Build your own quantum computers, Part 2: N-qubit projects and Shor's algorithm [1]

D. Candela

November 28, 2016

[1] These handouts were created for the honors colloquium for Physics 424. The material in Parts 1 and 2 can also be found in the published article: American Journal of Physics **83**, 688 (2015). Equations before Eq. 34 and figures before Fig. 7 are in Part 1, also in the AJP article.

Increasing the number of qubits

The projects in Part 1 all used $N = 3$ qubits, and required negligible time and memory to simulate on a classical computer. To understand how things scale with N, and also to try out Shor's algorithm, you need more qubits.

The challenge of making N as large as possible is addressed in two stages. First the techniques of Part 1 are extended to $N > 3$ simply by using bigger vectors and matrices. On a 2012-era PC, it was possible to reach $N = 12$ before the matrices became too large to store in the computer. Next, sparse-matrix techniques are used to increase N further by taking advantage of the many zeros in the matrices.

Matrices for gates with an arbitrary number of qubits

For $N > 3$ we need the computer program to generate the $2^N \times 2^N$ matrices for gates. This section gives the algorithm to carry this out.

Let p and q be binary digits, each 0 or 1. A 2×2 matrix M can be represented in the form $M_{p,q}$,

$$M = \begin{bmatrix} M_{0,0} & M_{0,1} \\ M_{1,0} & M_{1,1} \end{bmatrix}. \tag{34}$$

Easy exercise: Show that the 2×2 identity matrix is $I_{p,q} = \delta_{pq}$, the Kronecker-δ.

Using $N = 3$ to illustrate, let $[pqr]$ be a 3-digit binary number, with each of p, q, and r either 0 or 1. Using this notation the tensor product of Eq. 19 can be translated as follows:

$$\begin{aligned} \widehat{H}^{(2)} &= \widehat{I} \otimes \widehat{H} \otimes \widehat{I} \quad \text{is computed as} \\ H^{(2)}_{[pqr],[p'q'r']} &= \delta_{pp'} H_{q,q'} \delta_{rr'} \end{aligned} \tag{35}$$

The Hadamard matrix operates on the indices q, q' corresponding to the second qubit, while Kronecker-δ's give the identity operators for all the other qubits.

Here is an example: Let us assume matrix indices start from zero so the element in the seventh row and fifth column of $\widehat{H}^{(2)}$ is $H^{(2)}_{6,4}$. Using Eq. 35 this element is computed as

$$H^{(2)}_{6,4} = H^{(2)}_{[110],[100]} = \delta_{11} H_{1,0} \delta_{00} = H_{1,0} = \tfrac{1}{\sqrt{2}} \tag{36}$$

in agreement with Eq. 22. Notice how 6 and 4 have been written in binary as 110 and 100.

Matrix indices start at zero in Python and C++, but they start at one in MATLAB and Mathematica. Therefore in MATLAB this would be the $(7,5)$ element of the matrix, not the $(6,4)$ element as it would be in Python. But we still must use $([110],[100])$ (i.e $(6,4)$ in binary) in Eq. 36. See Appendix for more discussion.

Exercise: Use the bottom line of Eq. 35 to write out the 8×8 matrix for $\widehat{H}^{(2)}$ and check that it agrees with Eq. 22. It helps to write $[pqr]$ ranging from [000] to [111] down the side of the 8×8 matrix, and $[p'q'r']$ ranging from [000] to [111] across the top.

Shor's algorithm will also require two-qubit gates. Consider a CNOT gate in an $N = 3$ system in which qubit 2 controls qubit 1, as used in Fig. 6(b). The equivalent of Eq. 35 is

$$C^{(2,1)}_{\mathrm{NOT}[pqr],[p'q'r']} = C_{\mathrm{NOT}[qp],[q'p']}\delta_{rr'}. \tag{37}$$

As with Eq. 35, we have Kronecker-δ's for every qubit that the CNOT does *not* operate on, in this case only the third qubit with indices r, r'. Since the second qubit with indices q, q' is the controlling qubit for the CNOT, it is used as the *first* index for the C_{NOT} matrix. For example, the element in the seventh row and third column of $\widehat{C}^{(2,1)}_{\mathrm{NOT}}$ which is $C^{(2,1)}_{\mathrm{NOT}\ 6,2}$ is

Again this is written for matrix indices starting at zero as in Python.

$$\begin{aligned} C^{(2,1)}_{\mathrm{NOT}\ 6,2} &= C^{(2,1)}_{\mathrm{NOT}[110],[010]} \\ &= C_{\mathrm{NOT}[11],[10]}\delta_{00} = C_{\mathrm{NOT}\ 3,2} = 1 \end{aligned} \tag{38}$$

in agreement with Eq. 33. As above 6 and 2 have been written in binary as 110 and 010, but now after the correct bits have been picked out according to Eq. 37 the binary numbers 11 and 10 are converted back to 3 and 2 to index the $\widehat{C}_{\mathrm{NOT}}$ matrix, Eq. 30.

Programming project 5: Handle an arbitrary number of qubits.

Program Grover's search for an arbitrary number of qubits N. Check that the optimum number of Grover iterations is $(\pi/4)\sqrt{2^N}$, for $N > 3$. Then increase N as much as possible.

