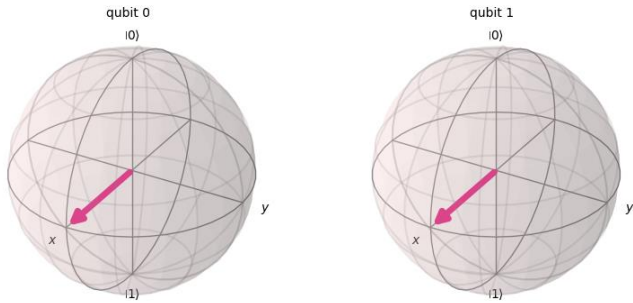
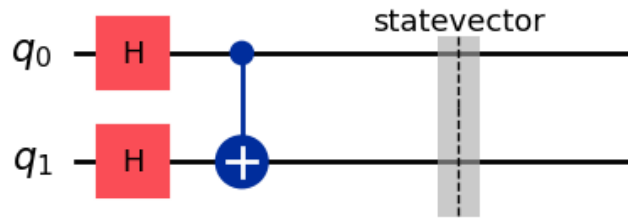

ECE 550/650 QC

Shor's Algorithm

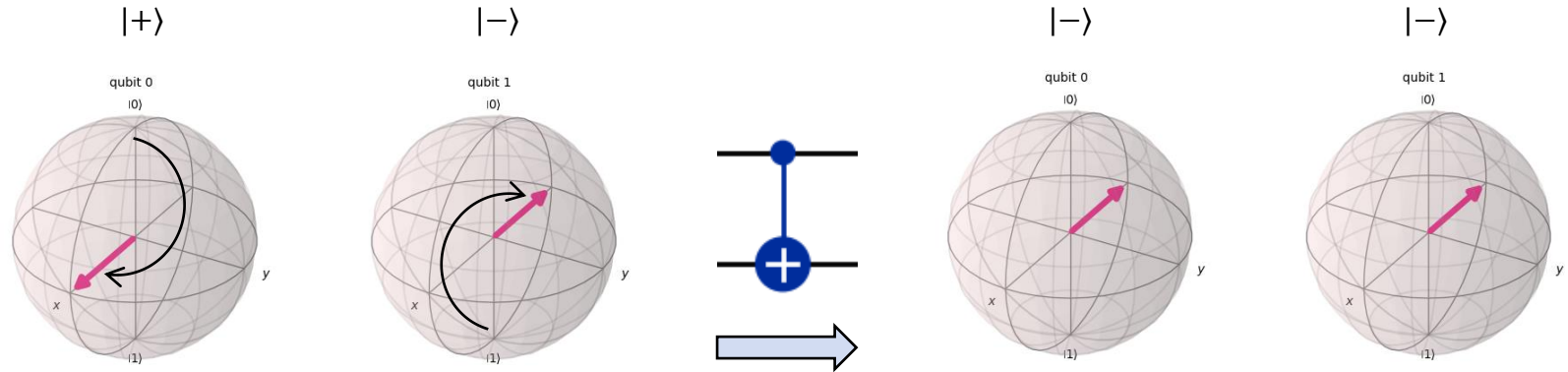
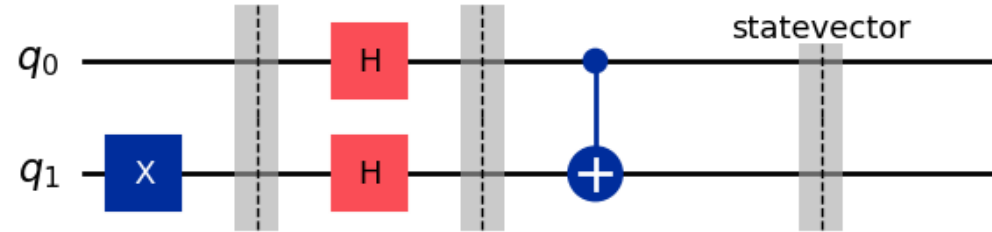
Robert Niffenegger



Review - Phase Kickback



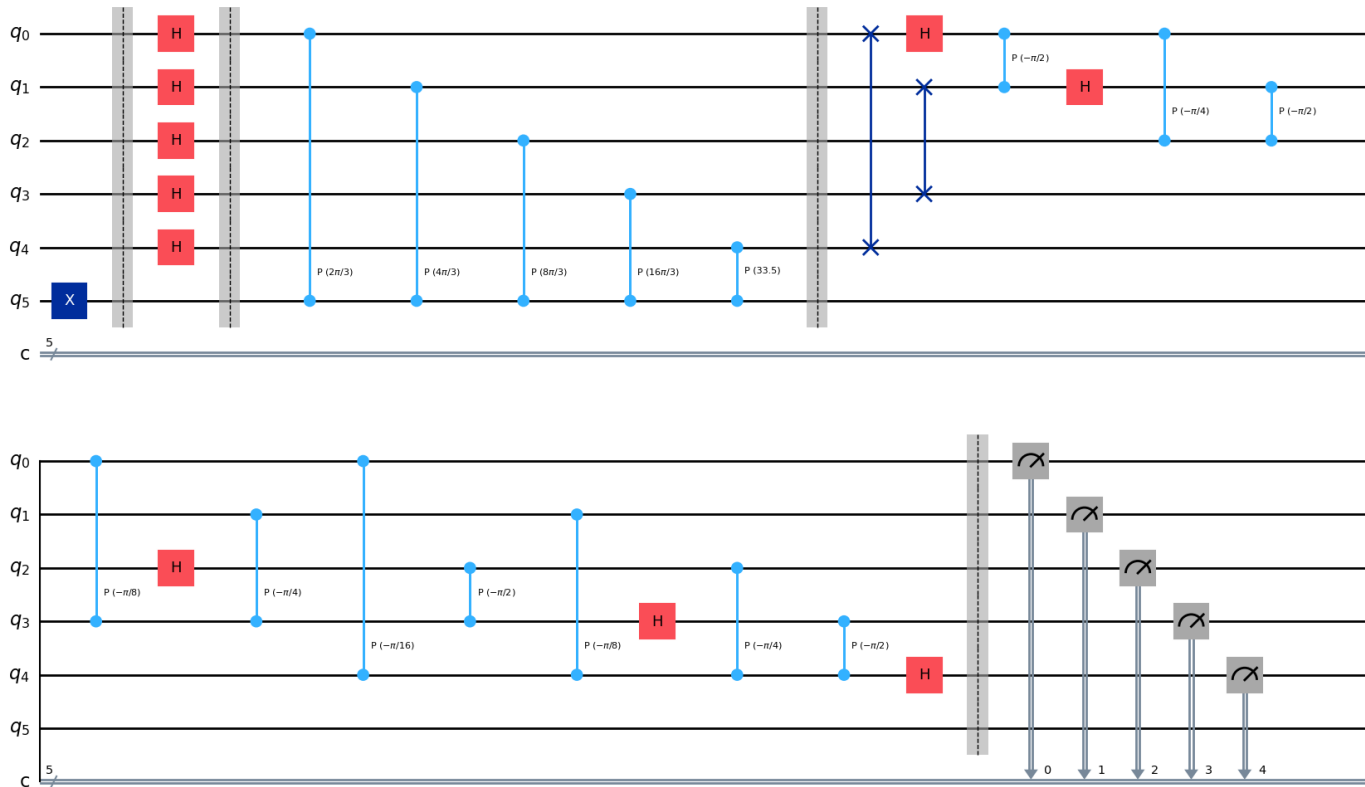
$$\frac{1}{2} [|00\rangle + |01\rangle + |10\rangle + |11\rangle]$$



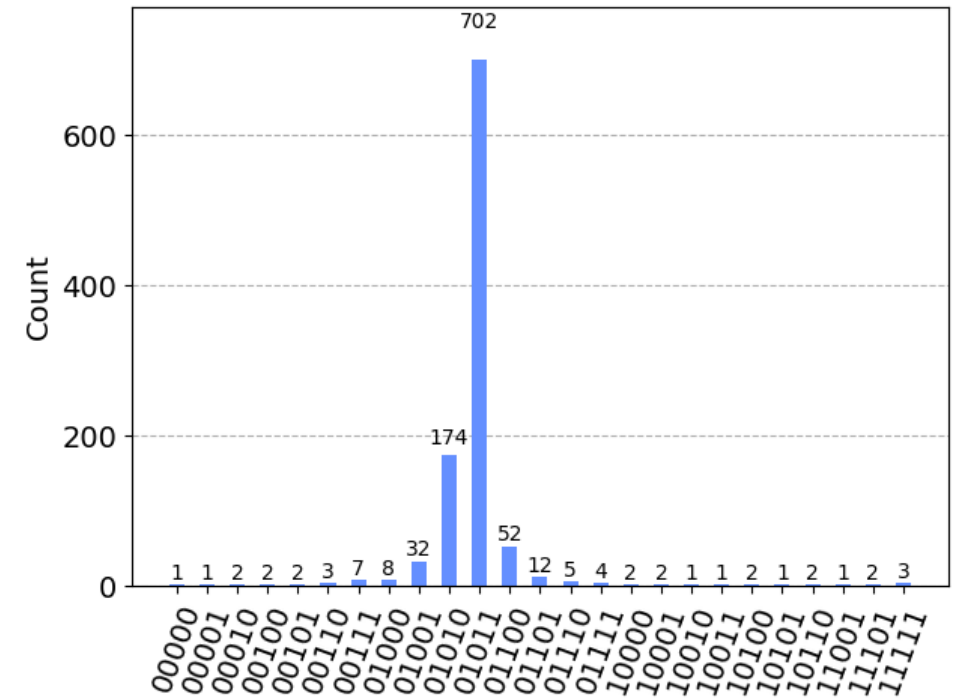
$$\begin{aligned} & \text{CNOT} | - + \rangle \\ &= \text{CNOT} \frac{1}{2} [|00\rangle + |01\rangle - |10\rangle - |11\rangle] \end{aligned}$$

$$\begin{aligned} &= \frac{1}{2} [|00\rangle - |0\mathbf{1}\rangle - |10\rangle + |1\mathbf{1}\rangle] \\ &= \frac{1}{\sqrt{2}} [|0\rangle - |1\rangle] \otimes \frac{1}{\sqrt{2}} [|0\rangle - |1\rangle] \\ &= | - \rangle \otimes | - \rangle \\ &= | - - \rangle \end{aligned}$$

Review – Quantum Phase Estimation



$$01011 = 11, 11 \cdot \frac{1}{2^5} = \frac{11}{32} = 0.34375$$



RSA Encryption (review)

RSA encryption is based on the difficulty of factoring large numbers that are the product of prime numbers.

The mathematical root is based on the observation that with integers e , d , and n :

- $(m^e)^d = m \pmod{n}$

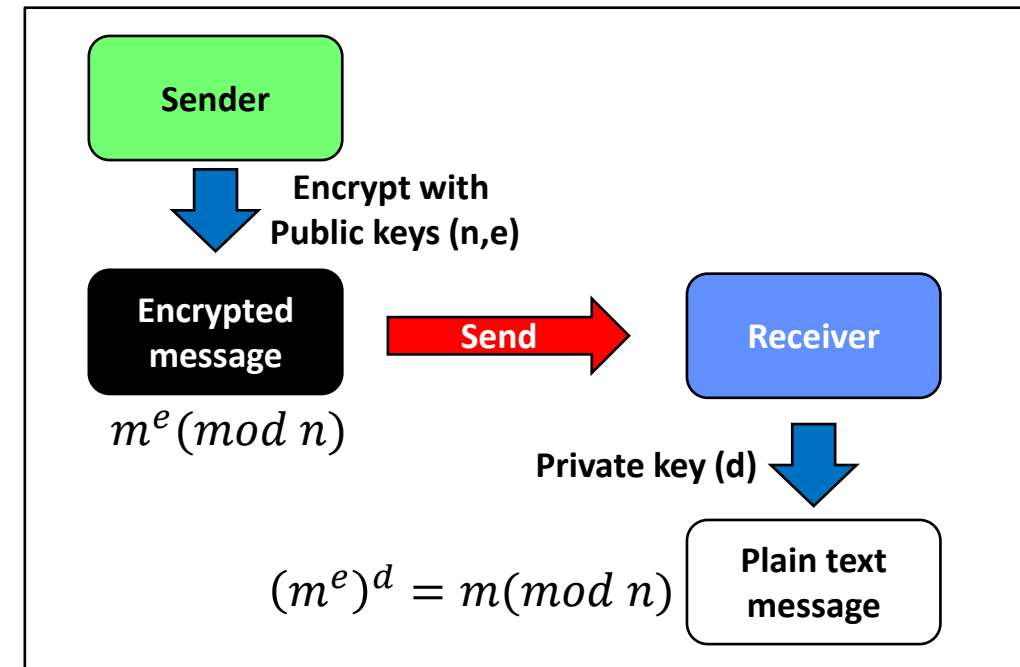
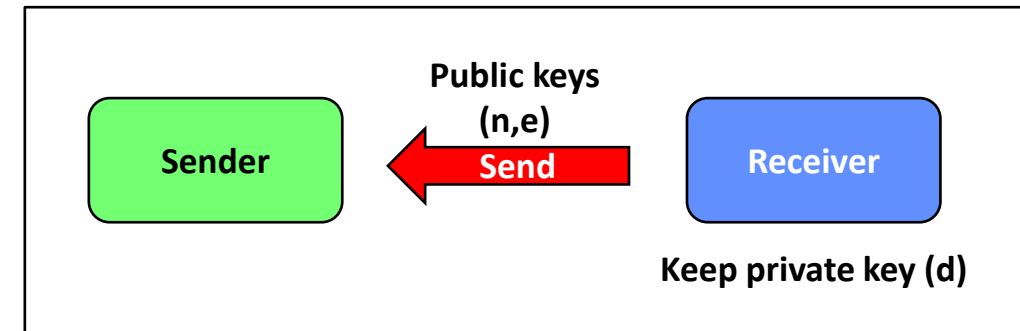
That is, modular exponentiation for all integers m to the power ' e ' and ' d ' is equal to $m \pmod{n}$.

- https://en.wikipedia.org/wiki/Modular_exponentiation

Therefore, if we want to encrypt ' m ' we exponentiate by a number ' e '.

- e can be public!
- Then upon exponentiation by ' d ' we recover m ! [\pmod{n} of course]

How is this possible? And how to get ' e ' and ' d '?



