```
        cz_and_swap(c, d, 0.125),
        cirq.H(a),
        cz_and_swap(a, b, 0.5),
        cz_and_swap(b, c, 0.25),
        cirq.H(a),
        cz_and_swap(a, b, 0.5),
        cirq.H(a),
        strategy=cirq.InsertStrategy.EARLIEST
    )

    return circuit
```

Finally, we can run this circuit by calling the main function:

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

The output of this program is shown below:

```
FinalState
[0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j
    0.25+0.j
 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j 0.25+0.j
     0.25+0.j]
```

Figure 8.4 delineates the process of applying QFT.

8.5 *Shor's Algorithm*

RSA Cryptography

Suppose Alice would like to send a private message to Bob via the internet. Alice's message could very well be intercepted by a malicious eavesdropper, Eve, along its journey. This is embarrassing, yet harmless, if Alice is sending Bob a note, but it is an issue if Alice is sending Bob her credit card number. How can we send messages securely via the internet?

Cryptography is the study of the making and breaking of secret codes. Cryptography refers to the writing of secret codes, while cryptanalysis refers to the breaking of those codes. RSA cryptography is a popular style of cryptography that allows for the secure transfer of information via the internet. RSA honors Rivest, Shah and Adelman, three pioneers in its development [188].

The core conjecture of RSA cryptography is that multiplying two large prime numbers is a *trapdoor* function; multiplying two large prime numbers is easy, yet finding the two factors after the multiplication has occurred is hard. Later in this chapter, we'll see that a fault-tolerant quantum computer will

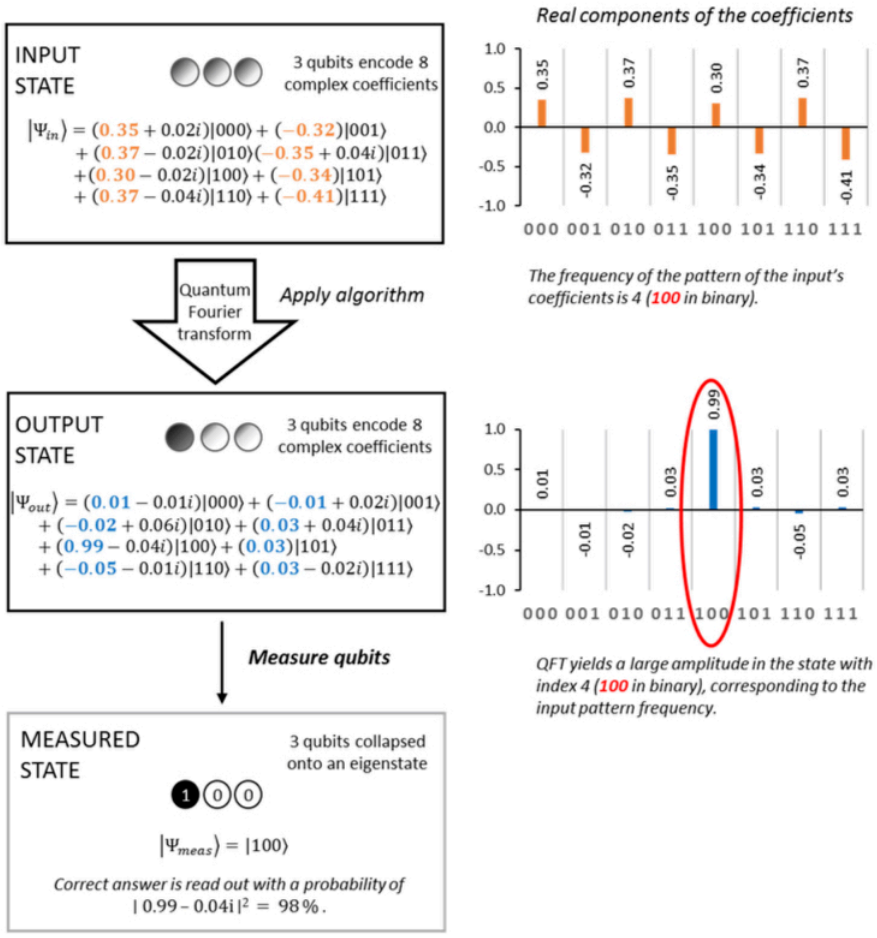


Figure 8.4: Measurement process Source: [159]