Table 1 shows results obtained on a 2012-era PC with 8 GB of RAM. Each additional qubit increased both the memory needed and the time to complete the calculation by a factor of about four. With 13 qubits, the memory of the PC was exceeded.

qubits N	Memory to store matrices	Time to compute matrices	Time to do Grover calculation
10	104 MB	53 s	4.3 s
11	450 MB	3.9 min	37 s
12	1.96 GB	17.6 min	3.8 min
13	**Failed**		

Table 1: Grover's algorithm with full matrices. All of the calculations in this handout were done using Mathematica, which is slow when handling very large arrays. So, you may find shorter computation times using a different programming language.

Using Sparse matrices

Most of the elements in the matrices of Eqs. 21-28 are zero. Matrices like this that have only a small fraction of non-zero elements are

called *sparse matrices*. It is wasteful of computer memory and time to store and compute with all these zeros. Some computer languages have sparse-matrix functions built in. This section explains how to implement sparse matrices without built-in functions, but either route can be pursued to increase N.

The $2^N \times 2^N$ matrix for a one-qubit gate (Eqs. 21-23) has at most two nonzero elements per row. Similarly, the $2^N \times 2^N$ matrix for a two-qubit gate has at most four nonzero elements per row. Therefore the Hadamard matrix of Eq. 21 can be stored as

$$\widehat{H}^{(1)} \leftrightarrow \begin{bmatrix} (0, \frac{1}{\sqrt{2}}),\ (4, \frac{1}{\sqrt{2}}) \\ (1, \frac{1}{\sqrt{2}}),\ (5, \frac{1}{\sqrt{2}}) \\ \cdots \\ \cdots \\ \cdots \\ \cdots \\ \cdots \\ (3, \frac{1}{\sqrt{2}}),\ (7, -\frac{1}{\sqrt{2}}) \end{bmatrix}. \tag{39}$$

N-qubit CNOT gates (Eqs. 32-33) are *permutation matrices*, with a single 1 in each row. A CNOT can be stored as in Eq. 39, or even more compactly as a one-dimensional array of integers giving the columns where the 1's are.

The first row of Eq. 39 indicates that the nonzero elements in row 0 are $\frac{1}{\sqrt{2}}$ in column 0 and $\frac{1}{\sqrt{2}}$ in column 4. (It may be more convenient to store the integer column indices and complex matrix elements in two separate arrays.) A *diagonal* matrix can be stored even more compactly, if desired, since it is known that the column index equals the row index for the non-zero elements.

In MATLAB these would be row 1, columns 1 and 5.

Programming project 6: Use sparse matrices to increase N.

Write functions to generate sparse matrix structures, or else figure out how to use built-in sparse matrix functions. Then carry out Grover's algorithm with N as large as possible.

Table 2 shows results obtained with the same PC as used for Table 1. Both the amount of memory used and the time needed to compute the matrices were drastically less than when full matrices were used. Extrapolating from this table, the storage limit of 8 GB would be reached at $N = 21$ and the calculation would take a week to finish.

qubits N	Memory to store matrices	Time to compute matrices	Time to do Grover calculation
10	1.82 MB	0.172 s	3.6 s
11	4.4 MB	0.48 s	11.4 s
12	9.9 MB	1.54 s	34 s
13	22 MB	5.4 s	1.78 min
14	46 MB	23 s	5.4 min

Table 2: Grover's algorithm with sparse matrices

Shor's quantum factoring algorithm

Pick two large prime numbers P_1, P_2 and compute their product $C = P_1 P_2$, for example $241 \times 683 = 164603$. Commonly used methods of data encryption depend on the fact that it takes a lot of computer power to find the factors P_1, P_2 if only C is known (using bigger numbers than this example, of course). Shor's algorithm[2] can factor a large composite (i.e. not prime) number C in far fewer steps than any known classical algorithm, potentially rendering some current encryption methods useless.

[2] Wikipedia article *Shor's algorithm*; and Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. 1995. URL `http://arxiv.org/abs/quant-ph/9508027`

Shor's algorithm step by step

To state Shor's algorithm, these two concepts are needed:

- The *greatest common divisor* of two integers $\gcd(p, q)$ is the largest integer that divides both p and q with zero remainder. For example, $\gcd(12, 18) = 6$ and $\gcd(12, 35) = 1$.
- The *modular congruence* $p \equiv q \pmod{C}$ means $p - q$ is an integer multiple of C. For example, $35 \equiv 13 \pmod{11}$. This can also be written $p \pmod{C} = q \pmod{C}$ where "$p \pmod{C}$" means the *remainder* when p is divided by C. So $35 \pmod{11} = 2$, and $13 \pmod{11} = 2$.

Given a composite integer C, the following steps will find a nontrivial factor of C (nontrivial means other than 1 or C):

See Nielsen and Chuang or the Wikipedia article on Shor's algorithm for a proof. The branch of mathematics that studies primes, factoring, and other aspects of the integers is called *number theory*. It is well known from number theory that steps 1-6 factor the number C. Shor's contribution was to prove that step 4 can be done much more quickly on a quantum computer than classically.