RSA decryption - time

Conversely:

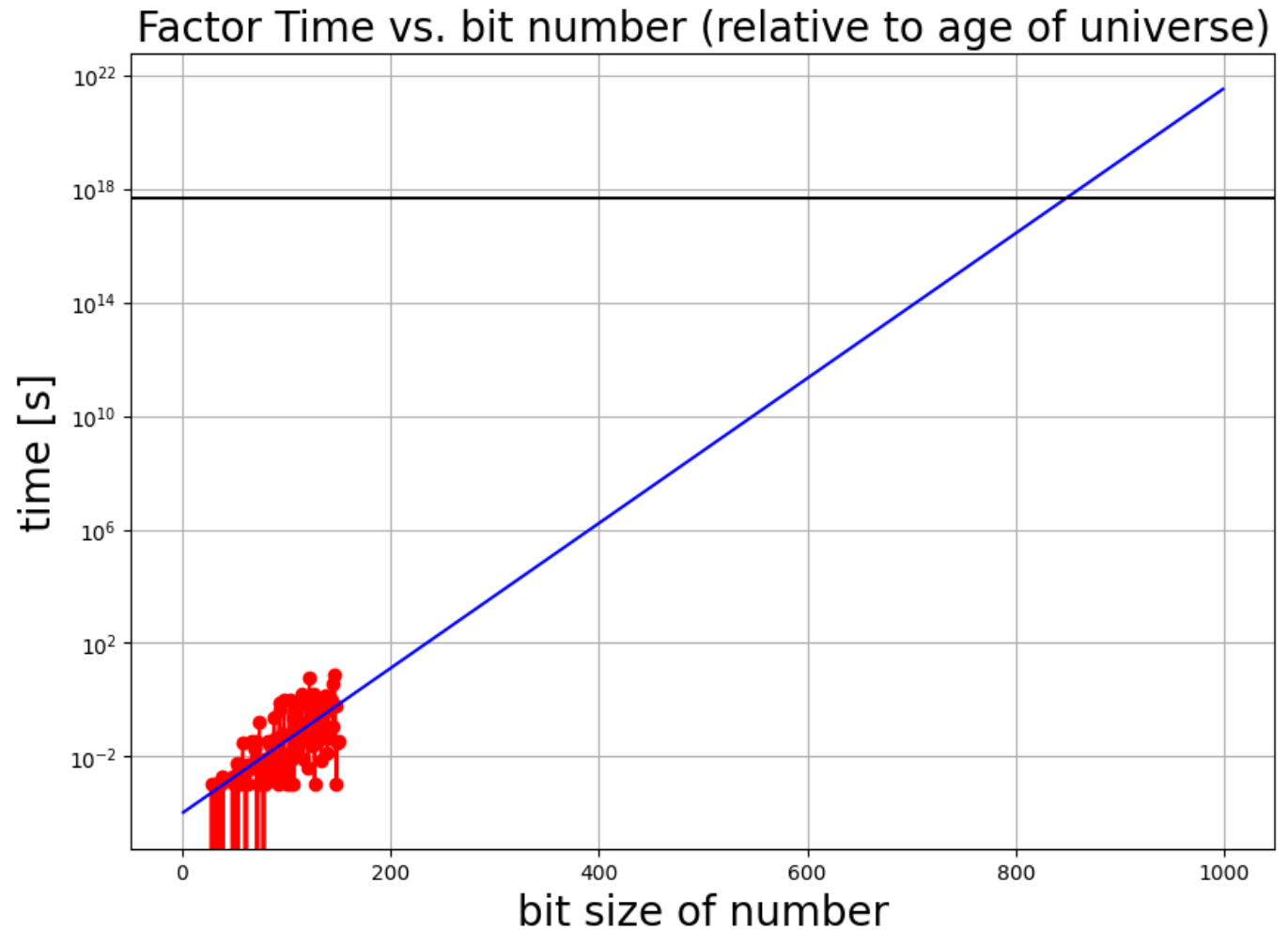
If we know 'p' and 'q' we can calculate 'd'

Goal for cracking will be to factor n

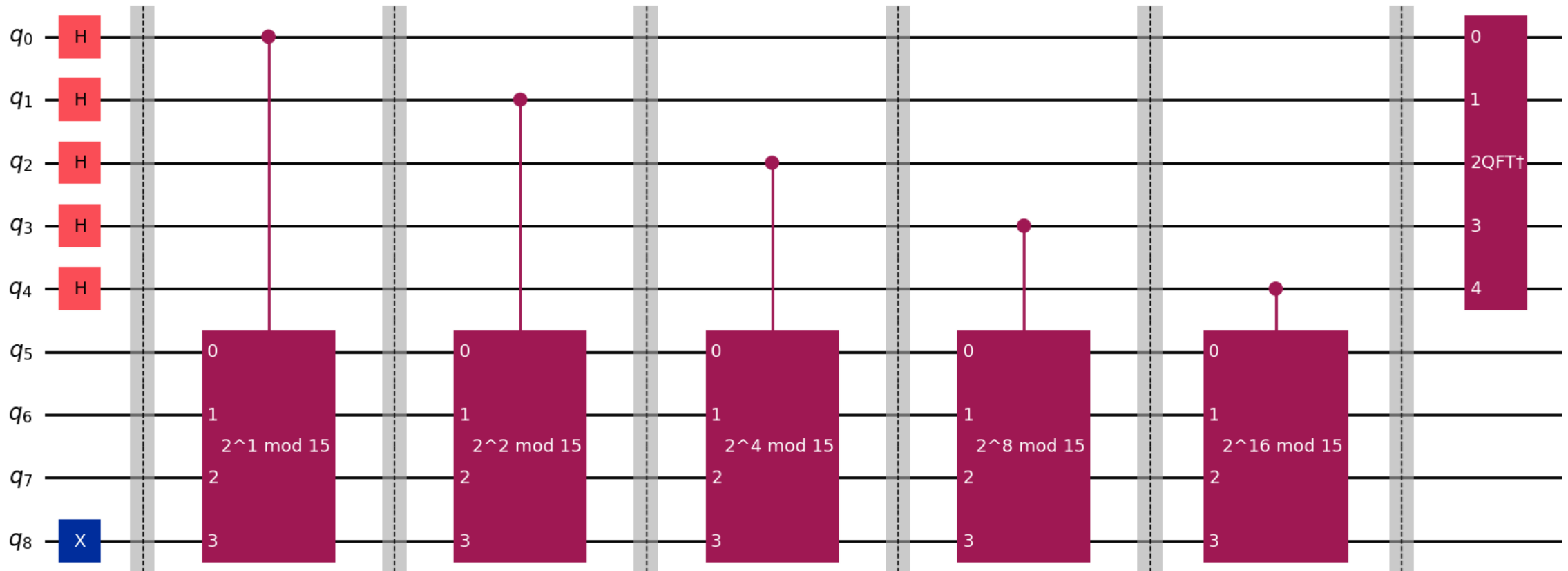
$$n = p \cdot q$$

to get 'p' and 'q'

to then calculate 'd'



Shor's Algorithm



Turning a Factoring Problem into Period Finding

Shor's algorithm is famous for factoring integers 'N' in polynomial time.

Since the best-known classical algorithm requires super polynomial time to factor the product of two primes, the widely used cryptosystem, RSA, relies on factoring being impossible for large enough integers.

Shor's algorithm factors N using period finding.

Since a factoring problem can be turned into a period finding problem in polynomial time, an efficient period finding algorithm can be used to factor integers efficiently too.

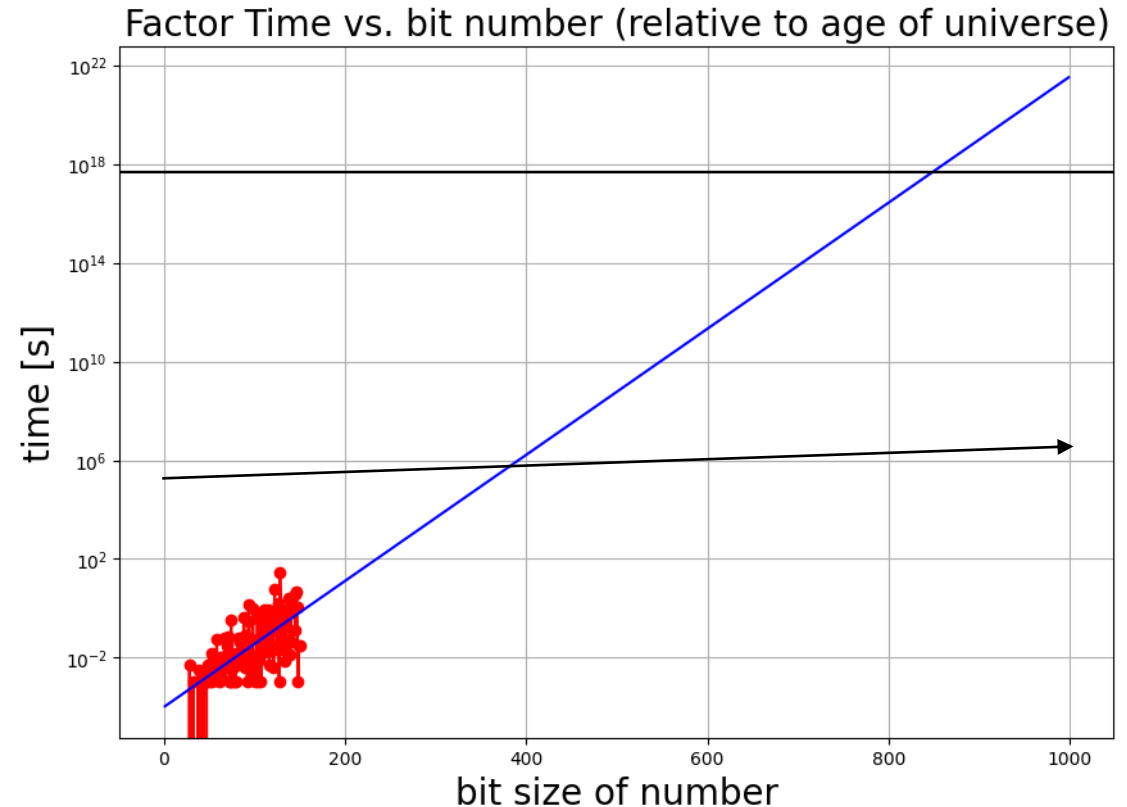
So efficiently computing the period of:

$$U^x = a^x \bmod N$$

allows us to efficiently factor N.

'a' can be any number less than N that has no common factors with N

We will start with $a = 2$



Period Finding

We will use the first non-trivial primes, 3 and 5 for this demo.

$p = 3$

$q = 5$

$N = p \cdot q$

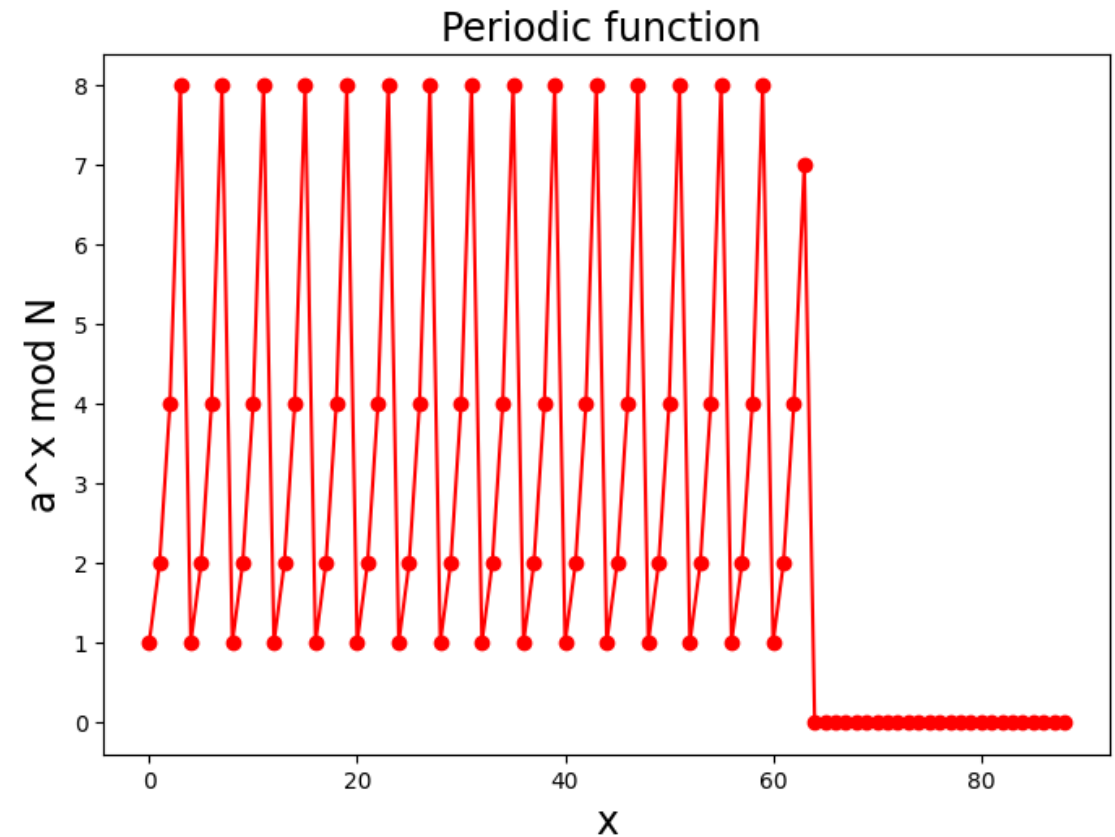
a is then any number $< N$ that does not share a factor with N

We'll start with $a = 2$

For small values it is very easy to see the period of the function

$$a^x \pmod{N}$$

just by plotting it out:



Period Finding – Modular Exponentiation

However, something is wrong. The function just fails after some point...

Turns out a^x is a big number for $a = 2$ and $x > 60$ and Python has overflow errors unless we start specifying more precision in our initialization of the variables.

Instead of taking the modulo at the end we can take it at intermediate steps using the identity:

$$(a \cdot b) \bmod m = [(a \bmod m) \cdot (b \bmod m)] \bmod m$$

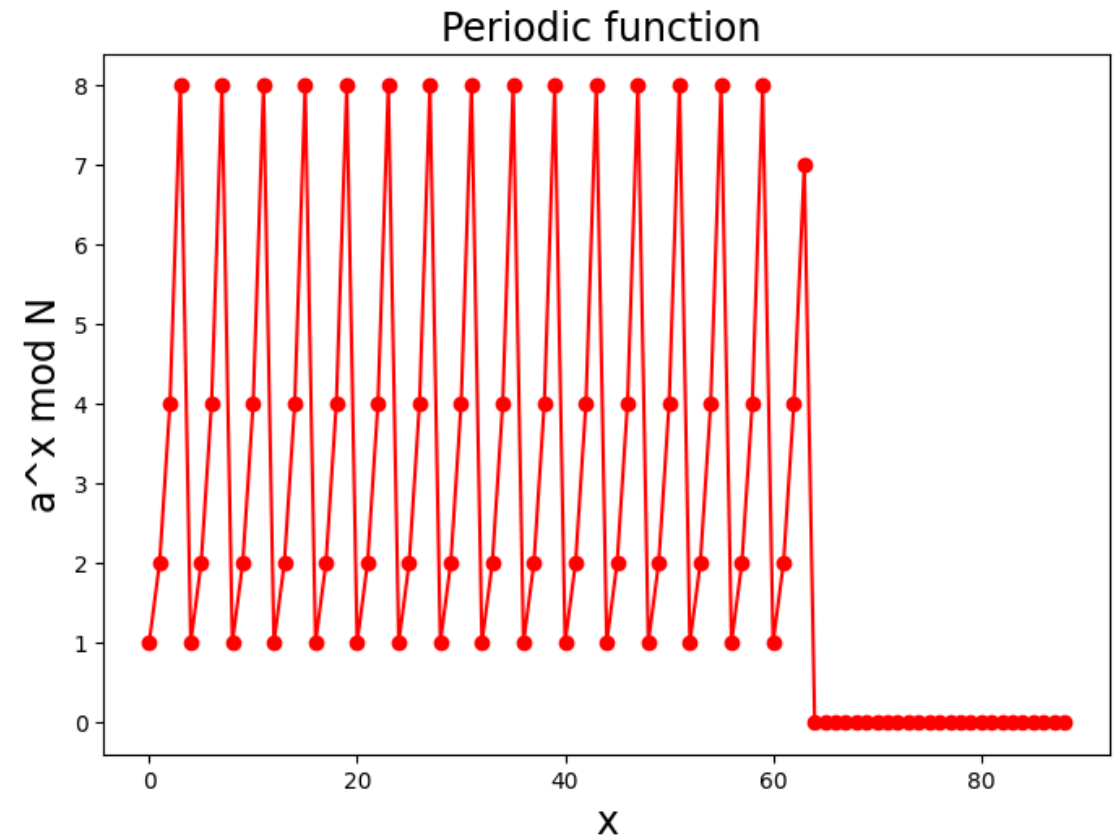
This is Modular Exponentiation:

So, we will use the `pow(a, x, N)` function to perform Modular Exponentiation.