have the capacity to surmount the difficulty posed by the factorization process which will put the entire RSA scheme at risk [200]! This leads us to post-quantum cryptography, a fascinating field at the intersection of mathematics, physics and computer science.

In 1994, Peter Shor published his landmark paper establishing a quantum algorithm for the prime factorization of numbers [200]. The problem of factoring a number into primes reduces to finding a factor, since if you can find one factor, you can use it to divide the original number and consider the smaller factors. Eventually, using this divide (literally) and conquer strategy, we can completely factor the number into primes.

His technique to find a factor of any number n is to find the period, r, of a certain function f and then use the knowledge of the period of said function to find a factor of the number.

The Period of a Function

The problem of factoring the product of two large prime numbers is, in some sense, equivalent to the problem of finding the *period* of a function. To get an idea of what the period of a function might be, consider raising some number, like 2, to higher and higher powers, and then taking the result modulo a product of two prime numbers such as $91 = 13 \cdot 7$. For example, we have:

$2^0 \pmod{91}$	1
$2^1 \pmod{91}$	2
$2^2 \pmod{91}$	4
$2^3 \pmod{91}$	8
$2^4 \pmod{91}$	16
$2^5 \pmod{91}$	32
$2^6 \pmod{91}$	64
$2^7 \pmod{91}$	37
$2^8 \pmod{91}$	74
$\vdots$	$\vdots$

We see that the numbers cannot grow forever due to the modulo operation. For example, $2^6 \pmod{91} = 64$ and the next highest power $2^7 \pmod{91} = 37$.

8.4 **Exercise** Figure out if the powers of 2 ever cycle back to the number 1 in the table above. More precisely: find the smallest number n beyond 0 such that

$$2^n \pmod{91} = 1$$

Persistence pays off here. If you tried the above exercise, you found that, yes, $2^{12} \pmod{91} = 1$, and that 12 is the smallest number beyond 0 that makes this happen. Was it even obvious that it would return to 1? We refer to the number 12 as the period of the function defined by

$$f(n) := 2^n \pmod{91}$$

In general, for a function defined by

$$f(n) := a^n \pmod{N}$$

for some $a \in \{1, 2, ..., N-1\}$ relatively prime to N, the *period* of the function f is the smallest number n beyond 0 such that $f(n) = 1$ once again. In other

words, since at $n = 0$

$$a^0 \pmod{N} = 1$$

we must find the next n that again satisfies

$$a^n \pmod{N} = 1$$

A number a is *relatively prime* to N iff[4] the greatest common divisor of a and N is equal to 1, written $\gcd(a, N) = 1$. We refer to these functions as *modular* functions. This number n is also referred to as the *order* of the element a in the group $(\mathbb{Z}/N\mathbb{Z})^\times$, which denotes the multiplicative group whose underlying set is the subset of numbers in $\{1, 2, ..., N-1\}$ relatively prime to N, and whose binary operation is multiplication modulo N, as above.

8.5 **Exercise** Check that $(\mathbb{Z}/N\mathbb{Z})^\times$, whose underlying set of elements is the subset of numbers in $\{1, 2, ..., N-1\}$ relatively prime to N and whose binary operation is multiplication modulo N, is actually a group! For any number N, how many elements does the group $(\mathbb{Z}/N\mathbb{Z})^\times$ have? Can you find a pattern relating the number of elements in $(\mathbb{Z}/N\mathbb{Z})^\times$ to the number N?

