

Modular arithmetic

Much of modern number theory, and many practical problems (including problems in cryptography and computer science), are concerned with *modular arithmetic*. While this is probably familiar to most people taking this course, I will review it briefly.

In arithmetic modulo N , we are concerned with arithmetic on the integers, where we identify all numbers which differ by an exact multiple of N . That is, $x \equiv y \pmod{N}$ if $x = y + mN$ for some integer m .

This identification divides all the integers into N equivalence classes. We usually denote these by their “simplest” members, that is, the numbers $0, 1, \dots, N - 1$. (Usually. In the case of clock arithmetic (modulo 12), we use $1, \dots, 12$ instead.)

Most ordinary arithmetic operations extend to modular arithmetic straightforwardly.

$$x + y \rightarrow x + y \bmod N,$$

$$xy \rightarrow xy \bmod N,$$

$$x^y \rightarrow x^y \bmod N.$$

This does lead to some things which are strange based on the intuition from *ordinary* integer arithmetic. For instance, all numbers have additive inverses, but these are now represented by positive numbers: $(-x) \equiv N - x$, so the additive inverse of 3 modulo 7 is 4.

And unlike ordinary arithmetic, it is possible for a non-zero integer to have a *multiplicative* inverse, as well: $3 \cdot 5 = 15 = 1 \bmod 7$. In fact, for *prime* N , all numbers $1, \dots, N - 1$ have multiplicative inverses. If N is composite, then all numbers which have no common factor with N have multiplicative inverses.

Order-finding

For integers x and N with no common factor, the *order* of x modulo N is the least positive integer r such that

$$x^r = 1 \pmod{N}.$$

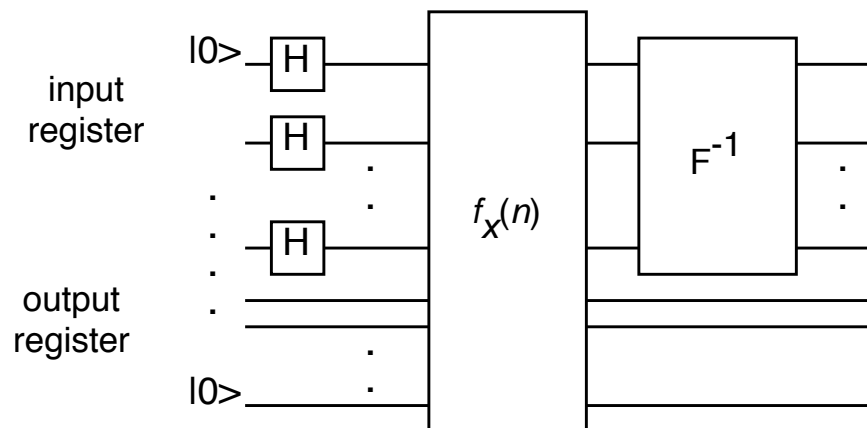
Obviously, $r \leq N$. (If not, the sequence of numbers $x^n \pmod{N}$ for $n = 1, \dots, r$ must all be distinct modulo N , which is impossible, since there are only N equivalence classes.)

Our problem is, given x and N , to find the order r of x modulo N . The description of the problem just requires the statement of x and N ; we parametrize the size of the problem by $L = \log_2 N$, the number of bits needed to state N . No classical algorithm for order-finding is known which is polynomial in L .

Building a quantum algorithm

The first thing to notice is that we can define a function $f_x(n) = x^n \bmod N$. Since the order r means that $x^r = 1 \bmod N$, this means that $f_x(n + r) = x^{n+r} \bmod N = x^n x^r \bmod N = x^n \bmod N = f_x(n)$. So the function is *periodic* with period r . Moreover, the $f(n)$ must all be distinct for $0 \leq n < r$.

We can therefore build a circuit for order-finding based on our circuit for period-finding:



We have divided the problem into two sub-circuits. The first performs the unitary $\hat{U}_{f_x}(|n\rangle|y\rangle) = |n\rangle|y \oplus f_x(n)\rangle$. The second performs the inverse Fourier transform on the input register. Both the input and the output registers start in the state $|0\rangle$, and the input register is put into a superposition of all $|n\rangle$ by Hadamard gates.

In fact, for this problem, it is better to have the second register start in the state $y = 1$, and create the unitary $\hat{U}'_{f_x}(|n\rangle|y\rangle) = |n\rangle|y \cdot f_x(n) \bmod N\rangle$. Note that this multiplication is invertible as long as x and N have no common factors; we would just multiply $yx^n \bmod N$ by x^{r-n} where r is the order of x . (Of course, we don't know what r is, but that doesn't change the fact that the multiplication is invertible in principle.) We call this unitary operator the circuit for *modular exponentiation*.