It can handle REALLY big numbers

```
'''
x = pow(a, d, n)
'''

d = 2**5
print(d)
x = pow(91, d, 131)
print(x)
```



Period Finding – Modular Exponentiation + Repeated Squares

Repeated Squaring

Another trick the 'pow' function uses is repeated squaring.

For instance, we can compute:

$$a^{2^j} \pmod{N}$$

by repeatedly squaring.

For example:

$a^{2^{2000}}$ is a very very very big number.

instead of calculating

$a^{114813069527425452423283320117768198402231770208869520047764273682576626139237031385665948631650626}$

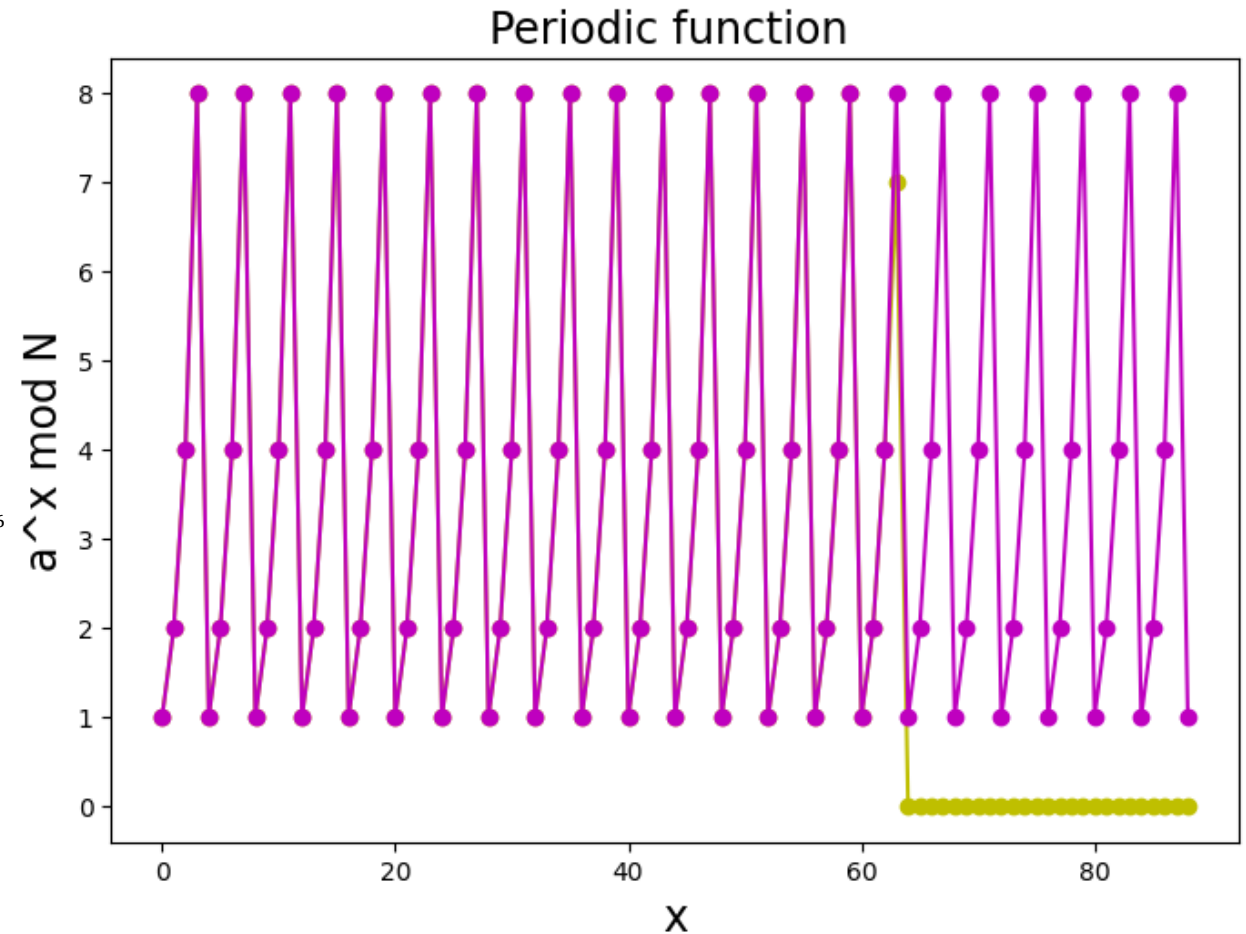
We can repeatedly square 'a', 2000 times, taking the modulo of N each iteration.

So instead of 2^{2000} operations we only have 2000 (AMAZING!)

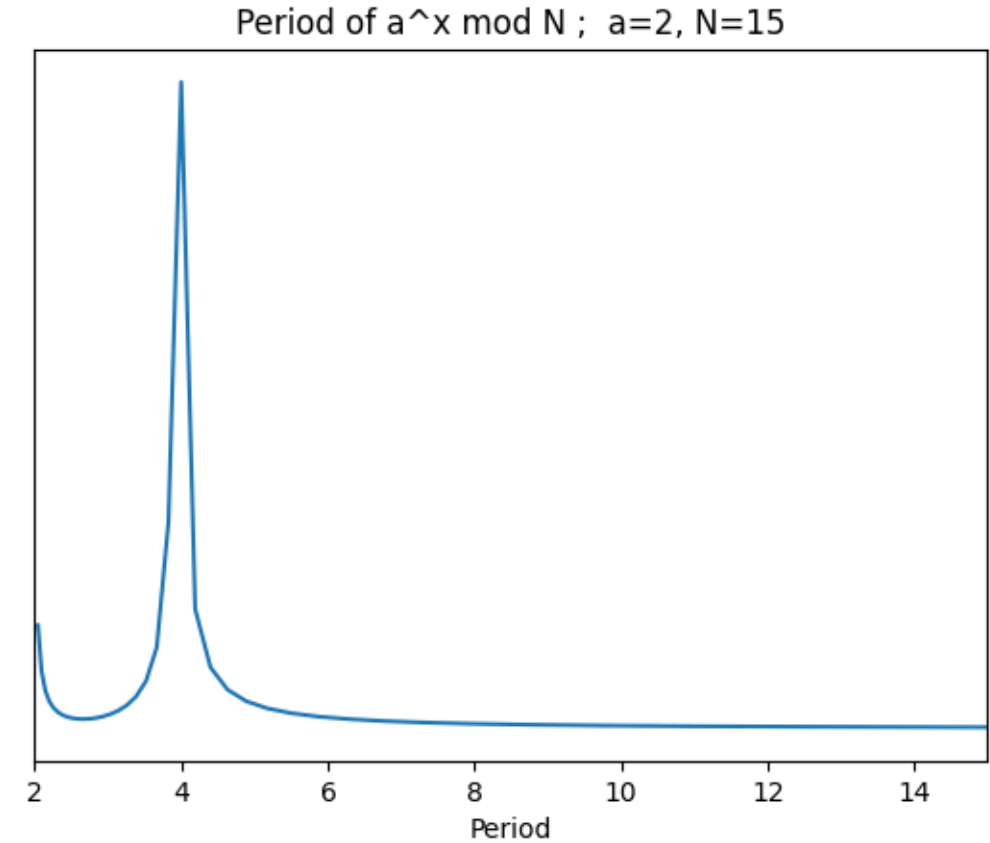
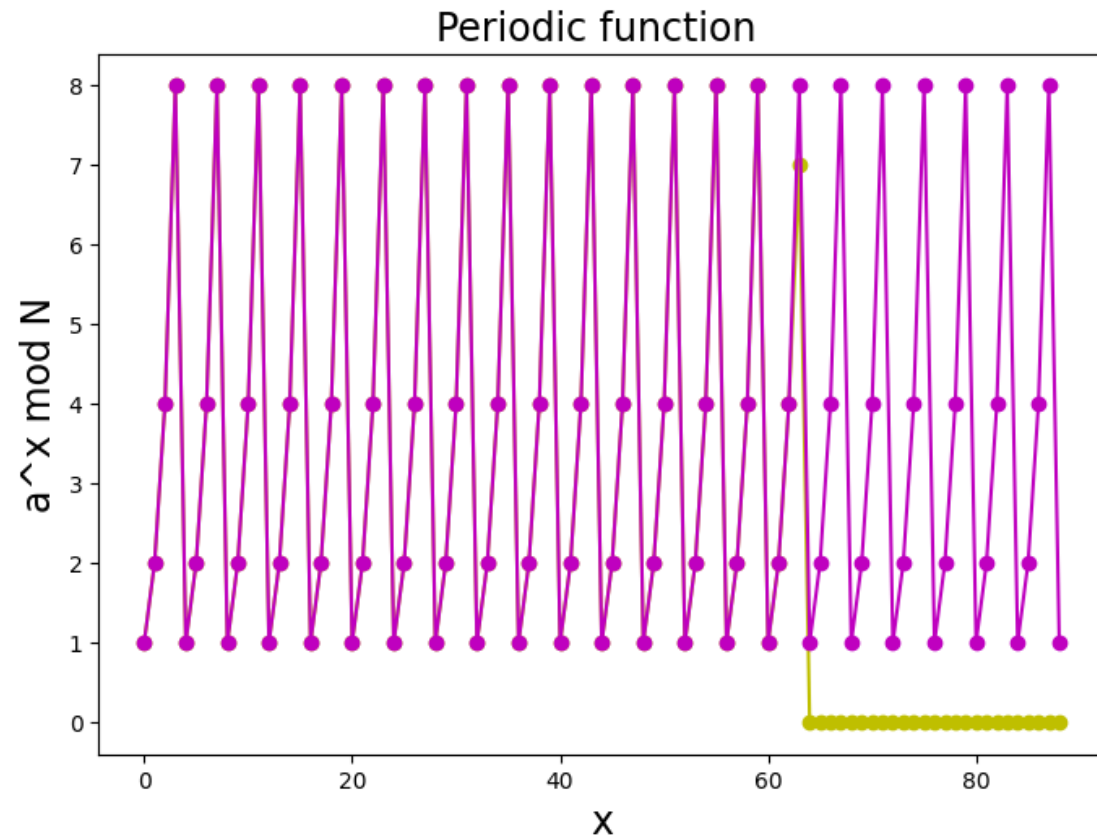
We will use it in the quantum algorithm as well.

https://en.wikipedia.org/wiki/Exponentiation_by_squaring

<https://mathlesstraveled.com/2018/08/18/modular-exponentiation-by-repeated-squaring/>



Period Finding – Classical FFT

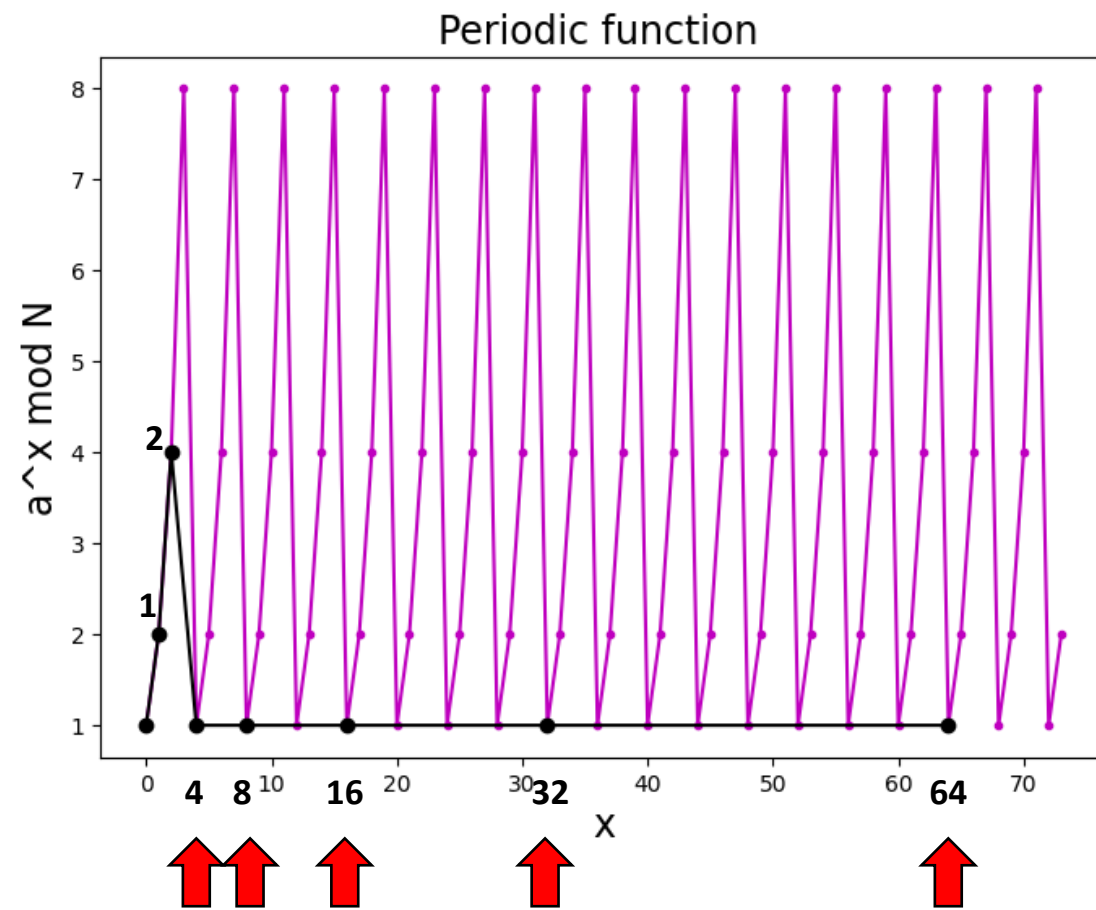


Period Finding – Repeated Squares

$i=0$ $f_mod = \text{mod}(a^{2^0}, N) = 2$ power = 1
 $i=1$ $f_mod = \text{mod}(a^{2^1}, N) = 4$ power = 2
 $i=2$ $f_mod = \text{mod}(a^{2^2}, N) = 1$ power = 4
 $i=3$ $f_mod = \text{mod}(a^{2^3}, N) = 1$ power = 8
 $i=4$ $f_mod = \text{mod}(a^{2^4}, N) = 1$ power = 16
 $i=5$ $f_mod = \text{mod}(a^{2^5}, N) = 1$ power = 32
 $i=6$ $f_mod = \text{mod}(a^{2^6}, N) = 1$ power = 64

So first value of '1' will just repeat when squared!
→ Periodically!

$$\text{mod}(a^{2^2}, N) = 1$$
$$\text{mod}(2^4, 15) = 1$$



Period Finding – Repeated Squares

So first value of '1' will just repeat when squared! → Periodically!

$$\text{mod}(a^4, 15) = 1$$

But how does the period give the factors of N?

So 'r' is the period when the function returns to 1 and therefore $a^r \text{ mod } N = 1$

In this case: $a^r = 2^4 = 16 \rightarrow 16 \text{ (mod } 15) = 1$

So $(a^r - 1) \text{ mod } N = 0$

So critically then → N must divide $(a^r - 1)$ **without** a remainder!

