

# Shor's Algorithm

*EE599-001 & EE699-010, Spring 2026*

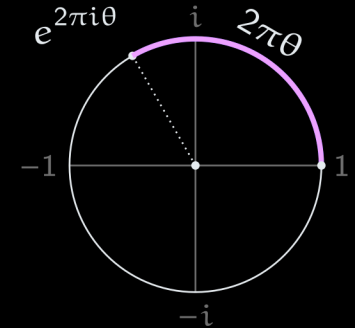
**Hank Dietz**

<http://aggregate.org/hankd/>

# Shor's Algorithm

- The quantum algorithm everyone hears about
  - Fastest algorithm to find prime factors
  - Modern encryption is based on the idea that, given the product of two large primes, it is very difficult to find the two primes
- Shor's algorithm has quantum subroutines, but it is mostly run on a conventional computer
- Not a trivial algorithm...

# It's About Phase Estimation

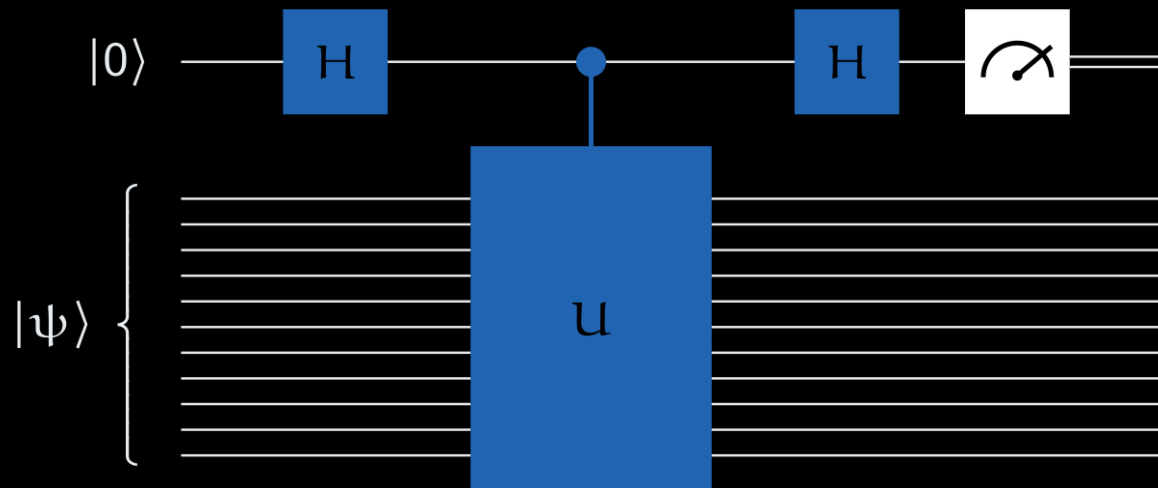


- Given:
  - A unitary quantum circuit  $U$  on  $n$  qubits
  - An  $n$ -qubit quantum state  $|\psi\rangle$
  - **Promise** that  $|\psi\rangle$  is an *eigenvector* of  $U$
- Find an  **$m$ -bit fraction** approximating  $\theta \in [0, 1)$  such that  **$U|\psi\rangle = e^{2\pi i \theta}|\psi\rangle$**

