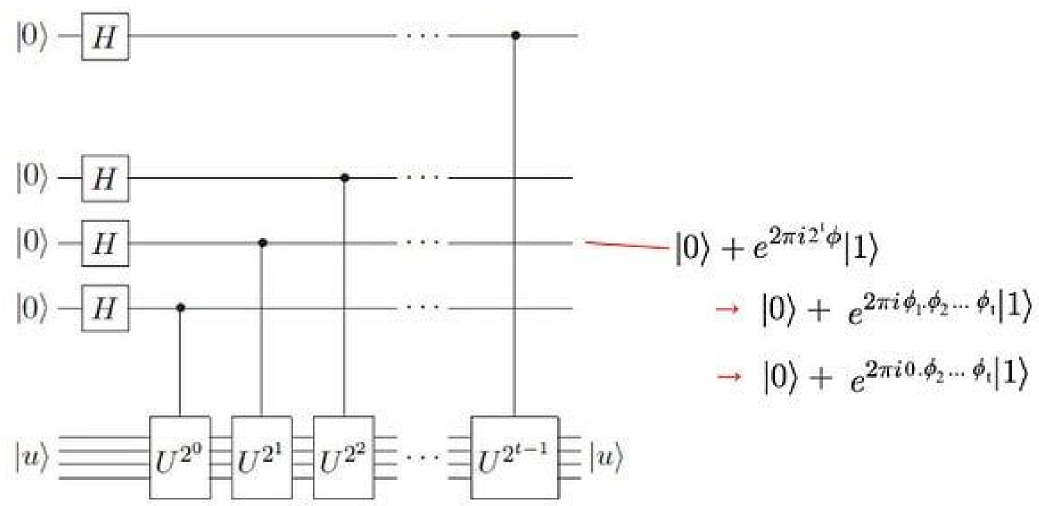


$$2^1\phi = \phi_1.\phi_2 \ldots \phi_t$$

(In our example, this becomes 1.0111...) and we apply

$$e^{2\pi i\phi_1.\phi_2 \ldots \phi_t} = e^{2\pi i\phi_1 + 2\pi i 0.\phi_2 \ldots \phi_t} = e^{2\pi i 0.\phi_2 \ldots \phi_t}$$

Therefore, the second lowest bit can be simplified to:



$|0\rangle$
H
$|0\rangle$
H
$|0\rangle$
H
$|0\rangle$
H
$|u\rangle$
U^{2^0}
U^{2^1}
U^{2^2}
$U^{2^{t-1}}$
$|u\rangle$
$|0\rangle + e^{2\pi i 2^t \phi}|1\rangle$
$\rightarrow |0\rangle + e^{2\pi i \phi_1 . \phi_2 \dots \phi_t}|1\rangle$
$\rightarrow |0\rangle + e^{2\pi i 0 . \phi_2 \dots \phi_t}|1\rangle$

i.e. we just left shift ϕ and remove the integer part since we can ignore any integral multiplication with 2π. So the circuit above just prepare the following supposition:

$$\frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} e^{2\pi i \phi j} |j\rangle$$

Recall previously that, Quantum Fourier Transform can be implemented as:

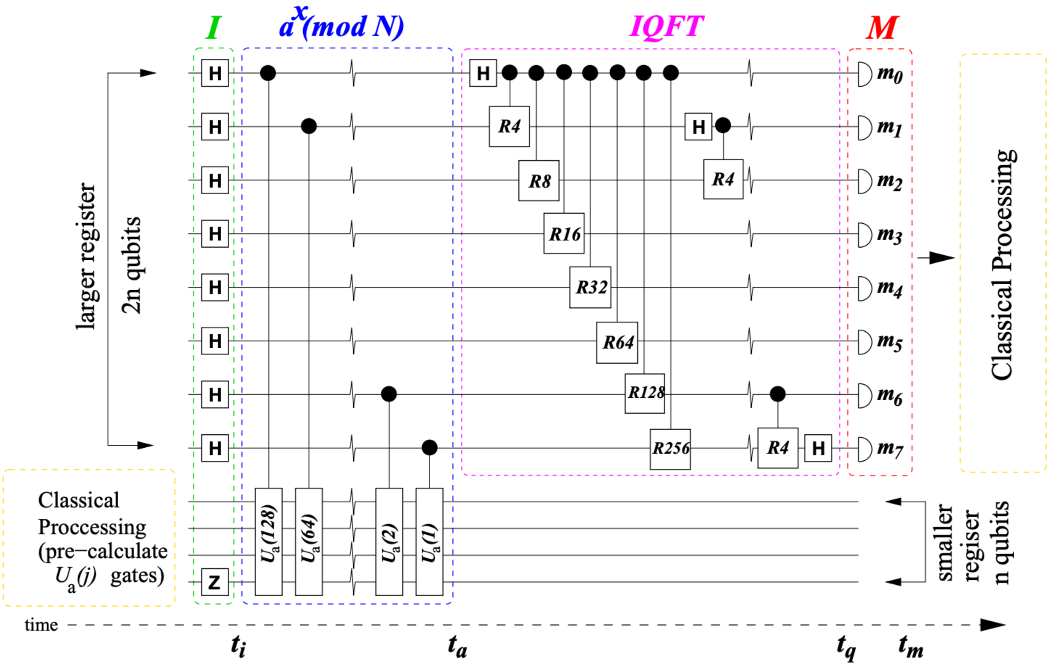
$$\boxed{|j_1\cdots j_n\rangle \rightarrow \frac{\left(|0\rangle + e^{2\pi i 0.j_n}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle\right)\cdots\left(|0\rangle + e^{2\pi i 0.j_1 j_2\cdots j_n}|1\rangle\right)}{2^{n/2}}}$$

So, our prepared superposition is just the inverse QFT of the phase.

$$\frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1} e^{2\pi i\phi j}|j\rangle \xrightarrow{\mathrm{QFT}^{-1}} |2^t\phi\rangle$$

E.g. $|10111\rangle$

So by applying the inverse of the QFT on the prepared superposition, we can find the phase value. As shown below, the left side circuit prepares the superposition and the right side apply the inverse of QFT.



To achieve accuracy to *m* bits with the probability at least (1 – ε), we need *t* qubits for the controls:

$$t = m + \log\left(1 + \frac{1}{2\epsilon}\right)$$

We will use this information later to find out the number of qubits we need to estimate the phase. For $|u\rangle$, we need enough bits to represent the eigenvector. Since the function *mod N* has a maximum value of *N-1*, we just need enough bits to represent *N-1*.

Issues with Shor’s Algorithm

Shor’s Algorithm is not the best method on classical computers. The algorithms like Quadratic Sieve, Lenstra Elliptic Curve Factorization and General Number Field Sieve perform better for this problem on a traditional machine.

Despite Shor's Algorithm's incredibly fast performance, the largest number factored by a Quantum Computer using Shor's Algorithm is 21.

Applications of Shor’s Algorithm with a quantum computer

Shor's algorithm can do prime factoring of very large numbers on a quantum computer.

Shor’s algorithm can potentially be used to hack RSA and other secure data forms

Shor’s algorithm can solve the problem of finding the period of a function.

BLOG LINK:

https://amalshoralgorithm.blogspot.com/2023/05/QUANTUMcomputingamalpro.html.