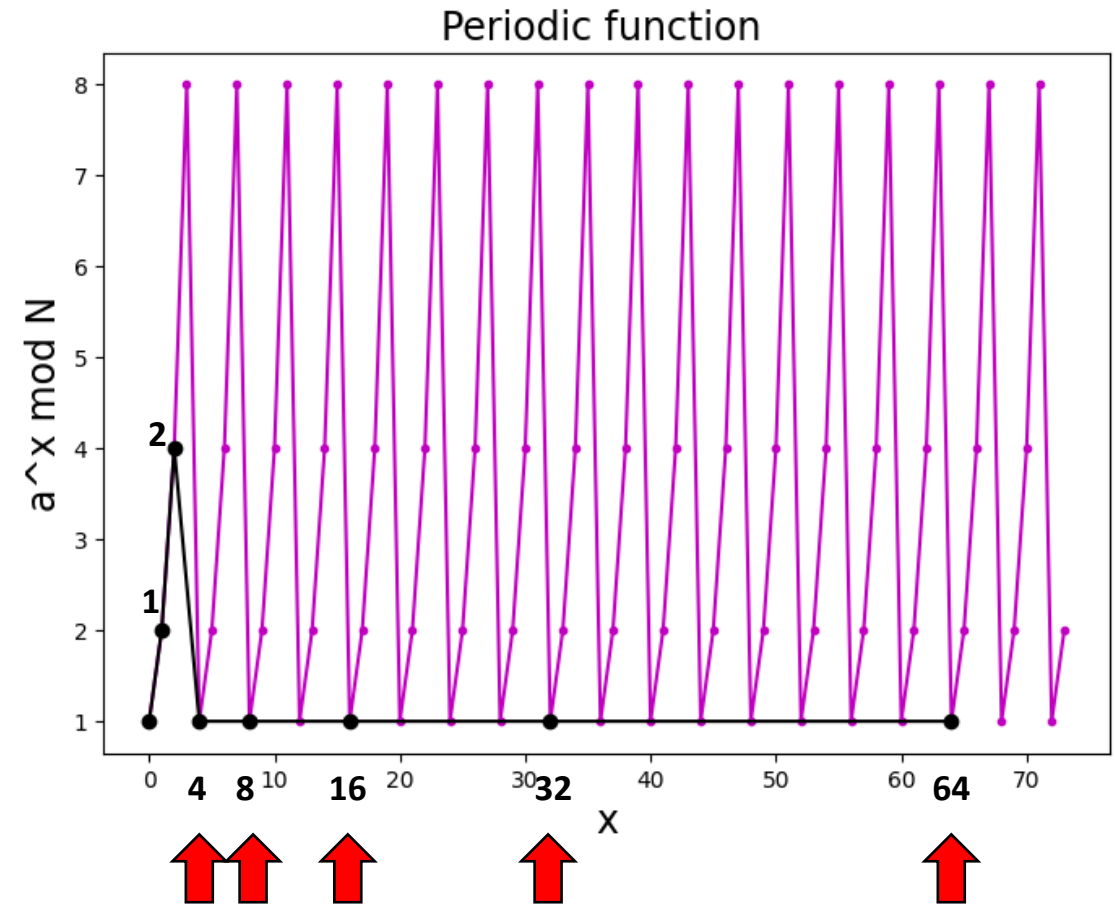
If r is also an even number, then $(a^r - 1)$ has two factors:

$$(a^r - 1) = (a^{r/2} - 1)(a^{r/2} + 1)$$

$$(2^4 - 1) = (2^2 - 1)(2^2 + 1) = (3)(5) = 15$$

Success!!! 3 and 5 are our factors p and q!!!!

If r is an odd number we must start over by picking a different 'a' and try again

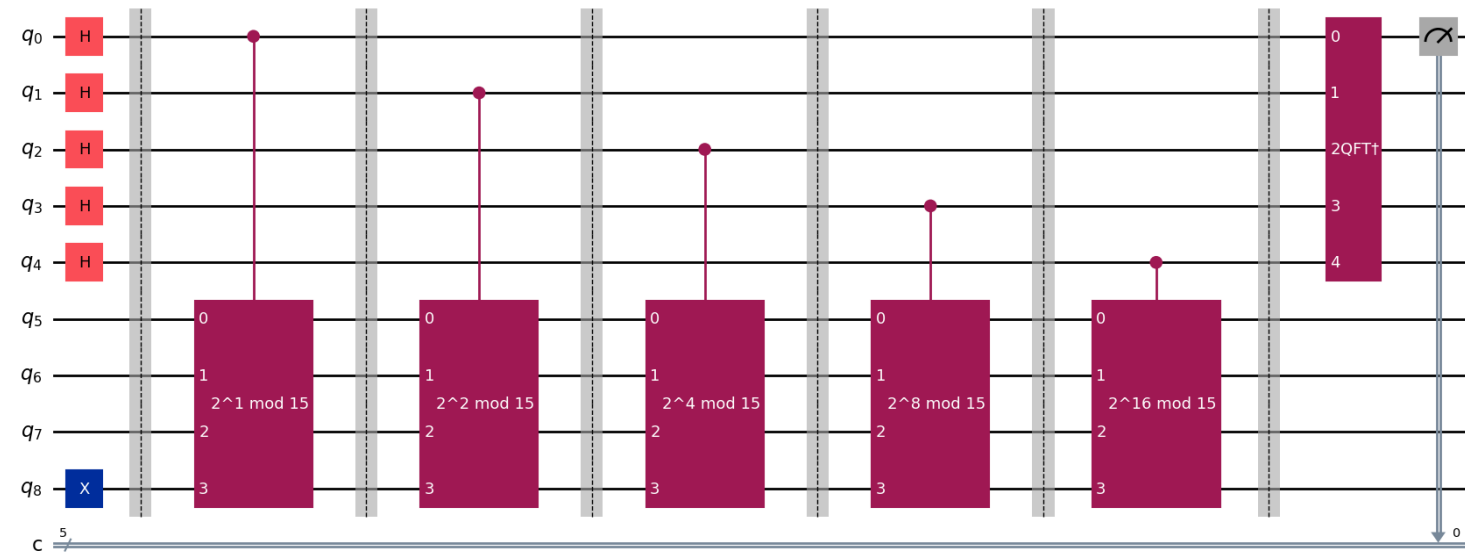
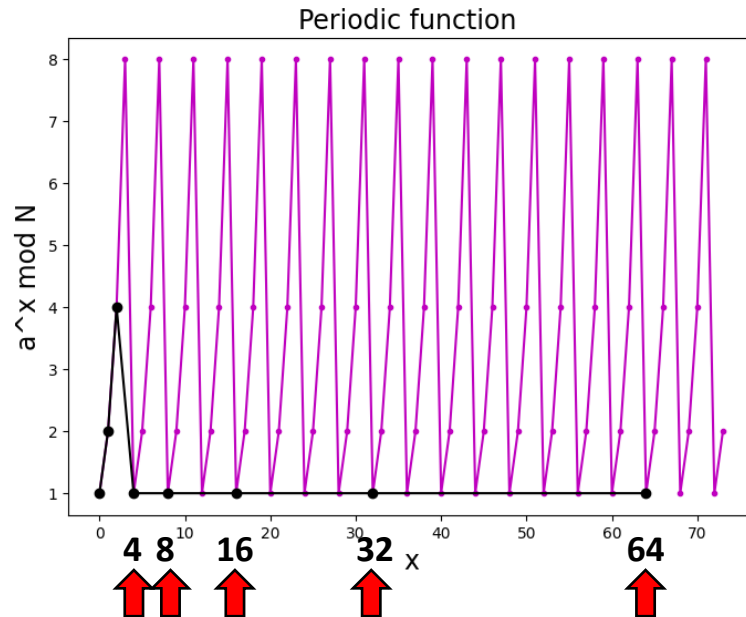


Period Finding – Repeated Squares

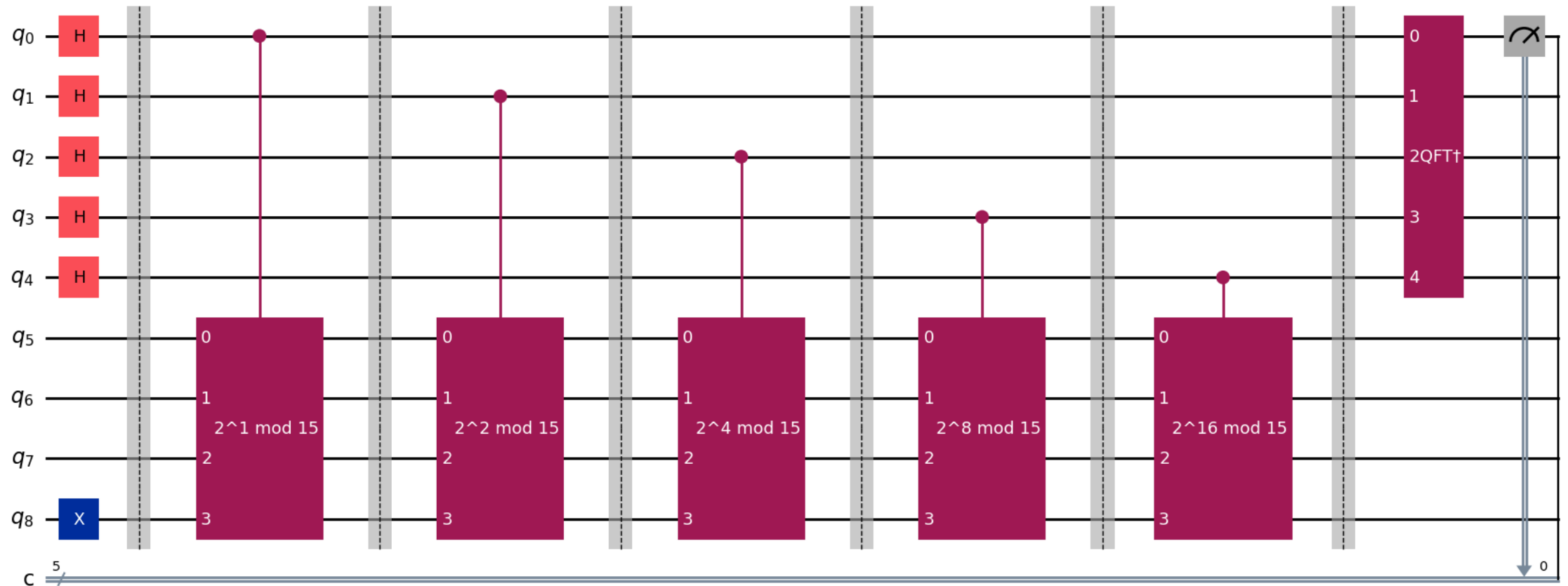
$\text{mod}(a^{2^0}, N) = 2$ power = 1
 $\text{mod}(a^{2^1}, N) = 4$ power = 2
 $\text{mod}(a^{2^2}, N) = 1$ power = 4
 $\text{mod}(a^{2^3}, N) = 1$ power = 8
 $\text{mod}(a^{2^4}, N) = 1$ power = 16

$q_0 = 2$ phase kickbacks ($\text{mod}(a^{2^0}, N) = 2$)
 $q_1 = 4$ phase kickbacks ($\text{mod}(a^{2^1}, N) = 4$)
 $q_2 = 1$ phase kickback ($\text{mod}(a^{2^2}, N) = 1$)
 $q_3 = 1$ phase kickback ($\text{mod}(a^{2^3}, N) = 1$)
 $q_4 = 1$ phase kickback ($\text{mod}(a^{2^4}, N) = 1$)

INV QFT → BIT space



Quantum Circuit Implementation



Modular Exponentiation with Quantum Circuits

Eigenstate of U

A superposition of states in a single cycle is an eigenstate of the modular exponentiation!

That is if we setup a state:

$$|u_0\rangle = \frac{1}{\sqrt{4}}(|1\rangle + |2\rangle + |4\rangle + |8\rangle)$$

$$|u_0\rangle = \frac{1}{\sqrt{4}}(|0001\rangle + |0010\rangle + |0100\rangle + |1000\rangle)$$

$$|u_0\rangle = \frac{1}{\sqrt{4}}(U|0001\rangle + U^2|0001\rangle + U^3|0001\rangle + U^4|0001\rangle)$$


$$|u_0\rangle = \frac{1}{\sqrt{r}} \sum U^x |1\rangle$$

$$|u_0\rangle = \frac{1}{\sqrt{r}} \sum |a^x \bmod N\rangle$$

When we apply the operator U everything just shifts, and we get the same superpositions of states:

$$U|y\rangle = |a \cdot y \bmod N\rangle$$

$$U|y\rangle = |2 \cdot y \bmod 15\rangle$$

$$\begin{aligned} U|0001\rangle &= |0010\rangle \\ U|0010\rangle &= |0100\rangle \\ U|0100\rangle &= |1000\rangle \\ U|1000\rangle &= |0001\rangle \end{aligned}$$


so:

$$U|u_0\rangle = |u_0\rangle$$

This means that $|u_0\rangle$ is an eigenvector of U, with an eigenvalue of 1.

Where are the phases?

However, there are other eigenvectors of U.

Consider a construction that each basis state within the superposition statevector has a phase that is proportional to x such that:

$$|u_1\rangle = \frac{1}{\sqrt{r}} \sum e^{\frac{-2\pi i x}{r}} |a^x \bmod N\rangle$$

$$|u_1\rangle = \frac{1}{\sqrt{4}} (|1\rangle + e^{\frac{-2\pi i 1}{4}} |2\rangle + e^{\frac{-2\pi i 2}{4}} |4\rangle + e^{\frac{-2\pi i 3}{4}} |8\rangle)$$

Now applying U shifts the state register as before leaving the phase.

$$U|u_1\rangle = \frac{1}{\sqrt{4}} (|2\rangle + e^{\frac{-2\pi i 1}{4}} |4\rangle + e^{\frac{-2\pi i 2}{4}} |8\rangle + e^{\frac{-2\pi i 3}{4}} |1\rangle)$$

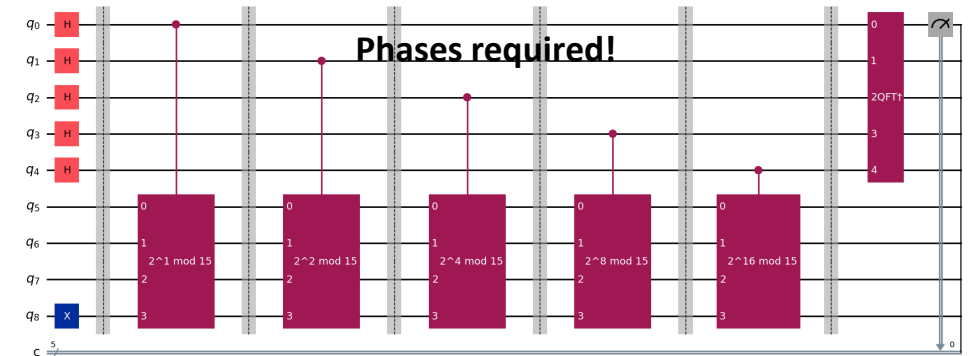
$$U|u_1\rangle = \frac{1}{\sqrt{4}} e^{\frac{2\pi i}{4}} (e^{\frac{-2\pi i}{4}} |2\rangle + e^{\frac{-2\pi i 2}{4}} |4\rangle + e^{\frac{-2\pi i 3}{4}} |8\rangle + e^{\frac{-2\pi i 4}{4}} |1\rangle)$$

$$U|u_1\rangle = \frac{1}{\sqrt{4}} e^{\frac{2\pi i}{4}} (e^{\frac{-2\pi i}{4}} |2\rangle + e^{\frac{-2\pi i 2}{4}} |4\rangle + e^{\frac{-2\pi i 3}{4}} |8\rangle + |1\rangle)$$

$$U|u_1\rangle = \frac{1}{\sqrt{4}} e^{\frac{2\pi i}{4}} (|u_1\rangle)$$

Notice how the phase factor has the period 1/4 which is 1/r in it! This is the period of the function and will give us the factors of N.