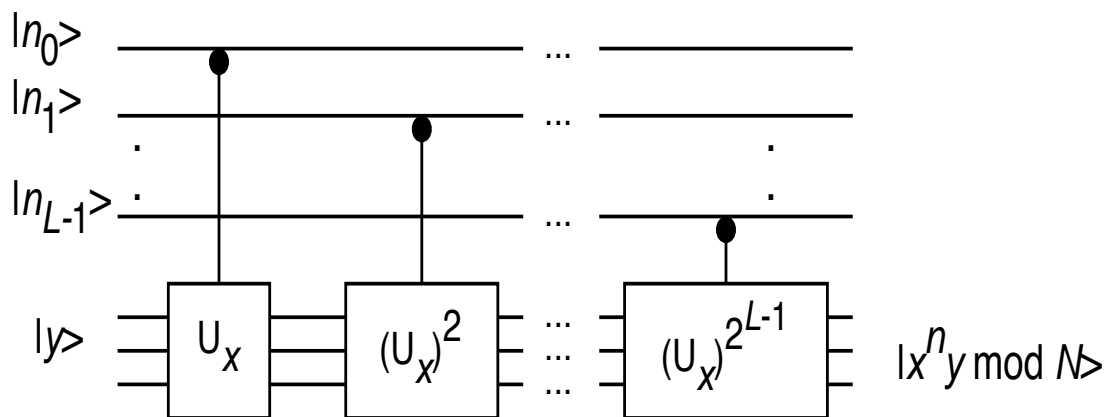
We can build modular exponentiation out of repeated applications of modular multiplication. Define a unitary operator \hat{U}_x such that

$$\hat{U}_x|y\rangle = |xy \bmod N\rangle.$$

Let n be the argument of the modular exponential function, with L -bit binary representation $n_{L-1}n_{L-2} \dots n_0 = n_{L-1}2^{L-1} + \dots + n_0$.

$$x^n y \bmod N = x^{n_{L-1}2^{L-1}} x^{n_{L-2}2^{L-2}} \dots x^{n_0} y.$$

That is, we successively multiply y by x^{2^j} if $n_j = 1$, and by 1 otherwise. Turning this into a quantum circuit we get:



This circuit looks very familiar—it is the same circuit used in phase estimation. This is no surprise, since we have seen that period-finding is an instance of phase estimation.

Of course, written in this form, the circuit is only efficient if we can do all these controlled- $\hat{U}_x^{2^j}$ operations efficiently. We will now see that this is the case.

We build these out circuits out of two other circuits: *modular multiplication* and *modular squaring*. These work as follows:

$$\hat{U}_m(|x\rangle|y\rangle) = |x\rangle|xy \bmod N\rangle,$$

$$\hat{U}_2|x\rangle = |x^2 \bmod N\rangle.$$

Again, these are invertible as long as x has no common factors with N . (We don't care what they do in other cases, so it is possible to construct unitaries that do what we want.)

We also need a scratch register, which we start in the state $|1\rangle$. Let us write this first, so our full state is $|1\rangle|y\rangle$. The first thing we do is apply \hat{U}_x to the scratch register:

$$|1\rangle|y\rangle \rightarrow |x\rangle|y\rangle.$$

We now apply \hat{U}_2 j times to perform $\hat{U}_x^{2^j}$:

$$\begin{aligned} |x\rangle|y\rangle &\rightarrow |x^2 \bmod N\rangle|y\rangle \rightarrow |x^4 \bmod N\rangle|y\rangle \\ &\rightarrow \dots \rightarrow |x^{2^j} \bmod N\rangle|y\rangle. \end{aligned}$$

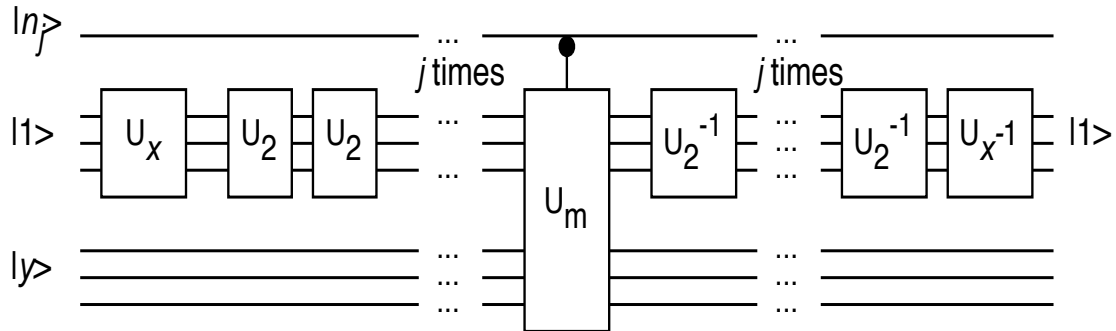
We then do modular multiplication \hat{U}_m :

$$|x^{2^j} \bmod N\rangle|y\rangle \rightarrow |x^{2^j} \bmod N\rangle|x^{2^j} y \bmod N\rangle.$$

Finally, we want to re-use the scratch space, so we “uncompute” $x^{2^j} \bmod N$ by applying \hat{U}_2^\dagger j times and then \hat{U}_x^\dagger , making the whole transformation

$$|1\rangle|y\rangle \rightarrow |1\rangle|x^{2^j} y \bmod N\rangle.$$

The circuit for controlled- $\hat{U}_x^{2^j}$ looks like this:



(It would be more efficient to keep our partial results for $x^{2^{j-1}}$ for use with each successive n_j , but that does not alter the principle.)

