Lecture 19, Thurs March 30: RSA and Shor's Algorithm

Today we'll see Shor's algorithm. Given a positive integer *N*, which we'll assume for simplicity is a product of two primes p and q, this algorithm lets you find p and q using only about $O(\log^2 N)$ steps. This application captured the world's attention because **RSA**, one of the most widely-used public-key encryption methods, relies on the assumption that factoring is hard.

So before we start in on Shor's algorithm, which will take a few lectures, let's briefly review RSA. The basic idea is:

Some website, like Amazon, wants you to be able to send them messages that only they can decrypt (say your credit card number). But they've never met with you in private to agree on a secret encryption key. So what they do instead is that they find two large primes *p* and *q* (say a thousand digits each), which they keep secret. They Amazon multiplies them to get $N = pq$, which they publish to the world. Note that the fact that Amazon *can* efficiently pick two huge random prime numbers *p* and *q*, and know for sure that they're prime, is already not quite obvious, but it follows from some classical number theory that we won't go into.

Now you can encrypt a message using the public key, *N*: if your plaintext message is x, then in the simplest version of RSA, your encrypted message would just be $x^3 \bmod N$. And given that encrypted message, and *also* knowing *p* and *q*, there's a way for Amazon to efficiently recover x—it's again some basic number theory that we won't go into right now, although we'll see aspects of it later. The key idea is that, if you know *p* and *q*, then you also know the order of the *multiplicative group* modulo N, and that knowledge lets you do things like efficiently take cube roots modulo N.

By contrast, an eavesdropper who doesn't know *p* and *q*, but only knows *N*, *seems* to face an exponentially hard problem in recovering the plaintext—though it's been proven neither that factoring is hard, nor even that breaking RSA is necessarily as hard as factoring.

Some number theorists conjecture that factoring is in P, or profess agnosticism.
The problem has only been seriously worked on for, like, 40 years.

In any case, if you can factor, then you can break RSA, and that certainly provides more than enough reason to be interested in the complexity of factoring.

The naive algorithm to factor *N* is trial division—in the worst case, requiring you to test all possible divisors up to $\sqrt{N}$ (note that every divisor greater than $\sqrt{N}$ must have an accompanying divisor that's smaller). We call this an "exponential-time" algorithm, since the running time is exponential in $n = \log(N)$, which is the number of digits needed to specify *N*.

Number theorists have discovered several faster algorithms.

The Quadratic Field Sieve, from 1981, takes time roughly $2^{O(\sqrt{n})}$, a milder exponential.

The Number Field Sieve brings that down to $2^{n^{1/3}}$, though its correctness depends on the proof of a yet-unproven conjecture.

This is why 512-bit, 768-bit, and maybe even 1024-bit encryption aren't quite secure anymore.

They can be cracked using known algorithms and sufficient money for hardware.

In the Snowden documents there's evidence of money allocated for this sort of thing.
In a way that's almost reassuring, to people who worry that the NSA can break anything...

Another public-key cryptosystem in widespread use is **Diffie-Hellman**, which is based on a different problem than factoring, called *discrete logarithms*. While we won't cover this in lecture, it turns out that Shor's algorithm *also* solves the discrete log problem in polynomial time, thereby breaking Diffie-Hellman as well.

No one has shown that factoring and discrete log are necessarily related, e.g. by giving a reduction between them. In practice, though, advances in solving one problem almost always seem to lead in short order to advances in solving the other.

We now know that both RSA and Diffie-Hellman were first discovered in secret, by a mathematician named Clifford Cocks at the GCHQ (the British NSA), before they were rediscovered in public.

What Shor's algorithm really does, under the hood, is something called *period-finding*. Shor observed that, for classical number theory reasons having nothing to do with quantum mechanics, a fast algorithm for period-finding leads to a fast algorithm for problems like factoring and discrete logarithm.

Period Finding
is yet another black box problem—one that should remind you of Simon's problem from the last lecture.
You're given oracle access to a function $f : \mathbb{N} \to \mathbb{N}$
You're also promised that there's a secret integer $s > 0$ (the "period") such that

$$f(x) = f(y) \Leftrightarrow s \mid y - x$$

for all *x*, *y*.
The problem is to find s.

f has *period* s, which is to say: it returns to the same value after every s steps.

How many queries to f do we need to solve period-finding classically?

Let's assume that $s \sim 2^n$ (i.e., that *s* is an n-bit integer), and give our answer in terms of n. Observe that once you find a pair x,   such that $f(x) = f(y)$, you're very close to solving the problem. Indeed, if you found a few such collisions, you could just take their greatest common divisor, like so:

$$\gcd(x_1 - y_1, x_2 - y_2, x_3 - y_3)$$

We won't give the analysis, but after not too many collisions, the odds are high that this will yield the period.

Incidentally, how do we get the gcd of two integers in polynomial time?
We use **Euclid's GCD Algorithm**—possibly the oldest interesting polynomial-time algorithm in history!
To find the gcd of *x* and *y* (with *x* > *y*), we find *q* and *r* that satisfy:

$$x = qy + r$$

such that *qy* is the greatest multiple of *y* less than *x*, which means *y* > *r*.