The problem is that we don't know $|u_1\rangle$



All you need is $|1\rangle$

The Magic

What is surprising is that we just measure the phase of the trivially easy state to prepare $|1\rangle$ (aka $|00001\rangle$) and get the phase factor with a denominator r without knowing eigenstate $|u_1\rangle$

r is the periodicity of the function and will allow us to factor N .

But how can everything be measured through $|1\rangle$?

First we can generalize $|u_1\rangle$ for other multiples (besides $s=1$) for s up to $r-1$

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum e^{\frac{-2\pi i x s}{r}} |a^x \bmod N\rangle$$

$$|u_s\rangle = \frac{1}{\sqrt{4}} (|1\rangle + e^{\frac{-2\pi i 1 s}{4}} |2\rangle + e^{\frac{-2\pi i 2 s}{4}} |4\rangle + e^{\frac{-2\pi i 3 s}{4}} |8\rangle)$$

$$U|u_s\rangle = \frac{1}{\sqrt{4}} e^{\frac{2\pi i s}{4}} (|u_1\rangle)$$

What is remarkable is that:

$$\frac{1}{\sqrt{r}} (|u_0\rangle + |u_1\rangle + |u_2\rangle + |u_3\rangle) = |1\rangle$$

Writing it all out:

$$\begin{aligned} \frac{1}{2} (|u_0\rangle &= \frac{1}{\sqrt{4}} (|1\rangle + |2\rangle + |4\rangle + |8\rangle) \dots \\ &+ |u_1\rangle = \frac{1}{\sqrt{4}} (|1\rangle + e^{\frac{-2\pi i 1}{4}} |2\rangle + e^{\frac{-2\pi i 2}{4}} |4\rangle + e^{\frac{-2\pi i 3}{4}} |8\rangle) \dots \\ &+ |u_2\rangle = \frac{1}{\sqrt{4}} (|1\rangle + e^{\frac{-2\pi i 1 \cdot 2}{4}} |2\rangle + e^{\frac{-2\pi i 4}{4}} |4\rangle + e^{\frac{-2\pi i 6}{4}} |8\rangle) \dots \\ &+ |u_3\rangle = \frac{1}{\sqrt{4}} (|1\rangle + e^{\frac{-2\pi i 1 \cdot 3}{4}} |2\rangle + e^{\frac{-2\pi i 6}{4}} |4\rangle + e^{\frac{-2\pi i 9}{4}} |8\rangle)) = \\ &= |1\rangle \end{aligned}$$

Checking the $|2\rangle$ phase terms :

$$e^0 + e^{\frac{-\pi i}{2}} + e^{-\pi i} + e^{\frac{-6\pi i}{4}} = 1 + i - 1 - i = 0$$

They all cancel out except $|1\rangle$!

Which means that we can do QPE on the function U just using the initial state $|1\rangle$ (aka $|00001\rangle$) because it is equivalent to testing all of the other eigenstates $|u_s\rangle$!

So we just initialize the first bit in the register $|1\rangle$ and we're ready to apply U !

Actual Quantum Circuit???

Back to the circuit

Ok, but how are we going to implement U?

$$U|y\rangle = |2 \cdot y \bmod 15\rangle$$

Looking back at the cyclical nature of U we can see that U is just *swapping* the bit up the register (multiplying by 2...mod 15)

$$U|0001\rangle = |0010\rangle$$

$$U|0010\rangle = |0100\rangle$$

$$U|0100\rangle = |1000\rangle$$

$$U|1000\rangle = |0001\rangle$$

equivalent to:

$$U(1) = 2$$

$$U(2) = 4$$

$$U(4) = 8$$

$$U(8) = 1$$

So we can construct U from swaps.

Working it out again but with repeating squares:

$$U = 2^x \bmod 15$$

$$U|1\rangle = |2\rangle$$

$$U^2|1\rangle = |4\rangle$$

$$U^4|1\rangle = |1\rangle$$

$$U^8|1\rangle = |1\rangle$$

Repeating. With a period of 4 U's since U^4 returns back to the original value 1.

Writing out in binary:

$$U|0001\rangle = |0010\rangle$$

$$U^2|0001\rangle = |0100\rangle$$

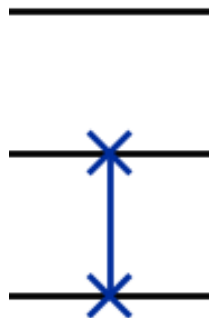
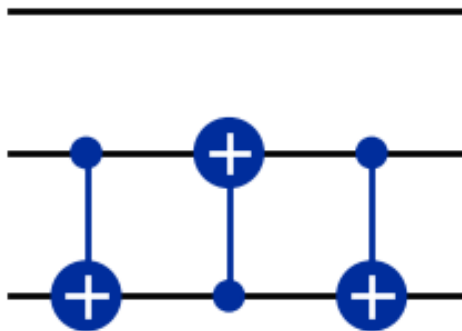
$$U^4|0001\rangle = |0001\rangle$$

Again, U is swapping the bit up the register.

Swaps → Control Swaps

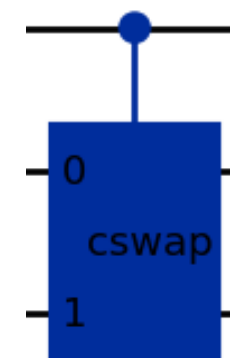
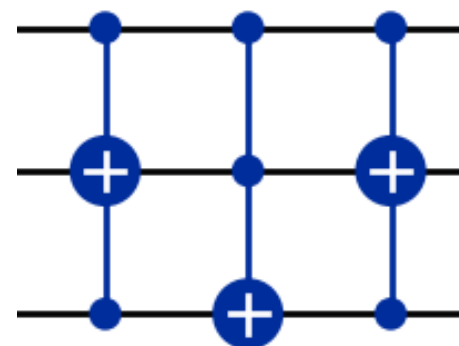
- **Swap**

- Created from 3 CNOT gates

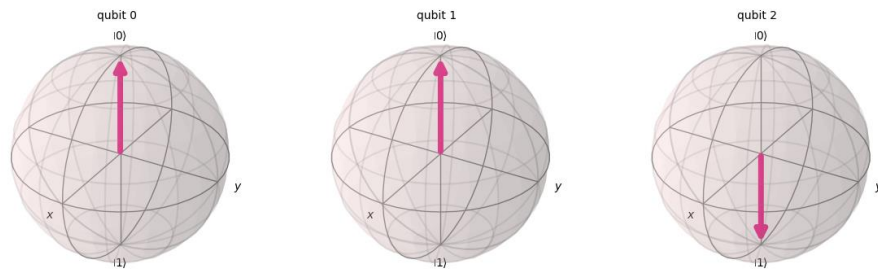
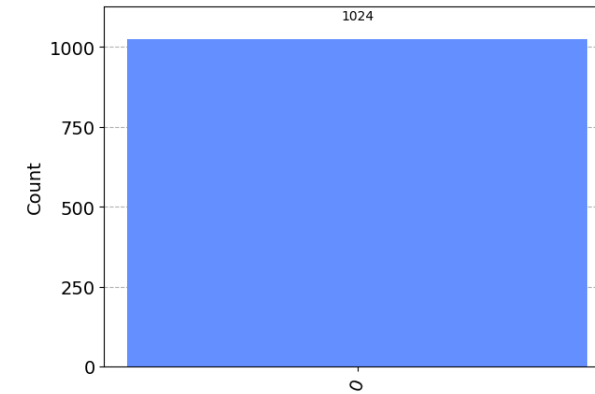
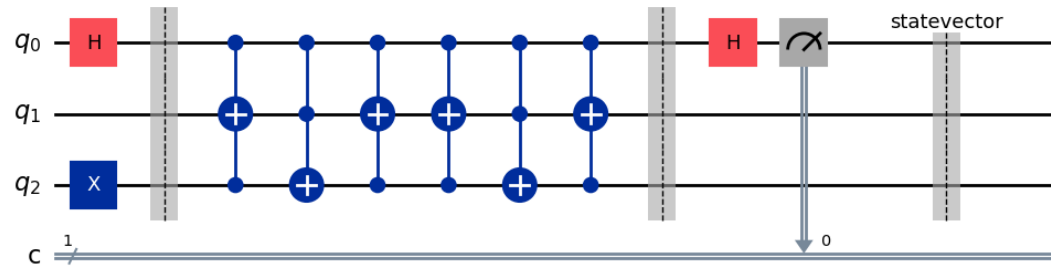


- **Control Swap**

- Created from 3 Toffoli gates (CCNOTs)



Phase Kickback from 2 Control SWAPs



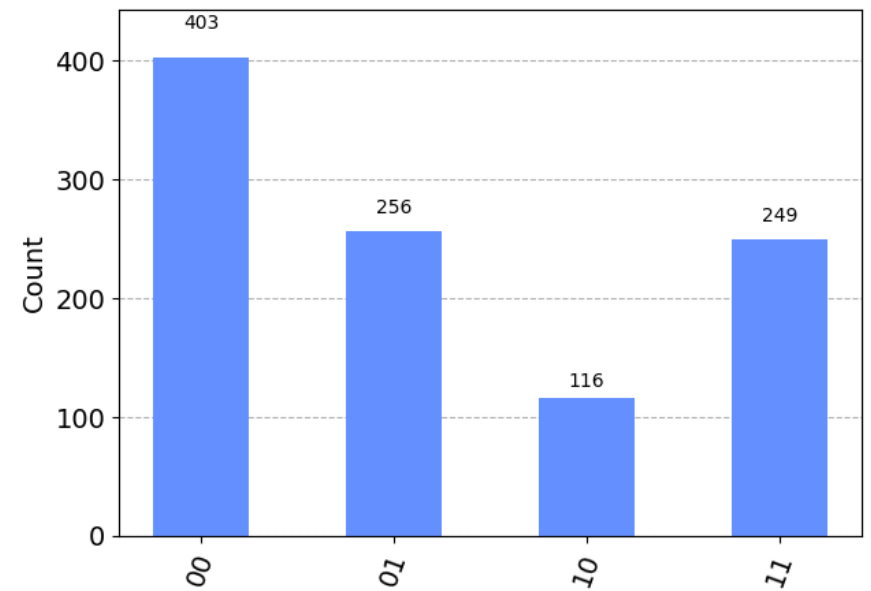
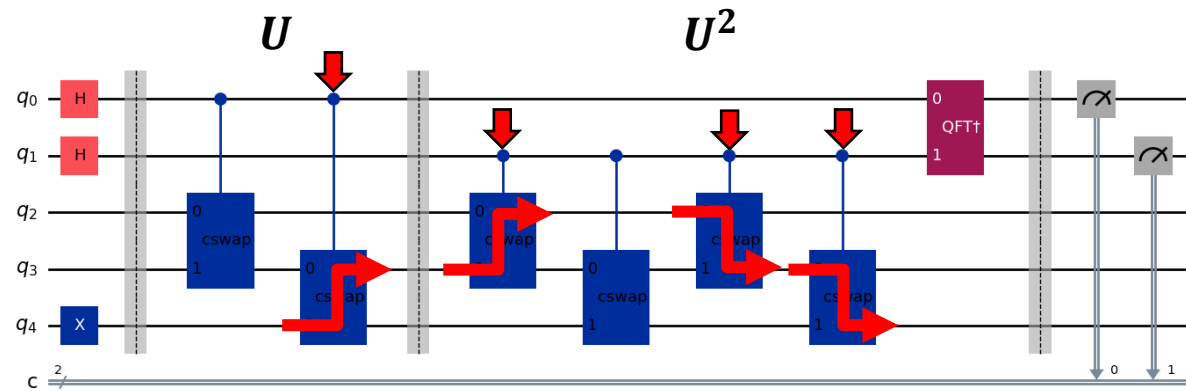
Now the 'QFT' qubit in the top register returns to 0 and the qubits in the bottom register become disentangled returning to their original states.

This shows that the $(\text{Control SWAP})^2$ is periodic!

$$U^2(1) = 2^2 \cdot 1 \bmod 3 = 1$$

This is what we want to do with more sophisticated functions U , find when they are periodic. In this case the order was just 2. Finding the order of U will allow us to factor N (because U is mod N).

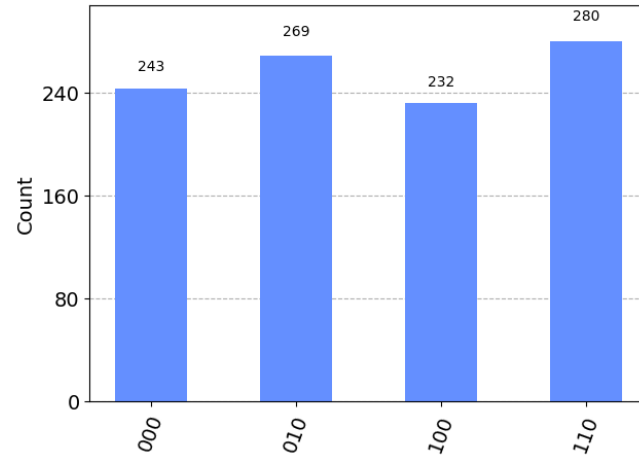
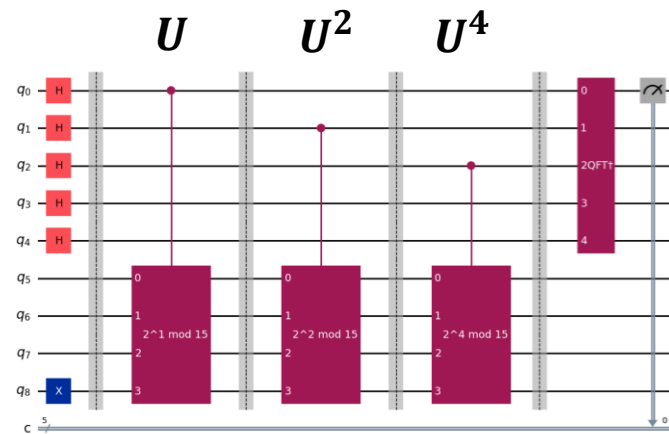
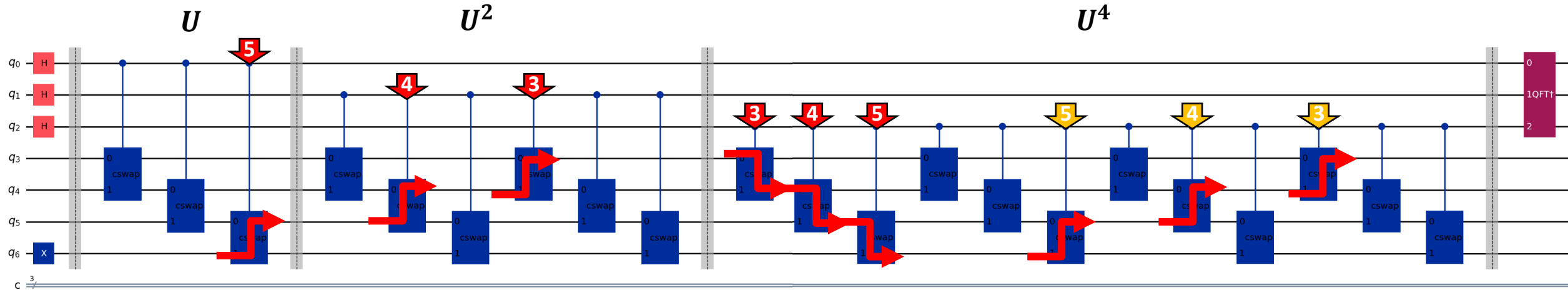
Ripple Swap



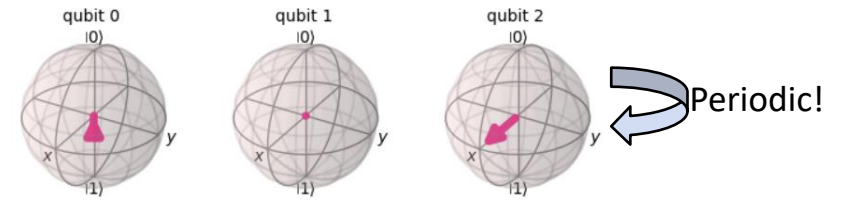
Register	Output	Phase
0	01(bin) = 1(dec)	$1/4 = 0.25$
1	10(bin) = 2(dec)	$2/4 = 0.50$
2	00(bin) = 0(dec)	$0/4 = 0.00$
3	11(bin) = 3(dec)	$3/4 = 0.75$

Phase	Fraction	Guess for r
0 0.25	$1/4$	4
1 0.50	$1/2$	2
2 0.00	$0/1$	1
3 0.75	$3/4$	4

Shor's Algorithm - Repeated Squares!



