High-Speed RSA Implementation

Çetin Kaya Koç
Koc@ece.orst.edu

RSA Laboratories
RSA Data Security, Inc.
100 Marine Parkway, Suite 500
Redwood City, CA 94065-1031

Version 2.0 – November 1994

Contents

Preface

This report is written for people who are interested in implementing modular exponentiation based cryptosystems. These include the RSA algorithm, the Diffie-Hellman key exchange scheme, the ElGamal algorithm, and the recently proposed Digital Signature Standard (DSS) of the National Institute for Standards and Technology. The emphasis of the report is on the underlying mathematics, algorithms, and their running time analyses. The report does not include any actual code; however, we have selected the algorithms which are particularly suitable for microprocessor and signal processor implementations. It is our aim and hope that the report will close the gap between the mathematics of the modular exponentiation operation and its actual implementation on a general purpose processor.

Chapter 1

The RSA Cryptosystem

1.1 The RSA Algorithm

The RSA algorithm was invented by Rivest, Shamir, and Adleman [41]. Let p and q be two distinct large random primes. The modulus n is the product of these two primes: $n = pq$. Euler's totient function of n is given by

$$\phi(n) = (p-1)(q-1) \ .$$

Now, select a number $1 < e < \phi(n)$ such that

$$\gcd(e, \phi(n)) = 1 \ ,$$

and compute d with

$$d = e^{-1} \bmod \phi(n)$$

using the extended Euclidean algorithm [19, 31]. Here, e is the public exponent and d is the private exponent. Usually one selects a small public exponent, e.g., $e = 2^{16} + 1$. The modulus n and the public exponent e are published. The value of d and the prime numbers p and q are kept secret. Encryption is performed by computing

$$C = M^e \pmod{n} \ ,$$

where M is the plaintext such that $0 \leq M < n$. The number C is the ciphertext from which the plaintext M can be computed using

$$M = C^d \pmod{n} \ .$$

The correctness of the RSA algorithm follows from Euler's theorem: Let n and a be positive, relatively prime integers. Then

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

Since we have $ed = 1 \bmod \phi(n)$, i.e., $ed = 1 + K\phi(n)$ for some integer K, we can write

$$\begin{aligned} C^d &= (M^e)^d \pmod n \\ &= M^{ed} \pmod n \\ &= M^{1+K\phi(n)} \pmod n \\ &= M \cdot (M^{\phi(n)})^K \pmod n \\ &= M \cdot 1 \pmod n \end{aligned}$$

provided that $\gcd(M, n) = 1$. The exception $\gcd(M, n) > 1$ can be dealt as follows. According to Carmichael's theorem

$$M^{\lambda(n)} = 1 \pmod n$$

where $\lambda(n)$ is Carmichael's function which takes a simple form for $n = pq$, namely,

$$\lambda(pq) = \frac{(p-1)(q-1)}{\gcd(p-1, q-1)} \ .$$

Note that $\lambda(n)$ is always a proper divisor of $\phi(n)$ when n is the product of distinct odd primes; in this case $\lambda(n)$ is smaller than $\phi(n)$. Now, the relationship between e and d is given by

$$M^{ed} = M \pmod n \ \text{if} \quad ed = 1 \pmod{\lambda(n)} \ .$$

Provided that n is a product of distinct primes, the above holds for all M, thus dealing with the above-mentioned exception $\gcd(M, n) > 1$ in Euler's theorem.

As an example, we construct a simple RSA cryptosystem as follows: Pick $p = 11$ and $q = 13$, and compute

$$\begin{aligned} n &= p \cdot q &&= 11 \cdot 13 &&= 143 \ , \\ \phi(n) &= (p-1) \cdot (q-1) &&= 10 \cdot 12 &&= 120 \ . \end{aligned}$$

We can also compute Carmichael's function of n as

$$\lambda(pq) = \frac{(p-1)(q-1)}{\gcd(p-1, q-1)} = \frac{10 \cdot 12}{\gcd(10, 12)} = \frac{120}{2} = 60 \ .$$

The public exponent e is selected such that $1 < e < \phi(n)$ and

$$\gcd(e, \phi(n)) = \gcd(e, 120) = 1 \ .$$

For example, $e = 17$ would satisfy this constraint. The private exponent d is computed by

$$\begin{aligned} d &= e^{-1} \pmod{\phi(n)} \\ &= 17^{-1} \pmod{120} \\ &= 113 \end{aligned}$$

which is computed using the extended Euclidean algorithm, or any other algorithm for computing the modular inverse. Thus, the user publishes the public exponent and the modulus: $(e, n) = (13, 143)$, and keeps the following private: $d = 113$, $p = 11$, $q = 13$. A typical encryption/decryption process is executed as follows:

$$\begin{array}{ll} \text{Plaintext:} & M = 50 \\ \text{Encryption:} & C := M^e \pmod{n} \\ & C := 50^{17} \pmod{143} \\ & C = 85 \end{array}$$

$$\begin{array}{ll} \text{Ciphertext:} & C = 85 \\ \text{Decryption:} & M := M^d \pmod{n} \\ & M := 85^{113} \pmod{143} \\ & M = 50 \end{array}$$

1.2 Exchange of Private Messages

The public-key directory contains the pairs (e, n) for each user. The users wishing to send private messages to one another refer to the directory to obtain these parameters. For example, the directory might be arranged as follows:

User	Public Keys
Alice	(e_a, n_a)
Bob	(e_b, n_b)
Cathy	(e_c, n_c)
$\cdots$	$\cdots$

The pair n_a and e_a respectively are the modulus and the public exponent for Alice. As an example, we show how Alice sends her private message M to Bob. In our simple protocol example Alice executes the following steps:

1. Alice locates Bob's name in the directory and obtains his public exponent and the modulus: (e_b, n_b).
2. Alice computes $C := M^{e_b} \pmod{n_b}$.
3. Alice sends C to Bob over the network.
4. Bob receives C.
5. Bob uses his private exponent and the modulus, and computes $M = C^{d_b} \pmod{n_b}$ in order to obtain M.

1.3 Signing Digital Documents

The RSA algorithm provides a procedure for signing a digital document, and verifying whether the signature is indeed authentic. The signing of a digital document is somewhat different from signing a paper document, where the same signature is being produced for all paper documents. A digital signature cannot be a constant; it is a function of the digital document for which it was produced. After the signature (which is just another piece of digital data) of a digital document is obtained, it is attached to the document for anyone wishing the verify the authenticity of the document and the signature. Here we will briefly illustrate the process of signing using the RSA cryptosystem. Suppose Alice wants to sign a message, and Bob would like to obtain a proof that this message is indeed signed by Alice. First, Alice executes the following steps:

1. Alice takes the message M and computes $S = M^{d_a} \pmod{n_a}$.
2. Alice makes her message M and the signature S available to any party wishing to verify the signature.

Bob executes the following steps in order to verify Alice's signature S on the document M:

1. Bob obtains M and S, and locates Alice's name in the directory and obtains her public exponent and the modulus (e_a, n_a).
2. Bob computes $M' = S^{e_a} \pmod{n_a}$.
3. If $M' = M$ then the signature is verified. Otherwise, either the original message M or the signature S is modified, thus, the signature is not valid.

