CHAPTER 2

Primality Testing and Factoring

Chapter Goals

- To explain the basics of primality testing.
- To describe the most used primality-testing algorithm, namely Miller–Rabin.
- To examine the relationship between various mathematical problems based on factoring.
- To explain various factoring algorithms.
- To sketch how the most successful factoring algorithm works, namely the Number Field Sieve.

2.1. Prime Numbers

The generation of prime numbers is needed for almost all public key algorithms, for example

- In the RSA encryption or the Rabin encryption system we need to find primes p and q to compute the public key $N = p \cdot q$.
- In ElGamal encryption we need to find primes p and q with q dividing $p - 1$.
- In the elliptic curve variant of ElGamal we require an elliptic curve over a finite field, such that the order of the elliptic curve is divisible by a large prime q.

Luckily we shall see that testing a number for primality can be done very fast using very simple code, but with an algorithm that has a probability of error. By repeating this algorithm we can reduce the error probability to any value that we require.

Some of the more advanced primality-testing techniques will produce a certificate which can be checked by a third party to prove that the number is indeed prime. Clearly one requirement of such a certificate is that it should be quicker to verify than it is to generate. Such a primality-testing routine will be called a primality-proving algorithm, and the certificate will be called a proof of primality. However, the main primality-testing algorithm used in cryptographic systems only produces certificates of compositeness and not certificates of primality.

For many years this was the best that we could do; i.e. either we could use a test which had a small chance of error, or we spent a lot of time producing a proof of primality which could be checked quickly. However, in 2002 Agrawal, Kayal and Saxena presented a deterministic polynomial-time primality test thus showing that the problem of determining whether a number was prime was in the complexity class $\mathcal{P}$. However, the so-called AKS Algorithm is not used in practice as the algorithms which have a small error are more efficient and the error can be made vanishingly small at little extra cost.

2.1.1. The Prime Number Theorem: Before discussing these algorithms, we need to look at some basic heuristics concerning prime numbers. A famous result in mathematics, conjectured by Gauss after extensive calculation in the early 1800s, is the Prime Number Theorem:

Theorem 2.1 (Prime Number Theorem)**.** *The function $\pi(X)$ counts the number of primes less than X, where we have the approximation*

$$\pi(X) \approx \frac{X}{\log X}.$$

This means primes are quite common. For example, the number of primes less than 2^{1024} is about 2^{1014}. The Prime Number Theorem also allows us to estimate the probability of a random number being prime: if p is a number chosen at random then the probability it is prime is about

$$\frac{1}{\log p}.$$

So a random number p of 1024 bits in length will be a prime with probability

$$\approx \frac{1}{\log p} \approx \frac{1}{709}.$$

So on average we need to select 354 odd numbers of size 2^{1024} before we find one which is prime. Hence, it is practical to generate large primes, as long as we can test primality efficiently.

2.1.2. Trial Division: The naive test for testing a number p to be prime is one of trial division. We essentially take all numbers between 2 and $\sqrt{p}$ and see whether one of them divides p, if not then p is prime. If such a number does divide p then we obtain the added bonus of finding a factor of the composite number p. Hence, trial division has the advantage (compared with more advanced primality-testing/proving algorithms) that it either determines that p is a prime, or determines a non-trivial factor of p.

However, primality testing by using trial division is a terrible strategy. In the worst case, when p is a prime, the algorithm requires $\sqrt{p}$ steps to run, which is an exponential function in terms of the size of the input to the problem. Another drawback is that it does not produce a certificate for the primality of p, in the case when the input p is prime. When p is not prime it produces a certificate which can easily be checked to prove that p is composite, namely a non-trivial factor of p. But when p is prime the only way we can verify this fact again (say to convince a third party) is to repeat the algorithm once more.

Despite its drawbacks, however, trial division is the method of choice for numbers which are very small. In addition, partial trial division up to a bound Y is able to eliminate all but a proportion

$$\prod_{p<Y} \left(1 - \frac{1}{p}\right)$$

of all composites. This method of eliminating composites is very old and is called the Sieve of Eratosthenes. Naively this is what we would always do, since we would never check an even number greater than two for primality, since it is obviously composite. Hence, many primality-testing algorithms first do trial division with all primes up to say 100, so as to eliminate all but the proportion

$$\prod_{p<100} \left(1 - \frac{1}{p}\right) \approx 0.12$$

of composites.

2.1.3. Fermat's Test: Most advanced probabilistic algorithms for testing primality make use of the converse to Fermat's Little Theorem. Recall Lagrange's Theorem from Chapter 1; this said that if G is a multiplicative group of size $\#G$ then

$$a^{\#G} = 1$$

for all values $a \in G$. So if G is the group of integers modulo n under multiplication then

$$a^{\phi(n)} = 1 \pmod{n}$$

for all $a \in (\mathbb{Z}/n\mathbb{Z})^*$. Fermat's Little Theorem is the case where $n = p$ is prime, in which case the above equality becomes

$$a^{p-1} = 1 \pmod{p}.$$

So if n is prime we have that

$$a^{n-1} = 1 \pmod{n}$$

always holds, whilst if n is not prime then we have that

$$a^{n-1} = 1 \pmod{n}$$

is "unlikely" to hold.

Since computing $a^{n-1} \pmod{n}$ is a very fast operation (see Chapter 6) this gives us a very fast test for compositeness called the Fermat Test to the base a. Running the Fermat Test can only convince us of the compositeness of n. It can never prove to us that a number is prime, only that it is not prime.

To see why it does not prove primality consider the case $n = 11 \cdot 31 = 341$ and the base $a = 2$: we have

$$a^{n-1} = 2^{340} = 1 \pmod{341}.$$

but n is clearly not prime. In such a case we say that n is a (Fermat) pseudo-prime to the base 2. There are infinitely many pseudo-primes to any given base. It can be shown that if n is composite then, with probability greater than 1/2, we obtain

$$a^{n-1} \neq 1 \pmod{n}.$$

This gives us Algorithm 2.1 to test n for primality. If Algorithm 2.1 outputs (Composite, a) then

Algorithm 2.1: Fermat's test for primality

```
for $i = 0$ to $k - 1$ do
    Pick $a \in [2, ..., n-1]$.
    $b \leftarrow a^{n-1} \mod n$.
    if $b \neq 1$ then return (Composite, $a$).
return "Probably Prime".
```

we know

- n is definitely a composite number,
- a is a witness for this compositeness, in that we can verify that n is composite by using the value of a.

If the above algorithm outputs "Probably Prime" then

- n is a composite with probability at most $1/2^k$,
- n is either a prime or a so-called probable prime.

For example if we take

$$n = 43\,040\,357,$$

then n is a composite, with one witness given by $a = 2$ since

$$2^{n-1} \pmod{n} = 9\,888\,212.$$

As another example take

$$n = 2^{192} - 2^{64} - 1,$$

then the algorithm outputs “Probably Prime” since we cannot find a witness for compositeness. Actually this n is a prime, so it is not surprising we did not find a witness for compositeness!

However, there are composite numbers for which the Fermat Test will always output

“Probably Prime”

for every a coprime to n. These numbers are called Carmichael numbers, and to make things worse there are infinitely many of them. The first three are $561, 1105$ and 1729. Carmichael numbers have the following properties

- They are always odd.
- They have at least three prime factors.
- They are square free.
- If p divides a Carmichael number N, then $p-1$ divides $N-1$.

To give you some idea of their density, if we look at all numbers less than 10^{16} then there are about $2.7 \cdot 10^{14}$ primes in this region, but only $246\,683 \approx 2.4 \cdot 10^5$ Carmichael numbers. Hence, Carmichael numbers are rare, but not rare enough to be ignored completely.

2.1.4. Miller–Rabin Test: Due to the existence of Carmichael numbers the Fermat Test is usually avoided. However, there is a modification of the Fermat Test, called the Miller–Rabin Test, which avoids the problem of composites for which no witness exists. This does not mean it is easy to find a witness for each composite, it only means that a witness must exist. In addition the Miller–Rabin Test has probability of 1/4 of accepting a composite as prime for each random base a, so again repeated application of the algorithm leads us to reduce the error probability down to any value we care to mention.

The Miller–Rabin Test is given by the pseudo-code in Algorithm 2.2. We do not show that the Miller–Rabin Test works. If you are interested in the reason see any book on algorithmic number theory for the details, for example that by Cohen or Bach and Shallit mentioned in the Further Reading section of this chapter. Just as with the Fermat Test, we repeat the method k times with k different bases, to obtain an error probability of $1/4^k$ if the algorithm always returns “Probably Prime”. Hence, we expect that the Miller–Rabin Test will output “Probably Prime” for values of $k \geq 20$ only when n is actually a prime.