1. Check that C is odd and not a power of some smaller integer. If C is even or a power, then a factor of C has been found and we are done.
2. Pick any integer a in the range $1 < a < C$.
3. Find $\gcd(a, C)$. If by luck the gcd is greater than 1, then a factor of C has been found (the gcd) and we are done.
4. Find the smallest integer $p > 1$ such that $a^p \equiv 1 \pmod{C}$.
5. If p is odd, or if p is even and $a^{p/2} \equiv -1 \pmod{C}$, go back to step 2 and pick a different a.
6. The numbers $P_\pm = \gcd(a^{p/2} \pm 1, C)$ are nontrivial factors of C.

Example: $C = 15$

1. 15 is odd and is not a power of a smaller integer, so proceed.
2. Arbitrarily pick $a = 7$.
3. $\gcd(7, 15) = 1$, so proceed.

4. Try p's starting with 2:
   - $7^2 = 49$ and $49 \pmod{15} = 4 \neq 1$.
   - $7^3 = 343$ and $343 \pmod{15} = 13 \neq 1$.
   - $7^4 = 2401$ and $2401 \pmod{15} = 1$, so the result of this step is $p = 4$.
5. $p = 4$ is even, and $7^{4/2} = 49$. Since $49 \not\equiv -1 \pmod{15}$, it is not necessary to go back and pick a different a.
6. $P_+ = \gcd(50, 15) = 5$ and $P_- = \gcd(48, 15) = 3$ are the sought-for factors of 15.

One should imagine implementing Shor's algorithm for a very large number C. A *classical* computer can quickly determine if C is a power, and quickly compute the gcd of two large numbers using Euclid's algorithm. So the only thing for which a quantum computer is needed is step 4.

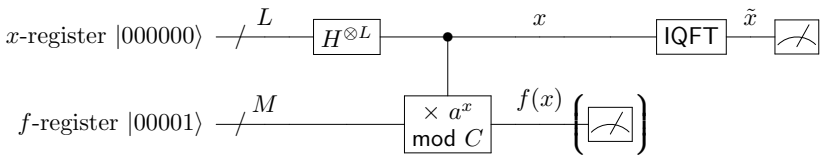

Figure 7: The quantum circuit for the period-finding part of Shor's algorithm. The top line marked with a slash and L denotes a group of L qubits. Similarly the bottom line denotes a group of M qubits.

The quantum part of Shor's algorithm: period-finding

Step 4 is called "period finding", because the function $f(x) = a^x \pmod{C}$ is periodic with period p. In the example $C = 15$, $a = 7$, $f(x)$ has period 4:

$$\begin{aligned} f(0) &= 7^0 \pmod{15} = 1 \\ f(1) &= 7^1 \pmod{15} = 7 \\ f(2) &= 7^2 \pmod{15} = 4 \\ f(3) &= 7^3 \pmod{15} = 13 \\ f(4) &= 7^4 \pmod{15} = 1 \\ f(5) &= 7^5 \pmod{15} = 7 \\ f(6) &= 7^6 \pmod{15} = 4 \\ &\vdots \end{aligned} \tag{40}$$

Fig. 7 shows the quantum circuit used to find p. The qubit register is divided into two parts, the x-register with L qubits initialized to the state $|0 \ldots 0\rangle$ and the f-register with M qubits initialized to the state $|0 \ldots 01\rangle$. The steps in the calculation are:

1. Apply a Hadamard gate to each of the L qubits in the x-register, as indicated by the box with $H^{\otimes L}$. This puts the x-register in a superposition of all 2^L possible x values.

2. Based on the value in the x-register, multiply the f-register by $a^x \pmod C$. This leaves the f-register containing $f(x)$.

3. Measure the f-register. It doesn't matter if the f-register is measured at this point, or later when the x-register is measured, or never, but the explanation of Fig. 7 is simpler if the f-register is measured as shown.

4. Perform an inverse quantum Fourier transform (IQFT) on the x-register. Since the Fourier transform of a function has peaks at the frequencies present in the function, the IQFT will make it possible to find the period = 1/frequency of $f(x)$. The IQFT is discussed in more detail below.

5. Measure the output $\tilde{x}$ of the IQFT. Shor proved that the measured $\tilde{x}/2^L \approx s/p$, where s is an unknown integer. This information is used to find p. For example if $\tilde{x}/2^L = 0.32 \approx \frac{1}{3} = \frac{2}{6} = \frac{3}{9} \ldots$ it can be guessed that p is one of 3, 6, 9.... These are checked to see which one satisfies $a^p \equiv 1 \pmod C$, and is therefore the true period p.

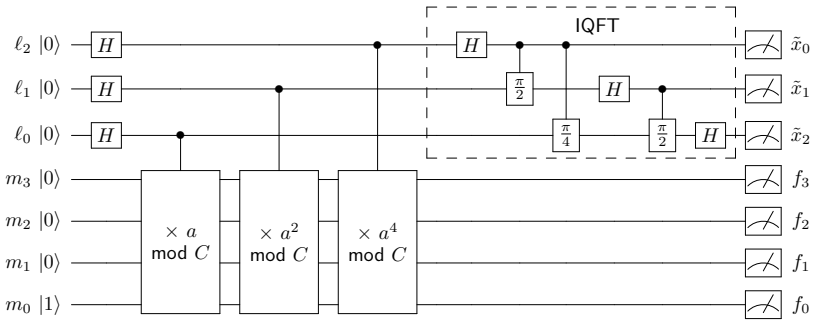

Figure 8: Quantum circuit to implement Shor's algorithm with $N = 7$ qubits.

