CHAPTER 6

Implementation Issues

Chapter Goals

- To show how exponentiation algorithms are implemented.
- To explain how modular arithmetic can be implemented efficiently on large numbers.
- To show how certain tricks can be used to speed up exponentiation operations.
- To show how finite fields of characteristic two can be implemented efficiently.

6.1. Introduction

In this chapter we examine how one actually implements cryptographic operations. We shall mainly be concerned with public key operations since those are the most complex to implement. For example, when we introduce RSA or DSA later we will have to perform a modular exponentiation with respect to a modulus of a thousand or more bits. This means we need to understand the implementation issues involved with both modular arithmetic and exponentiation algorithms.

There is another reason to focus on public key algorithms rather than private key ones: in general public key schemes run much more slowly than symmetric schemes. In fact they can be so slow that their use can make networks and web servers unusable. Hence, efficient implementation is crucial unless one is willing to pay a large performance penalty. The chapter focuses on algorithms used in software; for hardware-based algorithms one often uses different techniques entirely.

6.2. Exponentiation Algorithms

So far in this book, e.g. when discussing primality testing in Chapter 2, we have assumed that computing

$$y = x^d \pmod{n}$$

is an easy operation. We will also need this operation both in RSA and in systems based on discrete logarithms such as ElGamal encryption and DSA. In this section we concentrate on the exponentiation algorithms and assume that we can perform modular arithmetic efficiently. In a later section we shall discuss how to perform modular arithmetic. Firstly note it does not make sense to perform this operation via the sequence

- Compute $r \leftarrow x^d$,
- Compute $y \leftarrow r \pmod{n}$.

To see this, consider

$$123^5 \pmod{511} = 28\,153\,056\,843 \pmod{511} = 359.$$

With this naive method one obtains a huge intermediate result, in our small case above this is

$$28\,153\,056\,843.$$

But in a real 2048-bit exponentiation this intermediate result would be in general $2^{2048} \cdot 2048$ bits long. Such a number requires over 10^{600} gigabytes simply to write down.

N.P. Smart, *Cryptography Made Simple*, Information Security and Cryptography, DOI 10.1007/978-3-319-21936-3_6

6.2.1. Binary Exponentiation: To stop this explosion in the size of any intermediate results we use the fact that we are working modulo n. But even here one needs to be careful; a naive algorithm would compute the above example by computing

$$\begin{aligned} x &= 123, \\ x^2 &= x \times x \pmod{511} = 310, \\ x^3 &= x \times x^2 \pmod{511} = 316, \\ x^4 &= x \times x^3 \pmod{511} = 32, \\ x^5 &= x \times x^4 \pmod{511} = 359. \end{aligned}$$

This requires four modular multiplications, which seems fine for our small example. But for a general exponentiation by a 2048-bit exponent using this method would require around 2^{2048} modular multiplications. If each such multiplication could be done in under one millionth of a second we would still require around 10^{600} years to perform this operation.

However, it is easy to see that, even in our small example, we can reduce the number of required multiplications by being a little more clever:

$$\begin{aligned} x &= 123, \\ x^2 &= x \times x \pmod{511} = 310, \\ x^4 &= x^2 \times x^2 \pmod{511} = 32, \\ x^5 &= x \times x^4 \pmod{511} = 359. \end{aligned}$$

Which only requires three modular multiplications rather than the previous four. To understand why we only require three modular multiplications notice that the exponent 5 has binary representation $0b101$ and so

- Has bit length $t = 3$,
- Has Hamming weight $h = 2$.

In the above example we required $1 = (h - 1)$ general multiplications and $2 = (t - 1)$ squarings. This fact holds in general, in that a modular exponentiation can be performed using

- $(h - 1)$ multiplications,
- $(t - 1)$ squarings,

where t is the bit length of the exponent and h is the Hamming weight. The average Hamming weight of an integer is $t/2$ so the number of multiplications and squarings is on average

$$t + t/2 - 1.$$

For a 2048-bit modulus this means that the average number of modular multiplications needed to perform exponentiation by a 2048-bit exponent is at most 4096 and on average 3072.

Algorithm 6.1: Binary exponentiation: Right-to-left variant

```
y ← 1.
while d ≠ 0 do
    if (d mod 2) ≠ 0 then
        y ← (y · x) mod n.
        d ← d − 1.
    d ← d/2.
    x ← (x · x) mod n.
```

The method used to achieve this improvement in performance is called the binary exponentiation method. This is because it works by reading each bit of the binary representation of the exponent in turn, starting with the least significant bit and working up to the most significant bit. Algorithm 6.1 explains the method by computing

$$y = x^d \pmod{n}.$$

The above binary exponentiation algorithm has a number of different names: some authors call it the *square and multiply* algorithm, since it proceeds by a sequence of squarings and multiplications, other authors call it the *Indian exponentiation* algorithm. Algorithm 6.1 is called a *right-to-left* exponentiation algorithm since it processes the bits of d from the least significant bit (the right one) up to the most significant bit (the left one).

6.2.2. Window Exponentiation Methods: Most of the time it is faster to perform a squaring operation than a general multiplication. Hence to reduce time even more one tries to reduce the total number of modular multiplications even further. This is done using *window* techniques which trade off precomputations (i.e. storage) against the time in the main loop.

To understand window methods better we first examine the binary exponentiation method again. But this time instead of a right-to-left variant, we process the exponent from the most significant bit first, thus producing a *left-to-right* binary exponentiation algorithm; see Algorithm 6.2. Again we assume we wish to compute

$$y = x^d \pmod{n}.$$

We first give a notation for the binary representation of the exponent

$$d = \sum_{i=0}^{t} d_i 2^i,$$

where $d_i \in \{0, 1\}$. The algorithm processes a single bit of the exponent on every iteration of the loop. Again the number of squarings is equal to t and the expected number of multiplications is equal to $t/2$.

Algorithm 6.2: Binary exponentiation: Left-to-right variant

$y \leftarrow 1$.
for $i = t$ **downto** 0 **do**
  $y \leftarrow (y \cdot y) \mod n$.
  if $d_i = 1$ **then** $y \leftarrow (y \cdot x) \mod n$.