We note that the protocol examples given here for illustration purposes only — they are simple 'textbook' protocols; in practice, the protocols are somewhat more complicated. For example, secret-key cryptographic techniques may also be used for sending private messages. Also, signing is applied to messages of arbitrary length. The signature is often computed by first computing a hash value of the long message and then signing this hash value. We refer the reader to the report [42] and Public Key Cryptography Standards [43] published by RSA Data Security, Inc., for answers to certain questions on these issues.

1.4 Computation of Modular Exponentiation

Once an RSA cryptosystem is set up, i.e., the modulus and the private and public exponents are determined and the public components have been published, the senders as well as the recipients perform a single operation for signing, verification, encryption, and decryption. The RSA algorithm in this respect is one of the simplest cryptosystems. The operation required is the computation of $M^e \pmod{n}$, i.e., the modular exponentiation. The modular exponentiation operation is a common operation for scrambling; it is used in several cryptosystems. For example, the Diffie-Hellman key exchange scheme requires modular exponentiation [8]. Furthermore, the ElGamal signature scheme [13] and the recently proposed Digital Signature Standard (DSS) of the National Institute for Standards and Technology [34] also require the computation of modular exponentiation. However, we note that the exponentiation process in a cryptosystem based on the discrete logarithm problem is slightly different: The base (M) and the modulus (n) are known in advance. This allows some precomputation since powers of the base can be precomputed and saved [6]. In the exponentiation process for the RSA algorithm, we know the exponent (e) and the modulus (n) in advance but not the base; thus, such optimizations are not likely to be applicable. The emphasis of this report is on the RSA cryptosystem as the title suggests.

In the following chapters we will review techniques for implementation of modular exponentiation operation on general-purpose computers, e.g., personal computers, microprocessors, microcontrollers, signal processors, workstations, and mainframe computers. This report does not include any *actual* code; it covers mathematical and algorithmic aspects of the *software* implementations of the RSA algorithm. There also exist hardware structures for performing the modular multiplication and exponentiations, for example, see [40, 28, 46, 15, 24, 25, 26, 50]. A brief review of the hardware implementations can be found in [5].

Chapter 2

Modular Exponentiation

2.1 Modular Exponentiation

The first rule of modular exponentiation is that we do *not* compute

$$C := M^e \pmod{n}$$

by first exponentiating

$$M^e$$

and then performing a division to obtain the remainder

$$C := (M^e) \ \% \ n \ .$$

The temporary results *must* be reduced modulo n at each step of the exponentiation. This is because the space requirement of the binary number M^e is enormous. Assuming, M and e have 256 bits each, we need

$$\log_2(M^e) = e \cdot \log_2(M) \approx 2^{256} \cdot 256 = 2^{264} \approx 10^{80}$$

bits in order to store M^e. This number is approximately equal to the number of particles in the universe [1]; we have no way of storing it. In order to compute the bit capacity of all computers in the world, we can make a generous assumption that there are 512 million computers, each of which has 512 MBytes of memory. Thus, the total number of bits available would be

$$512 \cdot 2^{20} \cdot 512 \cdot 2^{20} \cdot 8 = 2^{61} \approx 10^{18} \ ,$$

which is only enough to store M^e when M and e are 55 bits.

2.2 Exponentiation

We raise the following question: *How many modular multiplications are needed to compute* $M^e \bmod n$ *?* A naive way of computing $C = M^e \pmod{n}$ is to start with $C := M$

(mod n) and keep performing the modular multiplication operations

$$C := C \cdot M \pmod{n}$$

until $C = M^e \pmod{n}$ is obtained. The naive method requires $e-1$ modular multiplications to compute $C := M^e \pmod{n}$, which would be prohibitive for large e. For example, if we need to compute $M^{15} \pmod{n}$, this method computes all powers of M until 15:

$$M \to M^2 \to M^3 \to M^4 \to M^5 \to M^6 \to M^7 \to \cdots \to M^{15}$$

which requires 14 multiplications. However, not all powers of M need to be computed in order to obtain M^{15}. Here is a faster method of computing M^{15}:

$$M \to M^2 \to M^3 \to M^6 \to M^7 \to M^{14} \to M^{15}$$

which requires 6 multiplications. The method by which M^{15} is computed is not specific for certain exponents; it can be used to compute M^e for any e. The algorithm is called the *binary method* or *square and multiply* method, and dates back to antiquity.

2.3 The Binary Method

The binary method scans the bits of the exponent either from left to right or from right to left. A squaring is performed at each step, and depending on the scanned bit value, a subsequent multiplication is performed. We describe the left-to-right binary method below. The right-to-left algorithm requires one extra variable to keep the powers of M. The reader is referred to Section 4.6.3 of Knuth's book [19] for more information. Let k be the number of bits of e, i.e., $k = 1 + \lfloor \log_2 e \rfloor$, and the binary expansion of e be given by

$$e \;=\; (e_{k-1}e_{k-2}\cdots e_1 e_0) = \sum_{i=0}^{k-1} e_i 2^i$$

for $e_i \in \{0, 1\}$. The binary method for computing $C = M^e \pmod{n}$ is given below:

The Binary Method
Input: M, e, n.
Output: $C = M^e \bmod n$.
1. **if** $e_{k-1} = 1$ **then** $C := M$ **else** $C := 1$
2. **for** $i = k-2$ **downto** 0
  2a. $C := C \cdot C \pmod{n}$
  2b. **if** $e_i = 1$ **then** $C := C \cdot M \pmod{n}$
3. **return** C

As an example, let $e = 250 = (11111010)$, which implies $k = 8$. Initially, we take $C := M$ since $e_{k-1} = e_7 = 1$. The binary method proceeds as follows:

i	e_i	Step 2a	Step 2b
6	1	$(M)^2 = M^2$	$M^2 \cdot M = M^3$
5	1	$(M^3)^2 = M^6$	$M^6 \cdot M = M^7$
4	1	$(M^7)^2 = M^{14}$	$M^{14} \cdot M = M^{15}$
3	1	$(M^{15})^2 = M^{30}$	$M^{30} \cdot M = M^{31}$
2	0	$(M^{31})^2 = M^{62}$	M^{62}
1	1	$(M^{62})^2 = M^{124}$	$M^{124} \cdot M = M^{125}$
0	0	$(M^{125})^2 = M^{250}$	M^{250}

The number of modular multiplications required by the binary method for computing M^{250} is found to be $7+5=12$. For an arbitrary k-bit number e with $e_{k-1}=1$, the binary method requires:

- Squarings (Step 2a): $k-1$ where k is the number of bits in the binary expansion of e.
- Multiplications (Step 2b): $H(e)-1$ where $H(e)$ is the Hamming weight (the number of 1s in the binary expansion) of e.

Assuming $e > 0$, we have $0 \leq H(e)-1 \leq k-1$. Thus, the total number of multiplications is found as:

$$\begin{aligned}
\text{Maximum:} \quad (k-1)+(k-1) &= 2(k-1) \ , \\
\text{Minimum:} \quad (k-1)+0 &= k-1 \ , \\
\text{Average:} \quad (k-1)+\tfrac{1}{2}(k-1) &= \tfrac{3}{2}(k-1) \ ,
\end{aligned}$$

where we assume that $e_{k-1}=1$.

2.4 The m-ary Method

The binary method can be generalized by scanning the bits of e

- 2 at a time: the quaternary method, or
- 3 at a time: the octal method, etc.