Remember that  **$e^{2\pi i \theta} = \cos(2\pi \theta) + i \sin(2\pi \theta)$**

# Controlled-U Phase Kickback

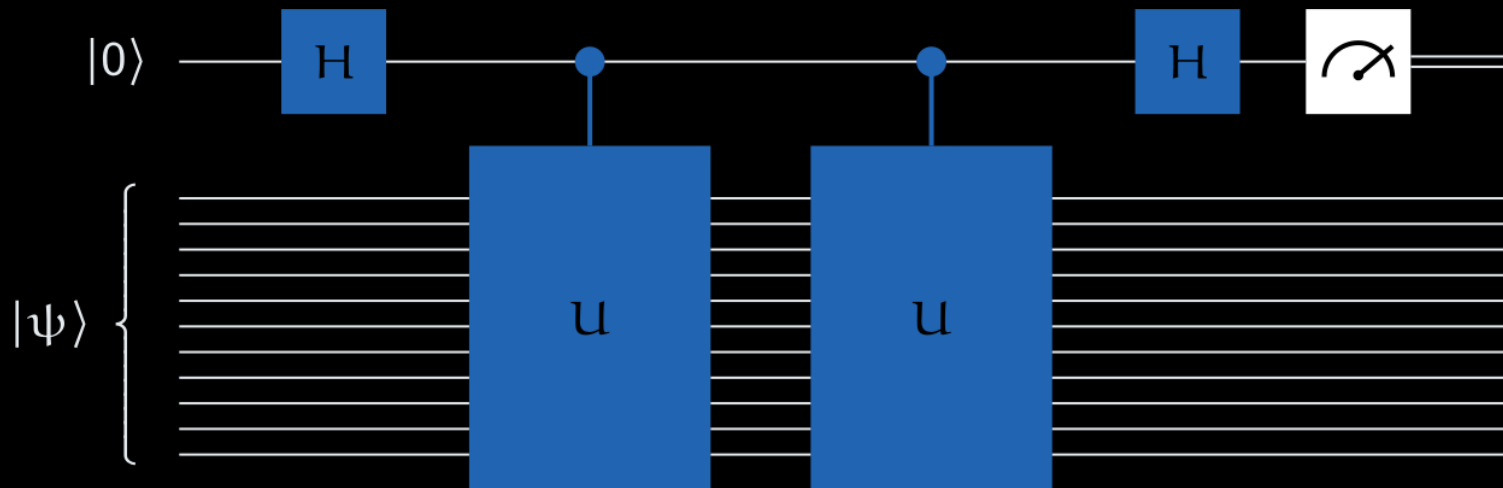
- To apply phase kickback
  - Create a controlled version of U
  - Sample it using this circuit:



- Result is  $p(0)=\cos^2(\pi\theta)$  and  $p(1)=\sin^2(\pi\theta)$

# Phase Kickback Frequency

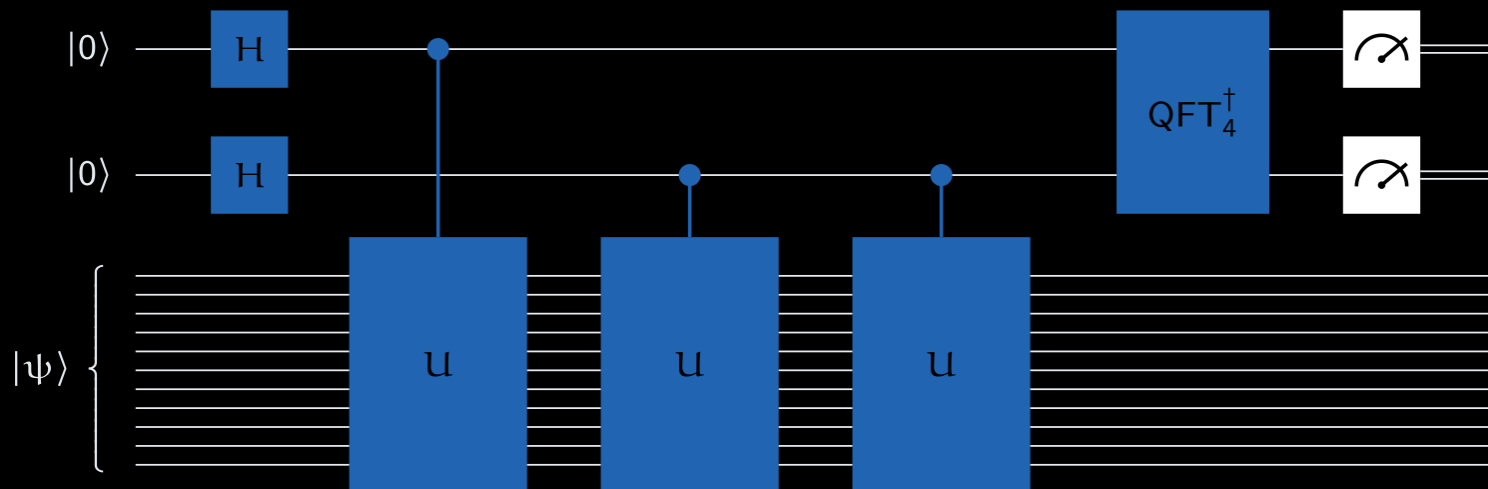
- Applying controlled-U twice
  - Squares the eigenvalue, doubles frequency
  - Sample it using this circuit:



