Quantum Programming Algorithms: Shor

Jens Palsberg

May 28, 2020

Outline

Hook: Shor's algorithm factors integers in polynomial time with high probability. This creates trouble for RSA cryptography, which relies on that factoring integers is hard. Indeed, part of the promise of quantum computing is that Shor's algorithm will break some forms of cryptography that are widely used on the Internet. But how does Shor's algorithm work?

Purpose: Persuade you that you can understand Shor's algorithm at a high level.

Preview:

1. Shor's algorithm consists of a classical algorithm that calls a quantum subroutine.
2. We will present an example run of the algorithm.
3. We will outline how the quantum subroutine works.

Transition to Body: Now let us look at the big picture of factoring integers.

Main Point 1: Shor's algorithm consists of a classical algorithm that calls a quantum subroutine.
[The algorithm is probabilistic and it guesses and checks candidates]
[We use known classical algorithms to isolate the core problem]
[A key subroutine finds the order of an element in a group]

Transition to MP2: This algorithm has many components that are nontrivial.

Main Point 2: We will present an example run of the algorithm.
[We will pick an input that requires more than a little bit of work]
[We will make a good guess in the first try]
[We will go through the steps at a high level]

Transition to MP3: And where does quantum come in?

Main Point 3: We will outline how the quantum subroutine works.
[Order finding uses phase estimation as a subroutine]
[Phase estimation uses the quantum Fourier transform as a subroutine]
[The quantum Fourier transform is exponentially faster than the classical FFT]

Transition to Close: So there you have it.

Review: At the core of Shor's algorithm is a subroutine for finding the order of an element in a group. This subroutine can be executed efficiently on a quantum computer.

Strong finish: Shor's algorithm prompted cryptographers to study post-quantum cryptography and it created a lot of excitement about the promise of quantum computing.

Call to action: Find an implementation of Shor's algorithm in Qiskit, or write it yourself, and run it on IBM's quantum computer with input 15, and see if you get 3,5.

Detailed presentation

Hook: Shor's algorithm factors integers in polynomial time with high probability. This creates trouble for RSA cryptography, which relies on that factoring integers is hard. Indeed, part of the promise of quantum computing is that Shor's algorithm will break some forms of cryptography that are widely used on the Internet. But how does Shor's algorithm work?

Purpose: Persuade you that you can understand Shor's algorithm at a high level.

Preview:

1. Shor's algorithm consists of a classical algorithm that calls a quantum subroutine.
2. We will present an example run of the algorithm.
3. We will outline how the quantum subroutine works.

Transition to Body: Now let us look at the big picture of factoring integers.

Main Point 1: Shor's algorithm consists of a classical algorithm that calls a quantum subroutine.

[The algorithm is probabilistic and it guesses and checks candidates]
The idea of Shor's algorithm is to guess a random integer that is less than the input and then check whether the guess leads to a factoring. The guess-and-check may well fail but it will succeed with probability greater than $\frac{1}{2}$. So if we repeat the guess-and-check a few times, we are highly likely to succeed.

[We use known classical algorithms to isolate the core problem]
Here is the algorithm for integer factorization. Notice that the algorithm calls itself recursively.

Algorithm: Integer Factorization.
Input: A positive integer $N \geq 2$.
Output: A prime factorization $N = p_1^{k_1} \times \ldots \times p_m^{k_m}$.
Method:

```
if (N = p^k where p is prime and k ≥ 1) {
    return p^k
}
else {
if (N is even) {
    return combine(2, IntegerFactorization(N/2))
}
else {
    int d = Shor(N)
    return combine(IntegerFactorization(d), IntegerFactorization(N/d))
}
```

The above algorithm calls Shor's algorithm as a subroutine, and it is informal about the following three aspects. First, it calls a classical subroutine to check whether N is the power of a prime. Second, it calls a classical subroutine to check whether N is even. Third, it calls a classical subroutine to combine two prime factorizations.

Here is Shor's algorithm.

Algorithm: Shor's Algorithm.
Input: An odd, composite integer N that is not the power of a prime.
Output: A nontrivial factor of N, that is,
an integer d such that $1 < d < N$ and $d \mid N$.
Method:

```
repeat
    int a = random choice from {2, ..., N − 1}
    int d = gcd(a,N)
    if (d > 1) {
        return d
    }
    else {
        int r = FindOrder(a,N)
        if (r is even) {
            int x = a^(r/2) − 1 (mod N)
            int d = gcd(x, N)
            if (d > 1) {
                return d
            }
        }
    }
until we give up
```

Shor's algorithm calls the subroutine FindOrder. The idea is that FindOrder(a,N) returns the smallest integer $r > 0$ such that $a^r \equiv 1 \pmod{N}$. This number r is known as the order of a in $\mathbb{Z}_N^*$. Here, $\mathbb{Z}_N = \{0, 1, \ldots, N-1\}$ and $\mathbb{Z}_N^* = \{\, a \in \mathbb{Z}_N \mid \gcd(a, N) = 1 \,\}$. If we equip $\mathbb{Z}_N^*$ with a binary operation which is multiplication modulo N, then it forms a group.