More generally,

- $\log_2 m$ at a time: the m-ary method.

The m-ary method is based on m-ary expansion of the exponent. The digits of e are then scanned and squarings (powerings) and subsequent multiplications are performed accordingly. The method was described in Knuth's book [19]. When m is a power of 2, the implementation of the m-ary method is rather simple, since M^e is computed by grouping the bits of the binary expansion of the exponent e. Let $e = (e_{k-1}e_{k-2}\cdots e_1e_0)$ be the binary expansion of the exponent. This representation of e is partitioned into s blocks of length r each for $sr = k$. If r does not divide k, the exponent is padded with at most $r-1$ 0s. We define

$$F_i = (e_{ir+r-1}e_{ir+r-2}\cdots e_{ir}) = \sum_{j=0}^{r-1} e_{ir+j}2^j \ .$$

Note that $0 \leq F_i \leq m-1$ and $e = \sum_{i=0}^{s-1} F_i 2^{ir}$. The m-ary method first computes the values of $M^w \pmod{n}$ for $w = 2, 3, \ldots, m-1$. Then the bits of e are scanned r bits at a time from the most significant to the least significant. At each step the partial result is raised to the 2^r power and multiplied y M^{F_i} modulo n where F_i is the (nonzero) value of the current bit section.

The m-ary Method
Input: M, e, n.
Output: $C = M^e \bmod n$.
1. Compute and store $M^w \pmod{n}$ for all $w = 2, 3, 4, \ldots, m-1$.
2. Decompose e into r-bit words F_i for $i = 0, 1, 2, \ldots, s-1$.
3. $C := M^{F_{s-1}} \pmod{n}$
4. **for** $i = s-2$ **downto** 0
  4a. $C := C^{2^r} \pmod{n}$
  4b. **if** $F_i \neq 0$ **then** $C := C \cdot M^{F_i} \pmod{n}$
5. **return** C

2.4.1 The Quaternary Method

We first consider the quaternary method. Since the bits of e are scanned two at a time, the possible digit values are $(00) = 0$, $(01) = 1$, $(10) = 2$, and $(11) = 3$. The multiplication step (Step 4b) may require the values M^0, M^1, M^2, and M^3. Thus, we need to perform some preprocessing to obtain M^2 and M^3. As an example, let $e = 250$ and partition the bits of e in groups of two bits as

$$e = 250 = \underline{11}\ \underline{11}\ \underline{10}\ \underline{10} \ .$$

Here, we have $s = 4$ (the number of groups $s = k/r = 8/2 = 4$). During the preprocessing step, we compute:

bits	w	M^w
00	0	1
01	1	M
10	2	$M \cdot M = M^2$
11	3	$M^2 \cdot M = M^3$

The quaternary method then assigns $C := M^{F_3} = M^3 \pmod{n}$, and proceeds to compute $M^{250} \pmod{n}$ as follows:

i	F_i	Step 4a	Step 4b
2	11	$(M^3)^4 = M^{12}$	$M^{12} \cdot M^3 = M^{15}$
1	10	$(M^{15})^4 = M^{60}$	$M^{60} \cdot M^2 = M^{62}$
0	10	$(M^{62})^4 = M^{248}$	$M^{248} \cdot M^2 = M^{250}$

The number of modular multiplications required by the quaternary method for computing $M^{250} \pmod{n}$ is found as $2 + 6 + 3 = 11$.

2.4.2 The Octal Method

The octal method partitions the bits of the exponent in groups of 3 bits. For example, $e = 250$ is partitioned as

$$e = 250 = \underline{011}\ \underline{111}\ \underline{010}\ ,$$

by padding a zero to the left, giving $s = k/r = 9/3 = 3$. During the preprocessing step we compute $M^w \pmod{n}$ for all $w = 2, 3, 4, 5, 6, 7$.

bits	w	M^w
000	0	1
001	1	M
010	2	$M \cdot M = M^2$
011	3	$M^2 \cdot M = M^3$
100	4	$M^3 \cdot M = M^4$
101	5	$M^4 \cdot M = M^5$
110	6	$M^5 \cdot M = M^6$
111	7	$M^6 \cdot M = M^7$

The octal method then assigns $C := M^{F_2} = M^3 \pmod{n}$, and proceeds to compute $M^{250} \pmod{n}$ as follows:

i	F_i	Step 4a	Step 4b
1	111	$(M^3)^8 = M^{24}$	$M^{24} \cdot M^7 = M^{31}$
0	010	$(M^{31})^8 = M^{248}$	$M^{248} \cdot M^2 = M^{250}$

The computation of $M^{250} \pmod{n}$ by the octal method requires a total of $6 + 6 + 2 = 14$ modular multiplications. However, notice that, even though we have computed $M^w \pmod{n}$ for all $w = 2, 3, 4, 5, 6, 7$, we have not used *all* of them. Thus, we can slightly modify Step 1 of the m-ary method and precompute $M^w \pmod{n}$ for only those w which appear in the partitioned binary expansion of e. For example, for $e = 250$, the partitioned bit values are $(011) = 3$, $(111) = 7$, and $(010) = 2$. We can compute these powers using only 4 multiplications:

bits	w	M^w
000	0	1
001	1	M
010	2	$M \cdot M = M^2$
011	3	$M^2 \cdot M = M^3$
100	4	$M^3 \cdot M = M^4$
111	7	$M^4 \cdot M^3 = M^7$

This gives the total number of multiplications required by the octal method for computing $M^{250} \pmod{n}$ as $4+6+2 = 12$. The method of computing $M^e \pmod{n}$ by precomputing $M^w \pmod{n}$ for only those w which appear in the partitioning of the exponent is termed a data-dependent or an adaptive algorithm. In the following section, we will explore methods of this kind which try to reduce the number of multiplications by making use of the properties of the given e. In general, we will probably have to compute $M^w \pmod{n}$ for all $w = 2, 3, \ldots, 2^r - 1$. This will be more of the case when k is very large. We summarize the average number of multiplications and squarings required by the m-ary method assuming $2^r = m$ and $\frac{k}{r}$ is an integer.

- Preprocessing Multiplications (Step 1): $m - 2 = 2^r - 2$
- Squarings (Step 4a): $(\frac{k}{r} - 1) \cdot r = k - r$
- Multiplications (Step 4b): $(\frac{k}{r} - 1)(1 - \frac{1}{m}) = (\frac{k}{r} - 1)(1 - 2^{-r})$

Thus, in general, the m-ary method requires

$$2^r - 2 + k - r + \left(\frac{k}{r} - 1\right)\left(1 - 2^{-r}\right)$$

multiplications plus squarings on the average. The average number of multiplications for the binary method can be found simply by substituting $r = 1$ and $m = 2$ in the above, which gives $\frac{3}{2}(k-1)$. Also note that there exists an optimal $r = r^*$ for each k such that the average number of multiplications required by the m-ary method is minimum. The optimal values of r can be found by enumeration [21]. In the following we tabulate the average values of multiplications plus squarings required by the binary method and the m-ary method with the optimal values of r.

k	binary	m-ary	r^*	Savings %
8	11	10	2	9.1
16	23	21	2	8.6
32	47	43	2,3	8.5
64	95	85	3	10.5
128	191	167	3,4	12.6
256	383	325	4	15.1
512	767	635	5	17.2
1024	1535	1246	5	18.8
2048	3071	2439	6	20.6

The asymptotic value of savings offered by the m-ary method is equal to 33 %. In order to prove this statement, we compute the limit of the ratio

$$\lim_{k\to\infty} \frac{2^r - 2 + k - r + (\frac{k}{r} - 1)(1 - 2^{-r})}{\frac{3}{2}(k-1)} = \frac{2}{3}\left(1 + \frac{1 - 2^{-r}}{r}\right) \approx \frac{2}{3} \ .$$

2.5 The Adaptive m-ary Methods

The adaptive methods are those which form their method of computation according to the input data. In the case of exponentiation, an adaptive algorithm will modify its structure according to the exponent e, once it is supplied. As we have pointed out earlier, the number of preprocessing multiplications can be reduced if the partitioned binary expansion of e do not contain all possible bit-section values w. However, there are also adaptive algorithms which partition the exponent into a series of zero and nonzero words in order to decrease the number multiplications required in Step 4b of the m-ary method. In the following we introduce these methods, and give the required number of multiplications and squarings.