- Want to decompose by both together...

# 2-bit Phase Estimation

- Composes the single and double applications
  - 2 qubits of phase output to sample
  - Distinguishes  $2^2$  phases



- What is that  $\text{QFT}_4$  block?

# Quantum Fourier Transform

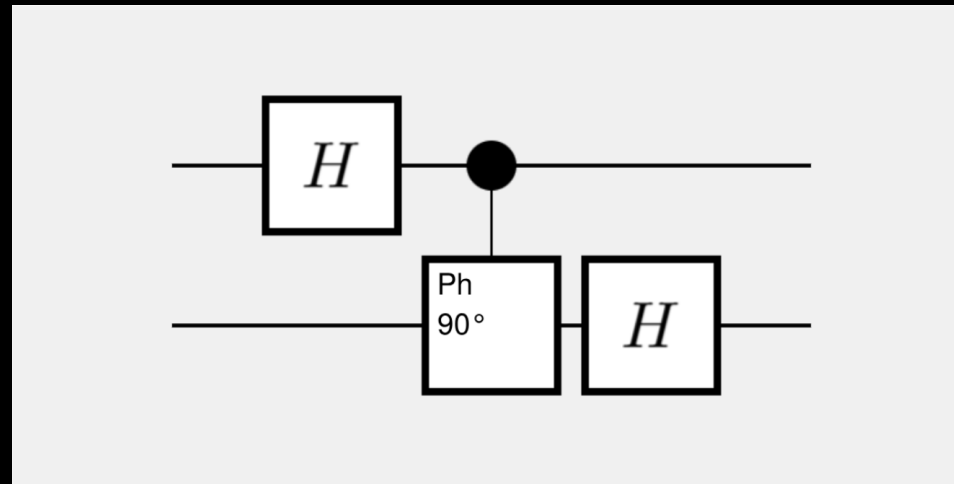
- $\text{QFT}_4$  is a unitary matrix applying the discrete Fourier transform distinguishing 4 phases

$$\text{QFT}_4 = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

- What gates implement that?

# Quantum Fourier Transform

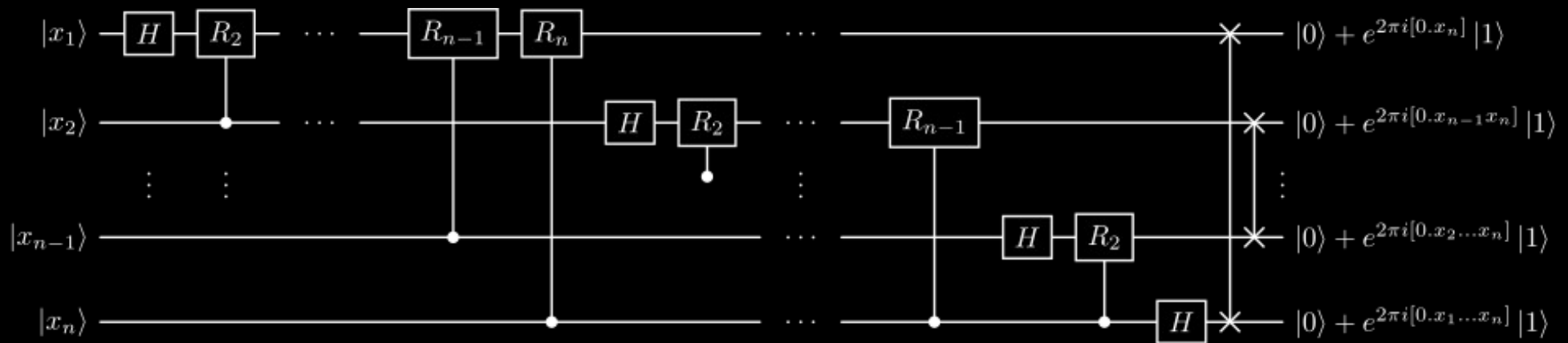
- $\text{QFT}_4$  can be built from  $H$  and controlled phase; phase gate is sometimes called  $R_k$  for  $360/2^{k^0}$