The fascinating point we make now is that the difficulty you experienced finding the period of the function $f(x) = 2^n \pmod{97}$ is not unique. Even a (classical) computer would have a difficult time finding the period of this function! Peter Shor realized that we can exploit quantum computing to quickly find the period of such a function [200]. We will now explain how the period of a function can be used as an input to the factorization algorithm that would crack RSA cryptography.

Period of a Function as an Input to a Factorization Algorithm

Suppose we are asked to factor some number N, and that we know how to find the period of any modular function, as described above. Remember that the problem of factoring N reduces to the simpler problem of finding *any* factor of N. So, let's see how we could leverage our ability to find the period of a modular function to find a factor of N:

1. Choose a random number $a < N$.
2. Compute $\gcd(a, N)$ using the extended Euclidean algorithm.
3. If $\gcd(a, N) \neq 1$, i.e., a and N are not relatively prime, a is already a nontrivial factor of N, and so we are done.

[4]Note: we abbreviate *if and only if* as iff throughout the book.

4. Otherwise, find the period r of the modular function

$$f(n) := a^n \ (\text{mod} N)$$

5. If r is an odd number, or if $a^{\frac{r}{2}} = -1 \ (\text{mod} N)$, choose a new random number and start over.
6. Otherwise, classical number theory guarantees that $\gcd(a^{\frac{r}{2}} + 1, N)$ and $\gcd(a^{\frac{r}{2}} - 1, N)$ are both nontrivial factors of N.

8.6 **Exercise** Run the above algorithm to factor the number $N = 21$.

A successful approach to the exercise above is the following:

1. began with $a = 2$, since
2. $\gcd(a, N) = \gcd(2, 21) = 1$,
3. (Step 3 is omitted, since $\gcd(a, N) = \gcd(2, 21) = 1$),
4. the period of the modular function $f(n) := 2^n \ (\text{mod } 21)$ is found to be $r = 6$ and
5. $r = 6$ is neither an odd number, nor does it satisfy the equation

$$a^{\frac{r}{2}} = -1 \ (\text{mod})N,$$

6.

$$\gcd(a^{\frac{r}{2}}+1, N) = \gcd(2^{\frac{6}{2}}+1, 21) = \gcd(8+1, 21) = \gcd(9, 21) = 3$$

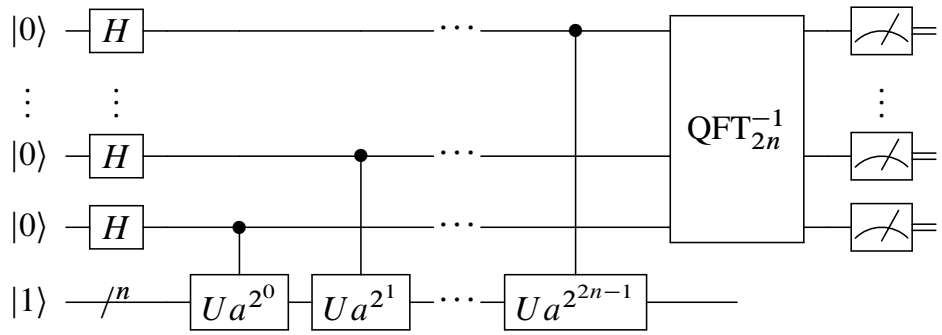
and

$$\gcd(a^{\frac{r}{2}}-1, N) = \gcd(2^{\frac{6}{2}}-1, 21) = \gcd(2^3-1, 21) = \gcd(7, 21) = 7$$

are the two nontrivial factors of $N = 21$.

We can see that being able to find the period of any modular function is the key to factoring. It is the *quantum Fourier transform (QFT)*, described earlier in this book, that allows us to find the period! Shor was inspired by Simon's algorithm and BV in his development of this algorithm. He built on the use of the QFT by BV and the period finding of Simon's approach to then arrive at his number-factoring algorithm.[5] Here is the circuit diagram for Shor's algorithm:

[5]Please see this book's GitHub site for an example of code for Simon's algorithm.