Phase kickbacks before InvQFT



Outputs

110(bin) = 6(dec)
 000(bin) = 0(dec)
 010(bin) = 2(dec)
 100(bin) = 4(dec)

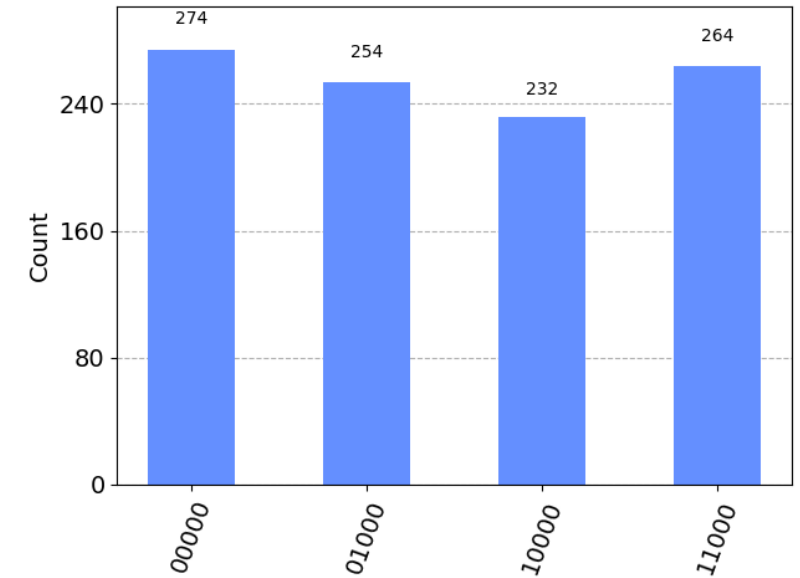
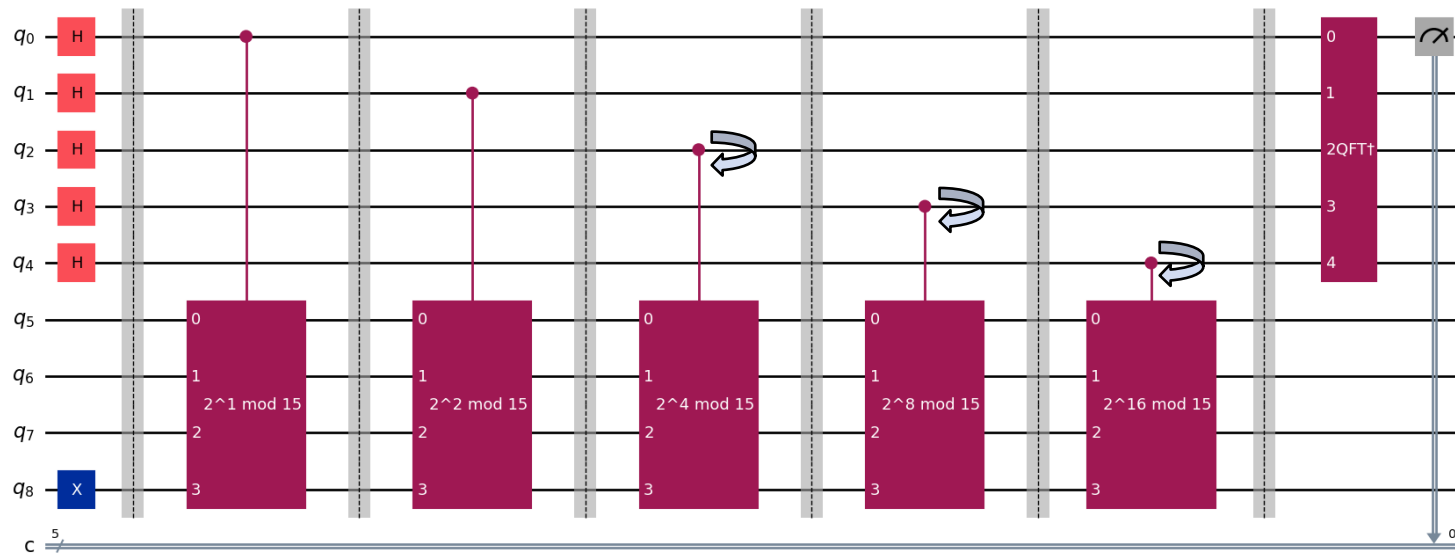
Phase

$6/8 = 3/4$
 $0/8 = 0$
 $2/8 = 1/4$
 $4/8 = 2/4$

r=

4
 0
 4
 2

Shor's Algorithm



Output	Phase	Fraction	r=
01000 (bin) = 8 (decimal)	$8/32 = 0.25$	$1/4$	4
00000 (bin) = 0 (decimal)	$0/32 = 0.00$	$0/1$	1
11000 (bin) = 24 (decimal)	$24/32 = 0.75$	$3/4$	4
10000 (bin) = 16 (decimal)	$16/32 = 0.50$	$1/2$	2

scientific reports



OPEN

Demonstration of Shor's factoring algorithm for $N = 21$ on IBM quantum processors

Unathi Skosana¹✉ & Mark Tame¹ ¹

We report a proof-of-concept demonstration of a quantum order-finding algorithm for factoring the integer 21. Our demonstration involves the use of a compiled version of the quantum phase estimation routine, and builds upon a previous demonstration. We go beyond this work by using a configuration of approximate Toffoli gates with residual phase shifts, which preserves the functional correctness and allows us to achieve a complete factoring of $N = 21$. We implemented the algorithm on IBM quantum processors using only five qubits and successfully verified the presence of entanglement between the control and work register qubits, which is a necessary condition for the algorithm's speedup in general. The techniques we employ may be useful in carrying out Shor's algorithm for larger integers, or other algorithms in systems with a limited number of noisy qubits.

Shor with N=21 run on IBM QPUs

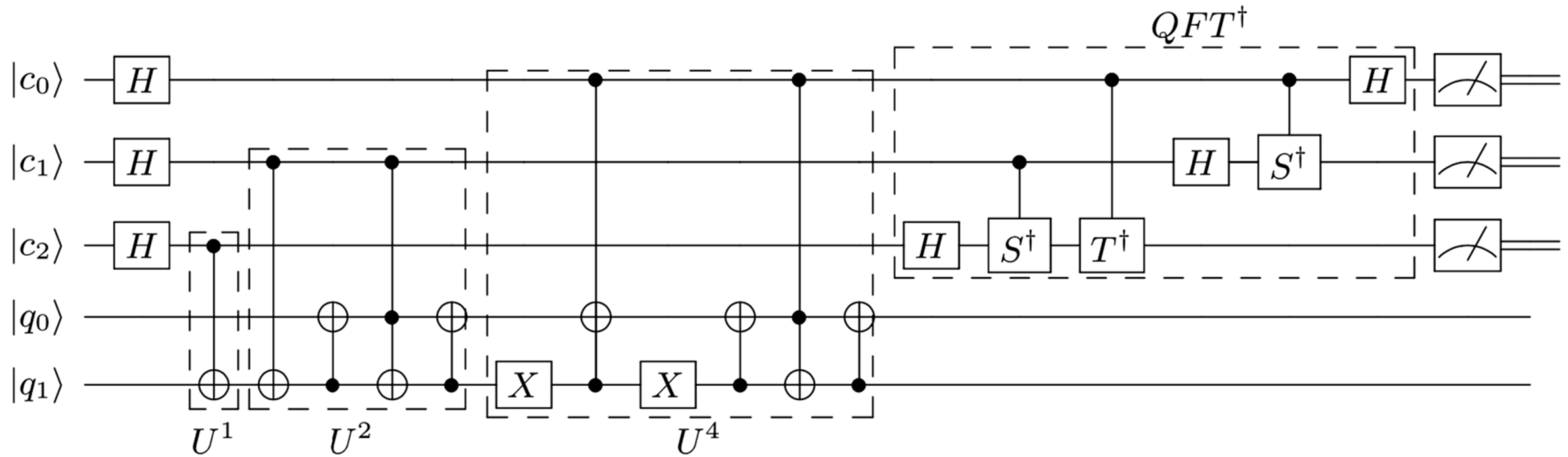
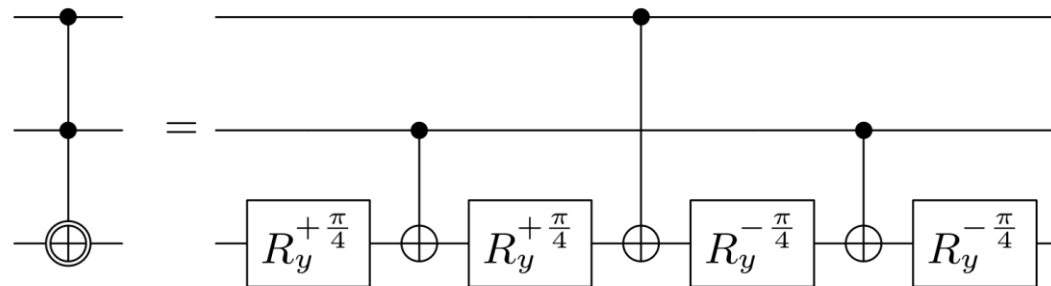
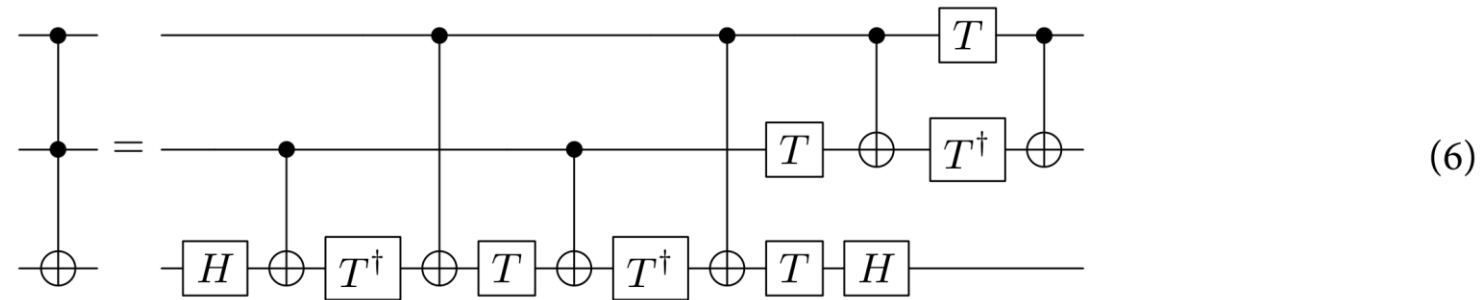


Figure 2. Compiled quantum order-finding routine for $N = 21$ and $a = 4$. This circuit uses five qubits in total; 3 for the control register and 2 for the work register. The above circuit determines $2^n s/r$ to three bits of accuracy, from which the order can be extracted. Here, up to a global phase, $S = R_z(\frac{\pi}{2})$ and $T = R_z(\frac{\pi}{4})$ are phase and $\pi/8$ gates, respectively.

Margolus gate (Toffoli with phase shift)

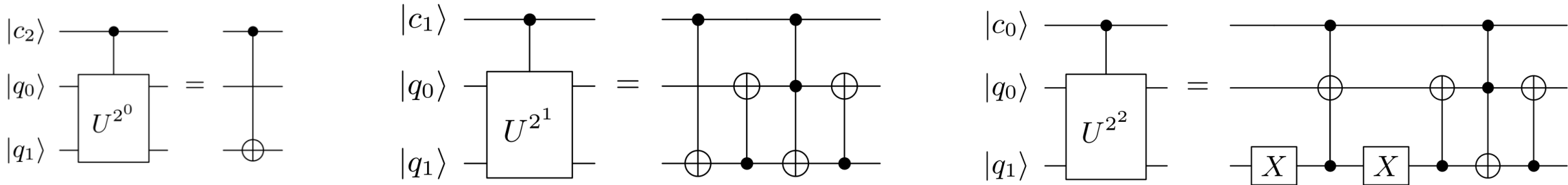
Modular exponentiation with relative phase Toffolis. In total, the modular exponentiation routine requires three Toffoli gates; traditionally a single Toffoli gate can be decomposed into six CX gates and several single-qubit gates² as follows