2.5.1 Reducing Preprocessing Multiplications

We have already briefly introduced this method. Once the binary expansion of the exponent is obtained, we partition this number into groups of d bits each. We then precompute and obtain $M^w \pmod{n}$ only for those w which appear in the binary expansion. Consider the following exponent for $k = 16$ and $d = 4$

$$\underline{1011}\ \underline{0011}\ \underline{0111}\ \underline{1000}$$

which implies that we need to compute $M^w \pmod{n}$ for only $w = 3, 7, 8, 11$. The exponent values $w = 3, 7, 8, 11$ can be sequentially obtained as follows:

$$\begin{array}{rcl} M^2 &=& M \cdot M \\ M^3 &=& M^2 \cdot M \\ M^4 &=& M^2 \cdot M^2 \\ M^7 &=& M^3 \cdot M^4 \\ M^8 &=& M^4 \cdot M^4 \\ M^{11} &=& M^8 \cdot M^3 \end{array}$$

which requires 6 multiplications. The m-ary method that disregards the necessary exponent values and computes all of them would require $16 - 2 = 14$ preprocessing multiplications. The number of multiplications that can be saved is upper-bounded by $m - 2 = 2^d - 2$, which is the case when all partitioned exponent values are equal to 1, e.g., when

$$\underline{0001}\ \underline{0001}\ \underline{0001}\ \underline{0001}$$

This implies that we do not precompute anything, just use M. This happens quite rarely. In general, we have to compute $M^w \pmod n$ for all $w = w_0, w_1, \ldots, w_{p-1}$. If the span of the set $\{w_i \mid i = 0, 1, \ldots, p-1\}$ is the values $2, 3, \ldots, 2^d - 1$, then there is no savings. We perform $2^d - 2$ multiplications and obtain all of these values. However, if the span is a subset (especially a small subset) of the values $2, 3, \ldots, 2^d - 1$, then some savings can be achieved if we can compute w_i for $i = 0, 1, \ldots, p-1$ using much fewer than $2^d - 2$ multiplications. An algorithm for computing any given p exponent values is called *a vectorial addition chain*, and in the case of $p = 1$, *an addition chain*. Unfortunately, the problem of obtaining an addition chain of minimal length is an NP-complete problem [9]. We will elaborate on addition and vectorial addition chains in the last section of this chapter.

2.5.2 The Sliding Window Techniques

The m-ary method decomposes the bits of the exponent into d-bit words. The probability of a word of length d being zero is equal to 2^{-d}, assuming that the 0 and 1 bits are produced with equal probability. In Step 4b of the m-ary method, we skip a multiplication whenever the current word is equal to zero. Thus, as d grows larger, the probability that we have to perform a multiplication operation in Step 4a becomes larger. However, the total number of multiplications increases as d decreases. The sliding window algorithms provide a compromise by allowing zero and nonzero words of variable length; this strategy aims to increase the average number of zero words, while using relatively large values of d.

A sliding window exponentiation algorithm first decomposes e into zero and nonzero words (*windows*) F_i of length $L(F_i)$. The number of windows p may not be equal to k/d. In general, it is also not required that the length of the windows be equal. We take d to be the length of the longest window, i.e., $d = \max(L(F_i))$ for $i = 0, 1, \ldots, k-1$. Furthermore, if F_i is a nonzero window, then the least significant bit of F_i must be equal to 1. This is because we partition the exponent starting from the least significant bit, and there is no point in starting a nonzero window with a zero bit. Consequently, the number of preprocessing multiplications (Step 1) are nearly halved, since x^w are computed for odd w only.

The Sliding Window Method
Input: M, e, n.
Output: $C = M^e \pmod n$.
1. Compute and store $M^w \pmod n$ for all $w = 3, 5, 7, \ldots, 2^d - 1$.
2. Decompose e into zero and nonzero windows F_i of length $L(F_i)$ for $i = 0, 1, 2, \ldots, p-1$.
3. $C := M^{F_{k-1}} \pmod n$
4. **for** $i = p-2$ **downto** 0
  4a. $C := C^{2^{L(F_i)}} \pmod n$
  4b. **if** $F_i \neq 0$ **then** $C := C \cdot M^{F_i} \pmod n$
5. **return** C

Two sliding window partitioning strategies have been proposed [19, 4]. These methods differ in whether the length of a nonzero window must be a constant ($= d$), or can be variable (however, $\leq d$). In the following sections, we give algorithmic descriptions of these two partitioning strategies.

2.5.3 Constant Length Nonzero Windows

The constant length nonzero window (CLNW) partitioning algorithm is due to Knuth [19]. The algorithm scans the bits of the exponent from the least significant to the most significant. At any step, the algorithm is either forming a zero window (ZW) or a nonzero window (NW). The algorithm is described below:

ZW: Check the incoming single bit: if it is a 0 then stay in ZW; else go to NW.

NW: Stay in NW until all d bits are collected. Then check the incoming single bit: if it is a 0 then go to ZW; else go to NW.

Notice that while in NW, we distinguish between staying in NW and going to NW. The former means that we continue to form the same nonzero window, while the latter implies the beginning of a new nonzero window. The CLNW partitioning strategy produces zero windows of arbitrary length, and nonzero windows of length d. There cannot be two adjacent zero windows; they are necessarily concatenated, however, two nonzero windows may be adjacent. For example, for $d = 3$, we partition $e = 3665 = (111001010001)$ as

$$e = \underline{111}\ 00\ \underline{101}\ 0\ \underline{001}\ .$$

The CLNW sliding window algorithm first performs the preprocessing multiplications and obtains $M^w \pmod{n}$ for $w = 3, 5, 7$.

bits	w	M^w
001	1	M
010	2	$M \cdot M = M^2$
011	3	$M \cdot M^2 = M^3$
101	5	$M^3 \cdot M^2 = M^5$
111	7	$M^5 \cdot M^2 = M^7$

The algorithm assigns $C = M^{F_4} = M^7 \pmod{n}$, and then proceeds to compute $M^{3665} \pmod{n}$ as follows:

i	F_i	$L(F_i)$	Step 4a	Step 4b
3	00	2	$(M^7)^4 = M^{28}$	M^{28}
2	101	3	$(M^{28})^8 = M^{224}$	$M^{224} \cdot M^5 = M^{229}$
1	0	1	$(M^{229})^2 = M^{458}$	M^{458}
0	001	3	$(M^{458})^8 = M^{3664}$	$M^{3664} \cdot M = M^{3665}$