- Larger QFT have sequences of phase shifts by correspondingly smaller angles

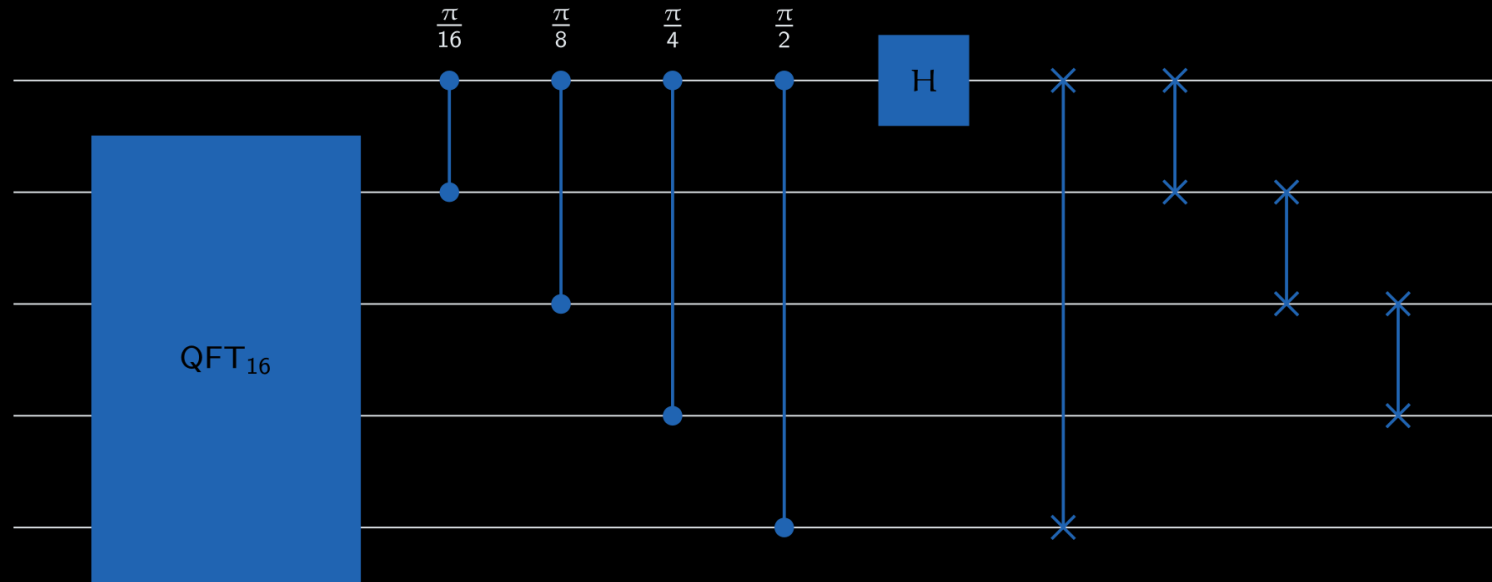
# Quantum Fourier Transform

- QFT over  $n$  qubits is:



# Quantum Fourier Transform

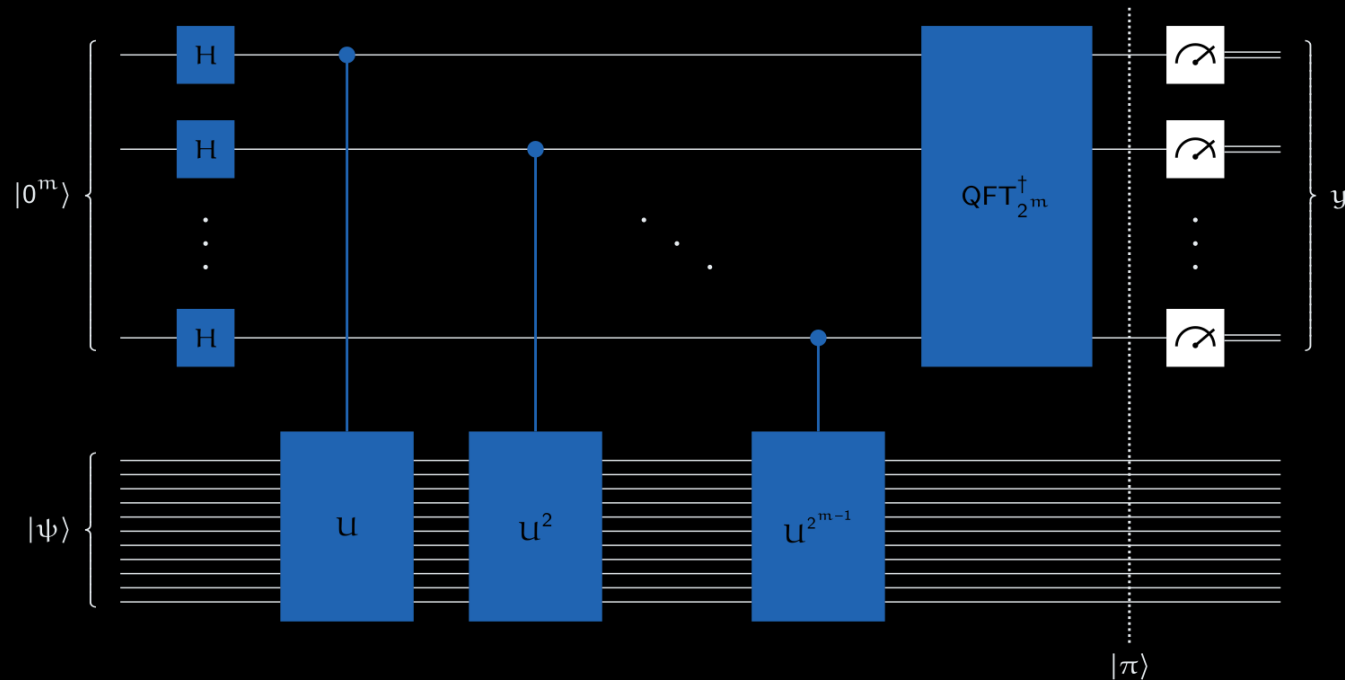
- IBM gives a recursive definition of  $\text{QFT}_{32}$ :



- Optimizations & approximations can be applied

# Phase Estimation

- Evaluation of powers of  $U$  gets expensive for estimation with many bits of precision



# Phase Estimation

- Probability of measuring the best approximation for  $\gamma$  is  $\geq 4/\pi^2 \approx 40.5\%$
- Wrong values on either side have lower probabilities
- Can filter by mode of multiple measurements...

# Order-Finding Problem

- For each positive integer  $N$ , define set  $\mathbb{Z}_N = \{0, 1, \dots, N-1\}$  and arithmetic on  $\mathbb{Z}_N \bmod N$
- Elements  $a \in \mathbb{Z}_N$  where  $\gcd(a, N) = 1$  is  $\mathbb{Z}_N^*$   
 $\mathbb{Z}_{21}^* = \{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}$
- For  $a \in \mathbb{Z}_N^* \exists$  positive integer  $k: a^k = 1$   
... the smallest such  $k$  is the order of  $a$  in  $\mathbb{Z}_N^*$