Compilation tricks: Log4 basis

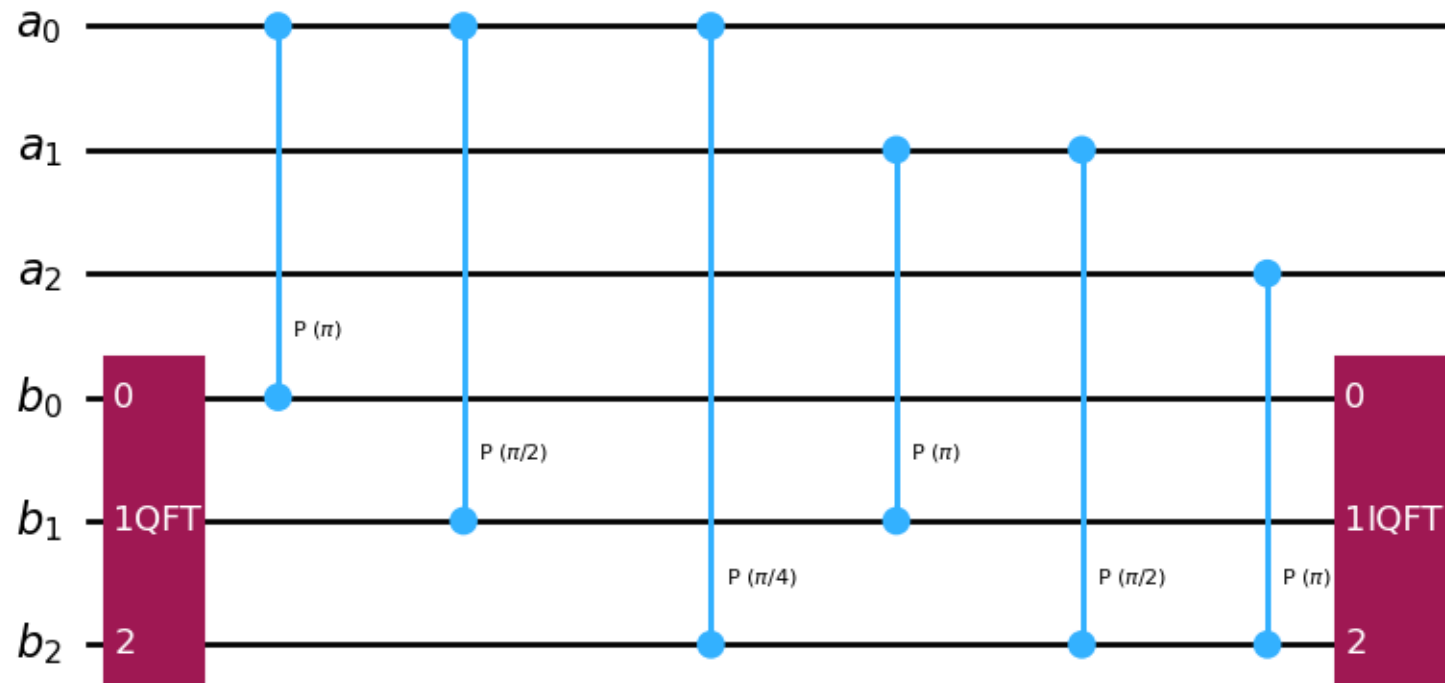
$$\begin{aligned} |1\rangle &\mapsto |\log_4 1\rangle = |00\rangle, \\ |4\rangle &\mapsto |\log_4 4\rangle = |01\rangle, \\ |16\rangle &\mapsto |\log_4 16\rangle = |10\rangle. \end{aligned}$$

$$\begin{aligned} \hat{U}^1 &: \{|1\rangle \mapsto |4\rangle, |4\rangle \mapsto |16\rangle, |16\rangle \mapsto |1\rangle\}, \\ \hat{U}^2 &: \{|1\rangle \mapsto |16\rangle, |4\rangle \mapsto |1\rangle, |16\rangle \mapsto |4\rangle\}, \\ \hat{U}^4 &: \{|1\rangle \mapsto |4\rangle, |4\rangle \mapsto |16\rangle, |16\rangle \mapsto |1\rangle\}. \end{aligned}$$

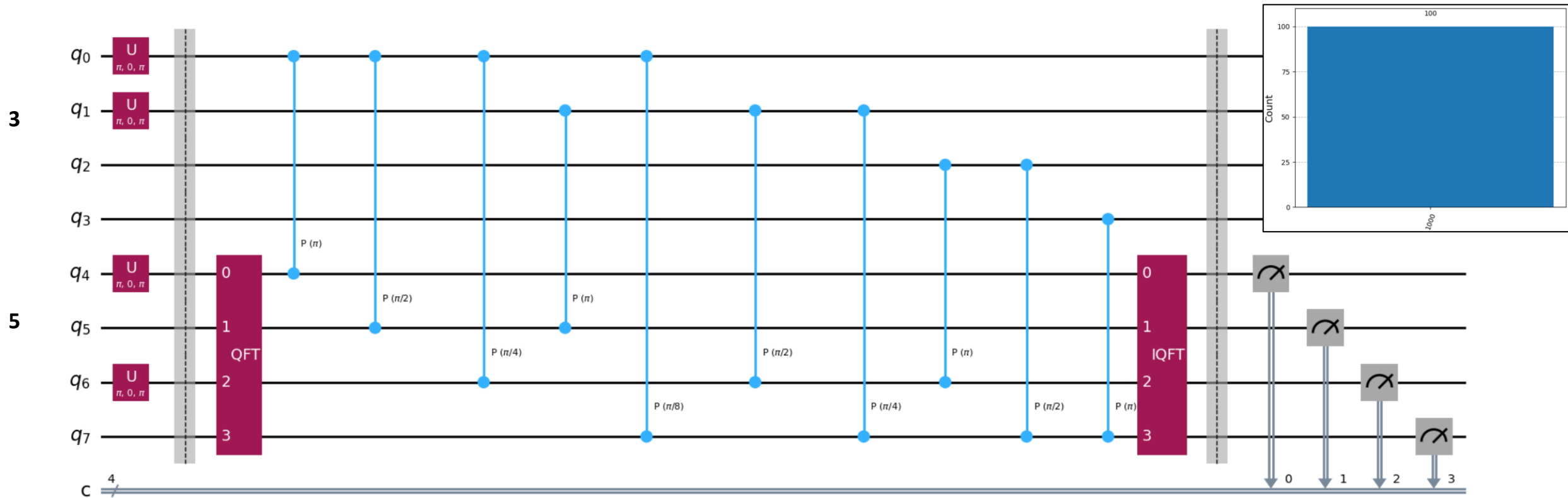


Draper Adder for more efficient arithmetic

from qiskit.circuit.library import DraperQFTAdder



Draper Adder : $3+5=8$



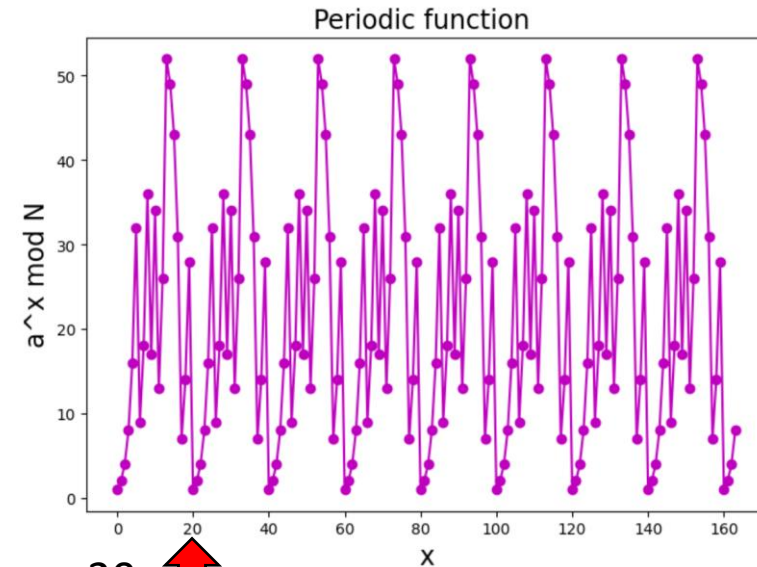
Larger N

$p = 5$
 $q = 11$
 $N = 55$

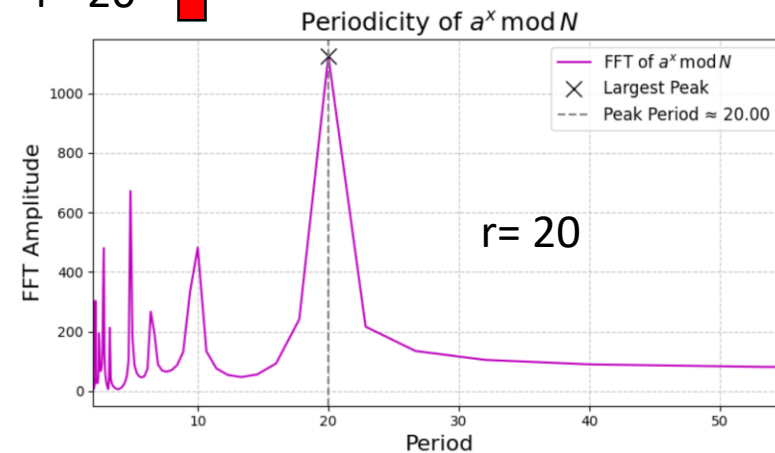
$a^{**}(r/2)-1 = 1023$
 $a^{**}(r/2)+1 = 1025$

$\text{mod}(a^{**}(r/2)-1, N) = 33$
 $\text{mod}(a^{**}(r/2)+1, N) = 35$

$\text{gcd}(\text{mod}(a^{**}(r/2)-1, N), N) = 11$
 $\text{gcd}(\text{mod}(a^{**}(r/2)+1, N), N) = 5$



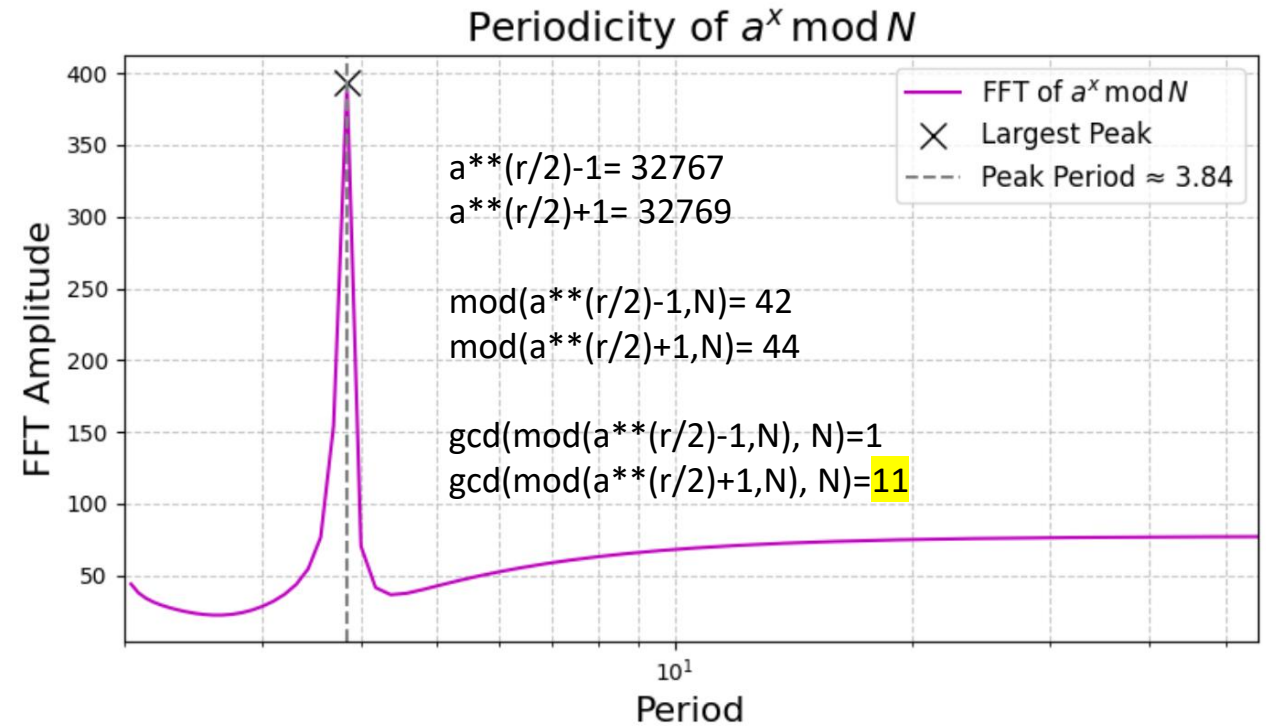
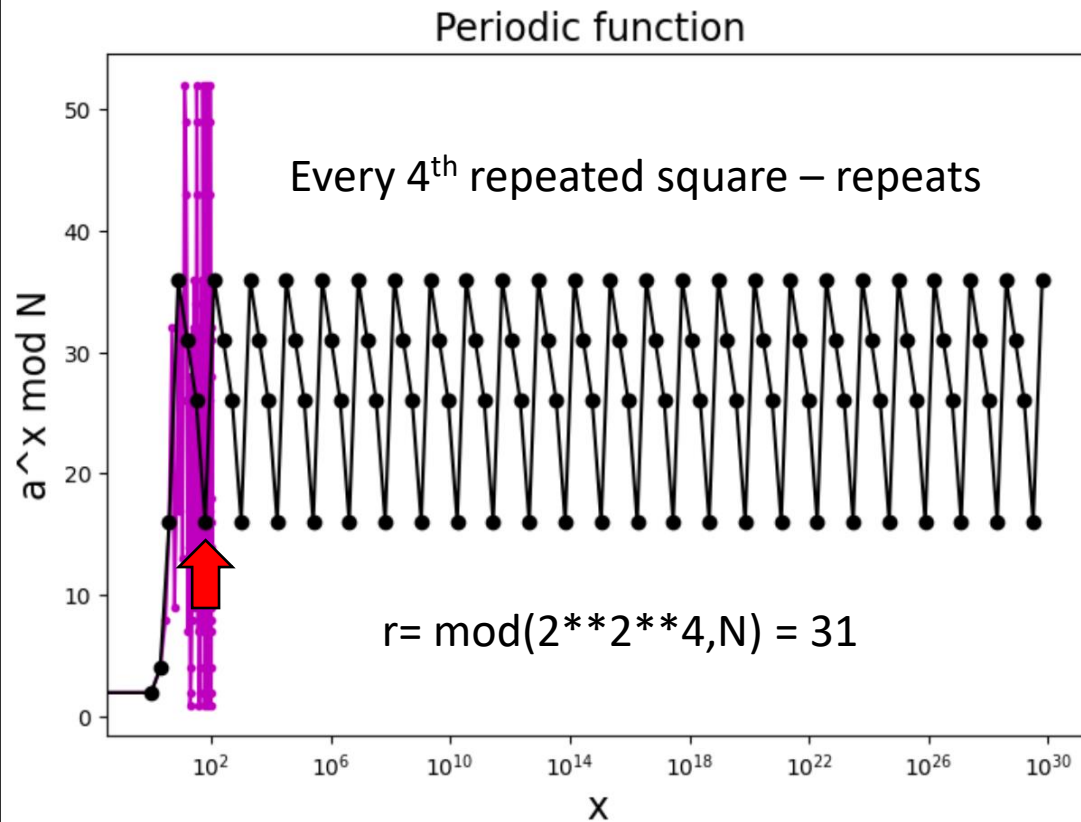
$r = 20$



Periodicity of Repeated Squares

$N = 55$

$$r = \text{int}(\text{mod}(2^{**2^{**4}}, N)) = 31$$



An Efficient Quantum Factoring Algorithm

Oded Regev*

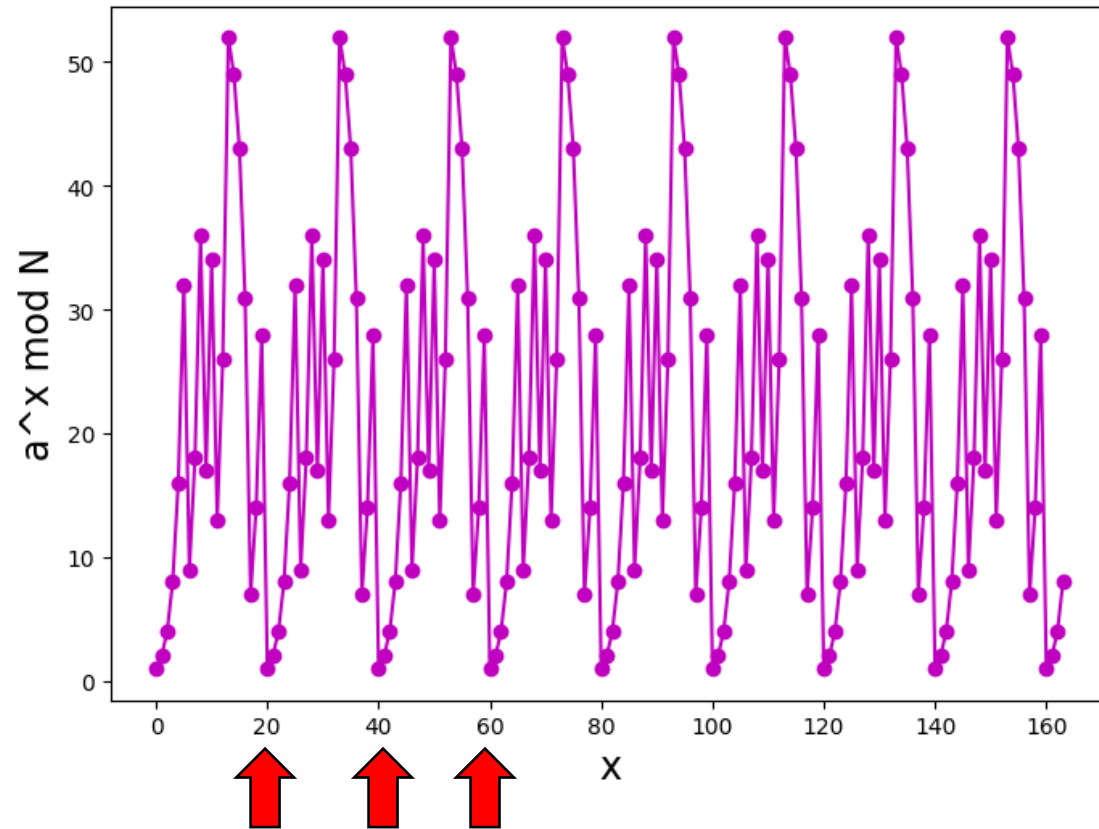
Abstract

We show that n -bit integers can be factorized by independently running a quantum circuit with $\tilde{O}(n^{3/2})$ gates for $\sqrt{n} + 4$ times, and then using polynomial-time classical post-processing. The correctness of the algorithm relies on a number-theoretic heuristic assumption reminiscent of those used in subexponential classical factorization algorithms. It is currently not clear if the algorithm can lead to improved physical implementations in practice.