We can build modular squaring out of modular multiplication; and modular multiplication can be built out of modular addition and modular multiplication by two. That is, if the binary expressions are $x = x_{L-1} \dots x_0$ and $y = x_{L-1} \dots y_0$, then $xy \bmod N$ is just

$$xy \bmod N = x_0y + 2x_1y + \dots + 2^{L-1}x_{L-1}y \bmod N.$$

Simple reversible circuits exist for both of these procedures.

We can now summarize the order-finding algorithm:

1. Prepare the input register in state $|0\rangle$ and the output register in state $|1\rangle$.
2. With Hadamard gates, put the input register in a superposition of all values $|n\rangle$.
3. Calculate the modular exponential function.
4. Do the inverse Fourier transform on the input register.
5. Measure the input register.
6. Find the order r using the continued fraction algorithm.

Our circuit uses $O(L^3)$ gates. The input register requires $t = 2L + 1 + \log(1 + 1/2\epsilon)$ bits to succeed with probability $p > 1 - \epsilon$.

Order-finding and factoring

While order-finding may seem of limited interest by itself, the problem of factoring large numbers reduces to order-finding for its most difficult cases. To state the problem concretely: given a (large) composite number N , we want to find one of its prime factors.

The algorithm proceeds in several steps, mostly eliminating special cases for which order-finding fails, but alternative efficient algorithms exist.

1. Check if N is even. If it is, obviously 2 is a factor.
2. Check if N is a power a^b for integers a and b . An efficient algorithm for this exists.
3. Choose a random integer x , $1 < x < N - 1$. Calculate $\text{GCD}(x, N)$ using Euclid's algorithm. If it is not 1, congratulations!

4. Use the order-finding algorithm to find the order r of x modulo N .

At this point, we use some number theory. If r is even, we calculate $x^{r/2} \bmod N$. If this is not $N - 1 \equiv -1 \bmod N$ then we calculate $\text{GCD}(x^{r/2} \pm 1, N)$. If one of these gives a non-trivial factor, that is our answer. Otherwise the algorithm fails.

This may seem like a lot of conditions. But in fact, r has at least a 50% probability of being even and not having $x^{r/2} = -1 \bmod N$. If the algorithm fails, we just pick a new value of x and try again. The probability is overwhelming that we will succeed after only a few repetitions.

This algorithm is $O(L^3)$ (from modular exponentiation and the continued fraction). The best classical algorithm has a complexity $O(e^{L^{1/3}})$, which is superpolynomial.

RSA

Factoring is of interest largely because the difficulty of factoring is used to guarantee the security of popular public-key cryptography codes, such as Diffie-Hellman and RSA. Let's briefly look at RSA to see how factoring plays a role.

1. Alice picks two large prime numbers, p and q , and calculates $N = pq$. She also picks another number e which is mutually prime to $(p - 1)(q - 1)$. She publishes the numbers N and e ; they form her *encryption key*.

2. Privately, Alice also calculates a number d such that $ed = 1 \pmod{(p - 1)(q - 1)}$. (She can do this using Euclid's algorithm.) This is her *decryption key*.

3. Bob encrypts messages by first converting them to binary form and considering them to be a large number M . This is the *plaintext*. He then calculates $C = M^e \bmod N$. This new number C is the *ciphertext*.

4. Bob transmits C to Alice. When she receives it, she calculates $M = C^d \bmod N$. This will be Bob's original message.

The function $M^e \bmod N$ is a *one-way function*; it is virtually impossible (using standard techniques) to invert the function without the value of d (or the numbers p and q). Obviously, an efficient factoring algorithm would destroy the security of RSA.

There are other public-key cryptosystems which do not rely on factoring. However, several of these can also be broken by quantum computers. For instance, another system relies on the difficulty of the *discrete logarithm* problem: given a and b , finding the smallest s such that $a^s = b$.

No known classical algorithm can solve this in polynomial time; but a quantum computer can do so in a time $O(L^3)$.

Factoring, the discrete logarithm, and period-finding are all examples of a general problem known as the *Abelian hidden subgroup problem*, and the algorithms for solving them all have the same general structure.

Next time: Grover's search algorithm.