Algorithm 2.2: Miller–Rabin algorithm

```
Write $n-1 = 2^s \cdot m$, with $m$ odd.
for $j = 0$ to $k-1$ do
    Pick $a \in [2, ..., n-2]$.
    $b \leftarrow a^m \mod n$.
    if $b \neq 1$ and $b \neq (n-1)$ then
        $i \leftarrow 1$.
        while $i < s$ and $b \neq (n-1)$ do
            $b \leftarrow b^2 \mod n$.
            if $b = 1$ then return (Composite, $a$).
            $i \leftarrow i + 1$.
        if $b \neq (n-1)$ then return (Composite, $a$).
return “Probable Prime”.
```

If n is a composite then the value of a output by Algorithm 2.2 is called a Miller–Rabin witness for the compositeness of n, and under the Generalized Riemann Hypothesis (GRH), a conjecture believed to be true by most mathematicians, there is always a Miller–Rabin witness a for the compositeness of n with

$$a \leq O((\log n)^2).$$

2.1.5. Primality Proofs: Up to now we have only output witnesses for compositeness, and we can interpret such a witness as a proof of compositeness. In addition we have only obtained probable primes, rather than numbers which are one hundred percent guaranteed to be prime. In practice this seems to be all right, since the probability of a composite number passing the Miller–Rabin Test for twenty bases is around 2^{-40} which should never really occur in practice. But theoretically (and maybe in practice if we are totally paranoid) this could be a problem. In other words we may want real primes and not just probable ones.

There are algorithms whose output is a witness for the primality of the number. Such a witness is called a proof of primality. In practice such programs are only used when we are morally certain that the number we are testing for primality is actually prime. In other words the number has already passed the Miller–Rabin Test for a number of bases and all we now require is a proof of the primality.

The most successful of these primality-proving algorithms is one based on elliptic curves called ECPP (for Elliptic Curve Primality Prover). This itself is based on an older primality-proving algorithm based on finite fields due to Pocklington and Lehmer; the elliptic curve variant is due to Goldwasser and Kilian. The ECPP algorithm is a randomized algorithm which is not mathematically guaranteed to always produce an output, i.e. a witness, even when the input is a prime number. If the input is composite then the algorithm is not guaranteed to terminate at all. Although ECPP runs in expected polynomial time, i.e. it is quite efficient, the proofs of primality it produces can be deterministically verified even faster.

There is an algorithm due to Adleman and Huang which, unlike the ECPP method, is guaranteed to terminate with a proof of primality on input of a prime number. It is based on a generalization of elliptic curves called hyperelliptic curves and has never (to my knowledge) been implemented. The fact that it has never been implemented is not only due to the far more complicated mathematics involved, but is also due to the fact that while the hyperelliptic variant is mathematically guaranteed to produce a proof, the ECPP method will always do so in practice for less work effort.

2.1.6. AKS Algorithm: The Miller–Rabin Test is a randomized primality-testing algorithm which runs in polynomial time. It can be made into a deterministic polynomial-time algorithm, but only on the assumption that the Generalized Riemann Hypothesis is true. The ECPP algorithm and its variants are randomized algorithms and are expected to have polynomial-time run-bounds, but we cannot prove they do so on all inputs. Thus for many years it was an open question whether we could create a primality-testing algorithm which ran in *deterministic* polynomial time, and provably so on all inputs without needing to assume any conjectures. In other words, the question was whether the problem PRIMES is in complexity class $\mathcal{P}$?

In 2002 this was answered in the affirmative by Agrawal, Kayal, and Saxena. The test they developed, now called the AKS Primality Test, makes use of the following generalization of Fermat's test. In the theorem we are asking whether two polynomials of degree n are the same. Taking this basic theorem, which is relatively easy to prove, and turning it into a polynomial-time test was a major breakthrough. The algorithm itself is given in Algorithm 2.3. In the algorithm we use the notation $F(X) \pmod{G(X), n}$ to denote taking the reduction of $F(X)$ modulo *both* $G(X)$ and n.

Theorem 2.2. *An integer $n \geq 2$ is prime if and only if the relation*

$$(X-a)^n = (X^n - a) \pmod{n}$$

holds for some integer a coprime to n; or indeed all integers a coprime to n.

Algorithm 2.3: AKS primality-testing algorithm

```
if $n = a^b$ for some integers $a$ and $b$ then return "Composite".
Find the smallest $r$ such that the order of $n$ modulo $r$ is greater than $(\log n)^2$.
if $\exists a \leq r$ such that $1 < \gcd(a, n) < n$ then return "Composite".
if $n \leq r$ then return "Prime".
for $a = 1$ to $\lfloor \sqrt{\phi(r)} \cdot \log(n) \rfloor$ do
    if $(X + a)^n \neq X^n + a \pmod{X^r - 1, n}$ then return "Composite"
return Prime
```

2.2. The Factoring and Factoring-Related Problems

The most important one-way function used in public key cryptography is that of factoring integers. By factoring an integer we mean finding its prime factors, for example

$$\begin{aligned} 10 &= 2 \cdot 5, \\ 60 &= 2^2 \cdot 3 \cdot 5, \\ 2^{113} - 1 &= 3391 \cdot 23\,279 \cdot 65\,993 \cdot 1\,868\,569 \cdot 1\,066\,818\,132\,868\,207. \end{aligned}$$

There are a number of other hard problems related to factoring which can be used to produce public key cryptosystems. Suppose you are given an integer N, which is known to be the product of two large primes, but not its factors p and q. There are four main problems which we can try to solve:

- **FACTOR:** Find p and q.
- **RSA:** Given e such that
  $$\gcd(e, (p-1)(q-1)) = 1$$
  and c, find m such that
  $$m^e = c \pmod{N}.$$
- **SQRROOT:** Given a such that
  $$a = x^2 \pmod{N},$$
  find x.
- **QUADRES:** Given $a \in J_N$, determine whether a is a square modulo N.

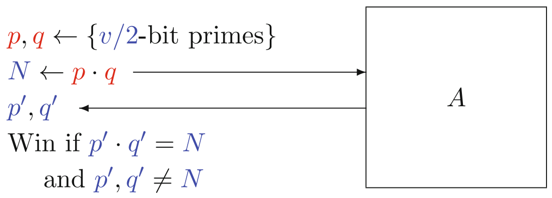


FIGURE 2.1. Security game to define the FACTOR problem

In Chapter 11, we use so-called security games to define security for cryptographic components. These are abstract games played between an adversary and a challenger. The idea is that the adversary needs to achieve some objective given only the data provided by the challenger. Such games tend to be best described using pictures, where the challenger (or environment) is listed on the outside and the adversary is presented as a box. The reason for using such diagrams will become clearer later when we consider security proofs, but for now they are simply going to be used to present security definitions.

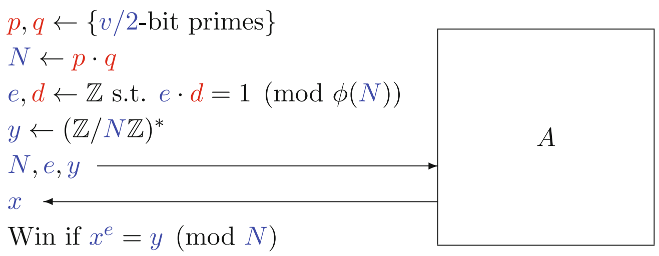

$p, q \leftarrow \{v/2\text{-bit primes}\}$
$N \leftarrow p \cdot q$
$e, d \leftarrow \mathbb{Z}$ s.t. $e \cdot d = 1 \pmod{\phi(N)}$
$y \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$
N, e, y
x
Win if $x^e = y \pmod{N}$
A

FIGURE 2.2. Security game to define the RSA problem

So for example, we could imagine a game which defines the problem of an adversary A trying to factor a challenge number N as in Figure 2.1. The challenger comes up with two secret prime numbers, multiplies them together and sends the product to the adversary. The adversary's goal is to find the original prime numbers. Similarly we can define games for the RSA and SQRROOT problems, which we give in Figures 2.2 and 2.3.