Period Finding – in Higher Dimensions

$$a^x \bmod (N)$$
$$2^x \bmod (55)$$

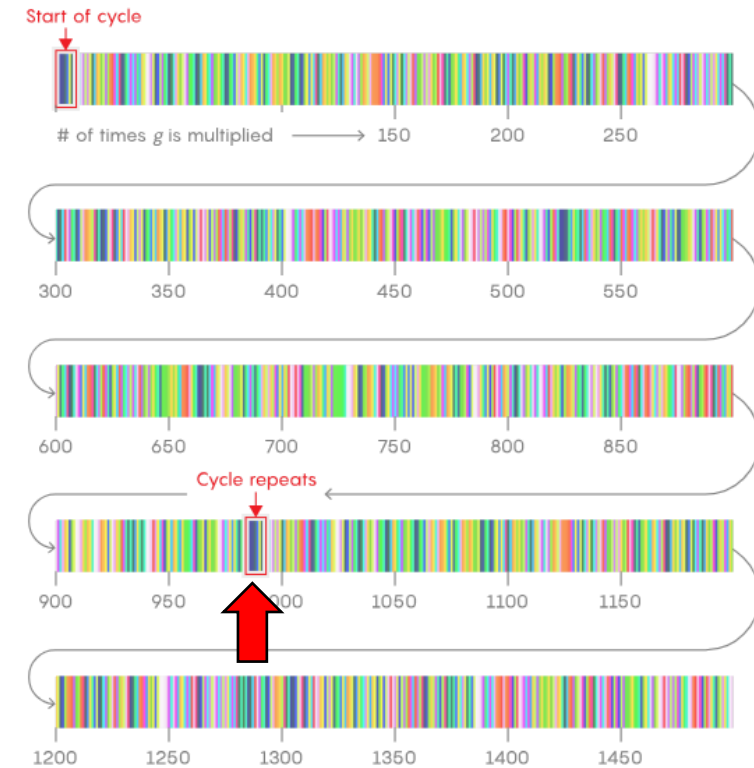
Periodic function



$$a^x \bmod (N)$$

Seeking Cycles

Shor's algorithm repeatedly multiplies a number g and looks for a repeating pattern, or cycle, in the output of a certain function.



Period Finding – in Higher Dimensions

$$g_1^x \bmod (N)$$

Seeking Cycles

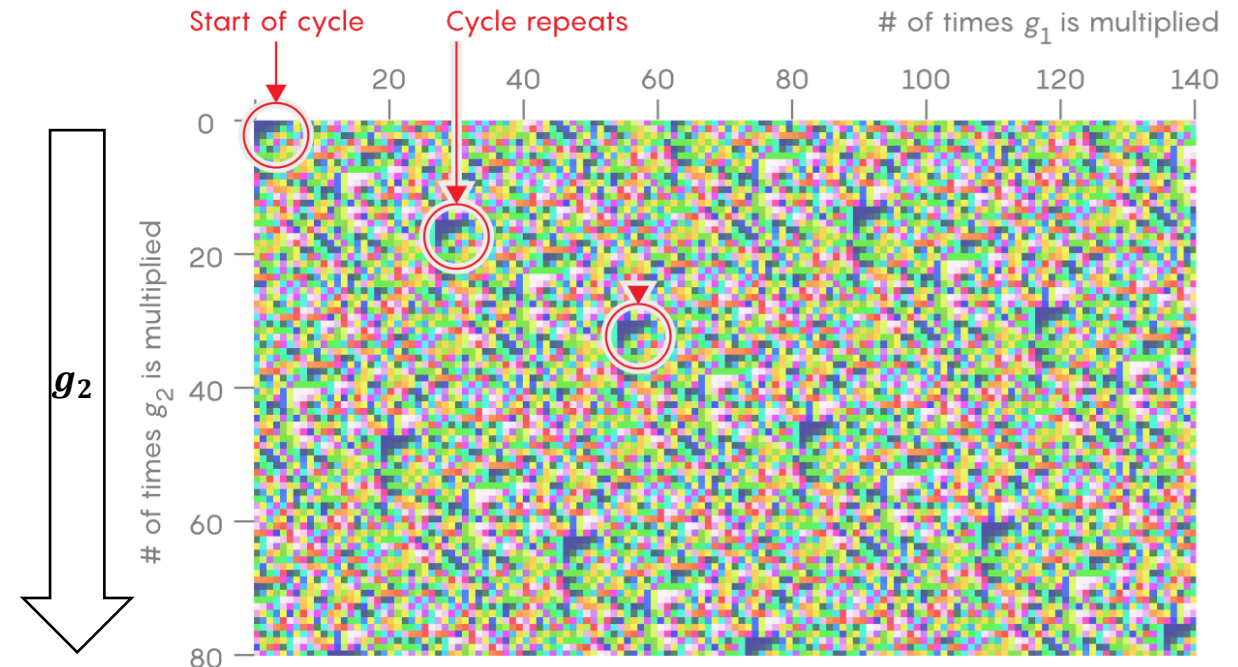
Shor's algorithm repeatedly multiplies a number g and looks for a repeating pattern, or cycle, in the output of a certain function.



$$g_1^x \cdot g_2^y \bmod (N)$$

Multiple dimensions $4^x \cdot 9^y \bmod (N)$ $4^{19} \cdot 9^{47} \bmod (8051) = 1$

Multidimensional algorithms multiply many numbers, called g_1, g_2 and so on. That can make it easier to find the length of the cycle.

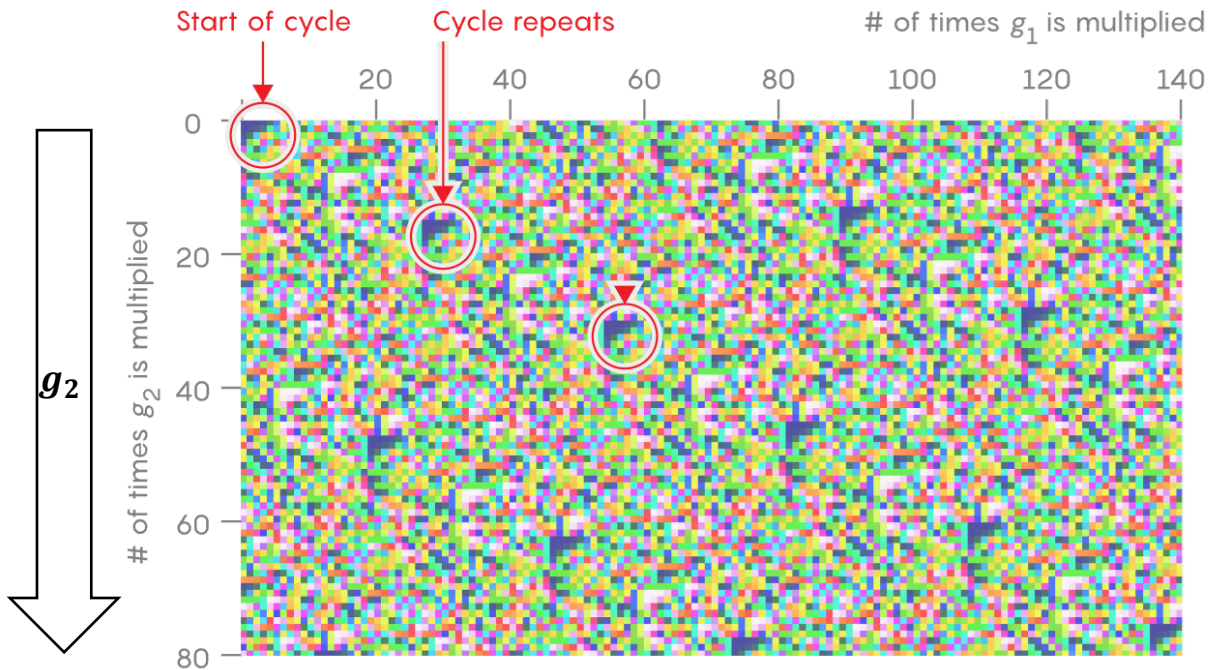


Period Finding – in Higher Dimensions

$$g_1^x \cdot g_2^y \bmod (N)$$

Multiple dimensions $4^x \cdot 9^y \bmod (N)$ $4^{19} \cdot 9^{47} \bmod (8051) = 1$

Multidimensional algorithms multiply many numbers, called g_1, g_2 and so on. That can make it easier to find the length of the cycle.



$$4^{19} \cdot 9^{47} \bmod (8051) = 1$$

Square root is trivial for 4 and 9:

$$2^{19} \cdot 3^{47} \bmod (8051) = 6888$$

$(6888 + 1)(6888 - 1)$ must share GCD with 8051

$$\text{GCD}(6887, 8051) = 97$$

$$8051/97 = 83 \text{ Done!}$$

The trick is that computation with smaller 'a' numbers: g_1 and g_2 is easier and faster because the exponents (x, y) are so much smaller.

New compilations of Shor's Algorithm

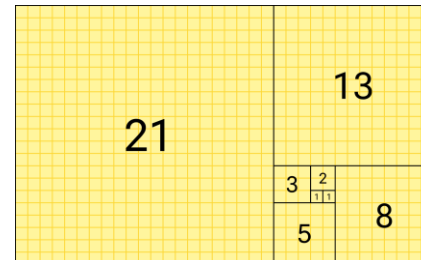
An Efficient Quantum Factoring Algorithm

Oded Regev*

Abstract

We show that n -bit integers can be factorized by independently running a quantum circuit with $\tilde{O}(n^{3/2})$ gates for $\sqrt{n} + 4$ times, and then using polynomial-time classical post-processing. The correctness of the algorithm relies on a number-theoretic heuristic assumption reminiscent of those used in subexponential classical factorization algorithms. It is currently not clear if the algorithm can lead to improved physical implementations in practice.

**Use Fibonacci Seq.
Instead of repeating squares**



Space-Efficient and Noise-Robust Quantum Factoring

Seyoon Ragavan
MIT

sragavan@csail.mit.edu

Vinod Vaikuntanathan
MIT

vinodv@csail.mit.edu

February 19, 2024

Algorithm	Mult. algorithm	Number of qubits	Circuit size
Shor [Sho97]	[HvdH21]	$O(n \log n)$	$O(n^2 \log n)$
Shor	Schoolbook	$O(n)$	$O(n^3)$
Shor	[Gid19]	$O(n)$	$O(n^{\log_2 3 + 1})$
Optimized Shor [Bea03, TK06, Gid17, HRS17]	Schoolbook	$2n + O(1)$	$\tilde{O}(n^3)$
Regev's algorithm [Reg23]	[HvdH21]	$O(n^{3/2})$	$O(n^{3/2} \log n)$
Our optimization of Regev	[HvdH21]	$O(n \log n)$	$O(n^{3/2} \log n)$
Our optimization of Regev	[Gid19]	$O(n)$	$O(n^{\log_2 3 + 1/2})$
Our optimization of Regev	Schoolbook [RNSL17]	$(10.32 + o(1))n$	$O(n^{5/2} \log n)$

Table 1: Comparison of our results with previous work. Asymptotically best-known results for either space or size are highlighted in bold. Note, importantly, that all values here are just for *one run of the circuit*. They do not account for the fact that a circuit for Shor's algorithm only requires $O(1)$ independent runs, while a circuit for Regev's algorithm as well as ours requires $\sqrt{n} + 4$ independent runs. The number of qubits in the last line is derived from Corollary 1.4 assuming that the constant C in Regev's number-theoretic assumption is $1 + o(1)$.

Noise

Let number of qubits, $n = 2000$

- $n^2 \log(n) = 1.3e7$ circuits
- $n^{1.5} \log(n) = 3e6$ circuits

Errors therefore:

- $1/(n^2 \log(n)) = 8e-7$
- $1/(n^{1.5} \log(n)) = 1e-6$

World record Fidelities (Trapped Ions)

- Single qubit gates = $1.5e-7$
 - <https://arxiv.org/abs/2412.04421>
- Two qubit gates = 99.97% ($3e4$)
 - <https://arxiv.org/abs/2407.07694v1>

Algorithm	Mult. algorithm	Number of qubits	Circuit size
Shor [Sho97]	[HvdH21]	$O(n \log n)$	$O(n^2 \log n)$
Shor	Schoolbook	$O(n)$	$O(n^3)$
Shor	[Gid19]	$O(n)$	$O(n^{\log_2 3+1})$
Optimized Shor [Bea03, TK06, Gid17, HRS17]	Schoolbook	$2n + O(1)$	$\tilde{O}(n^3)$
Regev's algorithm [Reg23]	[HvdH21]	$O(n^{3/2})$	$O(n^{3/2} \log n)$
Our optimization of Regev	[HvdH21]	$O(n \log n)$	$O(n^{3/2} \log n)$
Our optimization of Regev	[Gid19]	$O(n)$	$O(n^{\log_2 3+1/2})$
Our optimization of Regev	Schoolbook [RNSL17]	$(10.32 + o(1))n$	$O(n^{5/2} \log n)$

Table 1: Comparison of our results with previous work. Asymptotically best-known results for either space or size are highlighted in bold. Note, importantly, that all values here are just for *one run of the circuit*. They do not account for the fact that a circuit for Shor's algorithm only requires $O(1)$ independent runs, while a circuit for Regev's algorithm as well as ours requires $\sqrt{n}+4$ independent runs. The number of qubits in the last line is derived from Corollary 1.4 assuming that the constant C in Regev's number-theoretic assumption is $1 + o(1)$.