In a window method we process w bits of the exponent at a time, as in Algorithm 6.3. We first precompute a table

$$x_i = x^i \pmod{n} \text{ for } i = 0, \ldots, 2^w - 1.$$

Then we write our exponent out, but this time taking w bits at a time,

$$d = \sum_{i=0}^{t/w} d_i \cdot 2^{i \cdot w},$$

where $d_i \in \{0, 1, 2, \ldots, 2^w - 1\}$.

It is perhaps easier to illustrate this with an example. Suppose we wish to compute

$$y = x^{215} \pmod{n}$$

with a window width of $w = 3$. We compute the d_i as

$$215 = 3 \cdot 2^6 + 2 \cdot 2^3 + 7.$$

Algorithm 6.3: Window exponentiation method

$y \leftarrow 1$.
for $i = t/w$ **downto** 0 **do**
  for $j = 0$ **to** $w - 1$ **do** $y \leftarrow (y \cdot y) \mod n$.
  $j \leftarrow d_i$.
  $y \leftarrow (y \cdot x_j) \mod n$.

Hence, our iteration to compute $x^{215} \pmod n$ computes in order

$$\begin{aligned} y &= 1, \\ y &= y \cdot x^3 = x^3, \\ y &= y^8 = x^{24}, \\ y &= y \cdot x^2 = x^{26}, \\ y &= y^8 = y^{208}, \\ y &= y \cdot x^7 = x^{215}. \end{aligned}$$

6.2.3. Sliding Window Method: With a window method as above, we still perform t squarings but the number of multiplications reduces to t/w on average. One can do even better by adopting a sliding window method, where we now encode our exponent as

$$d = \sum_{i=0}^{l} d_i \cdot 2^{e_i}$$

where $d_i \in \{1, 3, 5, \ldots, 2^w - 1\}$ and $e_{i+1} - e_i \geq w$. By choosing only odd values for d_i and having a variable window width we achieve both decreased storage for the precomputed values and improved efficiency. After precomputing $x_i = x^i$ for $i = 1, 3, 5, \ldots, 2^w - 1$, we execute Algorithm 6.4.

Algorithm 6.4: Sliding window exponentiation

$y \leftarrow 1$.
for $i = l$ **downto** 0 **do**
  for $j = 0$ **to** $e_{i+1} - e_i - 1$ **do** $y \leftarrow (y \cdot y) \mod n$.
  $j \leftarrow d_i$.
  $y \leftarrow (y \cdot x_j) \mod n$.
for $j = 0$ **to** $e_0 - 1$ **do** $y \leftarrow (y \cdot y) \mod n$.

The number of squarings remains again at t, but now the number of multiplications reduces to l, which is about $t/(w + 1)$ on average. In our example of computing $y = x^{215} \pmod n$ we have

$$215 = 2^7 + 5 \cdot 2^4 + 7,$$

and so we execute the steps

$$\begin{aligned} y &= 1, \\ y &= y \cdot x = x^1, \\ y &= y^8 = x^8, \\ y &= y \cdot x^5 = x^{13}, \\ y &= y^{16} = x^{208}, \\ y &= y \cdot x^7 = x^{215}. \end{aligned}$$

6.2.4. Generalizations to Any Group: Notice that all of the above window algorithms apply to exponentiation in any abelian group and not just the integers modulo n. Hence, we can use these algorithms to compute α^d in a finite field or to compute $[d]P$ on an elliptic curve; in the latter case we call this point multiplication rather than exponentiation.

An advantage with elliptic curve variants is that negation comes for free, in that given P it is easy to compute $-P$. This leads to the use of signed binary and signed window methods. We only present the signed window method. We precompute

$$P_i = [i]P \text{ for } i = 1, 3, 5, \ldots, 2^{w-1} - 1,$$

which requires only half the storage of the equivalent sliding window method or one quarter of the storage of the equivalent standard window method. We now write our multiplicand d as

$$d = \sum_{i=0}^{l} d_i \cdot 2^{e_i}$$

where $d_i \in \{\pm 1, \pm 3, \pm 5, \ldots, \pm(2^{w-1} - 1)\}$. The signed sliding window method for elliptic curves is then given by Algorithm 6.5.

Algorithm 6.5: Signed sliding window method

$Q \leftarrow 0$.
for $i = l$ **downto** 0 **do**
  for $j = 0$ **to** $e_{i+1} - e_i - 1$ **do** $Q \leftarrow [2]Q$.
  $j \leftarrow d_i$.
  if $j > 0$ **then** $Q \leftarrow Q + P_j$.
  else $Q \leftarrow Q - P_{-j}$.
for $j = 0$ **to** $e_0 - 1$ **do** $Q \leftarrow [2]Q$.

6.3. Special Exponentiation Methods

To speed up public key algorithms even more in practice, various tricks are used, the precise one depending on whether we are performing an operation with a public exponent or a private exponent.

6.3.1. Small Exponents: When we compute $y = x^e \pmod{n}$ where the evaluator does not know the factors of n, but knows the exponent e, we often select e to be very small (this does not seem to create any major attacks), for example $e = 3, 17$ or $65\,537$. The reason for these particular values is that they have small Hamming weight, in fact the smallest possible for a non-trivial exponent, namely two. This means that the binary method, or any other exponentiation algorithm, will

require only one general multiplication, but it will still need k squarings where k is the bit size of the exponent e. For example

$$\begin{aligned} x^3 &= x^2 \times x, \\ x^{17} &= x^{16} \times x, \\ &= (((x^2)^2)^2)^2 \times x. \end{aligned}$$

6.3.2. Knowing p and q: In the case of RSA decryption, or signing, the exponent will be a general and secret 2000-bit number. Hence, we need some way of speeding up the computation. Luckily, since we are considering a private key operation we have access to the prime factors of n,

$$n = p \cdot q.$$

Suppose we wish to compute

$$y = x^d \pmod{n}.$$

We speed up the calculation by first computing y modulo p and q:

$$\begin{aligned} y_p &= x^d \pmod{p} = x^{d \pmod{p-1}} \pmod{p}, \\ y_q &= x^d \pmod{q} = x^{d \pmod{q-1}} \pmod{q}. \end{aligned}$$

Since p and q are 1024-bit numbers, the above calculation requires two exponentiations modulo 1024-bit moduli and 1024-bit exponents. This is faster than a single exponentiation modulo a 2048-bit number with a 2048-bit exponent.