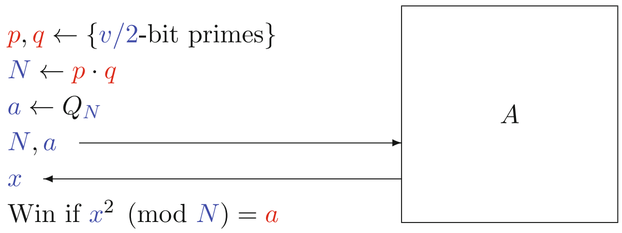

$p, q \leftarrow \{v/2\text{-bit primes}\}$
$N \leftarrow p \cdot q$
$a \leftarrow Q_N$
N, a
x
Win if $x^2 \pmod{N} = a$
A

FIGURE 2.3. Security game to define the SQRROOT problem

In all these games we define the advantage of a specific adversary A to be a function of the time t which the adversary spends trying to solve the input problem. For the Factoring, RSA and SQRROOT games it is defined as the probability (defined over the random choices made by A) that the adversary wins the game given that it runs in time bounded by t (we are not precise on what units t is measured in). We write

$$\mathrm{Adv}_v^X(A, t) = \Pr[A \text{ wins the game } X \text{ for } v = \log_2 N \text{ in time less than } t].$$

If the adversary is always successful then the advantage will be one, if the adversary is never successful then the advantage will be zero.

In the next section we will see that there is a trivial algorithm which always factors a number in time $\sqrt{N}$. So we know that *there is* an adversary A such that

$$\mathrm{Adv}_v^{\mathsf{FACTOR}}(A, 2^{v/2}) = 1.$$

However if t is any *polynomial* function p_1 of $v = \log_2 N$ then we expect that there is no efficient adversary A, and hence for such t we will have

$$\mathrm{Adv}_v^{\mathsf{FACTOR}}(A, p_1(v)) < \frac{1}{p_2(v)},$$

for *any* polynomial $p_2(x)$ and *for all* adversaries A. A function which grows less quickly than $1/p_2(x)$ for any polynomial function of $p_2(x)$ is said to be *negligible*, so we say the advantage of solving the factoring problem is negligible. Note that, even if the game was played again and again (but a polynomial in v number of times), the adversary would still obtain a negligible probability of winning since a negligible function multiplied by a polynomial function is still negligible.

In the rest of this book we will drop the time parameter from the advantage statement and implicitly assume that all adversaries run in polynomial time; thus we simply write $\mathrm{Adv}_Y^X(A)$, $\mathrm{Adv}_v^{\mathsf{FACTOR}}(A)$, $\mathrm{Adv}_v^{\mathsf{RSA}}(A)$ and $\mathrm{Adv}_v^{\mathsf{SQRROOT}}(A)$. We call the subscript the problem class; in the above this is the size v of the composite integers, in Chapter 3 it will be the underlying abelian group. The superscript defines the precise game which the adversary A is playing.

A game X for a problem class Y is said to be *hard* if the advantage is a negligible function *for all* polynomial-time adversaries A. The problem with this definition is that the notion of negligible is asymptotic, and when we consider cryptosystems we usually talk about concrete parameters; for example the fixed size of integers which are to be factored.

Thus, instead, we will deem a class of problems Y to be hard if for all polynomial-time adversaries A, the advantage $\mathrm{Adv}_Y^X(A)$ is a very small value ϵ; think of ϵ as being $1/2^{128}$ or some such number. This means that even if the run time of the adversary was one time unit, and we repeatedly ran the adversary a large number of times, the advantage that the adversary would gain would still be very very small. In this chapter we leave aside the issue of how small "small" is, but in later chapters we examine this in more detail.

The QUADRES problem is a little different as we need to define the probability distribution from which the challenge numbers a come. The standard definition is for the challenger to pick a to be a quadratic residue with probability 1/2. In this way the adversary has a fifty-fifty chance of simply guessing whether a is a quadratic residue or not. We present the game in Figure 2.4.

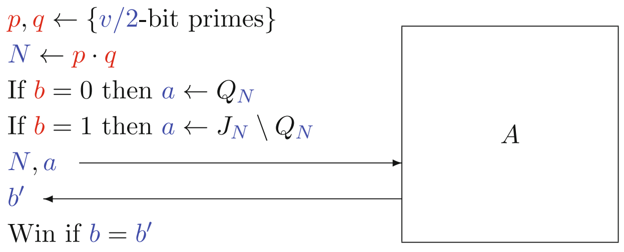

$p, q \leftarrow \{v/2\text{-bit primes}\}$
$N \leftarrow p \cdot q$
If $b = 0$ then $a \leftarrow Q_N$
If $b = 1$ then $a \leftarrow J_N \setminus Q_N$
N, a
b'
Win if $b = b'$
A

FIGURE 2.4. Security game to define the QUADRES problem

When defining the advantage for the QUADRES problem we need to be a bit careful, as the adversary can always win with probability one half by simply just guessing the bit b at random. Instead of using the above definition of advantage (i.e. the probability that the adversary wins the game), we use the definition

$$\mathrm{Adv}_v^{\mathsf{QUADRES}}(A) = 2 \cdot \left| \Pr[A \text{ wins the QUADRES game for } v = \log_2 N] - \frac{1}{2} \right|.$$

Notice that, with this definition, if the adversary just guesses the bit with probability 1/2 then its advantage is zero as we would expect, since $2 \cdot |1/2 - 1/2| = 0$. If however the adversary is always right, or indeed always wrong, then the advantage is one, since $2 \cdot |1 - 1/2| = 2 \cdot |0 - 1/2| = 1$. Thus the advantage is normalized to lie between zero and one, like in the earlier games, with one being always successful and zero being no better than random.

We call this type of game a *decision game* as the adversary needs to decide which situation it is being placed in. We can formulate the advantage statement for decision games in another way, as the following lemma explains.

Lemma 2.3. *Let A be an adversary in the QUADRES game. Then, if b' is the bit chosen by A and b is the bit chosen by the challenger in the game, we have*

$$\mathrm{Adv}_v^{\mathsf{QUADRES}}(A) = \left|\Pr[b' = 1|b = 1] - \Pr[b' = 1|b = 0]\right|.$$

Proof. The proof is a straightforward application of definitions of probabilities:

$$\begin{aligned}
\mathrm{Adv}_v^{\mathsf{QUADRES}}(A) &= 2 \cdot \left|\Pr[A \text{ wins}] - \frac{1}{2}\right| \\
&= 2 \cdot \left|\Pr[b' = 1 \text{ and } b = 1] + \Pr[b' = 0 \text{ and } b = 0] - \frac{1}{2}\right| \\
&= 2 \cdot \left|\Pr[b' = 1|b = 1] \cdot \Pr[b = 1] + \Pr[b' = 0|b = 0] \cdot \Pr[b = 0] - \frac{1}{2}\right| \\
&= 2 \cdot \left|\Pr[b' = 1|b = 1] \cdot \frac{1}{2} + \Pr[b' = 0|b = 0] \cdot \frac{1}{2} - \frac{1}{2}\right| \\
&= \left|\Pr[b' = 1|b = 1] + \Pr[b' = 0|b = 0] - 1\right| \\
&= \left|\Pr[b' = 1|b = 1] + \big(1 - \Pr[b' = 1|b = 0]\big) - 1\right| \\
&= \left|\Pr[b' = 1|b = 1] - \Pr[b' = 1|b = 0]\right|.
\end{aligned}$$

□

To see how this Lemma works consider the case when A is a perfect adversary, i.e. it wins the QUADRES game all the time. In this case we have $\Pr[A \text{ wins}] = 1$, and the advantage is equal to $2 \cdot |1 - 1/2| = 1$ by definition. However, in this case we also have $\Pr[b' = 1|b = 1] = 1$ and $\Pr[b' = 1|b = 0] = 0$. Hence, the formula from the Lemma holds. Now examine what happens when A just returns a random result. We obtain $\Pr[A \text{ wins}] = 1/2$, and the advantage is equal to $2 \cdot |1/2 - 1/2| = 0$. The Lemma gives the same result as $\Pr[b' = 1|b = 1] = \Pr[b' = 1|b = 0] = 1/2$.

When giving these problems it is important to know how they are related. We relate them by giving complexity-theoretic reductions from one problem to another. This allows us to say that "Problem B is no harder than Problem A". Assuming an oracle (or efficient subroutine) to solve Problem A, we create an efficient algorithm for Problem B. The algorithms which perform these reductions should be efficient, in that they run in polynomial time, where we treat each oracle query as a single time unit.

We can also show *equivalence* between two problems A and B, by showing an efficient reduction from A to B and an efficient reduction from B to A. If the two reductions are both polynomial-time reductions then we say that the two problems are *polynomial-time equivalent*. The most important result of this form for our factoring related problems is the following.