Let us now do a walkthrough of a sample encoding of Shor's algorithm. The following code comes from [234]:

```
"""
toddwildey/shors-python

@toddwildey toddwildey Implemented Shor's algorithm in Python 3.X
    using state vectors

470 lines (353 sloc) 12.1 KB
#!/usr/bin/env python

shors.py: Shor's algorithm for quantum integer factorization"""

import math
import random
import argparse

__author__ = "Todd Wildey"
__copyright__ = "Copyright 2013"
__credits__ = ["Todd Wildey"]

__license__ = "MIT"
__version__ = "1.0.0"
__maintainer__ = "Todd Wildey"
__email__ = "toddwildey@gmail.com"
__status__ = "Prototype"

def printNone(str):
  pass

def printVerbose(str):
  print(str)

printInfo = printNone


#     Quantum Components



class Mapping:
  def __init__(self, state, amplitude):
    self.state = state
    self.amplitude = amplitude
```

```
class QuantumState:
  def __init__(self, amplitude, register):
    self.amplitude = amplitude
    self.register = register
    self.entangled = {}

  def entangle(self, fromState, amplitude):
    register = fromState.register
    entanglement = Mapping(fromState, amplitude)
    try:
      self.entangled[register].append(entanglement)
    except KeyError:
      self.entangled[register] = [entanglement]

  def entangles(self, register = None):
    entangles = 0
    if register is None:
      for states in self.entangled.values():
        entangles += len(states)
    else:
      entangles = len(self.entangled[register])

    return entangles


class QubitRegister:
  def __init__(self, numBits):
    self.numBits = numBits
    self.numStates = 1 << numBits
    self.entangled = []
    self.states = [QuantumState(complex(0.0), self) for x in
        range(self.numStates)]
    self.states[0].amplitude = complex(1.0)

  def propagate(self, fromRegister = None):
    if fromRegister is not None:
      for state in self.states:
        amplitude = complex(0.0)

        try:
          entangles = state.entangled[fromRegister]
          for entangle in entangles:
            amplitude += entangle.state.amplitude *
                entangle.amplitude

          state.amplitude = amplitude
        except KeyError:
          state.amplitude = amplitude

    for register in self.entangled:
      if register is fromRegister:
        continue

      register.propagate(self)
```

```
  # Map will convert any mapping to a unitary tensor given each
      element v
  # returned by the mapping has the property v * v.conjugate() = 1
  #
  def map(self, toRegister, mapping, propagate = True):
    self.entangled.append(toRegister)
    toRegister.entangled.append(self)

    # Create the covariant/contravariant representations
    mapTensorX = {}
    mapTensorY = {}
    for x in range(self.numStates):
      mapTensorX[x] = {}
      codomain = mapping(x)
      for element in codomain:
        y = element.state
        mapTensorX[x][y] = element

        try:
          mapTensorY[y][x] = element
        except KeyError:
          mapTensorY[y] = { x: element }

    # Normalize the mapping:
    def normalize(tensor, p = False):
      lSqrt = math.sqrt
      for vectors in tensor.values():
        sumProb = 0.0
        for element in vectors.values():
          amplitude = element.amplitude
          sumProb += (amplitude * amplitude.conjugate()).real

        normalized = lSqrt(sumProb)
        for element in vectors.values():
          element.amplitude = element.amplitude / normalized

    normalize(mapTensorX)
    normalize(mapTensorY, True)

    # Entangle the registers
    for x, yStates in mapTensorX.items():
      for y, element in yStates.items():
        amplitude = element.amplitude
        toState = toRegister.states[y]
        fromState = self.states[x]
        toState.entangle(fromState, amplitude)
        fromState.entangle(toState, amplitude.conjugate())

    if propagate:
      toRegister.propagate(self)

  def measure(self):
    measure = random.random()
    sumProb = 0.0

    # Pick a state
    finalX = None
```

```
      finalState = None
      for x, state in enumerate(self.states):
        amplitude = state.amplitude
        sumProb += (amplitude * amplitude.conjugate()).real

        if sumProb > measure:
          finalState = state
          finalX = x
          break

      # If state was found, update the system
      if finalState is not None:
        for state in self.states:
          state.amplitude = complex(0.0)

        finalState.amplitude = complex(1.0)
        self.propagate()

      return finalX

    def entangles(self, register = None):
      entangles = 0
      for state in self.states:
        entangles += state.entangles(None)

      return entangles

    def amplitudes(self):
      amplitudes = []
      for state in self.states:
        amplitudes.append(state.amplitude)

      return amplitudes

  def printEntangles(register):
    printInfo("Entagles: " + str(register.entangles()))

  def printAmplitudes(register):
    amplitudes = register.amplitudes()
    for x, amplitude in enumerate(amplitudes):
      printInfo('State #' + str(x) + '\'s amplitude: ' +
          str(amplitude))

  def hadamard(x, Q):
    codomain = []
    for y in range(Q):
      amplitude = complex(pow(-1.0, bitCount(x & y) & 1))
      codomain.append(Mapping(y, amplitude))

    return codomain

  # Quantum Modular Exponentiation
  def qModExp(a, exp, mod):
    state = modExp(a, exp, mod)
    amplitude = complex(1.0)
    return [Mapping(state, amplitude)]
```

```
# Quantum Fourier Transform
def qft(x, Q):
  fQ = float(Q)
  k = -2.0 * math.pi
  codomain = []

  for y in range(Q):
    theta = (k * float((x * y) % Q)) / fQ
    amplitude = complex(math.cos(theta), math.sin(theta))
    codomain.append(Mapping(y, amplitude))

  return codomain
```