The idea of Shor's algorithm is that if we know that the order r of $a \in \mathbb{Z}_N^*$ is even, then we have

$$a^r \equiv 1 \pmod{N}$$

so

$$N \mid a^r - 1$$

and since $a^r - 1 = (a^{\frac{r}{2}} + 1) \times (a^{\frac{r}{2}} - 1)$, we have

$$N \mid (a^{\frac{r}{2}} + 1) \times (a^{\frac{r}{2}} - 1)$$

Notice that $N \mid (a^{\frac{r}{2}} - 1)$ is false because otherwise $a^{\frac{r}{2}} \equiv 1 \pmod{N}$, which implies that r is not the order of a after all, because $a^{\frac{r}{2}} \equiv 1 \pmod{N}$ and $\frac{r}{2} < r$, which contradicts that r is the order of a.

So, we let $x = a^{\frac{r}{2}} - 1 \pmod{N}$, and then we compute $d = \gcd(x,N)$. If $d > 1$, then we have found a nontrivial factor of N. But if $d = 1$, then we have $N \mid (a^{\frac{r}{2}} + 1)$ and we had no luck with a.

Let us analyze the success rate of the repeat-loop. Each iteration of the loop fails to give an answer if either r is odd, or r is even but $N \mid (a^{\frac{r}{2}} + 1)$. The probability that neither of these events occur is at least $\frac{1}{2}$. In more detail, the probability is at least $1 - 2^{m-1}$, where m is the number of distinct primes dividing N. So, the probability is at least $\frac{1}{2}$ when N is an odd, composite integer that is not the power of a prime, which is exactly the precondition for Shor's algorithm.

[A key subroutine finds the order of an element in a group]
The subroutine FindOrder takes a long time to do on a classical computer. Shor's insight is that FindOrder can be done efficiently on a quantum computer. Specifically, on a classical computer, FindOrder(a,N) takes time that is polynomial in N. But, on a quantum computer, FindOrder(a,N) takes time that is logarithmic in N.

Transition to MP2: This algorithm has many components that are nontrivial.

Main Point 2: We will present an example run of the algorithm.

[We will pick an input that requires more than a little bit of work]
Let us go with $N = 21$. We can see that N is not the power of a prime and N is not even. So, we call Shor's algorithm on N.

[We will make a good guess in the first try]
Let us pick $a = 2$. This pick will lead to a single iteration of the `repeat`-loop.

[We will go through the steps at a high level]
Let us calculate:

$$d \;=\; \gcd(a, N) \;=\; \gcd(2, 21) \;=\; 1$$

Now we call FindOrder, and we see that

$$r \;=\; \text{FindOrder}(a, N) \;=\; \text{FindOrder}(2, 21) \;=\; 6,$$

because $2^6 = 1 \pmod{21}$.

Next we notice that r is even, so we calculate

$$\begin{aligned} x &= a^{\frac{r}{2}} - 1 \pmod{N} \;=\; 2^{\frac{6}{2}} - 1 \pmod{21} \;=\; 7 \\ d &= \gcd(x, N) \;=\; \gcd(7, 21) \;=\; 7 \end{aligned}$$

We see that d is greater than 1, so we return $d = 7$.

Transition to MP3: And where does quantum come in?

Main Point 3: We will outline how the quantum subroutine works.

[Order finding uses phase estimation as a subroutine]

Algorithm:	Order Finding.
Input:	An integer $N > 1$ and an element $a \in \mathbb{Z}_N^*$.
Output:	The smallest integer $r > 0$ such that $a^r \equiv 1 \pmod{N}$.

Let n be the number of bits needed to encode an element of $\mathbb{Z}_N^*$ in binary.
Define a unitary operation M_a

$$M_a|x\rangle \;=\; |ax \pmod{N}\rangle$$

where $x \in \mathbb{Z}$ and the order of a is r. From now on, the operations inside kets are implicity assumed to be modulo N.

Notice that, for $\omega = e^{2\pi i/r}$, if

$$|\psi_k\rangle \;=\; \frac{1}{\sqrt{r}}(|1\rangle + \omega^{-k}|a\rangle + \omega^{-2k}|a^2\rangle + \ldots \omega^{-(r-1)k}|a^{r-1}\rangle)$$

then

$$M_a|\psi_k\rangle \;=\; \omega^k|\psi_k\rangle$$

so $|\psi_k\rangle$ is an eigenvector of M_a.