But we now need to recover y from y_p and y_q, which is done using the Chinese Remainder Theorem as follows: We compute $t = p^{-1} \pmod{q}$ and store it with the values p and q. The value y can then be recovered from y_p and y_q via

- $u = (y_q - y_p) \cdot t \pmod{q}$,
- $y = y_p + u \cdot p$.

This is why later on in Chapter 15 we say that when you generate a private key it is best to store p and q even though they are not mathematically needed.

6.3.3. Multi-exponentiation: Sometimes we need to compute

$$r = g^a \cdot y^b \pmod{n}.$$

This can be accomplished by first computing g^a and then y^b and then multiplying the results together. However, often it is easier to perform the two exponentiations simultaneously. There are a number of techniques to accomplish this, using various forms of window techniques etc. But all are essentially based on the following idea, called Shamir's trick.

We first compute the look-up table

$$G_i = g^{i_0} \cdot y^{i_1}$$

where $i = (i_1, i_0)$ is the binary representation of i, for $i = 0, 1, 2, 3$. We then compute an exponent array from the two exponents a and b. This is a two-by-t array, where t is the maximum bit length of a and b. The rows of this array are the binary representation of the exponents a and b. We then let I_j, for $j = 1, \ldots, t$, denote the integers whose binary representation is given by the columns of this array. The exponentiation is then computed by setting $r = 1$ and computing

$$r = r^2 \cdot G_{I_j}$$

for $j = 1$ to t. As an example suppose we wish to compute

$$r = g^{11} \cdot y^7,$$

hence we have $t = 4$. We precompute

$$G_0 = 1,\ G_1 = g,\ G_2 = y,\ G_3 = g \cdot y.$$

Since the binary representation of 11 and 7 is given by 1011 and 111, our exponent array is given by

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}.$$

The integers I_j then become

$$I_1 = 1,\ I_2 = 2,\ I_3 = 3,\ I_4 = 3.$$

Hence, the four steps of our algorithm become

$$\begin{aligned} r &= G_1 = g, \\ r &= r^2 \cdot G_2 = g^2 \cdot y, \\ r &= r^2 \cdot G_3 = (g^4 \cdot y^2) \cdot (g \cdot y) = g^5 \cdot y^3, \\ r &= r^2 \cdot G_3 = (g^{10} \cdot y^6) \cdot (g \cdot y) = g^{11} \cdot y^7. \end{aligned}$$

Note that elliptic curve analogues of Shamir's trick and its variants exist, which make use of signed representations for the exponent. We do not give these here, but leave them for the interested reader to investigate.

6.4. Multi-precision Arithmetic

We shall now explain how to perform modular arithmetic on 2048-bit numbers. We show how this is accomplished using modern processors, and then go on to show why naive algorithms are usually replaced with a special technique due to Montgomery.

In a cryptographic application it is common to focus on a fixed length for the integers in use, for example 2048 bits in an RSA/DSA implementation or 256 bits for an ECC implementation. This leads to different programming choices than when we implement a general-purpose multi-precision arithmetic library. For example, we no longer need to worry so much about dynamic memory allocation, and we can now concentrate on particular performance enhancements for the integer sizes we are dealing with.

It is common to represent all integers in little-wordian format. This means that if a large integer is held in memory locations $x_0, x_1, \ldots, x_n$, then x_0 is the least significant word and x_n is the most significant word. For a 64-bit machine and 128-bit numbers we would represent x and y as $[x_0, x_1]$ and $[y_0, y_1]$ where

$$\begin{aligned} x &= x_1 \cdot 2^{64} + x_0, \\ y &= y_1 \cdot 2^{64} + y_0. \end{aligned}$$

6.4.1. Addition: Most modern processors have a carry flag which is set by any overflow from an addition operation. Also most have a special instruction, usually called something like **addc**, which adds two integers together and adds on the contents of the carry flag. So if we wish to add our two 128-bit integers given earlier then we need to compute

$$z = x + y = z_2 \cdot 2^{128} + z_1 \cdot 2^{64} + z_0.$$

The values of z_0, z_1 and z_2 are then computed via

```
z0 <- add   x0,y0
z1 <- addc  x1,y1
z2 <- addc  0,0
```

Note that the value held in z_2 is at most one, so the value of z could be a 129-bit integer. The above technique for adding two 128-bit integers can clearly be scaled to adding integers of any fixed length, and can also be made to work for subtraction of large integers.

6.4.2. Schoolbook Multiplication: We now turn to the next simplest arithmetic operation, after addition and subtraction, namely multiplication. Notice that two 64-bit words multiply together to form a 128-bit result, and so most modern processors have an instruction which will perform this operation.

$$w_1 \cdot w_2 = (High, Low) = (H(w_1 \cdot w_2), L(w_1 \cdot w_2)).$$

When we use schoolbook long multiplication, for our two 128-bit numbers, we obtain something like

$$\begin{array}{rrrr}
 & & x_1 & x_0 \\
 & \times & y_1 & y_0 \\
\hline
 & & H(x_0 \cdot y_0) & L(x_0 \cdot y_0) \\
 & H(x_0 \cdot y_1) & L(x_0 \cdot y_1) & \\
 & H(x_1 \cdot y_0) & L(x_1 \cdot y_0) & \\
H(x_1 \cdot y_1) & L(x_1 \cdot y_1) & &
\end{array}$$

Then we add up the four rows to get the answer, remembering we need to take care of the carries. This then becomes, for

$$z = x \cdot y,$$

something like the following pseudo-code

```
(z1,z0) <- mul x0,y0
(z3,z2) <- mul x1,y1
(h,l)   <- mul x1,y0
z1      <- add z1,l
z2      <- addc z2,h
z3      <- addc z3,0
(h,l)   <- mul x0,y1
z1      <- add z1,l
z2      <- addc z2,h
z3      <- addc z3,0
```

If n denotes the bit size of the integers we are operating on, the above technique for multiplying large integers together clearly requires $O(n^2)$ bit operations, whilst it requires $O(n)$ bit operations to add or subtract integers. It is a natural question as to whether one can multiply integers faster than $O(n^2)$.