Theorem 2.4. *The FACTOR and SQRROOT problems are polynomial-time equivalent.*

The next two lemmas present reductions in both directions. By examing the proofs it is easy to see that both of the reductions can be performed in expected polynomial time. Hence, the problems FACTOR and SQRROOT are polynomial-time equivalent. First, in the next lemma, we show how to reduce SQRROOT to FACTOR; if there is no algorithm which can solve SQRROOT then there is no algorithm to solve FACTOR.

Lemma 2.5. *If A is an algorithm which can factor integers of size v, then there is an efficient algorithm B which can solve SQRROOT for integers of size v. In particular*

$$\mathrm{Adv}_v^{\mathsf{FACTOR}}(A) = \mathrm{Adv}_v^{\mathsf{SQRROOT}}(B).$$

PROOF. Assume we are given a factoring algorithm A; we wish to show how to use this to extract square roots modulo a composite number N. Namely, given

$$a = x^2 \pmod N$$

we wish to compute x. First we factor N into its prime factors p_1, p_2, ..., p_k, using the factoring oracle A. Then we compute

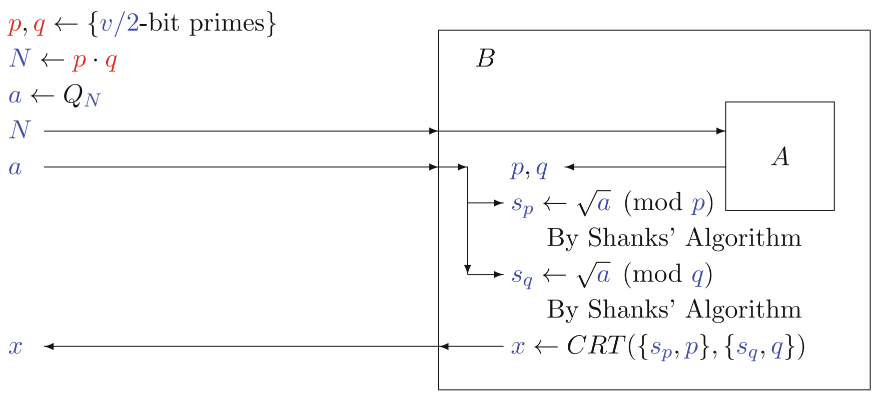
$$s_i \leftarrow \sqrt{a} \pmod{p_i} \text{ for } 1 \le i \le k.$$

This can be done in expected polynomial time using Shanks' Algorithm (Algorithm 1.3 from Chapter 1). Then we compute the value of x using the Chinese Remainder Theorem on the data

$$(s_1, p_1), \dots, (s_k, p_k).$$

We have to be a little careful if powers of p_i greater than one divide N. However, this is easy to deal with and will not concern us here, since we are mainly interested in integers N which are the product of two primes. Hence, finding square roots modulo N is no harder than factoring.

The entire proof can be represented diagramatically in terms of our game diagrams as in Figure 2.5; where we have specialized the game to one of integers N which are the product of two prime factors.



$p, q \leftarrow \{v/2\text{-bit primes}\}$
$N \leftarrow p \cdot q$
$a \leftarrow Q_N$
N
a
x
B
A
p, q
$s_p \leftarrow \sqrt{a} \pmod p$
By Shanks' Algorithm
$s_q \leftarrow \sqrt{a} \pmod q$
By Shanks' Algorithm
$x \leftarrow CRT(\{s_p, p\}, \{s_q, q\})$

FIGURE 2.5. Constructing an algorithm B to solve SQRROOT from an algorithm A to solve FACTOR

□

We now show how to reduce FACTOR to SQRROOT; if there is no algorithm which can solve FACTOR then there is no algorithm to solve SQRROOT.

Lemma 2.6. *Let A be an algorithm which can solve SQRROOT for integers of size v; then there is an efficient algorithm B which can factor integers of size v. In particular for N a product of two primes we have*

$$\mathrm{Adv}_v^{\mathsf{SQRROOT}}(A) = 2 \cdot \mathrm{Adv}_v^{\mathsf{FACTOR}}(B).$$

The proof of this result contains an important tool used in the factoring algorithms of the next section, namely the construction of a difference of two squares.

PROOF. Assume we are given an algorithm A for extracting square roots modulo a composite number N. We shall assume for simplicity that N is a product of two primes, which is the most difficult case. The general case is only slightly more tricky mathematically, but it is computationally easier since factoring numbers with three or more prime factors is usually easier than factoring numbers with two prime factors.

We wish to use our algorithm A for the problem SQRROOT to factor the integer N into its prime factors, i.e. given $N = p \cdot q$ we wish to compute p. First we pick a random $x \in (\mathbb{Z}/N\mathbb{Z})^*$ and compute

$$a \leftarrow x^2 \pmod{N}.$$

Now we compute

$$y \leftarrow \sqrt{a} \pmod{N}$$

using the SQRROOT algorithm. There are four such square roots, since N is a product of two primes. With fifty percent probability we obtain

$$y \neq \pm x \pmod{N}.$$

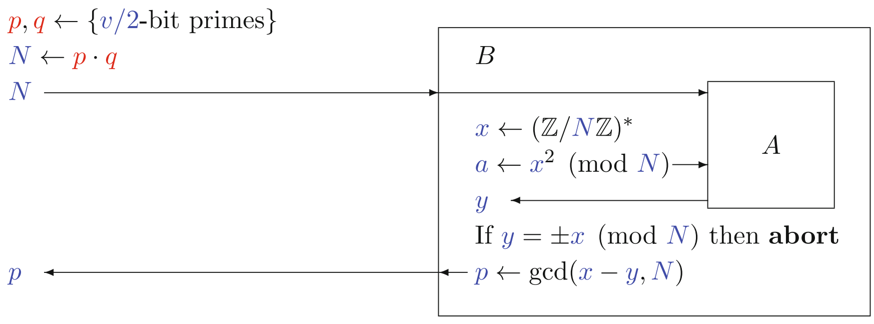
If we do not obtain this inequality then we abort.

We now assume that the inequality holds, but we note that we have the equality $x^2 = y^2 \pmod{N}$. It is then easy to see that N divides

$$x^2 - y^2 = (x - y)(x + y).$$

But N does not divide either $x - y$ or $x + y$, since $y \neq \pm x \pmod{N}$. So the factors of N must be distributed over $x - y$ and $x + y$. This means we can obtain a non-trivial factor of N by computing $\gcd(x - y, N)$

It is because of the above fifty percent probability that we get a factor of two in our advantage statement, since B is only successful if A is successful *and* we obtain $y \neq \pm x \pmod{N}$. Thus $\Pr[B \text{ wins}] = \Pr[A \text{ wins}]/2$. Diagrammatically we represent this reduction in Figure 2.6.



$p, q \leftarrow \{v/2\text{-bit primes}\}$
$N \leftarrow p \cdot q$
N
B
A
$x \leftarrow (\mathbb{Z}/N\mathbb{Z})^*$
$a \leftarrow x^2 \pmod{N}$
y
If $y = \pm x \pmod{N}$ then **abort**
$p \leftarrow \gcd(x - y, N)$
p

FIGURE 2.6. Constructing an algorithm B to solve FACTOR from an algorithm A to solve SQRROOT

□

Before leaving the problem SQRROOT, note that QUADRES is easier than SQRROOT, since an algorithm to compute square roots modulo N can trivially be used to determine quadratic residuosity.

Finally we end this section by showing that the RSA problem can be reduced to FACTOR. Recall the RSA problem is given $c = m^e \pmod{N}$, find m. There is some evidence, although slight, that the RSA problem may actually be easier than FACTOR for some problem instances. It is a major open question as to how much easier it is.

Lemma 2.7. *The RSA problem is no harder than the FACTOR problem. In particular, if A is an algorithm which can solve FACTOR for integers of size v, then there is an efficient algorithm B which can solve the RSA problem for integers of size v. In particular for N a product of two primes we have*

$$\mathrm{Adv}_v^{\mathsf{FACTOR}}(A) = \mathrm{Adv}_v^{\mathsf{RSA}}(B).$$

Proof. Using the factoring algorithm A we first find the factorization of N. We can now compute $\Phi = \phi(N)$ and then compute

$$d \leftarrow 1/e \pmod{\Phi}.$$

Once d has been computed it is easy to recover m via

$$c^d = m^{e \cdot d} = m^{1 \pmod{\Phi}} = m \pmod{N},$$

with the last equality following by Lagrange's Theorem, Theorem 1.4. Hence, the RSA problem is no harder than FACTOR. We leave it to the reader to present a diagram of this reduction similar to the ones above. □

2.3. Basic Factoring Algorithms

Finding factors is an expensive computational operation. To measure the complexity of algorithms to factor an integer N we often use the function

$$L_N(\alpha, \beta) = \exp\left((\beta + o(1))(\log N)^\alpha (\log\log N)^{1-\alpha}\right).$$

Note that

- $L_N(0, \beta) = (\log N)^{\beta + o(1)}$, i.e. essentially polynomial time,
- $L_N(1, \beta) = N^{\beta + o(1)}$, i.e. essentially exponential time.