Thus, a total of $4+9+2=15$ modular multiplications are performed. The average number of multiplications can be found by modeling the partitioning process as a Markov chain. The details of this analysis are given in [23]. In following table, we tabulate the average number of multiplications for the m-ary and the CLNW sliding window methods. The column for the m-ary method contains the optimal values d^* for each k. As expected, there exists an optimal value of d for each k for the CLNW sliding window algorithm. These optimal values are also included in the table. The last column of the table contains the percentage difference in the average number of multiplications. The CLNW partitioning strategy reduces the average number of multiplications by 3–7 % for $128 \le k \le 2048$. The overhead of the partitioning is negligible; the number of bit operations required to obtain the partitioning is proportional to k.

	m-ary		CLNW		$(T-T_1)/T$
k	d^*	T	d^*	T_1	%
128	4	168	4	156	7.14
256	4	326	5	308	5.52
512	5	636	5	607	4.56
768	5	941	6	903	4.04
1024	5	1247	6	1195	4.17
1280	6	1546	6	1488	3.75
1536	6	1844	6	1780	3.47
1792	6	2142	7	2072	3.27
2048	6	2440	7	2360	3.28

2.5.4 Variable Length Nonzero Windows

The CLNW partitioning strategy starts a nonzero window when a 1 is encountered. Although the incoming $d-1$ bits may all be zero, the algorithm continues to append them to the current nonzero window. For example, for $d=3$, the exponent $e = (111001010001)$ was partitioned as

$$e = \underline{111}\ 00\ \underline{101}\ 0\ \underline{001}\ .$$

However, if we allow variable length nonzero windows, we can partition this number as

$$e = \underline{111}\ 00\ \underline{101}\ 000\ \underline{1}\ .$$

We will show that this strategy further decreases the average number of nonzero windows. The variable length nonzero window (VLNW) partitioning strategy was proposed by Bos and Coster in [4]. The strategy requires that during the formation of a nonzero window (NW), we switch to ZW when the remaining bits are all zero. The VLNW partitioning strategy has two integer parameters:

- d : maximum nonzero window length,

- q : minimum number of zeros required to switch to ZW.

The algorithm proceeds as follows:

ZW: Check the incoming single bit: if it is zero then stay in ZW; else go to NW.

NW: Check the incoming q bits: if they are all zero then go to ZW; else stay in NW. Let $d = lq + r + 1$ where $1 < r \leq q$. Stay in NW until $lq + 1$ bits are received. At the last step, the number of incoming bits will be equal to r. If these r bits are all zero then go to ZW; else stay in NW. After all d bits are collected, check the incoming single bit: if it is zero then go to ZW; else go to NW.

The VLNW partitioning produces nonzero windows which start with a 1 and end with a 1. Two nonzero windows may be adjacent; however, the one in the least significant position will necessarily have d bits. Two zero windows will not be adjacent since they will be concatenated. For example, let $d = 5$ and $q = 2$, then $5 = 1 + 1 \cdot 2 + 2$, thus $l = 1$ and $r = 2$. The following illustrates the partitioning of a long exponent according to these parameters:

$$\underline{101}\ 0\ \underline{11101}\ 00\ \underline{101}\ \underline{10111}\ 000000\ \underline{1}\ 00\ \underline{111}\ 000\ \underline{1011}\ .$$

Also, let $d = 10$ and $q = 4$, which implies $l = 2$ and $r = 1$. A partitioning example is illustrated below:

$$\underline{1011011}\ 0000\ \underline{11}\ 0000\ \underline{11110111}\ 00\ \underline{1111110101}\ 0000\ \underline{11011}\ .$$

In order to compute the average number of multiplications, the VLNW partitioning process, like the CLNW process, can be modeled using a Markov chain. This analysis was performed in [23], and the average number of multiplications have been calculated for $128 \leq k \leq 2048$. In the following table, we tabulate these values together with the optimal values of d and q, and compare them to those of the m-ary method. Experiments indicate that the best values of q are between 1 and 3 for $128 \leq k \leq 2048$ and $4 \leq d \leq 8$. The VLNW algorithm requires 5–8 % fewer multiplications than the m-ary method.

	m-ary		VLNW				$(T_2-T)/T_2$
					T_2/k		for q^*
k	d^*	T/k	d^*	$q=1$	$q=2$	$q=3$	%
128	4	1.305	4	1.204	1.203	1.228	7.82
256	4	1.270	4	1.184	1.185	1.212	6.77
512	5	1.241	5	1.163	1.175	1.162	6.37
768	5	1.225	5	1.155	1.167	1.154	5.80
1024	5	1.217	6	1.148	1.146	1.157	5.83
1280	6	1.207	6	1.142	1.140	1.152	5.55
1536	6	1.200	6	1.138	1.136	1.148	5.33
1792	6	1.195	6	1.136	1.134	1.146	5.10
2048	6	1.191	6	1.134	1.132	1.144	4.95

The sliding window algorithms are easy to program, introducing negligible overhead. The reduction in terms of the number of multiplications is notable, for example, for $n = 512$, the m-ary method requires 636 multiplications whereas the CLNW and VLNW sliding window algorithms require 607 and 595 multiplications, respectively. In Figure 2.1, we plot the scaled average number of multiplications T/k, i.e., the average number of multiplications T divided by the total number of bits k, for the m-ary and the sliding window algorithms as a function of $n = 128, 256, \ldots, 2048$.

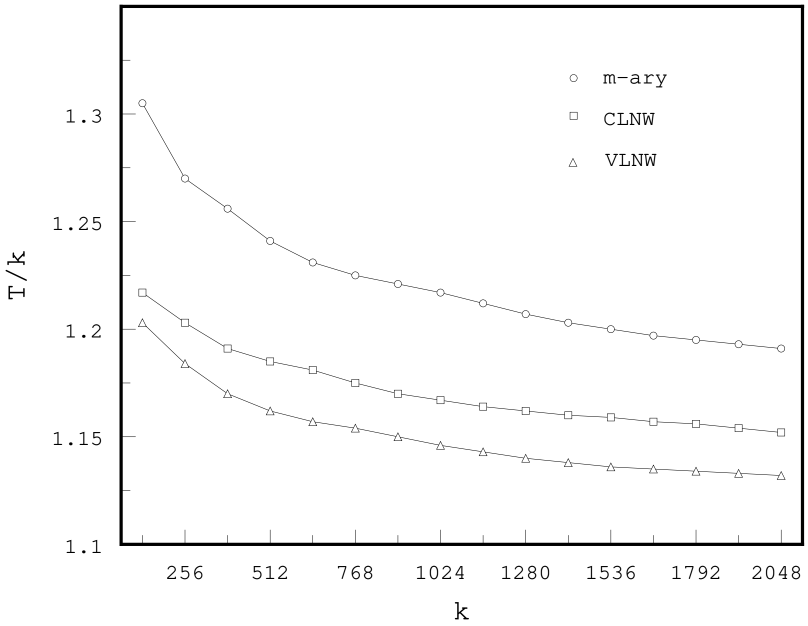

Figure 2.1: The values of T/k for the m-ary and the sliding window algorithms.