6.4.3. Karatsuba Multiplication: One technique to speed up multiplication is called Karatsuba multiplication. Suppose we have two n-bit integers x and y that we wish to multiply. We write these integers as

$$x = x_0 + 2^{n/2} \cdot x_1,$$
$$y = y_0 + 2^{n/2} \cdot y_2,$$

where $0 \leq x_0, x_1, y_0, y_1 < 2^{n/2}$. We then multiply x and y by computing

$$A \leftarrow x_0 \cdot y_0,$$
$$B \leftarrow (x_0 + x_1) \cdot (y_0 + y_1),$$
$$C \leftarrow x_1 \cdot y_1.$$

The product $x \cdot y$ is then given by

$$\begin{aligned} C \cdot 2^n + (B - A - C) \cdot 2^{n/2} + A &= x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 \\ &= (x_0 + 2^{n/2} \cdot x_1) \cdot (y_0 + 2^{n/2} \cdot y_1) \\ &= x \cdot y. \end{aligned}$$

Hence, to multiply two n-bit numbers we require three $n/2$-bit multiplications, two $n/2$-bit additions and three n-bit additions/subtractions. If we denote the cost of an n-bit multiplication by $M(n)$ and the cost of an n-bit addition/subtraction by $A(n)$, we have

$$M(n) = 3 \cdot M(n/2) + 2 \cdot A(n/2) + 3 \cdot A(n).$$

Now if we make the approximation that $A(n) \approx n$ then

$$M(n) \approx 3 \cdot M(n/2) + 4 \cdot n.$$

If the multiplication of the $n/2$-bit numbers is accomplished in a similar fashion then to obtain the final complexity of multiplication we solve the above recurrence relation to obtain

$$\begin{aligned} M(n) &\approx 9 \cdot n^{\frac{\log(3)}{\log(2)}} \text{ as } n \longrightarrow \infty \\ &= 9 \cdot n^{1.58}. \end{aligned}$$

So we obtain an algorithm with asymptotic complexity $O(n^{1.58})$. Karatsuba multiplication becomes faster than the $O(n^2)$ method for integers of sizes greater than a few hundred bits. However, one can do even better for very large integers since the fastest known multiplication algorithm takes time

$$O(n \cdot \log n \cdot \log \log n).$$

But neither this latter technique nor Karatsuba multiplication are used in many cryptographic applications. The reason for this will become apparent as we discuss integer division.

6.4.4. Division: After having looked at multiplication we are left with the division operation, which is the hardest of all the basic algorithms. After all division is required in order to be able to compute the remainder on division. Given two large integers x and y we wish to be able to compute q and r such that

$$x = q \cdot y + r$$

where $0 \leq r < y$; such an operation is called a Euclidean division. If we write our two integers x and y in the little-wordian format

$$x = (x_0, \ldots, x_n) \text{ and } y = (y_0, \ldots, y_t)$$

where the base for the representation is $b = 2^w$ then the Euclidean division can be performed by Algorithm 6.6. We let $u \ll_w v$ denote a large integer u shifted to the left by v words, in other words the result of multiplying u by b^v. As one can see this is a complex operation, hence one should try to avoid divisions as much as possible.

6.4.5. Montgomery Arithmetic: That division is a complex operation means our cryptographic operations run very slowly if we use standard division operations as above. Virtually all of the public key systems we will consider will make use of arithmetic modulo another number. What we require is the ability to compute remainders (i.e. to perform modular arithmetic) without having to perform any costly division operations. This at first sight may seem a state of affairs which is impossible to reach, but it can be achieved using a special form of arithmetic called Montgomery arithmetic.

Montgomery arithmetic works by using an alternative representation of integers, called the Montgomery representation. Let us fix some notation; we let b denote 2 to the power of the word

Algorithm 6.6: Euclidean division algorithm

```
r ← x.
/* Cope with the trivial case */
if t > n then
    q ← 0.
    return.
q ← 0, s ← 0.
/* Normalize the divisor */
while y_t < b/2 do
    y ← 2 · y, r ← 2 · r, s ← s + 1.
if r_{n+1} ≠ 0 then n ← n + 1.
/* Get the most significant word of the quotient */
while r ≥ (y ≪_w (n − t)) do
    q_{n−t} ← q_{n−t} + 1.
    r ← r − (y ≪_w n − t).
/* Deal with the rest */
for i = n to t + 1 do
    if r_i = y_t then q_{i−t−1} ← b − 1.
    else q_{i−t−1} ← ⌊(r_i · b + r_{i−1})/y_t⌋.
    if t ≠ 0 then h_m ← y_t · b + y_{t−1}.
    else h_m ← y_t · b.
    h ← q_{i−t−1} · h_m.
    if i ≠ 1 then l ← r_i · b^2 + r_{i−1} · b + r_{i−2}.
    else l ← r_i · b^2 + r_{i−1} · b.
    while h > l do
        q_{i−t−1} ← q_{i−t−1} − 1.
        h ← h − h_m.
    r ← r − (q_{i−t−1} · y) ≪_w (i − t − 1).
    if r < 0 then
        r ← r + (y ≪_w i − t − 1).
        q_{i−t−1} ← q_{i−t−1} − 1.
/* Renormalize */
for i = 0 to s − 1 do r ← r/2.
```

size of our computer, for example $b = 2^{64}$. To perform arithmetic modulo N we choose an integer R which satisfies

$$R = b^t > N,$$

for some integer value of t. Now instead of holding the value of the integer x in memory, we instead hold the value

$$x_R \leftarrow x \cdot R \pmod{N}.$$

Again this is usually held in a little-wordian format. The value x_R is called the Montgomery representation of the integer $x \pmod{N}$. Adding two elements in Montgomery representation is

easy; see Algorithm 6.7. If

$$z = x + y \pmod N$$

then given $x \cdot R \pmod N$ and $y \cdot R \pmod N$ we need to compute $z \cdot R \pmod N$.

Algorithm 6.7: Addition in Montgomery representation

$z_R \leftarrow x_R + y_R$.
if $z_R \geq N$ **then** $z_R \leftarrow z_R - N$.