Shor's algorithm with seven qubits

The smallest numbers satisfying step 1 of the algorithm (odd composite integers that are not powers of another integer) are $C = 15, 21, 33, 35, 39, \ldots$. To date, physical quantum computers using Shor's algorithm have factored the numbers 15 and 21. Vandersypen, et al.[3] made physical quantum computers with $N = 7$ using nuclear spins as the qubits. They used Shor's algorithm to factor $C = 15$ using $L = 3$ and $M = 4$.

[3] Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Constantino S. Yannoni, Mark H. Sherwood, and Isaac L. Chuang. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883–887, 2001

Figure 8 shows the quantum circuit for the $N = 7$ Shor's-algorithm quantum computation, based on the design of Vandersypen et al. In addition to things like Hadamard gates familiar from earlier projects there are two new elements:

- There are three *controlled phase shift gates*, all in the IQFT block. These are phase shift gates as used earlier, but now controlled by another qubit.
- There are three quantum gates using the x-register bits ℓ_2, ℓ_1, ℓ_0 to control the f-register bits m_3, m_2, m_1, m_0. The gate controlled by bit ℓ_k conditionally multiplies the the f-register by $a^{2^k} \pmod{15}$. Together the three gates multiply the f-register by $a^{\ell_0+2\ell_1+4\ell_2} \pmod{15} = a^x \pmod{15} = f(x)$, as required for step 2 of the period-finding calculation.

In physical quantum computers gates typically operate on no more than 1-3 qubits at a time, making this part of the period-finding calculation more complicated than the IQFT. As discussed in Part 3 of these handouts, it takes at least $17 \times (40L^3)$ one- and two-qubit gates to carry out the functions of these gates.

Table 3: Usable a values for $C = 15$

a	p	$f(x)$ values
2	4	1, 2, 4, 8
4	2	1, 4
7	4	1, 7, 4, 13
8	4	1, 8, 4, 2
11	2	1, 11
13	4	1, 13, 4, 7
14	2	1, 14

Programming project 7: Implement Shor's algorithm with seven qubits.

Set up the quantum computation diagrammed in Fig. 8, and use it to factor $C = 15$. The computation should work for all values of a that pass steps 2 and 3 of Shor's algorithm as listed in Table 3. The measured values of the seven qubits labeled on the right side of Fig. 8 should be used to print out the two numbers $\tilde{x}/2^L = (\tilde{x}_2\tilde{x}_1\tilde{x}_0)/2^L$ and $f = (f_3f_2f_1f_0)$. For example, if the measured output is $|0110100\rangle$ then $f = 0100_2 = 4$ from the last four qubits and $\tilde{x} = 110_2 = 6$ from the first three qubits in reversed order, so $\tilde{x}/2^L = 6/8 = 0.75$. Here's what should happen:

- The measured f should always be one of the p different values listed in Table 3. For example when $a = 7$, f should vary randomly among 1, 7, 4, and 13. This is just a check, as the f value is not used.
- The measured $\tilde{x}/2^L$ should come out close to s/p for some integer s. This is the output of the program, used to find the period p.

For this project it is necessary to compute $2^7 \times 2^7$ matrices for the new gates in Fig. 8:

Controlled phase-shift gates The 4×4 controlled phase-shift gate is given by Eq. 31 with the phase shift matrix (Eq. 16) used as the U submatrix. Then the full 128×128 matrix is constructed using an equation like Eq. 37.

Gates to compute $f(x)$ The 128×128 matrices for the gates that compute $f(x)$ are *permutation matrices*, which means that each column is all 0's except for one 1, and the 1 is in a different row in each column (like Eqs. 32-33). Using $C = 15$:

1. Compute $A_0 = a \pmod{15}$, $A_1 = a^2 \pmod{15}$, and $A_2 = a^4 \pmod{15}$.
2. Start with the first controlled gate in Fig. 8, controlled by ℓ_0. In each column $k = 0 \ldots 127$ of the matrix, the entries are all 0's except for a 1 in some row j. To compute j:
   (a) Write the column k as a 7-bit binary number $k = \ell_2\ell_1\ell_0 m_3 m_2 m_1 m_0$ as on the left side of Fig. 8.
   (b) If $\ell_0 = 0$, then $j = k$. This puts the 1 in column k on the diagonal, as in an identity matrix. (Conceptually if $\ell_0 = 0$, the gate does nothing to the f-register.)
   (c) If $\ell_0 = 1$, then express the four-bit binary number $m_3 m_2 m_1 m_0$ as an integer f. If $f \geq 15$ (i.e. if $f \geq C$), then again set $j = k$.
   (d) If $\ell_0 = 1$ and $f < 15$, compute a new f value $f' = A_0 f \pmod{15}$. Write this in binary as $f' = m_3' m_2' m_1' m_0'$. Then j is given by the 7-bit binary number $j = \ell_2\ell_1\ell_0 m_3' m_2' m_1' m_0'$. (Conceptually if $\ell_0 = 1$ then the f-register is multiplied by $A_0 \pmod{15}$.)
3. Use the same procedure to compute the other two controlled gates, substituting ℓ_1, ℓ_2 for ℓ_0 and A_1, A_2 for A_0.