Then we find the gcd of *y* and *r*, and we keep recursing in this way until $r = 0$.

Each time you run this, the size of the numbers goes down by about a constant factor, which means the whole algorithm runs in time linear in n (i.e., the bit-length of x and y).

OK, but how do we find collisions? This is the birthday paradox all over again. Recall from the last lecture that something like $\sqrt{2^n}$ queries to f is both necessary and sufficient.

This is *still* a huge number of queries, if (say) $n = 2000$.

Now we're ready to talk about…

Shor's Algorithm
which has two very different ingredients:

- A reduction from factoring to period-finding.

  This part is purely classical.

- A quantum algorithm for period-finding that uses only a *constant* ($O(1)$) number of queries to f, as well as a polynomial ($n^{O(1)}$) number of computational steps.

Why is factoring ≤ period finding? Here ≤ means "reducible to"

The main connection between the two is the multiplicative group modulo n. This is a finite abelian group—that is, a finite set with a commutative, associative, invertible multiplication operation—that's one of the most basic examples of a group in all of math.

For a prime , the multiplicative group consists of the set $\{1, 2, \ldots, p-1\}$, with the group operation being multiplication mod .

To be in the multiplicative group mod N where N is composite, you again need to be less than N, and now also relatively prime with N, since otherwise you won't have a multiplicative inverse mod N.

For example, when $n = 15$, the multiplicative group mod N consists of the 8-element set

$$\{1, 2, 4, 7, 8, 11, 13, 14\},$$

with the group operation being multiplication mod 15.

To check that this is a group, you can fill out the multiplication table and see for yourself!

In general, the size of this group, given $N = pq$, is going to be $\left|\mathbb{Z}_N^\times\right| = (p-1)(q-1)$, since that's how many positive integers less than N are relatively prime to N. For example, when $N = 15 = 5 \times 3$, we saw that the size of the group is $(5-1)(3-1) = 8$.

An extremely useful fact about finite groups G is that, for any element $x \epsilon G$, we have $x^{|G|} = 1$.
This has a few corollaries, such as

Fermat's Little Theorem

$x^{p-1} \equiv 1 \pmod{p}$, for all primes p and integers $x \epsilon \{1, \ldots, p-1\}$

and its generalization, **Euler's Theorem**

$x^{(p-1)(q-1)} \equiv 1 \pmod{pq}$, for all primes p, q and integers x relatively prime to them

Euler's Theorem is super important in RSA encryption as well.

More generally, **Euler's Totient Function** $\varphi(N)$ returns the order of the multiplicative group mod N, which is the number of integers from 1 to N that are relatively prime to N. For example, if p and q are prime, then $\varphi(p) = p - 1$ and $\varphi(pq) = (p-1)(q-1)$. And we can write:

$$x^{\varphi(N)} \equiv 1 \pmod N$$

Why is this important?

Well, let's say we want to factor some number $N = pq$, a product of distinct primes.

Then here's an approach: pick an x such that $\gcd(x, N) = 1$

We can assume such x's are easy to find. Indeed, if $\gcd(x, N) > 1$, then we can run Euclid's algorithm on x and *N* to factor *N* right then and there.

Shor's algorithm is based on a careful study of the **modular exponentiation function**:

$$f(r) := x^r \pmod N$$

What can we say about this function? First of all, how hard is it to compute? A naïve approach would use r–1 multiplications, which is exponential in $n = \log(N)$. But there's a much, much faster approach, called **Repeated Squaring**. It's best illustrated with an example.

Say we want to calculate $13^{21} \pmod{15}$.

We could calculate $13 \times 13 \times \cdots \times 13 \pmod{15}$, by alternating multiplication with reducing mod 15, but that's still 20 multiplications.

Instead, let's rewrite it as a product of 13 raised to various powers of 2:

$$13^{16} \times 13^4 \times 13 \pmod{15}.$$

Then we can further rewrite that as $\left(\left(\left(13^2\right)^2\right)^2\right)^2 \times \left(13^2\right)^2 \times 13 \pmod{15}$.

This may be ugly, but it requires considerably fewer multiplications, an advantage that grows rapidly as the exponent increases. Indeed, the total time needed is polynomial in $\log(n)$.

Once again, repeated squaring also plays a central role in RSA decryption. Ironically, many of the same number theory facts that led to RSA also allow lead to Shor's algorithm, which breaks RSA.

OK, so $f(r) = x^r \pmod N$ is efficiently computable. It's also, clearly, a periodic function. What could we learn by figuring out its period? Here's the key point: we claim that **finding the period of f will help us factor *N*.**

Why?

First an intuition. Suppose we were able to learn $\varphi(N)$, the order of the multiplicative group mod *N*. Then we'd surely be able to factor *N* into *pq*. Indeed,

$$\varphi(N) = (p-1)(q-1) = N - p - q + 1.$$

So now we know both *pq* and *p*+*q*, and we can use the quadratic formula to solve for *p* and *q* themselves.

Unfortunately, this doesn't quite work with Shor's algorithm, because the period of f might not be the same as $\varphi(N)$: the most we can say is that the period *divides* $\varphi(N)$.