Example: Let us take a simple example with

$$N = 18\,443\,759\,776\,216\,676\,723,$$
$$b = R = 2^{64} = 18\,446\,744\,073\,709\,551\,616.$$

The following is the map from the normal to Montgomery representation of the integers $1, 2$ and 3.

$$1 \longrightarrow 1 \cdot R \pmod N = 2\,984\,297\,492\,874\,893,$$
$$2 \longrightarrow 2 \cdot R \pmod N = 5\,968\,594\,985\,749\,786,$$
$$3 \longrightarrow 3 \cdot R \pmod N = 8\,952\,892\,478\,624\,679.$$

We can now verify that addition works since we have in the standard representation

$$1 + 2 = 3$$

whilst this is mirrored in the Montgomery representation as

$$2\,984\,297\,492\,874\,893 \; + \; 5\,968\,594\,985\,749\,786 = 8\,952\,892\,478\,624\,679 \pmod N.$$

Montgomery Reduction: Now we look at multiplication in Montgomery arithmetic. If we simply multiply two elements in Montgomery representation we will obtain

$$(x \cdot R) \cdot (y \cdot R) = x \cdot y \cdot R^2 \pmod N$$

but we want $x \cdot y \cdot R \pmod N$. Hence, we need to divide the result of the standard multiplication by R. Since R is a power of 2 we hope this should be easy. The process of computing

$$z = y/R \pmod N$$

given y and the earlier choice of R, is called Montgomery reduction. We first precompute the integer $q = 1/N \pmod R$, which is simple to perform with no divisions using the binary Euclidean algorithm. Then, performing a Montgomery reduction is done using Algorithm 6.8.

Algorithm 6.8: Montgomery reduction

$u \leftarrow (-y \cdot q) \mod R$.
$z \leftarrow (y + u \cdot N)/R$.
if $z \geq N$ **then** $z \leftarrow z - N$.

Note that the reduction modulo R in the first line is easy: we compute $y \cdot q$ using standard algorithms, the reduction modulo R being achieved by truncating the result. This latter trick works since R is a power of b. The division by R in the second line can also be simply achieved: since $y + u \cdot N = 0 \pmod R$, we simply shift the result to the right by t words, again since $R = b^t$.

Example: As an example we again take

$$N = 18\,443\,759\,776\,216\,676\,723,$$
$$R = b = 2^{64} = 18\,446\,744\,073\,709\,551\,616.$$

We wish to compute $2 \cdot 3$ in Montgomery representation. Recall

$$2 \longrightarrow 2 \cdot R \pmod{N} = 5\,968\,594\,985\,749\,786 = x,$$
$$3 \longrightarrow 3 \cdot R \pmod{N} = 8\,952\,892\,478\,624\,679 = y.$$

We then compute, using a standard multiplication algorithm, that

$$w = x \cdot y = 53\,436\,189\,155\,876\,232\,216\,612\,898\,568\,694 = 2 \cdot 3 \cdot R^2.$$

We now need to pass this value of w into our technique for Montgomery reduction, so as to find the Montgomery representation of $x \cdot y$. We find

$$w = 53\,436\,189\,155\,876\,232\,216\,612\,898\,568\,694,$$
$$q = (1/N) \pmod{R} = 14\,241\,249\,658\,089\,591\,739,$$
$$u = -w \cdot q \pmod{R} = 17\,905\,784\,957\,249\,358,$$
$$z = (w + u \cdot N)/R = 17\,905\,784\,957\,249\,358.$$

So the multiplication of x and y in Montgomery arithmetic should be

$$17\,905\,784\,957\,249\,358.$$

We can check that this is the correct value by computing

$$6 \cdot R \pmod{N} = 17\,905\,784\,957\,249\,358.$$

Hence, we see that Montgomery arithmetic allows us to add and multiply integers modulo an integer N without the need for costly division algorithms.

Optimized Montgomery Reduction: Our above method for Montgomery reduction requires two full multi-precision multiplications. So to multiply two numbers in Montgomery arithmetic we require three full multi-precision multiplications. If we are multiplying 2048-bit numbers, this means the intermediate results can grow to be 4096-bit numbers. We would like to do better, and we can.

Suppose y is given in little-wordian format

$$y = (y_0, y_1, \ldots, y_{2t-2}, y_{2t-1}).$$

Then a better way to perform Montgomery reduction is to first precompute $N' = -1/N \pmod{b}$, which is easy and only requires operations on word-sized quantities, and then to execute Algorithm 6.9.

Algorithm 6.9: Word-oriented Montgomery reduction

$z \leftarrow y$.
for $i = 0$ **to** $t - 1$ **do**
  $u \leftarrow (z_i \cdot N') \mod b$.
  $z \leftarrow z + u \cdot N \cdot b^i$.
$z \leftarrow z/R$.
if $z \geq N$ **then** $z \leftarrow z - N$.

Note that since we are reducing modulo b in the first line of the for loop we can execute this initial multiplication using a simple word multiplication algorithm. The second step of the for loop

requires a shift by one word (to multiply by b) and a single *word* $\times$ *bigint* multiply. Hence, we have reduced the need for large intermediate results in the Montgomery reduction step.

Montgomery Multiplication: We can also interleave the multiplication with the reduction to perform a single loop to produce

$$Z = X \cdot Y/R \pmod{N}.$$

So if $X = x \cdot R$ and $Y = y \cdot R$ this will produce

$$Z = (x \cdot y) \cdot R.$$

This procedure is called Montgomery multiplication and allows us to perform a multiplication in Montgomery arithmetic without the need for larger integers, as in Algorithm 6.10. Whilst Montgomery multiplication has complexity $O(n^2)$ as opposed to the $O(n^{1.58})$ of Karatsuba multiplication, it is still preferable to use Montgomery arithmetic since it deals more efficiently with modular arithmetic.

Algorithm 6.10: Montgomery multiplication

$Z \leftarrow 0$.
for $i = 0$ **to** $t - 1$ **do**
  $u \leftarrow ((z_0 + X_i \cdot Y_0) \cdot N') \mod b$.
  $Z \leftarrow (Z + X_i \cdot Y + u \cdot N)/b$.
if $Z \geq N$ **then** $Z \leftarrow Z - N$.

6.5. Finite Field Arithmetic

Apart from the integers modulo a large prime p the other type of finite field used in cryptography are those of characteristic two. These occur in the AES algorithm and in certain elliptic curve systems. In AES the field is so small that one can use look-up tables or special circuits to perform the basic arithmetic tasks, so in this section we shall concentrate on fields of large degree over $\mathbb{F}_2$, such as those used for elliptic curves. In addition we shall concern ourselves with software implementations only. Fields of characteristic two can have special types of hardware implementations based on optimal normal bases, but we shall not concern ourselves with these.