# Order-Finding Example

- For  $N=21$ :

$$\mathbb{Z}_{21}^* = \{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}$$

$$\begin{array}{cccccc} 1^1=1 & 2^6=1 & 4^3=1 & 5^6=1 & 8^2=1 & 10^6=1 \\ 11^6=1 & 13^2=1 & 16^3=1 & 17^6=1 & 19^6=1 & 20^2=1 \end{array}$$

- No efficient classical algorithm is known...  
solving this would imply an efficient algorithm  
for integer factorization

# Order-Finding by Phase Est.

- Two positive integers  $a$ ,  $b$  are **relatively prime** *iff* they have no common factors:  $\gcd(a,b)=1$
- Relative primes are often used for rehashing when there is a collision in a hash function with  $b$  buckets:

If  $\text{Hash}(v)$  collides, then try  $(\text{Hash}(v)+a)\%b$ ,  
 $(\text{Hash}(v)+a+a)\%b$ ,  $(\text{Hash}(v)+a+a+a)\%b$ , ...

All buckets of  $b$  will be tried before a repeat...

# Order-Finding by Phase Est.

- For a given  $a \in \mathbb{Z}_N^*$   
 $M_a |x\rangle = |ax\rangle$  (for each  $x \in \mathbb{Z}_N$ )

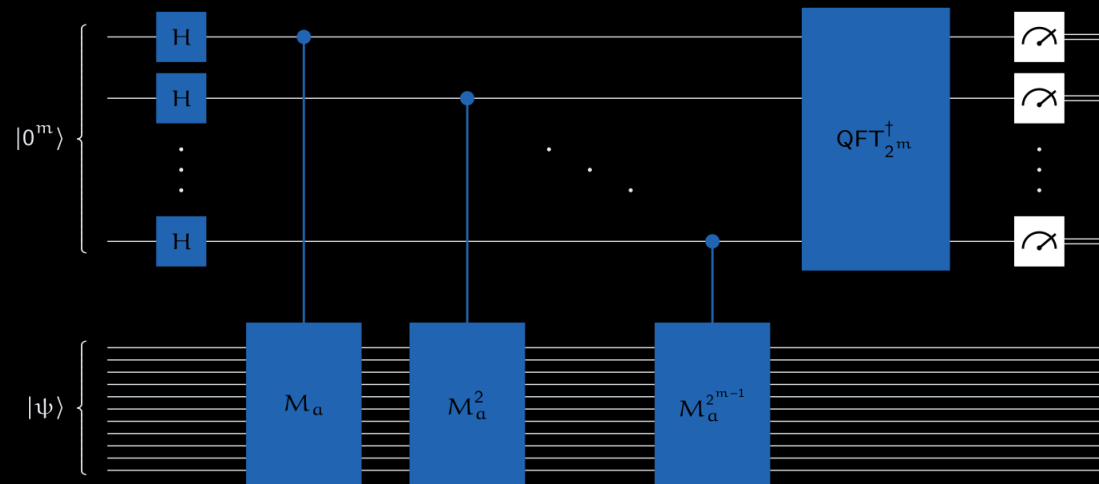
- For  $N=8$  and  $a=3$ :

$$\begin{array}{cccc} M_3 |0\rangle = |0\rangle & M_3 |1\rangle = |3\rangle & M_3 |2\rangle = |6\rangle & M_3 |3\rangle = |1\rangle \\ M_3 |4\rangle = |4\rangle & M_3 |5\rangle = |7\rangle & M_3 |6\rangle = |2\rangle & M_3 |7\rangle = |5\rangle \end{array}$$

- Because 3,8 are **relatively prime**,  $M_a |x\rangle$  is a **permutation**, so it is a **unitary matrix!**