So here's what we'll do instead. We pick a random x relatively prime to N, and find the period s of $f(r) = x^r \pmod{N}$. We then have that $x^s \equiv 1 \pmod{N}$.

Now let's imagine we're lucky, and s is even (which intuitively should happen maybe half the time?). In that case we can write:

$$x^s - 1 \equiv 0 \pmod{N}$$
$$\Rightarrow (x^{s/2} - 1)(x^{s/2} + 1) \equiv 0 \pmod{N}$$

Now let's imagine we get lucky a second time, and neither $x^{s/2} - 1$ nor $x^{s/2} + 1$ are multiples of N.

Then that means we've learned a factor of N.

For we just compute $\gcd(x^{s/2} - 1, N)$, which will give us either or .

Because if neither $x^{s/2} - 1$ nor $x^{s/2} + 1$ contains a full N, then one must contain a multiple of and the other a multiple of .

Furthermore, both of these "imagine we get lucky" steps can be shown to happen with a constant probability over the choice of x, by using a little number theory that we won't go into here. The precise statement is as follows:

For any N, if x is randomly chosen relatively prime to N, then with probability at least $\frac{3}{8}$:

- s is even
- Neither $x^{s/2} - 1$ nor $x^{s/2} + 1$ is a multiple of N

This gives us a plan of attack for…

Shor's Algorithm (The Quantum Part)

Basically, we'll give a quantum algorithm that solves the black-box problem of period-finding, in time polynomial in $\log(N)$. We'll then apply that algorithm to find the period of the function

$$f(r) = x^r \pmod{N},$$

for a randomly chosen base x. (I.e., we'll use the above f to "instantiate" the black box.) By the previous discussion, this f can be computed in polynomial time—and better yet, the ability to find its period implies the ability to factor N.

When we write f, what we really mean is f_x for a specific base x that we choose. As we saw before, to factor a given N, we might need to try several different values of x until we find one that works.

The first step is to make an equal superposition over all positive integers r less than some upper bound Q, with each integer written in binary notation: $\frac{1}{\sqrt{Q}} \sum_{r=0}^{Q-1} |r\rangle|f(r)\rangle$

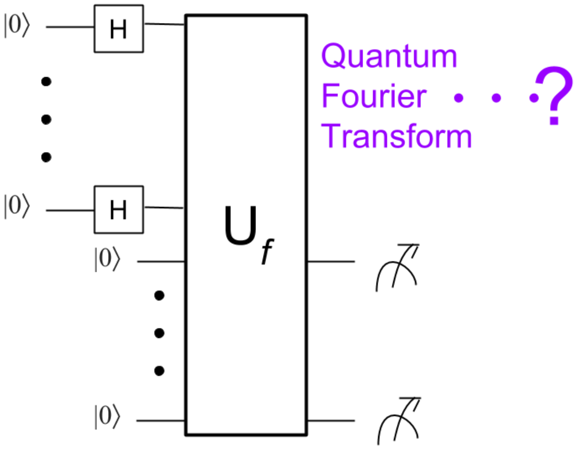
For technical reasons, we set Q to be the smallest power of 2 larger than N^2.

We can prepare this state by Hadamarding the qubits in the first register, then querying f.

Unlike with Simon's algorithm and so forth, in Shor's algorithm we don't need to treat U_f as an abstract black box. We can find an actual quantum circuit to implement U_f, because f is not arbitrary:

$$f(r) = x^r \pmod{N}$$

By using the repeated squaring trick, we can create actual circuit U_f that maps $|r\rangle|0\ldots0\rangle$ to $|r\rangle|f(r)\rangle$, out of a network of $\mathrm{polylog}(N)$ Toffoli gates.



We'll use ancilla qubits to store the output $|f(r)\rangle$. Then, just like with Simon's algorithm, we won't care at all about the actual value of $f(r)$—only about the effect that computing $f(r)$ had on the $|r\rangle$ register. So for pedagogical purposes, we'll immediately measure the $|f(r)\rangle$ register and then discard the result.

What's left in the $|r\rangle$ register?

By the partial measurement rule, what's left is an equal superposition over all the possible r's that could've led to the observed value $f(r)$. Since f is periodic with a secret period s, these values will differ from each other by multiples of s. In other words, we now have:

$$\frac{1}{\sqrt{\text{The quantity of these}}}\left[|r\rangle + |r+s\rangle + |r+2s\rangle + \ldots\right]$$

The central challenge of Shor's algorithm is to measure the above state in a way that reveals useful information about the period s. Just like with Simon's algorithm, *if* we could measure the state multiple times, then we could compare the results to each other and take some gcd's to find s. But we can't do that! We can repeat the whole algorithm from the beginning, but if we do, then we'll almost certainly end up with a different offset r, preventing useful comparisons.

This simply means that, just like with everything else in quantum computing, to see a speedup at some point we'll have to exploit the magic of minus signs: interference, cancellations, change of basis, whatever term you want to use. But how do we change the basis to one that reveals the period s—and moreover, do so efficiently, using a number of gates that's only polynomial in $\log(N)$?

That brings us to the topic of the next lecture: the **Quantum Fourier Transform**!