Recall that to define a finite field of characteristic two we first pick an irreducible polynomial $f(x)$ over $\mathbb{F}_2$ of degree n. The field is defined to be

$$\mathbb{F}_{2^n} = \mathbb{F}_2[x]/f(x),$$

i.e. we look at binary polynomials modulo $f(x)$. Elements of this field are usually represented as bit strings, which represent a binary polynomial. For example the bit string

$$101010111$$

represents the polynomial

$$x^8 + x^6 + x^4 + x^2 + x + 1.$$

Addition and subtraction of elements in $\mathbb{F}_{2^n}$ is accomplished by simply performing a bit-wise exclusive-or, written $\oplus$, between the two bitstrings. Hence, the difficult tasks are multiplication and division.

6.5.1. Characteristic-Two Field Division: It turns out that division, although slower than multiplication, is easier to describe, so we start with division. To compute α/β, where $\alpha, \beta \in \mathbb{F}_{2^n}$, we first compute β^{-1} and then perform the multiplication $\alpha \cdot \beta^{-1}$. So division is reduced to multiplication and the computation of β^{-1}. One way of computing β^{-1} is to use Lagrange's Theorem which tells us, for $\beta \neq 0$, that we have

$$\beta^{2^n-1} = 1.$$

But this means that

$$\beta \cdot \beta^{2^n-2} = 1,$$

or in other words

$$\beta^{-1} = \beta^{2^n-2} = \beta^{2\cdot(2^{n-1}-1)}.$$

Another way of computing β^{-1} is to use the binary Euclidean algorithm. We take the polynomial $a = f$ and the polynomial b which represents β and then perform Algorithm 6.11, which is a version of the binary Euclidean algorithm, where $\text{lsb}(b)$ refers to the least significant bit of b (in other words the coefficient of x^0).

Algorithm 6.11: Inversion of $b(x)$ modulo $f(x)$

$B \leftarrow 0$, $D \leftarrow 1$.
/* At least one of a and b will have a constant term on every execution of the loop */
while $a \neq 0$ **do**
  while $\text{lsb}(a) = 0$ **do**
    $a \leftarrow a \gg 1$.
    if $\text{lsb}(B) \neq 0$ **then** $B \leftarrow B \oplus f$.
    $B \leftarrow B \gg 1$.
  while $\text{lsb}(b) = 0$ **do**
    $b \leftarrow b \gg 1$.
    if $\text{lsb}(D) \neq 0$ **then** $D \leftarrow D \oplus f$.
    $D \leftarrow D \gg 1$.
  /* Now both a and b have a constant term */
  if $\deg(a) \geq \deg(b)$ **then**
    $a \leftarrow a \oplus b$.
    $B \leftarrow B \oplus D$.
  else
    $b \leftarrow a \oplus b$.
    $D \leftarrow D \oplus B$.
return D.

6.5.2. Characteristic-Two Field Multiplication: We now turn to the multiplication operation. Unlike the case of integers modulo N or p, where we use a special method of Montgomery arithmetic, in characteristic two we have the opportunity to choose a polynomial $f(x)$ which has "nice" properties. Any irreducible polynomial of degree n can be used to implement the finite field $\mathbb{F}_{2^n}$, we just need to select the best one.

Almost always one chooses a value of $f(x)$ which is either a trinomial

$$f(x) = x^n + x^k + 1$$

or a pentanomial

$$f(x) = x^n + x^{k_3} + x^{k_2} + x^{k_1} + 1.$$

It turns out that for all fields of degree less than 10 000 we can always find such a trinomial or pentanomial to make the multiplication operation very efficient. Table 6.1 at the end of this chapter gives a list for all values of n between 2 and 500 of an example pentanomial or trinomial which defines the field $\mathbb{F}_{2^n}$. In all cases where a trinomial exists we give one, otherwise we present a pentanomial.

Now to perform a multiplication of α by β we first multiply the polynomials representing α and β together to form a polynomial $\gamma(x)$ of degree at most $2 \cdot n - 2$. Then we reduce this polynomial by taking the remainder on division by the polynomial $f(x)$.

We show how this remainder on division is efficiently performed for trinomials, and leave the pentanomial case for the reader. We write

$$\gamma(x) = \gamma_1(x) \cdot x^n + \gamma_0(x).$$

Hence, $\deg(\gamma_1(x)), \deg(\gamma_0(x)) \leq n - 1$. We can then write, as $x^n = x^k + 1 \pmod{x^n + x^k + 1}$,

$$\gamma(x) \pmod{f(x)} = \gamma_0(x) + (x^k + 1) \cdot \gamma_1(x).$$

The right-hand side of this equation can be computed from the bit operations

$$\delta = \gamma_0 \oplus \gamma_1 \oplus (\gamma_1 \ll k).$$

Now δ, as a polynomial, will have degree at most $n - 1 + k$. So we need to carry out this procedure again by first writing

$$\delta(x) = \delta_1(x) \cdot x^n + \delta_0(x),$$

where $\deg(\delta_0(x)) \leq n - 1$ and $\deg(\delta_1(x)) \leq k - 1$. We then compute as before that γ is equivalent to

$$\delta_0 \oplus \delta_1 \oplus (\delta_1 \ll k).$$

This latter polynomial will have degree $\max(n - 1, 2k - 1)$, so if we select our trinomial so that

$$k \leq n/2,$$

then Algorithm 6.12 will perform our division-with-remainder step. Let g denote the polynomial of degree $2 \cdot n - 2$ that we wish to reduce modulo f, where we assume a bit representation for these polynomials.

Algorithm 6.12: Reduction of g by a trinomial

$g_1 \leftarrow g \gg n$.
$g_0 \leftarrow g[n - 1 \ldots 0]$.
$g \leftarrow g_0 \oplus g_1 \oplus (g_1 \ll k)$.
$g_1 \leftarrow g \gg n$.
$g_0 \leftarrow g[n - 1 \ldots 0]$.
$g \leftarrow g_0 \oplus g_1 \oplus (g_1 \ll k)$.