Now that we have defined functions for entanglement and QFT, we can define the core period-finding function. Recall that this is the key subroutine that must run on quantum hardware.

```
def findPeriod(a, N):
  nNumBits = N.bit_length()
  inputNumBits = (2 * nNumBits) - 1
  inputNumBits += 1 if ((1 << inputNumBits) < (N * N)) else 0
  Q = 1 << inputNumBits

  printInfo("Finding the period...")
  printInfo("Q = " + str(Q) + "\ta = " + str(a))

  inputRegister = QubitRegister(inputNumBits)
  hmdInputRegister = QubitRegister(inputNumBits)
  qftInputRegister = QubitRegister(inputNumBits)
  outputRegister = QubitRegister(inputNumBits)

  printInfo("Registers generated")
  printInfo("Performing Hadamard on input register")

  inputRegister.map(hmdInputRegister, lambda x: hadamard(x, Q),
      False)
  # inputRegister.hadamard(False)

  printInfo("Hadamard complete")
  printInfo("Mapping input register to output register, where f(x)
      is a^x mod N")

  hmdInputRegister.map(outputRegister, lambda x: qModExp(a, x, N),
      False)

  printInfo("Modular exponentiation complete")
  printInfo("Performing quantum Fourier transform on output
      register")

  hmdInputRegister.map(qftInputRegister, lambda x: qft(x, Q), False)
  inputRegister.propagate()

  printInfo("Quantum Fourier transform complete")
  printInfo("Performing a measurement on the output register")
```

```
  y = outputRegister.measure()

  printInfo("Output register measured\ty = " + str(y))

  # Interesting to watch - simply uncomment
  # printAmplitudes(inputRegister)
  # printAmplitudes(qftInputRegister)
  # printAmplitudes(outputRegister)
  # printEntangles(inputRegister)

  printInfo("Performing a measurement on the periodicity register")

  x = qftInputRegister.measure()

  printInfo("QFT register measured\tx = " + str(x))

  if x is None:
    return None

  printInfo("Finding the period via continued fractions")

  r = cf(x, Q, N)

  printInfo("Candidate period\tr = " + str(r))

  return r
```