# Order-Finding by Phase Est.

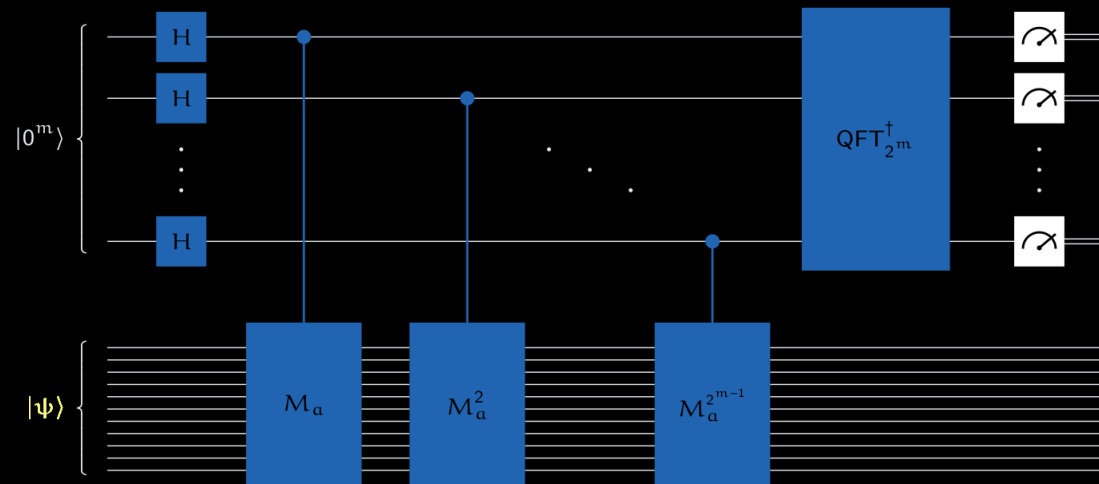
- Find order of  $a \in \mathbb{Z}_N^*$  by estimating phase of  $M_a$



- Need  $M_a^k$  for  $k$  is  $2^0, 2^1, 2^2, \dots, 2^{m-1}$   
where  $M_a^k$  is  $a^k \% N$ , computable with  $O(n^2)$  cost

# Order-Finding by Phase Est.

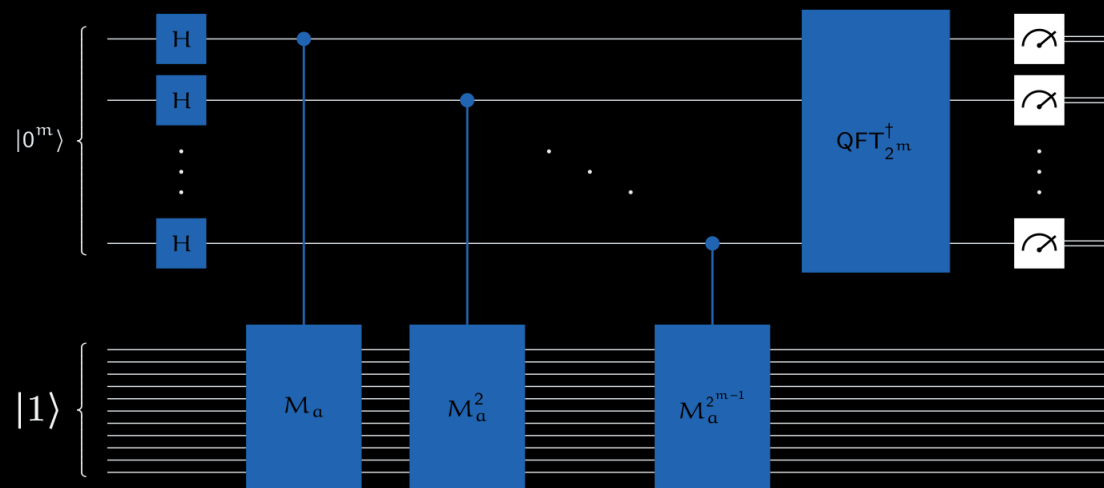
- Find order of  $a \in \mathbb{Z}_N^*$  by estimating phase of  $M_a$



- But how do I find an eigenvector  $|\psi\rangle$ ?  
We can simply use  $|1\rangle$

# Order-Finding by Phase Est.

- Total cost is  $O(n^3)$  for  $n$ -bit values



- $m$  Hadamard gates, cost  $O(n)$
- $m$  controlled unitary operations, cost  $O(n^3)$
- one Quantum Fourier transform, cost  $O(n^2)$

