Likewise, primality *proof* methods should only be attempted after a number is known to be a *pseudoprime* (that is, a **probable prime**).

And, from a practical viewpoint, whenever it is helpful we can assume that all 'small' prime factors have already been removed, perhaps simply by trial division. (The precise meaning of 'small' depends upon the context.)

The two factorization methods we show here are *not* the state of the art but have the virtue that they are relatively easy to explain, to understand, and to implement. Further, the *rho method* make a charming use of the birthday-paradox idea. The $p-1$ method is important because in public key cryptosystems the primes used must be chosen to avoid vulnerability to this and related attacks.

24.1 Pollard's Rho Method

Pollard's rho method quickly finds relatively small factors of composite numbers. It is a very simple factorization method which already runs several times faster than trial division for numbers whose smallest prime factor is about $1,000,000$. Further, it has the practical virtue that if a number has a small prime factor, then the method finds such a factor faster than it would find a large factor. On the other hand, it has the minor disadvantage of being probabilistic, so that one must always be alert to occurrence of 'bad cases'. Finally, it has a peculiar disadvantage that it is difficult to *prove* that it works as well as it does: On one hand, from an experimental viewpoint this may be perceived as being of no consequence. On the other hand, in applications where correctness really matters this is obviously a fatal flaw. This method was first described in [Pollard 1975].

The description of the algorithm is simple. Given integer n, initialize by setting $x = 2$, $y = x^2 + 1$.

- Compute $g = \gcd(x - y, n)$
- If $1 < g < n$, stop: g is a proper factor of n
- If $g = 1$, replace x by $x^2 + 1$ and y by $(y^2 + 1)^2 + 1$ and repeat.
- If $g = n$, we have *failure*, and the algorithm needs to be reinitialized. This rarely occurs.

Remark: Playing upon the birthday paradox, the number of cycles necessary to find a factor p of n should be roughly of the order of $\sqrt{p}$. If n is composite, then there is a prime factor $p \leq \sqrt{n}$, so this algorithm takes on the order of $\sqrt[4]{n}$ cycles to find a proper factor. (See further comments below.)

Remark: As described, we haven't imposed a limit on the number of cycles to run through. If the number n is known to be composite, this is most likely (!?) acceptable, since the worst thing that will happen is that this algorithm will take as much time as trial division. The most problematical case is the case where $g = n$, which fortunately seems to occur with small probability (difficult to estimate!).

Remark: In implementing this algorithm, probably a limit should be imposed upon the number of cycles to run through before making some adjustments, for which there are several choices. $\sqrt{n}$ steps is too many, and in any case would be no faster than trial division. At the other extreme, $\sqrt[4]{n}$ is probably not enough to allow for 'bad' cases. In practice, $100\sqrt[4]{n}$ cycles seems to be enough to find a proper factor. Still, it is quite reasonable to simply let the algorithm run to either success or the 'failure' state for numbers n *known* to be composite, both because it appears to run fast and without failure, and because it seems hard to give rigorous estimates for a good limitation to put on the number of cycles to allow. For a rare rigorous result about Pollard's rho method, see [Bach 1991].

Remark: Since, as just noted, the worst-case scenario for this algorithm is complete failure, it would be unwise to test primality by this algorithm, since failure can occur for composite numbers. Thus, in practice, before using Pollard's rho one should first apply a primality test to n. The Fermat test can be used here, since for n *failing* the Fermat test n, is definitely composite. Or the Miller–Rabin test can be used.

Remark: In H. Cohen, *A Course in Computational Algebraic Number Theory*, Springer-Verlag, 1993, it is asserted that the best response to the 'failure' state is to use a function other than the $x \to x^2 + 1$ above. The choices $f(y) = y^2 + c$ are computationally simplest, although the values $c = 0, -2$ are bad. (Why?) That source asserts that simply changing the initial value of x from the $x = 2$ above to something else is not wise.

Remark: There is no guarantee that a proper factor g found by this method will be prime, although in practice that tends to be the case. But finding a non-prime proper factor is perfectly fine progress toward factorization in any case.

Why does this work? The clearest explanation is unfortunately not at all rigorous, and attempts to be more rigorous do not succeed. Perhaps this is part of the charm of this algorithm. In any case, there are two key ingredients: the probabilistic idea involved in the birthday paradox, and the Floyd cycle-detection method, which is used to make exploitation of the birthday-paradox idea practical.

So suppose that n is a positive integer with proper divisor d. It does not matter whether d is prime or not, but only that d is much smaller than n. From the birthday paradox, we know that if we have more than $\sqrt{d}$ integers $x_1, x_2, \ldots, x_t$, then the probability is greater than $1/2$ that two of these will be the same modulo d. The idea of Pollard's rho method is that $\sqrt{d}$ is much smaller than $\sqrt{n}$, so we should expect that if we choose a 'random sequence' of integers $x_1, x_2, \ldots$, there will be two the same modulo d long before there are two the same modulo n. That is, supposing that $x_i = x_k \bmod d$ but not already knowing what d is, we can (often) *find* d by computing

$$g = \gcd(x_i - x_j, n)$$

(Keep in mind that computation of *gcd*'s via the Euclidean algorithm is relatively cheap.) More precisely, if $x_i = x_k \bmod d$ but $x_i \neq x_k \bmod n$, then this *gcd* will be

a multiple of d, and will be a divisor of n strictly smaller than n itself. This counts as good progress toward factorization of n.

