Factoring

Lecturer: Michel Goemans

We've seen that it's possible to efficiently check whether an integer n is prime or not. What about factoring a number? If this could be done efficiently (for example, in say D^4 operations, where $D \sim \log n$ is the number of digits of n), then the RSA encryption scheme could be broken, just by factoring $N = pq$ in the public key.

Probably the first algorithm that comes to mind is the brute-force approach: for each positive integer d greater than 1 and not larger than $\sqrt{n}$, check if d divides n. The reason we go up to $\sqrt{n}$ is simply that if n is composite, it must have a divisor no larger than $\sqrt{n}$. This can be done very efficiently for any particular choice of d, using the Euclidean algorithm: compute $\gcd(d, n)$, and if it's not 1, the result is a nontrivial[1] factor of n. Unfortunately, this method is extremely slow, just because of the number of potential divisors to check: $\sqrt{n}$.

We will see two faster algorithms. The first, *Pollard's rho algorithm* will require roughly $n^{1/4}$ gcd operations (rather than $n^{1/2}$ as above). The second, the *quadratic sieve*, will run roughly in time $e^{\sqrt{\log n \log\log n}}$. This looks a bit complicated, but notice that

$$(\log n)^C = e^{C \log\log n} \quad \text{and} \quad n^\epsilon = e^{\epsilon \log n}.$$

So $e^{\sqrt{\log n \log\log n}}$ lies inbetween these two, in terms of its speed of growth as n gets large.

Pollard's rho algorithm

Suppose n is the number to be factored, and $n = pq$, where p is a prime factor not larger than $\sqrt{n}$ (q need not be prime here, if n has more than 2 prime factors). Note that we *do not know* p; once we figure out p, we're done!

Consider the following thought experiment. Pick some numbers $x_1, x_2, \ldots, x_\ell$ uniformly at random in $\mathbb{Z}_n$ (the choice of ℓ will be decided later). Assume that all the numbers are distinct (if n is large compared to ℓ, this is very unlikely anyway). Now suppose that

$$\text{there exists some } 1 \le i < j \le \ell \text{ such that } x_i \equiv x_j \pmod{p}. \tag{1}$$

Then $p \mid x_i - x_j$, and since $p \mid n$ also, we have that

$$p \mid \gcd(x_i - x_j, n).$$

Moreover, since $-n < x_i - x_j < n$ and $x_i \neq x_j$, $\gcd(x_i - x_j, n) < n$. Thus $\gcd(x_i - x_j, n)$ provides a nontrivial factor of n, and our job is done.

Again, keep in mind that we do not know p. But, the above tells us that if we computed $\gcd(x_{i'} - x_{j'}, n)$ for every pair $1 \le i' < j' \le \ell$, we would find the nontrivial factor. But how big does ℓ need to be such that the chance that (1) holds is reasonable?

[1] "Nontrivial" here just means that the factor is neither 1, nor n itself.

The answer is roughly $\sqrt{p}$. This is essentially a reformulation of the "Birthday paradox": the somewhat surprising fact that in a group of only 30 people, the probability that two people in the group have the same birthday is already quite large—more than 60%. Let's calculate the probability that (1) does not hold, i.e., that the residue classes ("birthdays") of $x_1, x_2, \ldots, x_\ell$ are all different. This is

$$\begin{aligned}
\mathbb{P}(\text{all different}) &= \left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right)\cdots\left(1 - \frac{\ell-1}{n}\right) \\
&\leq e^{-1/n} \cdot e^{-2/n} \cdots e^{-(\ell-1)/n} \qquad \text{using } 1 - y \leq e^{-y} \\
&= e^{-\frac{\ell(\ell-1)}{2n}} \\
&\sim e^{-\ell^2/(2n)}.
\end{aligned}$$

So if $\ell = \sqrt{n}$, the probability is roughly $e^{-1/2} \leq 2/3$, and so there is a substantial probability that (1) occurs.

Since $p \leq \sqrt{n}$, we only need to choose $\ell = n^{1/4}$ (remember we don't know p!). But—then the number of pairs i, j is $\binom{\ell}{2}$ which is about $\frac{1}{2}\sqrt{n}$. Checking all these pairs takes again about $\sqrt{n}$ gcd computations—this is no better than brute search! What whas the point?

The clever trick here is to *not* pick the x_i's randomly, but instead in a way that "looks" random. Let $f : \mathbb{Z}_n \to \mathbb{Z}_n$ be defined by

$$f(x) = x^2 + 1 \bmod n.$$

Fix any $x_0 \in \mathbb{Z}_n$. Now consider the sequence

$$x_1 = f(x_0), \qquad x_2 = f(x_1), \qquad \ldots \qquad x_i = f(x_{i-1}), \ldots.$$

Empirical "fact": The sequence $x_0, x_1, \ldots$ "looks random".

