Shor's Algorithm

Ian Tillman

December 9, 2020

1 Introduction

In 1978, three computer scientists (Rivest, Shamir, and Adleman) published an encryption algorithm now known as RSA that was based on the apparent challenge of factoring large numbers into their prime divisors [1]. Even today there is no efficient method of factoring large numbers, where 'efficient' normally refers to polynomial complexity in time relative to the length of the number (worst-case time to solve is a polynomial function of the input length). This was the foundation of internet security for many years and modern cryptography relies on similar principles [2], so if an efficient method for solving these was found then the backbone of modern encryption would fail. Shor's Algorithm, first published by Peter Shor in 1994, is a proposed method to quickly factor large numbers. Although it scales very well with input size, its reliance on large-scale, fast quantum computers lead some to believe it will not be practically useful for a long time, if ever.

This brief summary will have 4 sections: A quick primer on Number Theory, the Quantum Fourier Transform and its decomposition, Shor's Algorithm itself, and a discussion on the feasibility of this algorithm. We will almost entirely stick to the revised 1996 paper by Shor [3], though other (maybe more intuitive) variants exist.[1] There is also a Matlab script attached at the end in an appendix that allows users to input a number to factor and look at a theoretical output of the full factoring algorithm. Though it must be noted that for the sake of brevity some key numerics were left out that might be necessary for factoring larger numbers.

2 The Algorithm

We begin this section by (quickly) looking at the number theory necessary to apply the algorithm to factoring and then move on to explaining the algorithm's parts. It should be noted here that we will be using *modular arithmetic* fairly heavily, so it's important to understand what the notation means. If $a = b \pmod{c}$, this means that both a and b are the same distance from multiples of c (i.e. both can be written as a (potentially different) multiple of c plus some shared constant k).

2.1 Number Theory

The Fundamental Theorem of Arithmetic states that any positive integer n can be factored into a product of primes. Modern methods for factoring are not much better than trivially testing all (reasonable) numbers less than $\sqrt{n}$, so a polynomial factoring algorithm would be a massive improvement. One can factor quasi-probabilistically with a clever strategy:

1. pick a random number $1 < x < n$ that is coprime to n
2. calculate its order mod n (i.e. the smallest r such that $x^r = 1 \pmod{n}$)
3. $x^r = 1 \pmod{n} \Rightarrow (x^{r/2} + 1)(x^{r/2} - 1) = 0 \pmod{n}$

[1] Here's a fantastic YouTube video by the channel MinutePhysics that gives a high-level explanation of a slight variant of what's described in this paper: https://youtu.be/lvTqbM5Dq4Q

This means $(x^{r/2}+1)$ and/or $(x^{r/2}-1)$ shares a factor with n, which we can find using the Euclidean Algorithm [4]. This will give us a non-trivial factor of n (not 1 or n itself) at least 50% of the time [3].

A keen eye will see that the hard part of the above algorithm is calculating the order, r, of a number x. Once again this is normally solved by trying all r until one works. Shor's algorithm is a method of calculating this order with the idea that a classical computer will likely be far more efficient with the steps of this approach after finding the order (though, of course, one could devise a fully quantum version if they wanted to).

2.2 Quantum Fourier Transform

One of the necessary breakthroughs to develop this algorithm was decomposing the Quantum Fourier Transform (QFT) into a polynomial amount of more basic elements which can each be applied with polynomial resources and time. The QFT is the keystone of this algorithm, so let's spend time looking at it closely.

Definition 1. *The Quantum Fourier Transform is a unitary operator acting on the space* $\{|0\rangle, |1\rangle\}^l$*, an* l*-qubit system, which we will call* A *and define as:*

$$A \equiv \frac{1}{\sqrt{q}} \sum_{a=0}^{q-1} \sum_{c=0}^{q-1} |c\rangle \langle a| e^{2\pi i ac/q}, \ q \equiv 2^l$$

where a *and* c *are the numbers that come from concatenating the* l *modes of the input/output state* $|1\rangle_1 \otimes |0\rangle_2 \otimes |1\rangle_3 \otimes \cdots \otimes |1\rangle_l \to |101\cdots 1\rangle$ *and converting from binary.*

In this work we pick l to be the integer in the range $2\log_2(n) < l \leq 2\log_2(n)+1 \Rightarrow n^2 < q \leq 2n^2$, where n is the number we're trying to factor.