This is written for Python or C++. For MATLAB or Mathematica, subtract one from the column index to get k, compute j from k as specified here, and then add one to j to get the row index where the 1 should be placed.

Optional exercise: (a) Show that a permutation matrix as defined above is unitary. (b) (subtle) Show that the procedure given above results in a permutation matrix, and therefore a unitary matrix.

$\tilde{x}$	$\tilde{x}/2^L$	Measurements with this result
0	0.0	27
1	0.125	0
2	0.25	25
3	0.375	0
4	0.5	30
5	0.625	0
6	0.75	18
7	0.875	0

Table 4: Shor's-algorithm results for $N = 7$, $C = 15$, $a = 7$

This is all that is needed to try out Shor's algorithm. Table 4 shows the results of 100 quantum measurements when this was tried with

$a = 7$. The measured $\tilde{x}/2^L$ values were always close to $\frac{0}{4}$, $\frac{1}{4}$,$\frac{2}{4}$, or $\frac{3}{4}$. Since these are multiples of $\frac{1}{4}$, they give a guess $p = 4$ for the period (the correct answer).

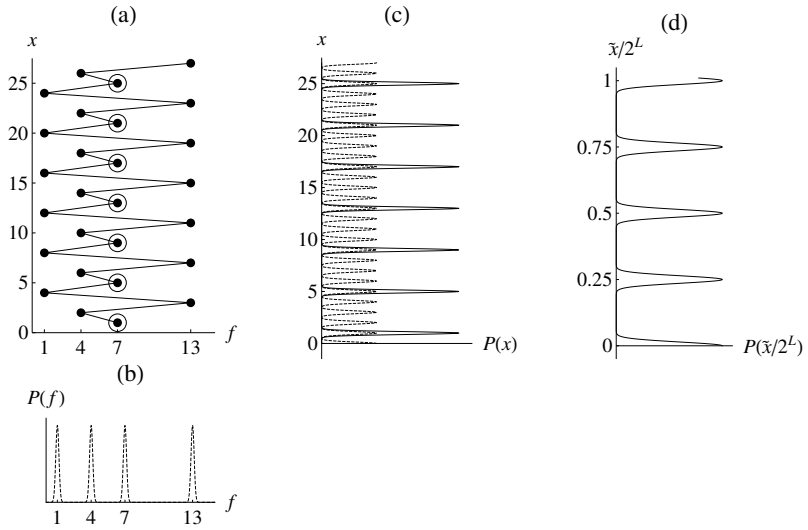

Figure 9: How the quantum period-finding calculation works. See text for details.

How the quantum part of Shor's algorithm works

This is a rough, hand-waving explanation. For an actual derivation see Nielsen and Chuang, Ch. 5, or the Wikipedia article on Shor's algorithm.

In Figs. 7 and 8 computations are carried out on the f-register, then no use is made of this register. A similar situation occurred earlier with Fig. 6(d): Using qubit 2 to control qubit 3 affected the measured value of qubit 2. Similarly, using the x-register to control the f-register affects the measured value of the x-register. Fig. 9 shows how this works.

- Fig. 9(a) represents the quantum state $|\Psi\rangle$ after the Hadamard operation on the x-register and the multiplication of the f-register by $f(x)$. Due to the Hadamard operation, every possible x value is equally likely. Due to the controlled-multiply operation, for every x value the f-register contains one of the p possible $f(x)$ values. (In Fig. 9(a) these are 1, 7, 4, 13.) Thus, $|\Psi\rangle$ at this point in the calculation is an equal superposition of many terms, each represented by one of the solid dots in Fig. 9(a). The dashed curves in Fig. 9(b,c) show that if the f-register is measured, the result will be 1, 4, 7, or 13, while if the x-register is measured, the result will be any integer.

- Now measure the f-register, and assume the result happens to be 7. According to basic quantum principles, a measurement causes the state $|\Psi\rangle$ to *collapse* so it agrees with the measured

value. Therefore after measuring $f = 7$ only the dots with $f = 7$ (circled in Fig. 9(a)) remain in the superposition.

- After measuring f all possible values of the x-register are no longer equally likely. Rather, $P(x)$ is only nonzero for the x's in the circled dots, so $P(x)$ has become a comb of peaks separated by the period $p = 4$ (solid curve in Fig. 9(c)).
- The IQFT takes the Fourier transform of this comb-like function. Because the function has period 4, its Fourier transform will contain the frequency $\omega = \frac{1}{4}$ and its harmonics $\omega = \frac{2}{4}, \frac{3}{4}$. In a discrete FT the frequencies are read out as $\omega = \tilde{x}/2^L$. Therefore when the x-register is measured the probability will have peaks at $\tilde{x}/2^L = s/4 = s/p$ for all integer harmonic numbers s, which is the claimed result.

More on the IQFT: In a classical discrete FT or IFT on an input register with L values, the FT is applied to L input numbers giving L output numbers. By contrast, here there are 2^L different x values represented by the 2^L quantum amplitudes in the superposition in the x-register, and the IQFT operates on all 2^L numbers at once. For the large prime numbers used in encryption L could be several thousand and a classical Fourier transform on 2^L numbers (or even storing 2^L numbers classically) is completely impossible.