2.6 The Factor Method

The factor method is given by Knuth [19]. It is based on factorization of the exponent $e = rs$ where r is the smallest prime factor of e and $s > 1$. We compute M^e by first computing M^r and then raising this value to the sth power:

$$\begin{aligned} C_1 &= M^r \ , \\ C_2 &= C_1^s = M^{rs} = M^e \ . \end{aligned}$$

If e is prime, then we first compute M^{e-1} then multiply this quantity by M. The algorithm is recursive, e.g., in order to compute M^r, we factor $r = r_1 \cdot r_2$ such that r_1 is the smallest prime factor of r and $r_2 > 1$. This process continues until the exponent value required is equal to 2. As an example, we illustrate the computation of M^e for $e = 55 = 5 \cdot 11$ in the following:

Compute:	$M \to M^2 \to M^4 \to M^5$
Assign:	$a := M^5$
Compute:	$a \to a^2$
Assign:	$b := a^2$
Compute:	$b \to b^2 \to b^4 \to b^5$
Compute:	$b^5 \to b^5 a = M^{55}$

The factor method requires 8 multiplications for computing M^{55}. The binary method, on the other hand, requires 9 multiplications since $e = 55 = (110111)$ implies 5 squarings (Step 2a) and 4 multiplications (Step 2b).

Unfortunately, the factor method requires factorization of the exponent, which would be very difficult for large numbers. However, this method could still be of use for the RSA cryptosystem whenever the exponent value is small. It may also be useful if the exponent is constructed carefully, i.e., in a way to allow easy factorization.

2.7 The Power Tree Method

The power tree method is also due to Knuth [19]. This algorithm constructs a tree according to a heuristic. The nodes of the tree are labeled with positive integers starting from 1. The root of the tree receives 1. Suppose that the tree is constructed down to the kth level. Consider the node e of the kth level, from left to right. Construct the $(k+1)$st level by attaching below node e the nodes

$$e + a_1, e + a_2, e + a_3, \ldots, e + a_k$$

where $a_1, a_2, a_3, \ldots, a_k$ is the path from the root of the tree to e. (Note: $a_1 = 1$ and $a_k = e$.) In this process, we discard any duplicates that have already appeared in the tree. The power tree down to 5 levels is given in Figure 2.2.

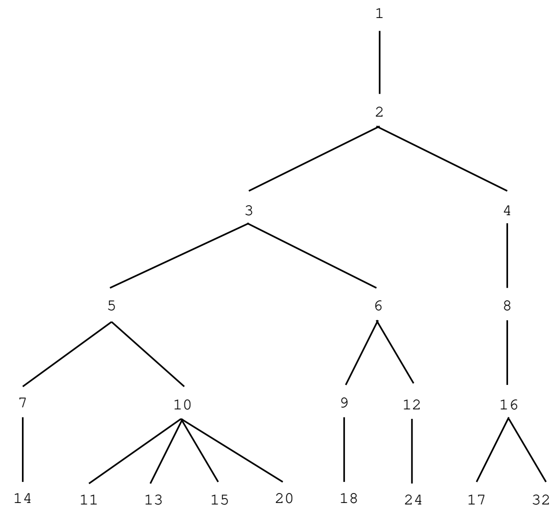

Figure 2.2: The Power Tree

In order to compute M^e, we locate e in the power tree. The sequence of exponents that occur in the computation of M^e is found on the path from the root to e. For example, the computation of M^{18} requires 5 multiplications:

$$M \to M^2 \to M^3 \to M^6 \to M^9 \to M^{18}$$

For certain exponent values of e, the power tree method requires fewer number of multiplications, e.g., the computation of M^{23} by the power tree method requires 6 multiplications:

$$M \to M^2 \to M^3 \to M^5 \to M^{10} \to M^{13} \to M^{23}$$

However, since $23 = (10111)$, the binary method requires $4 + 3 = 7$ multiplications:

$$M \to M^2 \to M^4 \to M^5 \to M^{10} \to M^{11} \to M^{22} \to M^{23}$$

Also, since $23 - 1 = 22 = 2 \cdot 11$, the factor method requires $1 + 5 + 1 = 7$ multiplications:

$$M \to M^2 \to M^4 \to M^8 \to M^{16} \to M^{20} \to M^{22} \to M^{23}$$

Knuth gives another variation of the power tree heuristics in Problem 6 in page 463 [19]. The power tree method is also applicable for small exponents since the tree needs to be "saved".

2.8 Addition Chains

Consider a sequence of integers

$$a_0, a_1, a_2, \ldots, a_r$$

with $a_0 = 1$ and $a_r = e$. If the sequence is constructed in such a way that for all k there exist indices $i, j < k$ such that

$$a_k = a_i + a_j \ ,$$

then the sequence is an addition chain for e. The length of the chain is equal to r. An addition chain for a given exponent e is an algorithm for computing M^e. We start with M^1, and proceed to compute M^{a_k} using the two previously computed values M^{a_i} and M^{a_j} as $M^{a_k} = M^{a_i} \cdot M^{a_j}$. The number of multiplications required is equal to r which is the length of the addition chain. The algorithms we have so far introduced, namely, the binary method, the m-ary method, the sliding window method, the factor and the power tree methods are in fact methods of generating addition chains for the given exponent value e. Consider for example $e = 55$, the addition chains generated by some of these algorithms are given below:

binary:	1	2	3	6	12	13	26	27	54	55	
quaternary:	1	2	3	6	12	13	26	52	55		
octal:	1	2	3	4	5	6	7	12	24	48	55
factor:	1	2	4	5	10	20	40	50	55		
power tree:	1	2	3	5	10	11	22	44	55		

Given the positive integer e, the computation of the *shortest* addition chain for e is established to be an NP-complete problem [9]. This implies that we have to compute all possible chains leading to e in order to obtain the shortest one. However, since the first introduction of the shortest addition chain problem by Scholz [19] in 1937, several properties of the addition chains have been established:

- The upper bound on the length of the shortest addition chain for e is equal to: $\lfloor \log_2 e \rfloor + H(e) - 1$ where $H(e)$ is the Hamming weight of e. This follows from the binary method. In the worst case, we can use the binary method to compute M^e using at most $\lfloor \log_2 e \rfloor + H(e) - 1$ multiplications.

- The lower bound was established by Schönhage [44]: $\log_2 e + \log_2 H(e) - 2.13$. Thus, no addition chain for e can be shorter than $\log_2 e + \log_2 H(e) - 2.13$.

The previously given algorithms for computing M^e are all *heuristics* for generating short addition chains. We call these algorithms heuristics because they do not guarantee minimality. Statistical methods, such as simulated annealing, can be used to produce short addition chains for certain exponents. Certain heuristics for obtaining short addition chains are discussed in [4, 52].

2.9 Vectorial Addition Chains

Another related problem (which we have briefly mentioned in Section 2.5.1) is the generation of *vectorial* addition chains. A vectorial addition chain of a given vector of integer components is the list of vectors with the following properties:

- The initial vectors are the unit vectors $[1, 0, \ldots, 0], [0, 1, 0, \ldots, 0], \ldots, [0, \ldots, 0, 1]$.

- Each vector is the sum of two earlier vectors.

- The last vector is equal to the given vector.