A too-naive implementation of this as a means of hunting for proper factors of n is to compute the greatest common divisors $\gcd(x_i - x_j, n)$ as we go along, for *all* $i < j$. This is a bad idea: if we hope that we need about $\sqrt{d}$ random integers in order to find the proper factor d, then we will have had to make on the order of

$$\frac{1}{2}(\sqrt{d})^2 = \frac{1}{2}d$$

computations of *gcd*s. That is, we could as well have found d by *trial division*. (Each step in trial division is cheaper than a Euclidean algorithm execution.)

So we need a clever way of taking advantage of the birthday paradox. It is here that we make use of the specific manner in which Pollard's rho algorithm creates the supposedly random integers. First, we are having to pretend that the function $f(x) = x^2 + 1 \ \% \ n$ used in the algorithm is a **random map** from $\mathbf{Z}/n$ to itself. Since it appears that no one can prove much of anything in this direction, we won't try to say precisely what this might mean. But in any case if we make a sequence of supposedly random numbers by

$$\begin{aligned} x_0 &= 2 \\ x_1 &= f(x_0) \\ x_2 &= f(x_1) \\ x_3 &= f(x_2) \\ &\dots \end{aligned}$$

then each element in the sequence determines the next one completely. That is, if ever $x_j = x_i \bmod d$ with $i < j$, then also inevitably $x_{j+1} = x_{i+1}$, $x_{j+2} = x_{i+2}$, $x_{j+3} = x_{i+3}$, and so on. modulo d. In particular, for all $t \geq j$,

$$x_t = x_{t-(j-i)} \bmod d$$

Further, for the same reason,

$$x_t = x_{t-(j-i)} = x_{t-2(j-i)} = x_{t-3(j-i)} = \dots = x_{t-\ell(j-i)} \bmod d$$

as long as $t - \ell(j-i) \geq i$. That is, there is more structure here than if we merely had a growing *set* of random numbers.

Keep in mind that we don't necessarily care about the very *first* case that some $x_i = x_j$, but only about a *relatively early* case. **Floyd's cycle-detection method** is the following efficient way of looking for matches. First, we do not keep in memory the whole list of x_i's, as this would be needlessly inefficient. Rather, we just remember the last one computed. But at the same time we separately compute a sequence

$$\begin{aligned} y_1 &= x_2 \\ y_2 &= x_4 \\ y_3 &= x_6 \\ y_4 &= x_8 \\ &\dots \\ y_i &= x_{2i} \end{aligned}$$

The efficient way to compute the sequence of y_i's is by noting that

$$y_{i+1} = (y_i^2 + 1)^2 + 1 \ \% \ n$$

And we only remember the last y_i computed.

At each step we only remember the last x_t and y_t computed, and consider $\gcd(x_t - y_t, n)$. Why will this most likely find a proper factor? Let j be the first index so that $x_j = x_i \bmod d$ for some $i < j$. As noted above, this means that

$$x_t = x_{t-\ell(j-i)} \bmod d$$

whenever $t - \ell(j-i) \geq i$. So, taking $t = 2s$:

$$y_s = x_{2s} = x_{2s-\ell(j-i)} \bmod d$$

for all s with $2s - \ell(j-i) \geq i$. So when $s = \ell(j-i)$ with $\ell(j-i) \geq i$ (which of course holds for ℓ sufficiently large)

$$y_s = x_{2s} = x_{2s-\ell(j-i)} = x_{2s-s} = x_s \bmod d$$

whenever $s = 2s - \ell(j-i) \geq i$. This proves that the trick used above really does succeed in finding x_i and x_j which are the same mod d, *assuming that there are such.*

Remark: Again, the latter heuristic discussion assumes that the function $x \to x^2 + 1$ behaves 'randomly', and we have no assurance that this is so. Also, assuming the randomness of this map, it is possible to do a more precise analysis of the *expected* number of cycles before a match occurs. This analysis is related to but more sophisticated than what we said above.

Exercises

24.1.01 Use Pollard's rho method to find a proper factor of 1133.

24.1.02 Use Pollard's rho method to find a proper factor of 1313.

24.1.03 Use Pollard's rho method to find a proper factor of 1649.

24.2 Pollard's $p-1$ method

This method is specialized, working well only to find prime factors p so that $p-1$ is divisible only by 'small' factors, and not working particularly well outside those cases. Still, application of this test is relatively simple, so to *prevent* factorization attacks it must be taken into account.

Fix an integer B. An integer n is **B-smooth** if all its prime factors are less than or equal B. In this context B is a **smoothness bound**. An integer n is **B-power smooth** if all its prime *power* factors are less than or equal B.