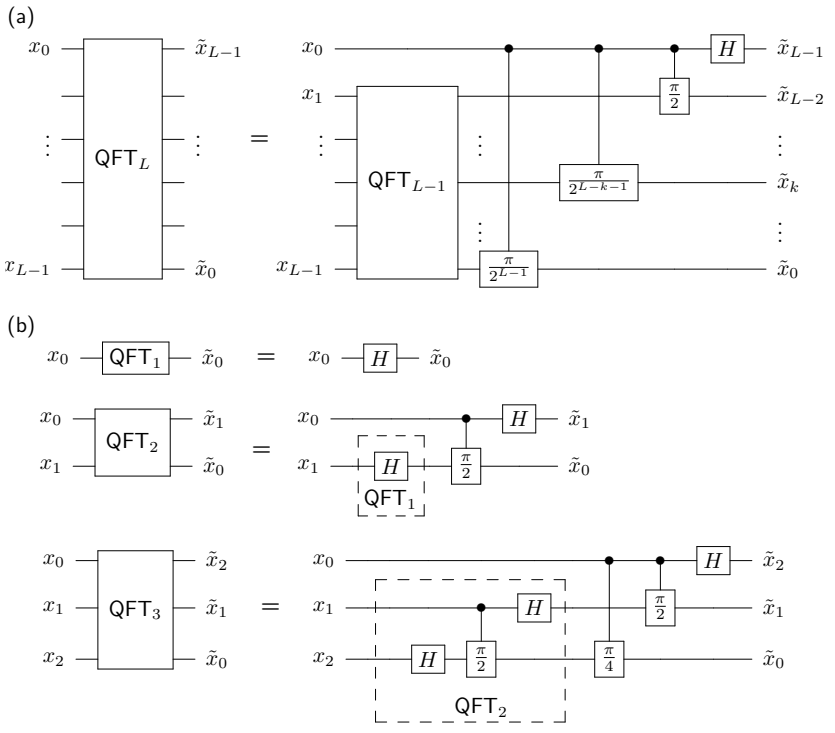

Figure 10: (a) Recursive definition of the quantum Fourier transform (QFT) for L qubits in terms of the QFT for $L-1$ qubits. For a derivation see Nielsen and Chuang Ch. 5, or `https://en.wikipedia.org/wiki/Quantum_Fourier_transform`. (b) QFT's for $L = 1, 2$ and 3 from the recursive definition.

Implementing Shor's algorithm for arbitrary N

By using $N > 7$ it is possible to factor numbers C greater than 15. The total number of qubits is $N = L + M$. The f-register holds binary values of $a^x \pmod C$, so M must satisfy $2^M \geq C$. To be confident[4] of finding the period p, the size L of the x-register should satisfy $2^L \geq C^2$. Table 5 shows the required L and M. However, as seen above Shor's algorithm can work with smaller L than this table implies: For $C = 15$, $L = 3 \rightarrow N = 7$ was used.

[4] Wikipedia article *Shor's algorithm*

Fig. 10 shows how to build the QFT for arbitrary L from Hadamard and controlled phase shift gates. For Shor's algorithm, the inverse quantum Fourier transform (IQFT) is needed. Quantum computing provides a way to invert a function that is not available classically. Since every quantum gate carries out a reversible operation, the QFT can be inverted by reversing the *time order* in which the gates are applied: Compare the IQFT in Fig. 8 with the bottom diagram in Fig. 10.

Programming project 8: Implement Shor's algorithm with $N > 7$.

Program Shor's algorithm for arbitrary N, and see how big a number can be factored. Table 6 shows a few results obtained in this way. In each case the L used was smaller than is listed in Table 5. In the results for $C = 39$, increasing L from 6 to 8 gave measured $\tilde{x}/2^L$ values much closer to the expected values $\frac{1}{6}, \frac{2}{6}, \frac{3}{6} \cdots = 0.1667, 0.3333, 0.5000 \ldots$

C	L	M	N	C	L	M	N
$15 = 3 \times 5$	8	4	12	$57 = 3 \times 19$	12	6	18
$21 = 3 \times 7$	9	5	14	$63 = 3 \times 3 \times 7$	12	6	18
$33 = 3 \times 11$	11	6	17	$65 = 5 \times 13$	13	7	20
$35 = 5 \times 7$	11	6	17	$69 = 3 \times 23$	13	7	20
$39 = 3 \times 13$	11	6	17	$75 = 3 \times 5 \times 5$	13	7	20
$45 = 3 \times 3 \times 5$	11	6	17	$77 = 7 \times 11$	13	7	20
$51 = 3 \times 17$	12	6	18	$85 = 5 \times 17$	13	7	20
$55 = 5 \times 11$	12	6	18	$87 = 3 \times 29$	13	7	20

Table 5: Numbers C suitable for Shor's-algorithm factoring and numbers of qubits required

Using continued fractions to guess the period

The output of the period-finding calculation is $\tilde{x}/2^L \approx s/p$ where s is an unknown integer, and is used to guess possible values for p. With a physical quantum computer, one would be trying to find p after running the computation just once. Trial values for p are typically derived from $\tilde{x}/2^L$ by using *continued fractions*. Having set up the computer to factor $C = 87$ and choosing $a = 13$, from Table 6 one