# Algorithm Factoring N

1. Pick random  $a \in \{2, \dots, N - 1\}$
2. Let  $d = \text{gcd}(a, N)$
3. If  $d \geq 2$ ,  $d$  is a factor and we're done
4. Let  $r = \text{order of } a \% N$
5. If  $r$  is odd, fail
6. Let  $d = \text{gcd}(a^{r/2} - 1, N)$
7. If  $d \geq 2$ ,  $d$  is a factor; else fail

# Greatest Common Divisor

- Euclidean algorithm for  $\text{gcd}(a, N)$  given  $N \geq a$

```
int
gcd(int a, int N)
{
    return (a ? gcd(N % a, a) : N);
}
```

- Quite fast on conventional hardware
- If  $\text{gcd}(a, N) \equiv 1$  then  $a, N$  are relatively prime

# Conventional Order Finding

- Sequential order(a,N) is  $O(N)$ , i.e.,  $O(2^n)$

```
int order(int a, int N) {
    int p=a;
    for (int i=1; i<N; ++i) {
        if (p == 1) return(i);
        p = (p * a) % N;
    }
    return(1); // failure
}
```

# Shor's Algorithm in bc

- Really easy, really slow for big N...

```
# recursive GCD function
define g(a,n) {
  if (a) return(g(n % a, a))
  return(n)
}

# order of a % n
define o(a,n) {
  v=a
  p=0
  for (i=1; i<n; ++i) {
    if (v==1) return(i)
    v = (v * a) % n
  }
  print "failed to find order "
  return(-1) # failed to find order
}

# power e of a mod n
define p(a,e,n) {
  q=1
  while (e > 0) {
    if (e % 2) q = (a * q) % n
    a = (a * a) % n
    e = e / 2
  }
  return(q)
}
```

```
# factor n using guess a
define f(n,a) {
  d = g(a,n)
  if (d > 1) return(d)
  r = o(a,n)
  if (r % 2) return(0)
  w = p(a,r/2,n)
  if (w < 2) return(-1)
  d = g(w-1,n)
  if (d > 1) return(d)
  return(1)
}

# factorization loop
define void factor(n) {
  print "factoring ", n, "\n"
  a = sqrt(n)
  print "starting with ", a, "\n"
  while (1) {
    print "trying ", a, ": "
    t = f(n, a)
    if (t == 0) print "odd"
    if (t == -1) print "power"
    if (t == 1) print "factor found was 1"
    if (t > 1) {
      print n, " is ", t, "*", (n/t), "\n"
      return
    }
    print "\n"
    a = (a + 1) % n
  }
}
```

```
factor(15)
factoring 15
starting with 3
trying 3: 15 is 3*5

factor(143)
factoring 143
starting with 11
trying 11: 143 is 11*13

factor(1679)
factoring 1679
starting with 40
trying 40: 1679 is 23*73

factor(200099)
factoring 200099
starting with 447
trying 447: 200099 is 499*401

factor(291311)
factoring 291311
starting with 539
trying 539: 291311 is 523*557

factor(8219999)
factoring 8219999
starting with 2867
trying 2867: 8219999 is 251*32749

factor(21428053)
factoring 21428053
starting with 4629
trying 4629: factor found was 1
trying 4630: factor found was 1
trying 4631: 21428053 is 5261*4073
```

# Algorithm Factoring N

- How often does this fail?  
Claimed less than 50% of the time!
- Very unlikely we're done in step 3, but ...
  - $a^r \% N \equiv 1 \iff (a^r - 1) \% N \equiv 0$
  - If  $r$  is even,  $(a^r - 1) = (a^{r/2} + 1)(a^{r/2} - 1)$  and a prime factor of  $N$  is likely to divide  $(a^{r/2} - 1)$

# Shor's Algorithm in `qiskit`

- Here's factoring 15 using `qiskit`:

<https://quantum.cloud.ibm.com/docs/en/tutorials/shors-algorithm>