Now we can define the functions that will run on classical hardware.

```
BIT_LIMIT = 12

def bitCount(x):
  sumBits = 0
  while x > 0:
    sumBits += x & 1
    x >>= 1

  return sumBits

# Greatest Common Divisor
def gcd(a, b):
  while b != 0:
    tA = a % b
    a = b
    b = tA

  return a

# Extended Euclidean
def extendedGCD(a, b):
  fractions = []
  while b != 0:
    fractions.append(a // b)
    tA = a % b
```

```
    a = b
    b = tA

  return fractions

# Continued Fractions
def cf(y, Q, N):
  fractions = extendedGCD(y, Q)
  depth = 2

  def partial(fractions, depth):
    c = 0
    r = 1

    for i in reversed(range(depth)):
      tR = fractions[i] * r + c
      c = r
      r = tR

    return c

  r = 0
  for d in range(depth, len(fractions) + 1):
    tR = partial(fractions, d)
    if tR == r or tR >= N:
      return r

    r = tR

  return r

# Modular Exponentiation
def modExp(a, exp, mod):
  fx = 1
  while exp > 0:
    if (exp & 1) == 1:
      fx = fx * a % mod
    a = (a * a) % mod
    exp = exp >> 1

  return fx

def pick(N):
  a = math.floor((random.random() * (N - 1)) + 0.5)
  return a

def checkCandidates(a, r, N, neighborhood):
  if r is None:
    return None

  # Check multiples
  for k in range(1, neighborhood + 2):
    tR = k * r
    if modExp(a, a, N) == modExp(a, a + tR, N):
      return tR

  # Check lower neighborhood
```

```
  for tR in range(r - neighborhood, r):
    if modExp(a, a, N) == modExp(a, a + tR, N):
      return tR

  # Check upper neighborhood
  for tR in range(r + 1, r + neighborhood + 1):
    if modExp(a, a, N) == modExp(a, a + tR, N):
      return tR

  return None
```

Now we are ready to define the function that will call all the other functions we have created. This function will iteratively test to see if the period has been found.

```
def shors(N, attempts = 1, neighborhood = 0.0, numPeriods = 1):
  if(N.bit_length() > BIT_LIMIT or N < 3):
    return False

  periods = []
  neighborhood = math.floor(N * neighborhood) + 1

  printInfo("N = " + str(N))
  printInfo("Neighborhood = " + str(neighborhood))
  printInfo("Number of periods = " + str(numPeriods))

  for attempt in range(attempts):
    printInfo("\nAttempt #" + str(attempt))

    a = pick(N)
    while a < 2:
      a = pick(N)

    d = gcd(a, N)
    if d > 1:
      printInfo("Found factors classically, re-attempt")
      continue

    r = findPeriod(a, N)

    printInfo("Checking candidate period, nearby values, and
        multiples")

    r = checkCandidates(a, r, N, neighborhood)

    if r is None:
      printInfo("Period was not found, re-attempt")
      continue

    if (r % 2) > 0:
      printInfo("Period was odd, re-attempt")
      continue

    d = modExp(a, (r // 2), N)
```

```
    if r == 0 or d == (N - 1):
      printInfo("Period was trivial, re-attempt")
      continue

    printInfo("Period found\tr = " + str(r))

    periods.append(r)
    if(len(periods) < numPeriods):
      continue

    printInfo("\nFinding least common multiple of all periods")

    r = 1
    for period in periods:
      d = gcd(period, r)
      r = (r * period) // d

    b = modExp(a, (r // 2), N)
    f1 = gcd(N, b + 1)
    f2 = gcd(N, b - 1)

    return [f1, f2]

  return None
```