So in some sense, the function $L_N(\alpha, \beta)$ interpolates between polynomial and exponential time. An algorithm with complexity $O(L_N(\alpha, \beta))$ for $0 < \alpha < 1$ is said to have sub-exponential behaviour. Note that multiplication, which is the inverse algorithm to factoring, is a very simple operation requiring time less than $O(L_N(0, 2))$.

There are a number of methods to factor numbers of the form

$$N = p \cdot q.$$

For now we just summarize the most well-known techniques.

- **Trial Division:** Try every prime number up to $\sqrt{N}$ and see whether it is a factor of N. This has complexity $L_N(1, 1)$, and is therefore an exponential algorithm.
- **Elliptic Curve Method:** This is a very good method if $p < 2^{50}$; its complexity is $L_p(1/2, c)$, for some constant c, which is a sub-exponential function. Note that the complexity is given in terms of the size of the smallest unknown prime factor p. If the number is a product of two primes of very unequal size then the elliptic curve method may be the best at finding the factors.
- **Quadratic Sieve:** This is probably the fastest method for factoring integers that have between 80 and 100 decimal digits. It has complexity $L_N(1/2, 1)$.
- **Number Field Sieve:** This is currently the most successful method for numbers with more than 100 decimal digits. It has factored numbers of size $10^{155} \approx 2^{512}$ and has complexity $L_N(1/3, 1.923)$.

Factoring methods are usually divided into Dark Age methods such as

- Trial division,
- $p - 1$ method,
- $p + 1$ method,
- Pollard rho method,

and modern methods such as

- Continued Fraction Method (CFRAC),
- Quadratic Sieve (QS),
- Elliptic Curve Method (ECM),
- Number Field Sieve (NFS).

We do not have space to discuss all of these in detail so we shall look at a couple of Dark Age methods and explain the main ideas behind some of the modern methods.

2.3.1. Trial Division: The most elementary algorithm is trial division, which we have already met in the context of testing primality. Suppose N is the number we wish to factor; we proceed as described in Algorithm 2.4. A moment's thought reveals that trial division takes time at worst

$$O(\sqrt{N}) = O\left(2^{(\log_2 N)/2}\right).$$

The input size to the algorithm is of size $\log_2 N$, hence this complexity is exponential. But just as in primality testing, we should not ignore trial division. It is usually the method of choice for numbers less than 10^{12}.

Algorithm 2.4: Factoring via trial division

```
for $p = 2$ to $\sqrt{N}$ do
    $e \leftarrow 0$.
    if $(N \mod p) = 0$ then
        while $(N \mod p) = 0$ do
            $e \leftarrow e + 1$.
            $N \leftarrow N/p$.
        output $(p, e)$.
```

2.3.2. Smooth Numbers: For larger numbers we would like to improve on the trial division algorithm. Almost all other factoring algorithms make use of other auxiliary numbers called smooth numbers. Essentially a smooth number is one which is easy to factor using trial division; the following definition makes this more precise.

Definition 2.8 (Smooth Number)**.** *Let B be an integer. An integer N is called B-smooth if every prime factor p of N is less than B.*

For example

$$N = 2^{78} \cdot 3^{89} \cdot 11^3$$

is 12-smooth. Sometimes we say that the number is just smooth if the bound B is small compared with N. The number of y-smooth numbers which are less than x is given by the function $\psi(x, y)$. This is a rather complicated function which is approximated by

$$\psi(x, y) \approx x\rho(u)$$

where ρ is the Dickman–de Bruijn function and

$$u = \frac{\log x}{\log y}.$$

The Dickman–de Bruijn function ρ is defined as the function which satisfies the following differential-delay equation

$$u \cdot \rho'(u) + \rho(u - 1) = 0,$$

for $u > 1$. In practice we approximate $\rho(u)$ via the expression

$$\rho(u) \approx u^{-u},$$

which holds as $u \to \infty$. This leads to the following result, which is important in analysing advanced factoring algorithms.

Theorem 2.9. *The proportion of integers less than x which are $x^{1/u}$-smooth is asymptotically equal to u^{-u}.*

Now if we set $y = L_N(\alpha, \beta)$ then

$$\begin{aligned} u &= \frac{\log N}{\log y} \\ &= \frac{1}{\beta}\left(\frac{\log N}{\log\log N}\right)^{1-\alpha}. \end{aligned}$$

Hence, we can show

$$\begin{aligned} \frac{1}{N}\psi(N, y) &\approx u^{-u} \\ &= \exp(-u \cdot \log u) \\ &\approx \frac{1}{L_N(1-\alpha, \gamma)}, \end{aligned}$$

for some constant γ.

Suppose we are looking for numbers less than N which are $L_N(\alpha, \beta)$-smooth. The probability that any number less than N is actually $L_N(\alpha, \beta)$-smooth is, as we have seen, given by $1/L_N(1 - \alpha, \gamma)$. This explains intuitively why some of the modern method complexity estimates for factoring are around $L_N(0.5, c)$, since to balance the smoothness bound against the probability estimate we take $\alpha = \frac{1}{2}$. The Number Field Sieve only obtains a better complexity estimate by using a more mathematically complex algorithm.

We shall also require, in discussing our next factoring algorithm, the notion of a number being B-power smooth:

Definition 2.10 (Power Smooth)**.** *A number is said to be B-power smooth if every prime power dividing N is less than B.*

For example $N = 2^5 \cdot 3^3$ is 33-power smooth.

2.3.3. Pollard's $P - 1$ Method: The most famous name in factoring algorithms in the late twentieth century was John Pollard. Almost all the important advances in factoring were made by him, for example

- The $P - 1$ method,
- The Rho-method,
- The Number Field Sieve.

In this section we discuss the $P - 1$ method and in a later section we consider the Number Field Sieve method.

Suppose the number we wish to factor is given by $N = p \cdot q$. In addition suppose we know (by some pure guess) an integer B such that $p - 1$ is B-power smooth, but that $q - 1$ is not B-power smooth. We can then hope that $p - 1$ divides $B!$, but $q - 1$ is unlikely to divide $B!$.

Suppose that we compute

$$a \leftarrow 2^{B!} \pmod{N}.$$

Imagine that we could compute this modulo p and modulo q, we would then have

$$a = 1 \pmod{p},$$

since

- $p - 1$ divides $B!$,
- $a^{p-1} = 1 \pmod{p}$ by Fermat's Little Theorem.

Algorithm 2.5: Pollard's $P-1$ factoring method

```
$a \leftarrow 2$.
for $j = 2$ to $B$ do
    $a \leftarrow a^j \mod N$.
$p \leftarrow \gcd(a-1, N)$.
if $p \neq 1$ and $p \neq N$ then return "$p$ is a factor of $N$".
else return "No Result".
```

But it is unlikely that we would have $a = 1 \pmod{q}$. Hence,

- p will divide $a-1$,
- q will not divide $a-1$.

We can then recover p by computing $p = \gcd(a-1, N)$, as in Algorithm 2.5

As an example, suppose we wish to factor $N = 15\,770\,708\,441$. We take $B = 180$ and running the above algorithm we obtain

$$a = 2^{B!} \pmod{N} = 1\,162\,022\,425.$$

Then we obtain

$$p = \gcd(a-1, N) = 135\,979.$$

To see why this works in this example we see that the prime factorization of N is given by

$$N = 135\,979 \cdot 115\,979$$

and we have

$$p - 1 = 135\,978 - 1 = 2 \cdot 3 \cdot 131 \cdot 173,$$
$$q - 1 = 115\,978 - 1 = 2 \cdot 103 \cdot 563.$$

Hence $p-1$ is indeed B-power smooth, whilst $q-1$ is not B-power smooth.

One can show that the complexity of the $P-1$ method is given by

$$O(B \cdot \log B \cdot (\log N)^2 + (\log N)^3).$$

So if we choose $B = O((\log N)^i)$, for some integer i, then this is a polynomial-time factoring algorithm, but it only works for numbers of a special form.