If we define the following gates that act on the jth (and kth) qubits:

$$H_j = \frac{1}{\sqrt{2}} \left(|0\rangle \langle 0|_j + |0\rangle \langle 1|_j + |1\rangle \langle 0|_j - |1\rangle \langle 1|_j \right)$$

$$S_{j,k} = |00\rangle \langle 00|_{j,k} + |01\rangle \langle 01|_{j,k} + |10\rangle \langle 10|_{j,k} + e^{i\pi 2^{j-k}} |11\rangle \langle 11|_{j,k}$$

then we can use this decomposition found by Coppersmith and Deutsch (independently) [5, 6]:

$$A \cong \prod_{j=0}^{l-1} \left[\prod_{k=j+1}^{l-1} S_{j,k} \right] H_j$$

where $\cong$ is used because we also need to reverse the bits, but this can be done within the algorithm so no actual time or resources are lost to that correction. It's also important to note that $\prod$, in our notation, acts successive terms on the left and $a > b \ \Rightarrow \prod_a^b \equiv 1$.[2] Counting the number of gates used we see that we have l H gates and $\frac{(l-1)l}{2}$ S gates, giving us $\frac{l(l+1)}{2}$ total gates. This, crucially, is polynomial in l, giving us a polynomial method to apply the normally exponential QFT. Copperfield also pointed out that we can approximate this by leaving out many of the $S_{j,k}$ gates (specifically ones where $k-j$ is large) and still be close enough to the true operator to use Shor's algorithm. This is useful for lowering computational complexity as well as the fact that modulating phase to the accuracy of $e^{i\pi 2^{j-k}}$ is very difficult when $k-j$ is large.

2.3 The Quantum Steps

This algorithm requires $l + \frac{l}{2} \approx 3\log_2(n)$ qubits, one register of size[3] l for the binary expansion of n and another of size $l/2$ for our random number $1 < x < n$ raised to the power of each of the q possible inputs modulo n. Starting with all qubits in the state $|0\rangle$, we begin by putting the system into the state

[2] For example, $l = 3$ gives us $A = H_2 S_{1,2} H_1 S_{0,2} S_{0,1} H_0$.

[3] The input register needs to be able to hold n^2 for the purposes of Shor's mathematical proof.

$$|\psi\rangle = \frac{1}{\sqrt{q}} \sum_{a=0}^{q-1} |a\rangle_i |0\rangle_o$$

where 'i' stands for input and 'o' stands for order. This can be done by applying a Hadamard gate to each of the individual qubits in the input register, leaving the input in the state $\bigotimes_{j=0}^{l-1} \frac{1}{\sqrt{2}} \left[|0\rangle_{ij} + |1\rangle_{ij} \right]$ (where 'ij' means the jth input qubit) which can be expanded into the sum of every element of the set $\{|0\rangle, |1\rangle\}^l$. We then put the system through a gate that sends the state $|a\rangle_i |0\rangle_o \to |a\rangle_i |x^a \pmod n\rangle_o$, followed by using the QFT on the input register[4], and ending by observing both registers. According to the derivation done by Shor the state observed in the input register, which we will call $|c\rangle$, will lead to *at most one* solution existing for the following inequality:

$$\left| \frac{c}{q} - \frac{d}{r} \right| < \frac{1}{2q}$$

where $1 < r < n$ is the order of $x \bmod n$ and d is some integer. There are fast ways to find $\frac{d}{r}$, namely using the continued fraction expansion of $\frac{c}{q}$, but one can also just try all d and r for relatively small n.

Shor's work shows that we will collapse to a state that can give us r in $O(\text{loglog}(n))$ attempts on average, so this is sub-polynomial in the number of bits. Getting r was the original goal so the quantum steps are done once there is a successful collapse and calculation of r.