For example, given the vector $[7, 15, 23]$, we obtain a vectorial addition chain as

$$\begin{array}{llllllllll} [1,0,0] & & & & & & & & & \\ [0,1,0] & [0,1,1] & [1,1,1] & [0,1,2] & [1,2,3] & [1,3,5] & [2,4,6] & [3,7,11] & [4,8,12] & [7,15,23] \\ [0,0,1] & & & & & & & & & \end{array}$$

which is of length 9. Short vectorial addition chains can be used to efficiently compute M^{w_i} for several integers w_i. This problem arises in conjunction with reducing the preprocessing multiplications in adaptive m-ary methods and as well as in the sliding window technique (refer to Section 2.5). If the exponent values appear in the partitioning of the binary expansion of e are just 7, 15, and 23, then the above vectorial addition chain can be used for obtaining these exponent values. This is achieved by noting a one-to-one correspondence between the addition sequences and the vectorial addition chains. This result was established by Olivos [35] who proved that the complexity of the computation of the multinomial

$$x_1^{n_1} x_2^{n_2} \cdots x_i^{n_i}$$

is the same as the simultaneous computation of the monomials

$$x^{n_1}, x^{n_2}, \ldots, x^{n_i} \ .$$

An addition sequence is simply an addition chain where the i requested numbers $n_1, n_2, \ldots, n_i$ occur somewhere in the chain [53]. Using the Olivos algorithm, we convert the above vectorial addition chain to the addition sequence with the requested numbers 7, 15, and 23 as

$$1 \quad 2 \quad 3 \quad 4 \quad 7 \quad 8 \quad 15 \quad 23$$

which is of length 7. In general an addition sequence of length r and i requested numbers can be converted to a vectorial addition sequence of length $r + i - 1$ with dimension i.

2.10 Recoding Methods

In this section we discuss exponentiation algorithms which are intrinsically different from the ones we have so far studied. The property of these algorithms is that they require the inverse of M modulo n in order to efficiently compute $M^e \pmod n$. It is established that $k-1$ is a lower bound for the number of squaring operations required for computing M^e where k is the number of bits in e. However, it is possible to reduce the number of consequent multiplications using a *recoding* of the the exponent [17, 33, 11, 21]. The recoding techniques use the identity

$$2^{i+j-1} + 2^{i+j-2} + \cdots + 2^i = 2^{i+j} - 2^i$$

to collapse a block of 1s in order to obtain a *sparse* representation of the exponent. Thus, a redundant signed-digit representation of the exponent using the digits $\{0, 1, -1\}$ will be obtained. For example, (011110) can be recoded as

$$\begin{aligned} (011110) &= 2^4 + 2^3 + 2^2 + 2^1 \\ (1000\bar{1}0) &= 2^5 - 2^1 \ . \end{aligned}$$

Once a recoding of the exponent is obtained, we can use the binary method (or, the m-ary method) to compute $M^e \pmod n$ provided that $M^{-1} \pmod n$ is supplied along with M. For example, the recoding binary method is given below:

The Recoding Binary Method
Input: M, M^{-1}, e, n.
Output: $C = M^e \bmod n$.

0. Obtain a signed digit representation f of e.
1. **if** $f_k = 1$ **then** $C := M$ **else** $C := 1$
2. **for** $i = k-1$ **downto** 0
  2a. $C := C \cdot C \pmod{n}$
  2b. **if** $f_i = 1$ **then** $C := C \cdot M \pmod{n}$
    else if $f_i = \bar{1}$ **then** $C := C \cdot M^{-1} \pmod{n}$
3. **return** C

Note that even though the number of bits of e is equal to k, the number of bits in the the recoded exponent f can be $k+1$, for example, (111) is recoded as $(100\bar{1})$. Thus, the recoding binary algorithm starts from the bit position k in order to compute $M^e \pmod{n}$ by computing $M^f \pmod{n}$ where f is the $(k+1)$-bit recoded exponent such that $f = e$. We give an example of exponentiation using the recoding binary method. Let $e = 119 = (1110111)$. The (nonrecoding) binary method requires $6+5 = 11$ multiplications in order to compute $M^{119} \pmod{n}$. In the recoding binary method, we first obtain a sparse signed-digit representation of 119. We will shortly introduce techniques for obtaining such recodings. For now, it is easy to verify the following:

$$\begin{array}{rcl} \text{Exponent:} \quad 119 & = & 01110111 \ , \\ \text{Recoded Exponent:} \quad 119 & = & 1000\bar{1}00\bar{1} \ . \end{array}$$

The recoding binary method then computes $M^{119} \pmod{n}$ as follows:

f_i	Step 2a	Step 2b
1	M	M
0	$(M)^2 = M^2$	M^2
0	$(M^2)^2 = M^4$	M^4
0	$(M^4)^2 = M^8$	M^8
$\bar{1}$	$(M^8)^2 = M^{16}$	$M^{16} \cdot M^{-1} = M^{15}$
0	$(M^{15})^2 = M^{30}$	M^{30}
0	$(M^{30})^2 = M^{60}$	M^{60}
$\bar{1}$	$(M^{60})^2 = M^{120}$	$M^{120} \cdot M^{-1} = M^{119}$

The number of squarings plus multiplications is equal to $7+2 = 9$ which is 2 less than that of the binary method. The number of squaring operations required by the recoding binary method can be at most 1 more than that of the binary method. The number of subsequent multiplications, on the other hand, can be significantly less. This is simply equal to the number of nonzero digits of the recoded exponent. In the following we describe algorithms for obtaining a sparse signed-digit exponent. These algorithms have been used to obtain efficient multiplication algorithms. It is well-known that the shift-add type of multiplication algorithms perform a shift operation for every bit of the multiplier; an addition is performed if the current bit of the multiplier is equal to 1, otherwise, no operation is performed, and the algorithm proceeds to the next bit. Thus, the number of addition operations can be reduced if we obtain a sparse signed-digit representation of the multiplier. We perform no operation if the current multiplier bit is equal to 0, an addition if it is equal to 1, and a subtraction if the current bit is equal to $\bar{1}$. These techniques are applicable to exponentiation, where we replace addition by multiplication and subtraction by division, or multiplication with the inverse.

2.10.1 The Booth Algorithm and Modified Schemes

The Booth algorithm [3] scans the bits of the binary number $e = (e_{k-1}e_{k-2}\cdots e_1e_0)$ from right to left, and obtains the digits of the recoded number f using the following truth table:

e_i	e_{i-1}	f_i
0	0	0
0	1	1
1	0	$\bar{1}$
1	1	0

To obtain f_0, we take $e_{-1} = 0$. For example, the recoding of $e = (111001111)$ is obtained as

$$\begin{array}{r} 111001111 \\ 100\bar{1}01000\bar{1} \end{array}$$

which is more sparse than the ordinary exponent. However, the Booth algorithm has a shortcoming: The repeated sequences of (01) are recoded as repated sequences of $(1\bar{1})$. Thus, the resulting number may be much less sparse. The worst case occurs for a number of the form $e = (101010101)$, giving

$$\begin{array}{r} 101010101 \\ 1\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1} \end{array}$$