Due to the $P-1$ method we often see it recommended that RSA primes are chosen to satisfy

$$p - 1 = 2 \cdot p_1 \text{ and } q - 1 = 2 \cdot q_1,$$

where p_1 and q_1 are both primes. In this situation the primes p and q are called safe primes. For a random 1024-bit prime p the probability that $p-1$ is B-power smooth, for a small value of B, is very small. Hence, choosing random 1024-bit primes would in all likelihood render the $P-1$ method useless, and so choosing p to be a safe prime is not really needed.

2.3.4. Difference of Two Squares: A basic trick in factoring algorithms, known for many centuries, is to produce two numbers x and y, of around the same size as N, such that

$$x^2 = y^2 \pmod{N}.$$

Since then we have

$$x^2 - y^2 = (x-y) \cdot (x+y) = 0 \pmod{N}.$$

If $N = p \cdot q$ then we have four possible cases

(1) p divides $x-y$ and q divides $x+y$.
(2) p divides $x+y$ and q divides $x-y$.
(3) p and q both divide $x-y$ but neither divides $x+y$.
(4) p and q both divide $x+y$ but neither divides $x-y$.

All these cases can occur with equal probability, namely $\frac{1}{4}$. If we then compute

$$d = \gcd(x - y, N),$$

our previous four cases then divide into the cases

(1) $d = p$.
(2) $d = q$.
(3) $d = N$.
(4) $d = 1$.

Since all these cases occur with equal probability, we see that with probability $\frac{1}{2}$ we will obtain a non-trivial factor of N. The only problem is, how do we find x and y such that $x^2 = y^2 \pmod{N}$?

2.4. Modern Factoring Algorithms

Most modern factoring methods use the following strategy based on the difference-of-two-squares method described at the end of the last section.

- Take a smoothness bound B.
- Compute a *factorbase* F of all prime numbers p less than B.
- Find a large number of values of x and y such that x and y are B-smooth and
$$x = y \pmod{N}.$$
These are called *relations* on the factorbase.
- Using linear algebra modulo 2, find a combination of the relations to give an X and Y with
$$X^2 = Y^2 \pmod{N}.$$
- Attempt to factor N by computing $\gcd(X - Y, N)$.

The trick in all algorithms of this form is how to find the relations. All the other details of the algorithms are basically the same. Such a strategy can be used to solve discrete logarithm problems as well, which we shall discuss in Chapter 3. In this section, we explain the parts of the modern factoring algorithms which are common and justify why they work.

One way of looking at such algorithms is in the context of computational group theory. The factorbase is essentially a set of generators of the group $(\mathbb{Z}/N\mathbb{Z})^*$, whilst the relations are relations between the generators of this group. Once a sufficiently large number of relations have been found, since the group is a finite abelian group, standard group-theoretic algorithms will compute the group structure and hence the group order. From the group order $\phi(N) = (p - 1)(q - 1)$, we are able to factor the integer N. These general group-theoretic algorithms could include computing the Smith Normal Form of the associated matrix. Hence, it should not be surprising that linear algebra is used on the relations to factor the integer N.

Combining Relations: The Smith Normal Form algorithm is far too complicated for factoring algorithms where a more elementary approach can be used, still based on linear algebra, as we shall now explain. Suppose we have the relations

$$\begin{aligned} p^2 \cdot q^5 \cdot r^2 &= p^3 \cdot q^4 \cdot r^3 \pmod{N}, \\ p \cdot q^3 \cdot r^5 &= p \cdot q \cdot r^2 \pmod{N}, \\ p^3 \cdot q^5 \cdot r^3 &= p \cdot q^3 \cdot r^2 \pmod{N}, \end{aligned}$$

where p, q and r are primes in our factorbase, $F = \{p, q, r\}$. Dividing one side by the other in each of our relations we obtain

$$\begin{aligned} p^{-1} \cdot q \cdot r^{-1} &= 1 \pmod{N}, \\ q^2 \cdot r^3 &= 1 \pmod{N}, \\ p^2 \cdot q^2 \cdot r &= 1 \pmod{N}. \end{aligned}$$

Multiplying the last two equations together we obtain

$$p^{0+2} \cdot q^{2+2} \cdot r^{3+1} = 1 \pmod{N}.$$

In other words

$$p^2 \cdot q^4 \cdot r^4 = 1 \pmod{N}.$$

Hence if $X = p \cdot q^2 \cdot r^2$ and $Y = 1$ then we obtain

$$X^2 = Y^2 \pmod{N}$$

as required and computing

$$\gcd(X - Y, N)$$

will give us a fifty percent chance of factoring N.

Whilst it was easy to see by inspection in the previous example how to combine the relations to obtain a square, in a real-life example our factorbase could consist of hundreds of thousands of primes and we would have hundreds of thousands of relations. We basically need a technique to automate this process of finding out how to combine relations into squares. This is where linear algebra can come to our aid.

We explain how to automate the process using linear algebra by referring to our previous simple example. Recall that our relations were equivalent to

$$\begin{aligned} p^{-1} \cdot q \cdot r^{-1} &= 1 \pmod{N}, \\ q^2 \cdot r^3 &= 1 \pmod{N}, \\ p^2 \cdot q^2 \cdot r &= 1 \pmod{N}. \end{aligned}$$

To find which equations to multiply together to obtain a square, we take a matrix A with $\#F$ columns and number of rows equal to the number of relations. Each relation is coded into the matrix as a row, modulo two, which in our example becomes

$$A = \begin{pmatrix} -1 & 1 & 1 \\ 0 & 2 & 3 \\ 2 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \pmod{2}.$$

We now try to find a (non-zero) binary vector z such that

$$z \cdot A = 0 \pmod{2}.$$

In our example we can take

$$z = (0, 1, 1)$$

since

$$\begin{pmatrix} 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \pmod{2}.$$

This solution vector $z = (0, 1, 1)$ tells us that multiplying the last two equations together will produce a square modulo N.

Finding the vector z is done using a variant of Gaussian Elimination. Hence in general this means that we require more equations (i.e. relations) than elements in the factorbase. This relation-combining stage of factoring algorithms is usually the hardest part since the matrices involved tend to be rather large. For example using the Number Field Sieve to factor a 100-decimal-digit number may require a matrix of dimension over 100 000. This results in huge memory problems and requires the writing of specialist matrix code and often the use of specialized super computers.

The matrix will have around 500 000 rows and as many columns, for cryptographically interesting numbers. As this is nothing but a matrix modulo 2 each entry could be represented by a single bit. If we used a dense matrix representation then the matrix alone would occupy around 29 gigabytes of storage. Luckily the matrix is very, very sparse and so the storage will not be so large.

As we said above, we can compute the vector z such that $z \cdot A = 0$ using a variant of Gaussian Elimination over $\mathbb{Z}/2\mathbb{Z}$. But standard Gaussian Elimination would start with a sparse matrix and end up with an upper triangular dense matrix, so we would be back with the huge memory problem again. To overcome this problem very advanced matrix algorithms are deployed that try not to alter the matrix at all. We do not discuss these here but refer the interested reader to the book of Lenstra and Lenstra mentioned in the Further Reading section of this chapter. The only thing we have not sketched is how to find the relations, a topic which we shall discuss in the next section.

2.5. Number Field Sieve

The Number Field Sieve is the fastest known factoring algorithm. The basic idea is to factor a number N by finding two integers x and y such that

$$x^2 = y^2 \pmod{N};$$

we then expect (hope) that $\gcd(x-y, N)$ will give us a non-trivial factor of N. To explain the basic method we shall start with the linear sieve and then show how this is generalized to the Number Field Sieve. The linear sieve is not a very good algorithm but it does show the rough method.

2.5.1. The Linear Sieve: We let F denote a set of "small" prime numbers which form the factorbase:

$$F = \{p : p \leq B\}.$$

A number which factorizes with all its factors in F is therefore B-smooth. The idea of the linear sieve is to find many pairs of integers a and λ such that

$$b = a + N \cdot \lambda$$

is B-smooth. If in addition we only select values of a which are "small", then we would expect that a will also be B-smooth and we could write

$$a = \prod_{p \in F} p^{a_p}$$

and

$$b = a + N \cdot \lambda = \prod_{p \in F} p^{b_p}.$$

We would then have a relation in $\mathbb{Z}/N\mathbb{Z}$

$$\prod_{p \in F} p^{a_p} = \prod_{p \in F} p^{b_p} \pmod{N}.$$

So the main question is how do we find such values of a and λ?

(1) Fix a value of λ to consider.
(2) Initialize an array of length $A+1$ indexed by 0 to A with zeros, for some value of A.
(3) For each prime $p \in F$ add $\log_2 p$ to every array location whose position is congruent to $-\lambda \cdot N \pmod{p}$.