3 Feasibility

Just as we have said in class, Shor mentions that the main difficulties in building a large-scale quantum computer will be dealing with *imprecision* and *decoherence*. Seeing as classical computers have factored numbers with 829 bits [7] and quantum computers have only factored numbers with 6 bits [8], there is a long way to go before this algorithm will be useful. Shor himself notes that this application would most likely not be enough to justify making quantum computers and that other applications will be the driving force. However, because of the relevance of encryption and the public's general understanding of the importance of prime numbers, this will most likely be a common experiment as we scale up quantum computers.

References

[1] R.L. Rivest et al. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. 1978.

[2] M. Giles. *Explainer: What is post-quantum cryptography?* 2019. URL: `https://www.technologyreview.com/2019/07/12/134211/explainer-what-is-post-quantum-cryptography/`.

[3] P. Shor. *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*. 1996. URL: `https://arxiv.org/pdf/quant-ph/9508027.pdf`.

[4] D.E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. 1981.

[5] D. Coppersmith. *An approximate Fourier transform useful in quantum factoring*. 1994.

[6] A. Ekert and R. Jozsa. *Shor's quantum algorithm for factorising numbers*. 1995.

[7] P. Zimmerman. 2020. URL: `https://listserv.nodak.edu/cgi-bin/wa.exe?A2=NMBRTHRY;dc42ccd1.2002`.

[8] M. Amico et al. *An Experimental Study of Shor's Factoring Algorithm on IBM Q*. 2019. URL: `https://arxiv.org/pdf/1903.00768.pdf`.

[4] This is where we start in the Matlab code below.

Appendix A MATLAB Code

```
% Using n=15 and x=11 recreates the experiment in (Chao-Yang Lu, et al. 2007):
% https://journals.aps.org/prl/pdf/10.1103/PhysRevLett.99.250504
n = 15;
x = 11;
l = ceil(2*log2(n));
q = 2^l;

%Initialize to the footnoted step in 2.3
coeffs = zeros(q,n);
for a=1:q
    for c=1:q
        oIndex = powermod(x,(a-1),n) + 1;
        coeffs(c,oIndex) = coeffs(c,oIndex) + exp(2*pi*1i*(a-1)*(c-1)/q);
    end
    disp(num2str(a)+"/"+num2str(q));
end
coeffs = coeffs/q;

% Find which state the system collapsed to using RNG (try to find valid r 100 times)
success = 0;
for measurement=1:100
    measurementOutcome = rand();
    temp = 0;
    qubit1MeasuredState = 1;
    qubit2MeasuredState = 1;
    while temp<measurementOutcome
        if qubit1MeasuredState>q
            qubit1MeasuredState = 1;
            qubit2MeasuredState = qubit2MeasuredState+1;
            if qubit2MeasuredState>q
                break; end
        end
        temp = temp + abs(coeffs(qubit1MeasuredState,qubit2MeasuredState))^2;
        qubit1MeasuredState = qubit1MeasuredState+1;
    end
    % Correcting the extra loop increment and the offset for starting arrays at 1
    qubit1MeasuredState = qubit1MeasuredState-2;

    % Find fraction d/r
    found = 0;
    for r=n-1:-1:2
       for d=1:r
          if abs(qubit1MeasuredState/q - d/r)<=1/(2*q)
              if mod(r/gcd(r,d),2)==0
                found = 1; r=r/gcd(r,d); break; end
          end
       end
       if found
           break; end
    end
    if found
        success = 1; break; end
end

if success
    disp("These possible factors of " + num2str(n) + " were found after "+num2str(measurement)+" attempts:")
    disp([gcd(x^(r/2)-1,n),gcd(x^(r/2)+1,n)]);
else
    disp("Try another x");
end
```

As expected this runs very slow relative to classical factoring. I didn't test numbers higher than 33 so there may be some errors I missed, namely with finding r since there are many caveats that I left out for space and readability.