So to complete our description of how to multiply elements in $\mathbb{F}_{2^n}$ we need to explain how to perform the multiplication of two binary polynomials of large degree $n - 1$. Again one can use a naive multiplication algorithm. Often however one uses a look-up table for multiplication of polynomials of degree less than eight, i.e. for operands which fit into one byte. Then multiplication of larger-degree polynomials is reduced to multiplication of polynomials of degree less than eight by using a variant of the standard long multiplication algorithm from school. This algorithm will have complexity $O(n^2)$, where n is the degree of the polynomials involved.

Suppose we have a routine which uses a look-up table to multiply two binary polynomials of degree less than eight, returning a binary polynomial of degree less than sixteen. This function we denote by $\mathsf{MultTab}(a, b)$ where a and b are 8-bit integers representing the input polynomials. To perform a multiplication of two n-bit polynomials represented by two n-bit integers x and y we

perform Algorithm 6.13, where $y \gg 8$ (resp. $y \ll 8$) represents shifting to the rightmost (resp. leftmost) by 8 bits.

Algorithm 6.13: Multiplication of two n-bit polynomials x and y over $\mathbb{F}_2$

```
i ← 0, a ← 0.
while x ≠ 0 do
    u ← y, j ← 0.
    while u ≠ 0 do
        w ← MultTab(x&255, u&255).
        w ← w ≪ (8 · (i + j)).
        a ← a ⊕ w.
        u ← u ≫ 8.
        j ← j + 1.
    x ← x ≫ 8.
    i ← i + 1.
return a.
```

6.5.3. Karatsuba Multiplication: Just as with integer multiplication one can use a divide-and-conquer technique based on Karatsuba multiplication, which again will have a complexity of $O(n^{1.58})$. Suppose the two polynomials we wish to multiply are given by

$$a = a_0 + a_1 \cdot x^{n/2},$$
$$b = b_0 + b_1 \cdot x^{n/2},$$

where a_0, a_1, b_0, b_1 are polynomials of degree less than $n/2$. We then multiply a and b by computing

$$A \leftarrow a_0 \cdot b_0,$$
$$B \leftarrow (a_0 + a_1) \cdot (b_0 + b_1),$$
$$C \leftarrow a_1 \cdot b_1.$$

The product $a \cdot b$ is then given by

$$\begin{aligned} C \cdot x^n + (B - A - C) \cdot x^{n/2} + A &= a_1 \cdot b_1 \cdot x^n + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot x^{n/2} + a_0 \cdot b_0 \\ &= (a_0 + a_1 \cdot x^{n/2}) \cdot (b_0 + b_1 \cdot x^{n/2}) \\ &= a \cdot b. \end{aligned}$$

Again to multiply a_0 and b_0 etc. we use the Karatsuba multiplication method recursively. Once we reduce to the case of multiplying two polynomials of degree less than eight we resort to using our look-up table to perform the polynomial multiplication. Unlike the integer case we now find that Karatsuba multiplication is more efficient than the schoolbook method even for polynomials of quite small degree, say $n \approx 32$.

6.5.4. Squaring in Characteristic Two: One should note that squaring polynomials in fields of characteristic two is particularly easy. Suppose we have a polynomial

$$a = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3,$$

where $a_i = 0$ or 1. Then to square a we simply "thin out" the coefficients, as $2 = 0 \pmod 2$, as follows:

$$a^2 = a_0 + a_1 \cdot x^2 + a_2 \cdot x^4 + a_3 \cdot x^6.$$

This means that squaring an element in a finite field of characteristic two is very fast compared with a multiplication operation.

Chapter Summary

- Modular exponentiation, or exponentiation in any group, can be computed using the binary exponentiation method. Often it is more efficient to use a window based method, or to use a signed-exponentiation method in the case of elliptic curves.
- There are some special optimizations in various cases. In the case of a known exponent we hope to choose one which is both small and has very low Hamming weight. For exponentiation by a private exponent we use knowledge of the prime factorization of the modulus and the Chinese Remainder Theorem.
- Simultaneous exponentiation is often more efficient than performing two single exponentiations and then combining the result.
- Modular arithmetic is usually implemented using the technique of Montgomery representation. This allows us to avoid costly division operations by replacing the division with simple shift operations. This however is at the expense of using a non-standard representation for the numbers.
- Finite fields of characteristic two can also be implemented efficiently, but now the modular reduction operation can be made simple by choosing a special polynomial $f(x)$. Inversion is also particularly simple using a variant of the binary Euclidean algorithm, although often inversion is still three to ten times slower than multiplication.

Further Reading

The standard reference work for the type of algorithms considered in this chapter is Volume 2 of Knuth. A more gentle introduction can be found in the book by Bach and Shallit, whilst for more algorithms one should consult the book by Cohen. The first chapter of Cohen gives a number of lessons learnt in the development of the PARI/GP calculator which can be useful, whilst Bach and Shallit provides an extensive bibliography and associated commentary.

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

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

D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1975.

TABLE 6.1. Trinomials and pentanomials

n	$k/k_1, k_2, k_3$	n	$k/k_1, k_2, k_3$	n	$k/k_1, k_2, k_3$
2	1	3	1	4	1
5	2	6	1	7	1
8	7,3,2	9	1	10	3
11	2	12	3	13	4,3,1
14	5	15	1	16	5,3,1
17	3	18	3	19	5,2,1
20	3	21	2	22	1
23	5	24	8,3,2	25	3
26	4,3,1	27	5,2,1	28	1
29	2	30	1	31	3
32	7,3,2	33	10	34	7
35	2	36	9	37	6,4,1
38	6,5,1	39	4	40	5,4,3
41	3	42	7	43	6,4,3
44	5	45	4,3,1	46	1
47	5	48	11,5,1	49	9
50	4,3,2	51	6,3,1	52	3
53	6,2,1	54	9	55	7
56	7,4,2	57	4	58	19
59	7,4,2	60	1	61	5,2,1
62	29	63	1	64	11,2,1
65	32	66	3	67	5,2,1
68	33	69	6,5,2	70	37,34,33
71	35	72	36,35,33	73	42
74	35	75	35,34,32	76	38,33,32
77	38,33,32	78	41,37,32	79	40,36,32
80	45,39,32	81	35	82	43,35,32
83	39,33,32	84	35	85	35,34,32
86	49,39,32	87	46,34,32	88	45,35,32
89	38	90	35,34,32	91	41,33,32
92	37,33,32	93	35,34,32	94	43,33,32
95	41,33,32	96	57,38,32	97	33
98	63,35,32	99	42,33,32	100	37
101	40,34,32	102	37	103	72
104	43,33,32	105	37	106	73,33,32
107	54,33,32	108	33	109	34,33,32
110	33	111	49	112	73,51,32
113	37,33,32	114	69,33,32	115	53,33,32
116	48,33,32	117	78,33,32	118	33