(4) Choose the candidates for a to be the positions of those elements that exceed some threshold bound.

The reasoning behind this method is that a position of the array that has an entry exceeding some bound will have a good chance of being B-smooth, when added to λN, as it is likely to be divisible by many primes in F. This is yet another application of the Sieve of Eratosthenes.

Linear Sieve Example: For example suppose we take $N = 1159$, $F = \{2, 3, 5, 7, 11\}$ and $\lambda = -2$. So we wish to find a smooth value of

$$a - 2N.$$

We initialize the sieving array as follows:

0	1	2	3	4	5	6	7	8	9
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

We now take the first prime in F, namely $p = 2$, and we compute $-\lambda \cdot N \pmod{p} = 0$. So we add $\log_2(2) = 1$ to every array location with index equal to 0 modulo 2. This results in our sieve array becoming:

0	1	2	3	4	5	6	7	8	9
1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0

We now take the next prime in F, namely $p = 3$, and compute $-\lambda \cdot N \pmod{p} = 2$. So we add $\log_2(3) = 1.6$ to every array location with index equal to 2 modulo 3. Our sieve array then becomes:

0	1	2	3	4	5	6	7	8	9
1.0	0.0	2.6	0.0	1.0	1.6	1.0	0.0	2.6	0.0

Continuing in this way with $p = 5, 7$ and 11, eventually the sieve array becomes:

0	1	2	3	4	5	6	7	8	9
1.0	2.8	2.6	2.3	1.0	1.6	1.0	0.0	11.2	0.0

Hence, the value $a = 8$ looks like it should correspond to a smooth value, and indeed it does, since we find

$$a - \lambda \cdot N = 8 - 2 \cdot 1159 = -2310 = -2 \cdot 3 \cdot 5 \cdot 7 \cdot 11.$$

So using the linear sieve we obtain a large collection of numbers, a_i and b_i, such that

$$a_i = \prod_{p_j \in F} p_j^{a_{i,j}} = \prod_{p_j \in F} p_j^{b_{i,j}} = b_i \pmod{N}.$$

We assume that we have at least $|B| + 1$ such relations with which we then form a matrix with the ith row being

$$(a_{i,1}, \ldots, a_{i,t}, b_{i,1}, \ldots, b_{i,t}) \pmod{2}.$$

We then find elements of the kernel of this matrix modulo 2. This will tell us how to multiply the a_i and the b_i together to obtain elements x^2 and y^2 such that $x, y \in \mathbb{Z}$ are easily calculated and

$$x^2 = y^2 \pmod{N}.$$

We can then try to factor N, but if these values of x and y do not provide a factor we just find a new element in the kernel of the matrix and continue.

The basic linear sieve gives a very small yield of relations. There is a variant called the *large prime variation* which relaxes the sieving condition to allow through pairs a and b which are almost B-smooth, bar say a single "large" prime in a and a single "large" prime in b. These large primes then have to be combined in some way so that the linear algebra step can proceed as above. This is done by constructing a graph and using an algorithm which computes a basis for the set of cycles in the graph. The basic idea for the large prime variation originally arose in the context of the quadratic sieve algorithm, but it can be applied to any of the sieving algorithms used in factoring.

It is clear that the sieving could be carried out in parallel, hence the sieving can be parcelled out to lots of slave computers around the world. The slaves then communicate any relations they find to the central master computer which performs the linear algebra step. In such a way the Internet can be turned into a large parallel computer dedicated to factoring numbers. As we have already remarked, the final (linear algebra) step often needs to be performed on specialized equipment with large amounts of disk space and RAM, so this final computation cannot be distributed over the Internet.

2.5.2. Higher-Degree Sieving: The linear sieve is simply not good enough to factor large numbers. Indeed, the linear sieve was never proposed as a real factoring algorithm, but its operation is instructive for other algorithms of this type. The Number Field Sieve (NFS) uses the arithmetic of algebraic number fields to construct the desired relations between the elements of the factor-base. All that changes is the way the relations are found. The linear algebra step, the large prime variations and the slave/master approach all go over to NFS virtually unchanged. We now explain the NFS, but in a much simpler form than is actually used in real life so as to aid the exposition. Those readers who do not know any algebraic number theory may wish to skip this section.

First we construct two monic, irreducible polynomials with integer coefficients f_1 and f_2, of degree d_1 and d_2 respectively, such that there exists an $m \in \mathbb{Z}$ such that

$$f_1(m) = f_2(m) = 0 \pmod{N}.$$

The Number Field Sieve will make use of arithmetic in the number fields K_1 and K_2 given by

$$K_1 = \mathbb{Q}(\theta_1) \text{ and } K_2 = \mathbb{Q}(\theta_2),$$

where θ_1 and θ_2 are defined by $f_1(\theta_1) = f_2(\theta_2) = 0$. We then have two homomorphisms ϕ_1 and ϕ_2 given by

$$\phi_i : \begin{cases} \mathbb{Z}[\theta_i] \longrightarrow \mathbb{Z}/N\mathbb{Z} \\ \quad \theta_i \longmapsto m. \end{cases}$$

We aim to use a sieve, just as in the linear sieve, to find a set

$$S \subset \{(a,b) \in \mathbb{Z}^2 : \gcd(a,b) = 1\}$$

such that

$$\prod_S (a - b \cdot \theta_1) = \beta^2$$

and

$$\prod_S (a - b \cdot \theta_2) = \gamma^2,$$

where $\beta \in K_1$ and $\gamma \in K_2$. If we found two such values of β and γ then we would have

$$\phi_1(\beta)^2 = \phi_2(\gamma)^2 \pmod{N}$$

and we hope

$$\gcd(N, \phi_1(\beta) - \phi_2(\gamma))$$

would be a factor of N.

This leads to three obvious problems, which we address in the following three sub-sections:

- How do we find the set S?
- Given $\beta^2 \in \mathbb{Q}[\theta_1]$, how do we compute β?
- How do we find the polynomials f_1 and f_2 in the first place?

How do we find the set S?: Similar to the linear sieve we can find such a set S using linear algebra provided we can find lots of a and b such that

$$a - b \cdot \theta_1 \text{ and } a - b \cdot \theta_2$$

are both "smooth". But what does it mean for these two objects to be smooth? This is rather complicated, and for the rest of this section we will assume the reader has a basic acquaintance with algebraic number theory. It is here that the theory of algebraic number fields comes in: by generalizing our earlier definition of smooth integers to algebraic integers we obtain the following definition:

Definition 2.11. *An algebraic integer is "smooth" if and only if the ideal it generates is only divisible by "small" prime ideals.*

Define $F_i(X,Y) = Y^{d_i} \cdot f_i(X/Y)$, then

$$N_{\mathbb{Q}(\theta_i)/\mathbb{Q}}(a - b \cdot \theta_i) = F_i(a,b).$$

We define two factorbases, one for each of the polynomials

$$\mathcal{F}_i = \{(p,r) : p \text{ a prime}, r \in \mathbb{Z} \text{ such that } f_i(r) = 0 \pmod{p}\}.$$

Each element of $\mathcal{F}_i$ corresponds to a degree-one prime ideal of $\mathbb{Z}[\theta_i]$, which is a sub-order of the ring of integers of $\mathcal{O}_{\mathbb{Q}(\theta_i)}$, given by

$$\langle p, \theta_i - r\rangle := p\mathbb{Z}[\theta_i] + (\theta_i - r)\mathbb{Z}[\theta_i].$$

Given values of a and b we can easily determine whether the ideal $\langle a - \theta_i \cdot b\rangle$ "factorizes" over our factorbase. Note factorizes is in quotes as unique factorization of ideals may not hold in $\mathbb{Z}[\theta_i]$, whilst it will hold in $\mathcal{O}_{\mathbb{Q}(\theta_i)}$. It will turn out that this is not really a problem. To see why this is not a problem you should consult the book by Lenstra and Lenstra.

If $\mathbb{Z}[\theta_i] = \mathcal{O}_{\mathbb{Q}(\theta_i)}$ then the following method does indeed give the unique prime ideal factorization of $\langle a - \theta_i \cdot b\rangle$.