C	L	M	N	$a \Rightarrow p$	Storage used	Calculation time	Most frequently measured $\tilde{x}/2^L$ values (occurrences in 100 measurements)
39	6	6	12	10 6	8 MB	5.8 s	0.8281 (16), 0.0 (16), 0.5 (15), 0.3281 (14), 0.6719 (11), 0.1719 (8)...
39	8	6	14	10 6	44 MB	72 s	0.5 (18), 0.0 (18), 0.1680 (16), 0.3320 (13), 0.8320 (11), 0.6680 (11)...
87	9	7	16	13 14	198 MB	32 min	0.6426 (9), 0.1426 (9), 0.0723 (8), 0.8574 (6), 0.5 (6), 0.2148 (6)...

Table 6: Shor's-algorithm results using sparse matrices

might get the single measured result $\tilde{x}/2^L = 0.6426$. A continued-fraction expansion of this number is

To compute a continued-fraction expansion like this, invert the number, subtract off the integer part (which is the next number in the continued fraction), invert the fractional part, subtract off the integer part....

$$0.6426 = \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{3+\cfrac{1}{1+\cfrac{1}{19+\dots}}}}}} \approx 1,\ \frac{1}{2},\ \frac{2}{3},\ \frac{7}{11},\ \frac{9}{14},\ \frac{178}{277}\dots \tag{41}$$

This gives a sequence of rational numbers closer and closer to 0.6426. From $0.6426 \approx 7/11$ one tries $p = 11, 22\dots$ and from $0.6426 \approx 9/14$ one tries $p = 14, 28\dots$ Even for large C a classical computer can check which of the trial p's satisfies $a^p \equiv 1 \pmod{C}$. This gives the correct $p = 14$, from which the factors of 87 are computed to be $\gcd(13^{14/2} \pm 1, 87) = 29, 3$.

This is the end of the projects for Part 2. In Part 3 we tackle the fascinating but complicated subject of quantum error correction.

Postscript: Is quantum computation truly powerful?

As the projects in these handouts illustrate, the classical resources needed to simulate a quantum calculation grow exponentially with the number of qubits (Tables 2 and 6). This suggests that a quantum computer should be exponentially more powerful than a classical computer with the same number of bits. However, there are classical algorithms to search databases and factor numbers that are far more efficient than simulating Grover's algorithm and Shor's algorithm, making the classical vs. quantum comparison subtle.[5]

Computer scientists classify problems by the way the classical computation time t increases with the size S of the input (e.g. digits

[5] Troels F. Rønnow et al. Defining and detecting quantum speedup. *Science*, 345:420–424, 2014. Also online at `https://arxiv.org/abs/1401.2910`

in the number to be factored). The complexity class **P** consists of problems for which t grows no faster than a power of S. The class **NP** includes **P** but also includes more difficult problems thought to require time that grows faster than any power of the input size, for example $t \propto e^S$. For such problems, even modest input size can result in an impossibly long classical computation time.

Factoring numbers is thought to lie in **NP** but not **P** and thus to be exponentially difficult on a classical computer. Conversely, the time needed to factor a number on a quantum computer using Shor's algorithm is only polynomial in the input size. In this sense, at least, it seems that gate-array quantum computation as described in this article is inherently more powerful than classical computation. Finally, there are other models of quantum computation[6] (adiabatic quantum computing, quantum annealing, quantum emulation...) that may also offer possibilities for exponentially exceeding the capabilities of any classical computer. Realizing these vast computational powers in the real world will depend on continued progress in building physical quantum computers.[7]

This is a simplified description of computational complexity classes. For example, it has not been proven that **NP** is actually a different class than **P**. Based on Shor's algorithm, factoring is thought to lie in the class **BQP** (larger than **P** but not equal to **NP**) of problems that can be solved with high probability in polynomial time on a quantum computer. See Wikipedia article *Computational complexity theory* and Nielsen and Chuang Sec. 3.2.

[6] Wikipedia article *Quantum computer*

[7] Elizabeth Gibney. Quantum computer quest. *Nature*, 516:24–26, 2014

Appendix: Programming hints

Binary numbers and array indices

In some computer languages (Python, C++...) array indices start from zero. If the variable `psi` is an 8-element array representing the quantum state $|\Psi\rangle$ in Eq. 5, `psi[0]`$= a$ and `psi[7]`$= h$. **In these languages to find the basis state represented by `psi[j]`, convert j into a binary number.** For example, $j = 6$ corresponds to $|110\rangle$.

In other computer languages (MATLAB, Mathematica...) array indices start from one. In these languages Eq. 5 implies `psi(1)`$= a$ and `psi(8)`$= h$. **In these languages to find the basis state represented by `psi(j)`, convert $j - 1$ into a binary number.** Now $j = 7$ corresponds to $|110\rangle$.

The general rule to use when doing these projects in MATLAB or Mathematica is: Add one to any number being that expresses the qubit values in binary form before using the number as an array index. Similarly, subtract one from an array index before converting it to a binary number representing the qubit values.