This is not a precise statement, and indeed there is no formal proof that the algorithm that we will describe now actually works efficiently. But it is well established empirically that it does, in that for most choices of x_0, there will be a pair $i < j < C\sqrt{p}$ (for some small constant C) with $x_i \equiv x_j \pmod{p}$. The plan now is to look for such a pair i, j in this sequence (rather than a randomly generated one). But we still can't check every pair i', j', so how does this help?

The observation is that we're actually looking for *cycles* in the sequence

$$x_0 \bmod p, \quad x_1 \bmod p, \quad x_2 \bmod p, \quad \ldots.$$

For suppose $x_i \equiv x_j \pmod{p}$, with $i < j$ and i chosen as small as possible so that this holds. Then $f(x_i) \equiv f(x_j) \pmod{p}$, i.e., $x_{i+1} \equiv x_{j+1} \pmod{p}$ (you should check this!). Figure 1 shows the situation schematically: the points in the picture represent the values $\mathbb{Z}_p$. Eventually the sequence $x_0 \bmod p, x_1 \bmod p, \ldots$ runs into itself, and from then one goes around a cycle. (The name "Pollard's rho algorithm" comes from this picture...). It's worth recalling again at this point that we don't know p, so we cannot directly see the cycle. Nevertheless, we can find it very efficiently with the following algorithm.

Tortoise and hare algorithm:

1. Let $y_0 = x_0$.

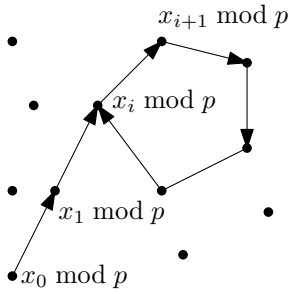

Figure 1: A cycle formed in the sequence $x_0 \bmod p, x_1 \bmod p, \ldots$.

2. For $i = 1, 2, \ldots$
   (a) $x_i = f(x_{i-1})$.
   (b) $y_i = f(f(y_{i-1}))$
   (c) If $\gcd(x_i - y_i, n) \neq 1$, return this discovered factor.

Informally, we start a tortoise (the sequence (x_i)) and a hare (the sequence (y_i)) at x_0, and the tortoise moves one step each turn, but the hare moves two. Once both the tortoise and the hare are on the cycle, it's just a matter of time until the hare runs into the tortoise. Our "empirical fact" implies that this algorithm will generally finish within $C\sqrt{p} \leq Cn^{1/4}$ steps for some small constant C; if we run for much longer than this without finding a cycle, we can give up and start again with a different choice of x_0.

The quadratic sieve

This algorithm is closely related to the currently fastest known method for factoring. As already mentioned, it takes time roughly $e^{\sqrt{\log n \log\log n}}$.

The approach is based on the following idea. If we could find two numbers a, b such that $a \not\equiv b \pmod{n}$, $a \not\equiv -b \pmod{n}$, but $a^2 \equiv b^2 \pmod{n}$, then we can easily get a factor. For we have that $n \mid (a-b)(a+b)$, but $n \not| a-b$ and $n \not| a+b$. Thus $\gcd(a+b, n)$ will be a nontrivial factor (as will $\gcd(a-b, n)$, but we only need one).

Let $g : \mathbb{Z}_n \to \mathbb{Z}$ be the function defined by $g(z) = z^2 \bmod n$. Now if we could find some integer z, with $\sqrt{n} \leq z \leq \sqrt{n} + K$ (K is something big, but much smaller than $\sqrt{n}$) and where $g(z)$ is a perfect square, then we are done. (Note: by perfect square, we really mean a perfect square as an integer, not as an element of $\mathbb{Z}_n$. So we mean $g(z) = y^2$ for some integer y, not $g(z) \equiv y^2 \pmod{n}$.) For let $g(z) = y^2$. Then $y^2 \equiv z^2 \pmod{n}$. But $z \not\equiv y \pmod{n}$, since $y < \sqrt{n} \leq z < n$. Moreover, if K is not too large, it can also be shown that $z \not\equiv -y \pmod{n}$. So then we can obtain a nontrivial factor from $\gcd(y - z, n)$ as already discussed.

Example. Suppose $n = 1817$. Then $\lceil \sqrt{n} \rceil = 43$. We get lucky: $g(51) = 784 = 28^2$. So $51^2 \equiv 28^2 \pmod{1817}$, hence $23 \cdot 79 \equiv 0 \pmod{1817}$, and we have a factor.