Let us check the above by doing a calculation for the case of $|\psi_1\rangle$:

$$\begin{aligned}
M_a|\psi_1\rangle &= M_a(\frac{1}{\sqrt{r}}(|1\rangle + \omega^{-1}|a\rangle + \omega^{-2}|a^2\rangle + \ldots \omega^{-(r-1)}|a^{r-1}\rangle) \\
&= \frac{1}{\sqrt{r}}(|a\rangle + \omega^{-1}|a^2\rangle + \omega^{-2}|a^3\rangle + \ldots \omega^{-(r-1)}|a^r\rangle \\
&= \frac{1}{\sqrt{r}}(|a\rangle + \omega^{-1}|a^2\rangle + \omega^{-2}|a^3\rangle + \ldots \omega^{-(r-1)}|1\rangle \\
&= \frac{\omega}{\sqrt{r}}(\omega^{-1}|a\rangle + \omega^{-2}|a^2\rangle + \omega^{-3}|a^3\rangle + \ldots \omega^{-r}|1\rangle \\
&= \frac{\omega}{\sqrt{r}}(\omega^{-1}|a\rangle + \omega^{-2}|a^2\rangle + \omega^{-3}|a^3\rangle + \ldots |1\rangle \\
&= \omega|\psi_1\rangle
\end{aligned}$$

In the third step, we use that $a^r = 1 \pmod N$, and in the fifth step, we use

$$\omega^{-r} \;=\; (e^{2\pi i/r})^{-r} \;=\; e^{-2\pi i} \;=\; (e^{\pi i})^{-2} \;=\; (-1)^{-2} \;=\; ((-1)^2)^{-1} \;=\; 1^{-1} \;=\; 1$$

where the fourth step uses $e^{\pi i} = -1$, which is Euler's identity.

Now we want to use Phase Estimation to get r.

Algorithm:	Phase Estimation.
Input:	A unitary operation U and an eigenvector of U, that is $U\|\psi\rangle = e^{2\pi i\theta}\|\psi\rangle$.
Output:	θ.

Ideally, we run Phase Estimation on M_a and $|\psi_1\rangle$, which returns $\frac{1}{r}$, from which we can get r. However, we don't have $|\psi_1\rangle$, so we do something else. Notice that

$$\begin{aligned}
\frac{1}{\sqrt{r}}\sum_{k=0}^{r-1}|\psi_k\rangle &= \frac{1}{r}\sum_{k=0}^{r-1}\sum_{l=0}^{r-1}\omega^{-lk}|a^l\rangle \;=\; \frac{1}{r}\sum_{l=0}^{r-1}\sum_{k=0}^{r-1}(\omega^{-l})^k|a^l\rangle \;=\; |1\rangle + \frac{1}{r}\sum_{l=1}^{r-1}\sum_{k=0}^{r-1}(\omega^{-l})^k)|a^l\rangle \\
&= |1\rangle + \frac{1}{r}\sum_{l=1}^{r-1}\frac{1-(\omega^{-l})^r}{1-\omega^{-l}}|a^l\rangle \;=\; |1\rangle
\end{aligned}$$

The fourth step uses the formula for a geometric sequence, and the fifth step uses Euler's identity: $(\omega^{-l})^r = (\omega^r)^{-l} = ((-1)^2)^{-l} = 1^{-l} = 1$.

So, instead of running Phase Estimation on M_a and $|\psi_1\rangle$, we run Phase Estimation on M_a and $|1\rangle$. This is a lot like running Phase Estimation on M_a and $|\psi_k\rangle$ for random choices of k, chosen uniformly. We will run Phase Estimation on M_a and $|1\rangle$ many times. Each run produces $\frac{k}{r}$,

where we don't know either of k and r. This is where the Continued Fraction Algorithm comes in. The Continued Fraction Algorithm plus some post-processing map a list of samples of $\frac{k}{r}$ to r (we skip the details). The Continued Fraction Algorithm runs in $O((\log N)^3)$ time, which is a major contributor to the time complexity of Shor's algorithm.

[Phase estimation uses the quantum Fourier transform as a subroutine]
The entire quantum part of Shor's algorithm boils down to Phase Estimation. Turns out that Phase Estimation uses the Quantum Fourier Transform as a critical component. We skip the details of how this works.

[The quantum Fourier transform is exponentially faster than the classical FFT]
A milestone in Fourier analysis was the 1965 paper by Cooley and Tukey, which presented the Fast Fourier Transform (FFT) algorithm. The FFT algorithm computes the Discrete Fourier Transform in $O(n \log n)$ time. The FFT algorithm has become a cornerstone of many disciplines within Electrical Engineering and Computer Science. The FFT paper has been cited 15,640 times, according to Google Scholar.

The FFT paper is amazing and the $O(n \log n)$ run time is blazingly fast, yet the quantum Fourier transform is exponentially faster. The quantum Fourier transform is due to Shor (1994).

Transition to Close: So there you have it.

Review: At the core of Shor's algorithm is a subroutine for finding the order of an element in a group. This subroutine can be executed efficiently on a quantum computer. So, Shor's algorithm is a hybrid algorithm that mixes classical computation and quantum computation. Notice the hierarchy of subroutines:

Integer Factorization	which calls
Shor's Algorithm	which calls
Order Finding	which calls
1) Phase Estimation and 2) Continued Fractions	where (1) calls
Quantum Fourier Transform	

Here, the quantum part of the algorithm lies in Phase Estimation and its use of the Quantum Fourier Transform.

Strong finish: Shor's algorithm prompted cryptographers to study post-quantum cryptography and it created a lot of excitement about the promise of quantum computing.

Call to action: Find an implementation of Shor's algorithm in Qiskit, or write it yourself, and run it on IBM's quantum computer with input 21, and see if you get 3×7.