119	38	120	41,35,32	121	35,34,32
122	39,34,32	123	42,33,32	124	37
125	79,33,32	126	49	127	63
128	55,33,32	129	46	130	61,33,32
131	43,33,32	132	44,33,32	133	46,33,32
134	57	135	39,33,32	136	35,33,32
137	35	138	57,33,32	139	38,33,32
140	45	141	85,35,32	142	71,33,32
143	36,33,32	144	59,33,32	145	52
146	71	147	49	148	61,33,32
149	64,34,32	150	53	151	39
152	35,33,32	153	71,33,32	154	109,33,32
155	62	156	57	157	47,33,32
158	76,33,32	159	34	160	79,33,32
161	39	162	63	163	48,34,32
164	42,33,32	165	35,33,32	166	37
167	35	168	134,33,32	169	34
170	105,35,32	171	125,34,32	172	81
173	71,33,32	174	57	175	57
176	79,37,32	177	88	178	87
179	80,33,32	180	33	181	46,33,32
182	81	183	56	184	121,39,32
185	41	186	79	187	37,33,32
188	46,33,32	189	37,34,32	190	47,33,32
191	51	192	147,33,32	193	73
194	87	195	50,34,32	196	33
197	38,33,32	198	65	199	34
200	57,35,32	201	59	202	55
203	68,33,32	204	99	205	94,33,32
206	37,33,32	207	43	208	119,34,32
209	45	210	49,35,32	211	175,33,32
212	105	213	75,33,32	214	73
215	51	216	115,34,32	217	45
218	71	219	54,33,32	220	33
221	63,33,32	222	102,33,32	223	33
224	39,33,32	225	32	226	59,34,32
227	81,33,32	228	113	229	64,35,32
230	50,33,32	231	34	232	191,33,32
233	74	234	103	235	34,33,32
236	50,33,32	237	80,34,32	238	73
239	36	240	177,35,32	241	70
242	95	243	143,34,32	244	111
245	87,33,32	246	62,33,32	247	82

248	155,33,32	249	35	250	103
251	130,33,32	252	33	253	46
254	85,33,32	255	52	256	91,33,32
257	41	258	71	259	113,33,32
260	35	261	89,34,32	262	86,33,32
263	93	264	179,33,32	265	42
266	47	267	42,33,32	268	61
269	207,33,32	270	53	271	58
272	165,35,32	273	53	274	67
275	81,33,32	276	63	277	91,33,32
278	70,33,32	279	38	280	242,33,32
281	93	282	35	283	53,33,32
284	53	285	50,33,32	286	69
287	71	288	111,33,32	289	36
290	81,33,32	291	168,33,32	292	37
293	94,33,32	294	33	295	48
296	87,33,32	297	83	298	61,33,32
299	147,33,32	300	45	301	83,33,32
302	41	303	36,33,32	304	203,33,32
305	102	306	66,33,32	307	46,33,32
308	40,33,32	309	107,33,32	310	93
311	78,33,32	312	87,33,32	313	79
314	79,33,32	315	132,33,32	316	63
317	36,34,32	318	45	319	36
320	135,34,32	321	41	322	67
323	56,33,32	324	51	325	46,33,32
326	65,33,32	327	34	328	195,37,32
329	50	330	99	331	172,33,32
332	89	333	43,34,32	334	43,33,32
335	113,33,32	336	267,33,32	337	55
338	86,35,32	339	72,33,32	340	45
341	126,33,32	342	125	343	75
344	135,34,32	345	37	346	63
347	56,33,32	348	103	349	182,34,32
350	53	351	34	352	147,34,32
353	69	354	99	355	43,33,32
356	112,33,32	357	76,34,32	358	57
359	68	360	323,33,32	361	56,33,32
362	63	363	74,33,32	364	67
365	303,33,32	366	38,33,32	367	171
368	283,34,32	369	91	370	139
371	116,33,32	372	111	373	299,33,32
374	42,33,32	375	64	376	227,33,32

377	41	378	43	379	44,33,32
380	47	381	107,34,32	382	81
383	90	384	295,34,32	385	51
386	83	387	162,33,32	388	159
389	275,33,32	390	49	391	37,33,32
392	71,33,32	393	62	394	135
395	301,33,32	396	51	397	161,34,32
398	122,33,32	399	49	400	191,33,32
401	152	402	171	403	79,33,32
404	65	405	182,33,32	406	141
407	71	408	267,33,32	409	87
410	87,33,32	411	122,33,32	412	147
413	199,33,32	414	53	415	102
416	287,38,32	417	107	418	199
419	200,33,32	420	45	421	191,33,32
422	149	423	104,33,32	424	213,34,32
425	42	426	63	427	62,33,32
428	105	429	83,33,32	430	62,33,32
431	120	432	287,34,32	433	33
434	55,33,32	435	236,33,32	436	165
437	40,34,32	438	65	439	49
440	63,33,32	441	35	442	119,33,32
443	221,33,32	444	81	445	146,33,32
446	105	447	73	448	83,33,32
449	134	450	47	451	406,33,32
452	97,33,32	453	87,33,32	454	128,33,32
455	38	456	67,34,32	457	61
458	203	459	68,33,32	460	61
461	194,35,32	462	73	463	93
464	143,33,32	465	59	466	143,33,32
467	156,33,32	468	33	469	116,34,32
470	149	471	119	472	47,33,32
473	200	474	191	475	134,33,32
476	129	477	150,33,32	478	121
479	104	480	169,35,32	481	138
482	48,35,32	483	288,33,32	484	105
485	267,33,32	486	81	487	94
488	79,33,32	489	83	490	219
491	61,33,32	492	50,33,32	493	266,33,32
494	137	495	76	496	43,33,32
497	78	498	155	499	40,33,32
500	75				