Unfortunately, starting from $\lceil \sqrt{n} \rceil$ and working our way up one at a time looking for a z s.t. $g(z)$ is a perfect square does not work well; the special values of z are rare, and so the number of

tries needed is again huge. But there is another hope. Suppose we find distinct $z_1, z_2, \ldots, z_\ell \in \mathbb{Z}_n$, with all $z_i \geq \sqrt{n}$, and where

$$\prod_{i=1}^{\ell} g(z_i) \quad \text{is a perfect square.} \tag{2}$$

(Again, note that we multiply the numbers $g(z_i)$ as normal integers, not modulo n; so this product can be larger than n). Then this is also (probably) good for us: let $z = \prod_{i=1}^{\ell} z_i$ and $y^2 = \prod_{i=1}^{\ell} g(z_i)$. Then $y^2 \equiv z^2 \pmod{n}$, and so as long as $y \not\equiv \pm z \pmod{n}$, we again find our nontrivial factor from $\gcd(y+z, n)$. It is (vaguely speaking) unlikely that $y \equiv \pm z \pmod{n}$, though we won't deal with this formally; if we get unlucky, we have to try for another collection of z_i's.

Is it any easier to find a collection of z_i's satisfying (2)? Notice that given an integer m with prime factorization $m = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_t^{\alpha_t}$, then m is a perfect square if and only iff α_j is even for each $1 \leq j \leq t$. The idea will be to work with z_i's so that $g(z_i)$ contain only small primes in their prime factorization, which will allow us to exploit this characterization of being a perfect square.

Definition 1. For $B \in \mathbb{N}$, an integer is called *B-smooth* if all its prime factors are at most B.

For example, 15 and 75 are 5-smooth, but 14 is not 5-smooth.

If B is not too large, we can i) check if an integer m is B-smooth, and ii) if it is, find its prime factorization, reasonably efficiently. Simply compute $\gcd(p_i, m)$ for each prime $p_i \leq B$ in turn, and if this is larger than 1, divide m by p_i (keeping track of the number of factors of p_i found). If m eventually reduces down to 1, we have a prime factorization of m in terms of primes not larger than B; otherwise, m is not B-smooth.

The plan is: pick some B (which will grow with n unfortunately, but much more slowly than n; roughly, $B = e^{\frac{1}{2}\sqrt{\log n \log\log n}}$), and find $B+1$ numbers $z_1, z_2, z_{B+1} \in \mathbb{Z}_n$, $z_i \geq \sqrt{n}$, so that $g(z_i)$ is B-smooth for each i. It turns out that these numbers are not so rare (depending on the choice of B), so we could do this just by trying each value z starting from $\lceil \sqrt{n} \rceil$, checking if $g(z)$ is B-smooth, as described above.

Once we have these numbers $z_1, \ldots, z_{B+1}$, we will in fact be able to find a *subset* of these numbers which satisfy (2)! This is a consequence of some linear algebra, and the reason we wanted precisely $B+1$ numbers. Let $p_1, \ldots, p_t$ be the list of primes of size at most B. Let $g(z_i) = \prod_{j=1}^{t} p_j^{\alpha_{i,j}}$ be the prime factorization of $g(z_i)$ (of course, some $\alpha_{i,j}$'s can be zero). Then for a given subset $I \subseteq \{1, 2, \ldots, B+1\}$,

$$\prod_{i \in I} g(z_i) = \prod_{j=1}^{t} p_j^{\left(\sum_{i \in I} \alpha_{i,j}\right)}.$$

Thus, applying our condition for a number to be a perfect square,

$$\prod_{i \in I} g(z_i) \quad \text{is a perfect square} \qquad \Leftrightarrow \qquad \sum_{i \in I} \alpha_{i,j} \text{ is even } \forall j \in \{1, 2, \ldots, t\}. \tag{3}$$

Now the linear algebra connection. (Warning: some knowledge of linear algebra will be assumed here). Consider the $(B+1) \times t$ matrix A, with entries $A_{ij} = \alpha_{i,j} \bmod 2$. We think of the entries of A as being in $\mathbb{Z}_2$. Let A_i denote the i'th row of A (so this is a vector). Then what we are looking for is a nonempty collection of rows $I \subseteq \{1, 2, \ldots, B+1\}$ so that $\sum_{i \in I} A_i = 0$. The existence of such a subset I follows from a key concept in linear algebra, *linear dependence*. Since A has only

B columns, the *rank* of A is at most B; since there are more than B rows, this means there must be a linear dependence amongst the rows, i.e., $w_i \in \mathbb{Z}_2$ so that

$$\sum_{i=1}^{B+1} w_i A_i = 0$$

(and not all the w_i are zero.) But that's precisely what we want; just take $I = \{i \mid w_i = 1\}$. There is also an efficient algorithm to find this linear dependence.