Finally, we define various flags for command line functionality.

```
def parseArgs():
  parser = argparse.ArgumentParser(description='Simulate Shor\'s
      algorithm for N.')
  parser.add_argument('-a', '--attempts', type=int, default=20,
      help='Number of quantum attempts to perform')
  parser.add_argument('-n', '--neighborhood', type=float,
      default=0.01, help='Neighborhood size for checking candidates
      (as percentage of N)')
  parser.add_argument('-p', '--periods', type=int, default=2,
      help='Number of periods to get before determining least common
      multiple')
  parser.add_argument('-v', '--verbose', type=bool, default=True,
      help='Verbose')
  parser.add_argument('N', type=int, help='The integer to factor')
  return parser.parse_args()

def main():
  args = parseArgs()

  global printInfo
  if args.verbose:
    printInfo = printVerbose
  else:
    printInfo = printNone

  factors = shors(args.N, args.attempts, args.neighborhood,
      args.periods)
  if factors is not None:
```

```
    print("Factors:\t" + str(factors[0]) + ", " + str(factors[1]))

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

So there it is — the famous Shor's algorithm. While we do not yet have the fault-tolerant hardware to run Shor's for any meaningfully large key, it is illustrative of the potential of quantum computing. Although Shor's algorithm is proven to run in polynomial time (i.e., polynomial in the number of bits in the integer to be factored), much work can be done to reduce the constant factors in this polynomial and overall resource requirements. See Gidney and Ekera's work [95] for a discussion of resource requirements in Shor's algorithm.

An example of using this program to factor 15 is shown below. This code (saved in a Python module called "shor.py" on this book's website) is set up to be run from a command line with the number to be factored as an argument. By executing

```
python shor.py 15
```

we see the following output:

```
N = 15
Neighborhood = 1
Number of periods = 2

Attempt #0
Finding the period...
Q = 256 a = 8
Registers generated
Performing Hadamard on input register
Hadamard complete
Mapping input register to output register, where f(x) is a^x mod N
Modular exponentiation complete
Performing quantum Fourier transform on output register
Quantum Fourier transform complete
Performing a measurement on the output register
Output register measured y = 1
Performing a measurement on the periodicity register
QFT register measured x = 192
Finding the period via continued fractions
Candidate period r = 4
Checking candidate period, nearby values, and multiples
Period found r = 4

Attempt #1
Found factors classically, re-attempt

Attempt #2
Found factors classically, re-attempt
```

```
Attempt #3
Finding the period...
Q = 256 a = 2
Registers generated
Performing Hadamard on input register
Hadamard complete
Mapping input register to output register, where f(x) is a^x mod N
Modular exponentiation complete
Performing quantum Fourier transform on output register
Quantum Fourier transform complete
Performing a measurement on the output register
Output register measured y = 2
Performing a measurement on the periodicity register
QFT register measured x = 128
Finding the period via continued fractions
Candidate period r = 2
Checking candidate period, nearby values, and multiples
Period found r = 4

Finding least common multiple of all periods
Factors: 5, 3
```

Here, we see that the quantum part of Shor's algorithm is executed four times (labeled "'Attempt #1" through "Attempt #4"). In two of the attempts, the circuit succeeds in finding the period, while in the other two, the factors are found classically by virtue of good luck, so the program re-attempts the quantum part. After finding the period twice, the classical part of Shor's algorithm ensues, in which the least common multiple of all periods found is computed. From this, the prime factors are determined correctly as 3 and 5.

8.6 *Grover's Search Algorithm*

In 1996, Lov Grover demonstrated that we can obtain a quadratic speedup in algorithmic search on a quantum computer compared with a classical one [98]. While this is not exponential speedup, it is still significant.

The search problem can be set up as follows. Given a function $f(x)$ such that $f(a^*) = -1$ and all other outputs of the function are 1, find a^*. In other words, we are looking for an exhaustive search algorithm; in particular, we are seeking an algorithmic search protocol. An algorithmic search protocol is one in which we can verify that we have found the item in question by evaluating a function on the search result. So we have exhausted the possibility of finding an analytic approach and now must do a brute force search.

On a classical computer, this would entail an exhaustive search using n operations for some range of $x = \{0, n\}$, or at best $\frac{n}{2}$ steps if we posit that on