Programming for speed and efficiency

As you work through these projects, your code may eventually take too long to finish, or run out of memory, or become too complicated to debug. Here are a few hints for dealing with these problems, with apologies to more experienced programmers:

- Often there is a trade-off between complexity and speed or storage needs. However, the execution time of a program is usually dominated by one small part of the program, and improving other parts will make the code more complicated without improving performance. You can use a "profiler" to figure out which parts of a code are using most of the time or space. More simply, you can insert functions to time how long the different parts of a program take. In general you should strive for simplicity and clarity in your code, only adding complexity where it demonstrably helps.

For example, it may be more complicated to use sparse matrix structures but they will execute faster and use less storage space than full matrices.

- You should start with the most natural and easy-to-manipulate representations of data, typically double precision floats for real and complex numbers. Only after your code is working in this form should you consider space-saving complexities like using integers (in a GPU, for example) to represent low-precision real numbers or using Booleans or even individual bits to represent real numbers known to be 1 or 0.

Mathematica stores numbers in a format that allows arbitrarily many digits of precision – convenient but probably not very efficient. Similarly variables in Mathematica are dynamically typed (e.g. x can switch mid-program between referencing a number or an expression). As a result, Mathematica seems slow to retrieve and store data in large arrays (individual dimensions of 10^5 - 10^6) which seem to be stored as linked lists.

- Once your code reaches a certain level of complexity it is hopeless to debug it by staring at it looking for mistakes. You must set up your code so you can test the different parts separately, and you need to check each part on input data which have known correct output (several of the projects have suggestions). To see what is going on during execution you can use a "debugger" or more simply add temporary print statements.

Convention used for numbering qubits

In a gate-array diagram like Fig. 1, qubit 1 is the top line and qubit N is the bottom line. For a basis state like $|011\rangle$, qubit 1 is the leftmost bit (0), and qubit N is the rightmost bit (1). It follows that basis states for an N-qubit register are read from top to bottom in a gate-array diagram. Thus in Fig. 1 the register is initialized to the state $|000b\rangle$.

Hints for project 5: Using an arbitrary number of qubits N

- It is necessary to find or write functions to convert integers into lists of their binary digits, and vice versa. For example, for $N = 7$ the conversions for the integer 23 are

$$23 \quad \Leftrightarrow \quad (0,0,1,0,1,1,1). \tag{42}$$

In Mathematica `IntegerDigits[]` provides this functionality; with other platforms you may need to write these functions. Note that the list of digits in Eq. 42 is *not* the same as a character string for the number in binary, although it may be possible to extract the bit values from such a character string.

  You may find it simplest to compute a $2^N \times N$ array with the N bits for all 2^N basis vectors at the start of you computation.

- Here is a way to implement Eq. 35, to create the $2^N \times 2^N$ matrix $\widehat{H}^{(n)}$ that applies the gate $\widehat{H}$ to qubit n (In Eq. 35, $N = 3$ and

$n = 2$). The program should be checked by printing out the $N = 3$ Hadamard matrices (Eqs. 21-23).

(i) Start with a $D \times D$ array initialized to all zeros, where $D = 2^N$.

(ii) For each element $H^{(n)}_{j,k}$ with $j = 0 \ldots D-1$ and $k = 0 \ldots D-1$, convert j and k into lists of 1's and 0's giving binary representations of these numbers.

(iii) Find the elements of each list at location n. These are the numbers q, q' in Eq. 35.

(iv) To enforce the Kronecker-δ's in Eq. 35, only put a nonzero element in the big array when the bit lists for j and k are equal to each other at all locations *other* than n. When this condition is met, put into the location j, k of the big array the q, q' element of the 1-qubit Hadamard array (Eq. 12).

This is written for Python or C++. For MATLAB or Mathematica, you still need a $D \times D$ array. But now to find the array element at row, column = r, c first compute $j = r - 1$ and $k = c - 1$, and then proceed as shown.

Hints for project 7: Shor's algorithm with $N = 7$ qubits

- For the three gates implementing modular multiplication print out the sum of each row and the sum of each column, and verify that every sum is 1.
- Start without the IQFT, to check that the first part of the calculation is working. The measured x should be random, and the measured f should be $f(x)$. For example, with $a = 7$ when $\tilde{x} = 6 = 110_2$, $x = 011_2 = 3$ so from Eq. 40 f should be $f(3) = 13$.

Hints for project 8: Shor's algorithm with $N > 7$ qubits

The recursive definition of the QFT of Fig. 10 must be implemented for arbitrary L, with the gates in reverse order to give the IQFT. Use Fig. 10 to write a routine (two nested FOR loops) to print out and check the *names* of the gates for any L. Then add the actual gates to the computation. The printed output for $L = 3$ should be:

```
Hadamard on 1
1 controls pi/2 on 2
1 controls pi/4 on 3
Hadamard on 2
2 controls pi/2 on 3
Hadamard on 3
```

(Compare with Fig. 10, bottom right, in reverse order)

Production notes

These handouts were produced with Latex using `\documentclass{tufte-handout}` documented at

`https://tufte-latex.github.io/tufte-latex/`

The quantum circuits were drawn using the Q-circuit macro package by Bryan Eastin and Steven T. Flammia, available at
`http://physics.unm.edu/CQuIC/Qcircuit/`

The source file for Parts 1 and 2 of these handouts is `byo1.tex` and `byo2.tex` with a shared bib file `byo12bib.bib`.