- Write
$$F_i(a,b) = \prod_{(p_j,r)\in\mathcal{F}_i} p_j^{s_j^{(i)}}.$$
- We have $(a : b) = (r : 1) \pmod{p}$, as an element in the projective space of dimension one over $\mathbb{F}_p$ (i.e. $a/b = r \pmod{p}$), if the ideal corresponding to (p,r) is included in a non-trivial way in the ideal factorization of $a - \theta_i b$.
- We have
$$\langle a - \theta_i \cdot b\rangle = \prod_{(p_j,r)\in\mathcal{F}_i} \langle p_j, \theta_i - r\rangle^{s_j^{(i)}}.$$

This leads to the following algorithm to sieve for values of a and b, such that $\langle a - \theta_i \cdot b\rangle$ is an ideal which factorizes over the factorbase. Just as with the linear sieve, the use of sieving allows us to avoid lots of expensive trial divisions when trying to determine smooth ideals. We end up only performing factorizations where we already know we have a good chance of being successful.

- Fix a.
- Initialize the sieve array for $-B \leq b \leq B$ by
$$S[b] = \log_2(F_1(a,b) \cdot F_2(a,b)).$$
- For every $(p,r) \in \mathcal{F}_i$ subtract $\log_2 p$ from every array element $S[b]$ where b is such that
$$a - r \cdot b = 0 \pmod{p}.$$
- The values of b we want are the ones such that $S[b]$ lies below some tolerance level.

If the tolerance level is set in a sensible way then we have a good chance that both $F_1(a,b)$ and $F_2(a,b)$ factor over the prime ideals in the factorbase, with the possibility of some large prime ideals creeping in. We keep these factorizations as a relation, just as we did with the linear sieve.

Then, after some linear algebra, we can find a subset S of all the pairs (a,b) we have found such that

$$\prod_{(a,b)\in S} \langle a - b\theta_i \rangle = \text{square of an } \textbf{ideal} \text{ in } \mathbb{Z}[\theta_i].$$

However, this is not good enough. Recall that we want the product $\prod(a - b\cdot\theta_i)$ to be the square of an **element** of $\mathbb{Z}[\theta_i]$. To overcome this problem we need to add information from the "infinite" places. This is done by adding in some quadratic characters, an idea introduced by Adleman. Let q be a rational prime (in neither $\mathcal{F}_1$ nor $\mathcal{F}_2$) such that there is an s_q with $f_i(s_q) = 0 \pmod q$ and $f_i'(s_q) \neq 0 \pmod q$ for either $i = 1$ or $i = 2$. Then our extra condition is that we require

$$\prod_{(a,b)\in S} \left(\frac{a - b\cdot s_q}{q}\right) = 1,$$

where $\left(\frac{\cdot}{q}\right)$ denotes the Legendre symbol. As the Legendre symbol is multiplicative this gives us an extra condition to put into our matrix. We need to add this condition for a number of primes q, hence we choose a set of such primes q and put the associated characters into our matrix as an extra column of 0s or 1s corresponding to:

$$\text{if } \left(\frac{a - b\cdot s_q}{q}\right) = \begin{cases} 1 & \text{then enter} \quad 0, \\ -1 & \text{then enter} \quad 1. \end{cases}$$

After finding enough relations we hope to be able to find a subset S such that

$$\prod_S (a - b\cdot\theta_1) = \beta^2 \text{ and } \prod_S (a - b\cdot\theta_2) = \gamma^2.$$

How do we take the square roots?: We then need to be able to take the square root of β^2 to recover β, and similarly for γ^2. Each β^2 is given in the form

$$\beta^2 = \sum_{j=0}^{d_1-1} a_j \cdot \theta_1^j$$

where the a_j are huge integers. We want to be able to determine the solutions $b_j \in \mathbb{Z}$ to the equation

$$\left(\sum_{j=0}^{d_1-1} b_j \cdot \theta_1^j\right)^2 = \sum_{j=0}^{d_1-1} a_j \cdot \theta_1^j.$$

One way this is overcome, due to Couveignes, is by computing such a square root modulo a large number of very, very large primes p. We then perform Hensel lifting and Chinese remaindering to hopefully recover our square root. This is the easiest method to understand although more advanced methods are available.

Choosing the initial polynomials: This is the part of the method that is a black art at the moment. We require only the following conditions to be met

$$f_1(m) = f_2(m) = 0 \pmod N.$$

However there are good heuristic reasons why it also might be desirable to construct polynomials with additional properties such as

- The polynomials have small coefficients.

- f_1 and f_2 have "many" real roots. Note, a random polynomial probably would have no real roots on average.
- f_1 and f_2 have "many" roots modulo lots of small prime numbers.
- The Galois groups of f_1 and f_2 are "small".

It is often worth spending a few weeks trying to find a good couple of polynomials before we start to attempt the factorization algorithm proper. There are a number of search strategies used for finding these polynomials. Once a few candidates are found, some experimental sieving is performed to see which appear to be the most successful, in that they yield the most relations. Then, once a decision has been made we can launch the sieving stage "for real".

Example: I am grateful to Richard Pinch for allowing me to include the following example. It is taken from his lecture notes from a course at Cambridge in the mid-1990s. Suppose we wish to factor the number $N = 290^2 + 1 = 84\,101$. We take $f_1(x) = x^2 + 1$ and $f_2(x) = x - 290$ with $m = 290$. Then

$$f_1(m) = f_2(m) = 0 \pmod{N}.$$

On one side we have the order $\mathbb{Z}[i]$ which is the ring of integers of $\mathbb{Q}(i)$ and on the other side we have the order $\mathbb{Z}$. We obtain the following factorizations:

x	y	$N(x - i \cdot y)$	Factors	$x - m \cdot y$	Factors
-38	-1	1445	$5 \cdot 17^2$	252	$2^2 \cdot 3^2 \cdot 7$
-22	-19	845	$5 \cdot 13^2$	5488	$2^4 \cdot 7^3$

We then obtain the two factorizations, which are real factorizations of elements, as $\mathbb{Z}[i]$ is a unique factorization domain,

$$-38 + i = -(2 + i) \cdot (4 - i)^2 \text{ and } -22 + 19 \cdot i = -(2 + i) \cdot (3 - 2 \cdot i)^2.$$

Hence, after a trivial bit of linear algebra, we obtain the following "squares"

$$(-38 + i) \cdot (-22 + 19 \cdot i) = (2 + i)^2 \cdot (3 - 2 \cdot i)^2 \cdot (4 - i)^2 = (31 - 12 \cdot i)^2$$

and

$$(-38 + m) \cdot (-22 + 19 \cdot m) = 2^6 \cdot 3^2 \cdot 7^4 = 1176^2.$$

We then apply the map ϕ_1 to $31 - 12 \cdot i$ to obtain

$$\phi_1(31 - 12 \cdot i) = 31 - 12 \cdot m = -3449.$$

But then we have

$$\begin{aligned}(-3449)^2 &= \phi_1(31 - 12 \cdot i)^2 \\ &= \phi_1((31 - 12 \cdot i)^2) \\ &= \phi_1((-38 + i) \cdot (-22 + 19 \cdot i)) \\ &= \phi_1(-38 + i) \cdot \phi_1(-22 + 19 \cdot i) \\ &= (-38 + m) \cdot (-22 + 19 \cdot m) \pmod{N} \\ &= 1176^2.\end{aligned}$$

So we compute

$$\gcd(N, -3449 + 1176) = 2273$$

and

$$\gcd(N, -3449 - 1176) = 37.$$

Hence 37 and 2273 are factors of $N = 84\,101$.

Chapter Summary

- Prime numbers are very common and the probability that a random n-bit number is prime is around $1/n$.
- Numbers can be tested for primality using a probable prime test such as the Fermat or Miller–Rabin algorithms. The Fermat Test has a problem in that certain composite numbers will always pass the Fermat Test, no matter how we choose the possible witnesses.
- If we really need to be certain that a number is prime then there are primality-proving algorithms which run in polynomial time.
- We introduced the problems FACTOR, SQRROOT and RSA, and the relations between them.
- Factoring algorithms are often based on the problem of finding the difference of two squares.
- Modern factoring algorithms run in two stages: In the first stage we collect many relations on a factorbase by using a process called sieving, which can be done using thousands of computers on the Internet. In the second stage these relations are processed using linear algebra on a big central server. The final factorization is obtained by finding a difference of two squares.

Further Reading

The definitive reference work on computational number theory which deals with many algorithms for factoring and primality proving is the book by Cohen. The book by Bach and Shallit also provides a good reference for primality testing. The main book explaining the Number Field Sieve is the book by Lenstra and Lenstra.

E. Bach and J. Shallit. *Algorithmic Number Theory. Volume 1: Efficient Algorithms.* MIT Press, 1996.

H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer, 1993.

A. Lenstra and H. Lenstra. *The Development of the Number Field Sieve.* Springer, 1993.