We are much better off not recoding this exponent. Another problem, which is related to this one, with the Booth algorithm is that when two trails of ones are separated by a zero, the Booth algorithm does not combine them even though they can be combined. For example, the number $e = (11101111)$ is recoded as

$$\begin{array}{r} 11101111 \\ 100\bar{1}1000\bar{1} \end{array}$$

even though a more sparse recoding exists:

$$\begin{array}{r} 100\bar{1}1000\bar{1} \\ 1000\bar{1}000\bar{1} \end{array}$$

since $(\bar{1}1) = -2 + 1 = -1 = (0\bar{1})$. In order to circumvent these shortcomings of the Booth algorithm, several modifications have been proposed [51, 16, 29]. These algorithms scan several bits at a time, and attempt to avoid introducing unnecessary nonzero digits to the recoded number. All of these algorithms which are designed for multiplication are applicable for exponentiation. Running time analyses of some of these modified Booth algorithms in the context of modular exponentiation have been performed [33, 21]. For example, the modified Booth scheme given in [21] scans the bits of the exponent four bits at a time sharing one bit with the previous and two bits with the next case:

e_{i+1}	e_i	e_{i-1}	e_{i-2}	f_i	e_{i+1}	e_i	e_{i-1}	e_{i-2}	f_i
0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0	1	0
0	0	1	0	0	1	0	1	0	0
0	0	1	1	1	1	0	1	1	1
0	1	0	0	1	1	1	0	0	$\bar{1}$
0	1	0	1	1	1	1	0	1	$\bar{1}$
0	1	1	0	0	1	1	1	0	0
0	1	1	1	0	1	1	1	1	0

This technique recodes the number in such a way that the isolated 1s stay untouched. Also 0110 is recoded as $10\bar{1}0$ and any trail of 1s of length $i \geq 3$ is recoded as $10 \cdots 0\bar{1}$. We have shown that the binary method requires $\frac{3}{2}(k-1)$ squarings plus multiplications on the average. The recoding binary method requires significantly fewer multiplications, and the number of squarings is increased by at most 1. In order to count the average number of consequent multiplications, we calculate the probability of the signed-digit value being equal to nonzero, i.e., 1 or $\bar{1}$. For the above recoding scheme, an analysis has been performed in [21]. The recoding binary method using the recoding strategy given in the able requires a total of $\frac{11}{8}(k-1)$ squarings plus multiplications. The average asymptotic savings in the number of squarings plus multiplications is equal to

$$\left(\frac{3}{2} - \frac{11}{8}\right) \div \frac{3}{2} = \frac{1}{12} \approx 8.3\ \% \ .$$

The average number of multiplications plus squarings are tabulated in the following table:

k	binary	recoding
8	11	10
16	23	21
32	47	43
64	95	87
128	191	175
256	383	351
512	767	703
1024	1535	1407
2048	3071	2815

2.10.2 The Canonical Recoding Algorithm

In a signed-digit number with radix 2, three symbols $\{\bar{1}, 0, 1\}$ are allowed for the digit set, in which 1 in bit position i represents $+2^i$ and $\bar{1}$ in bit position i represents -2^i. A minimal signed-digit vector $f = (f_k f_{k-1} \cdots f_1 f_0)$ that contains no adjacent nonzero digits (i.e. $f_i f_{i-1} = 0$ for $0 < i \leq k$) is called a *canonical signed-digit vector*. If the binary expansion of E is viewed as padded with an initial zero, then it can be proved that there exists a unique canonical signed-digit vector for e [38]. The canonical recoding algorithm [38, 16, 29] computes the signed-digit number

$$f = (f_k f_{k-1} f_{k-2} \cdots f_0)$$

starting from the least significant digit. We set the auxiliary variable $c_0 = 0$ and examine the binary expansion of e two bits at a time. The canonically recoded digit f_i and the next value of the auxiliary binary variable c_{i+1} for $i = 0, 1, 2, \ldots, n$ are computed using the following truth table.

c_i	e_{i+1}	e_i	c_{i+1}	f_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	$\bar{1}$
1	0	0	0	1
1	0	1	1	0
1	1	0	1	$\bar{1}$
1	1	1	1	0

As an example, when $e = 3038$, i.e.,

$$e = (0101111011110) \quad = \quad 2^{11} + 2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 \;,$$

we compute the canonical signed-digit vector f as

$$f = (10\bar{1}0000\bar{1}000\bar{1}0) \quad = \quad 2^{12} - 2^{10} - 2^5 - 2^1 \;.$$

Note that in this example the exponent e contains 9 nonzero bits while its canonically recoded version contains only 4 nonzero digits. Consequently, the binary method requires $11 + 8 = 19$ multiplications to compute M^{3038} when applied to the binary expansion of E, but only $12 + 3 = 15$ multiplications when applied to the canonical signed-digit vector f, provided that $M^{-1} \pmod{n}$ is also supplied. The canonical signed-digit vector f is optimal in the sense that it has the minimum number of nonzero digits among all signed-digit vectors representing the same number. For example, the following signed-digit number for $e = 3038$ produced by the original Booth recoding algorithm contains 5 nonzero digits instead of 4:

$$f = (011000\bar{1}1000\bar{1}0) \quad = \quad 2^{11} + 2^{10} - 2^6 + 2^5 - 2^1 \;.$$

Certain variations of the Booth algorithm also produce recodings which are suboptimal in terms of the number of zero digits of the recoding. For example, the first of the two algorithms given in [33] replaces the occurrences of 01^a0 by $10^{a-1}\bar{1}0$, and consequently recodes

(01111011110) as $(1000\bar{1}1000\bar{1}0)$. Since $(\bar{1}1) = (0\bar{1})$, the optimal recoding is $(10000\bar{1}000\bar{1}0)$. The second algorithm in [33] recodes (01111011110) correctly but is suboptimal on binary numbers in which two trails of 1s are separated by (010). For example (0111101011110) is recoded as $(1000\bar{1}011000\bar{1}0)$, which can be made more sparse by using the identity $(\bar{1}011) = (\bar{1}0\bar{1})$. We note that Reitwiesner's canonical recoding algorithm has none of these shortcomings; the recoding f it produces is provably the optimal signed-digit number [38].

It has been observed that when the exponent is recoded using the canonical bit recoding technique then the average number of multiplications for large k can be reduced to $\frac{4}{3}k + O(1)$ provided that M^{-1} is supplied along with M. This is proved in [11] by using formal languages to model the Markovian nature of the generation of canonically recoded signed-digit numbers from binary numbers and counting the average number of nonzero bits. The average asymptotical savings in the number of squarings plus multiplications is equal to

$$\left(\frac{3}{2} - \frac{4}{3}\right) \div \frac{3}{2} = \frac{1}{9} \approx 11\ \% \ .$$

The average number of squarings plus multiplications are tabulated in the following table:

k	binary	canonical
8	11	11
16	23	22
32	47	43
64	95	86
128	191	170
256	383	342
512	767	683
1024	1535	1366
2048	3071	2731

2.10.3 The Canonical Recoding m-ary Method

The recoding binary methods can be generalized to their respective recoding m-ary counterparts. Once the digits of the exponent are recoded, we scan them more than one bit at a time. In fact, more sophisticated techniques, such as the sliding window technique can also be used to compute $M^e \pmod{n}$ once the recoding of the exponent e is obtained. Since the partitioned exponent values are allowed to be negative numbers as well, during the preprocessing step M^w for certain $w < 0$ may be computed. This is easily accomplished by computing $(M^{-1})^w \pmod{n}$ because $M^{-1} \pmod{n}$ is assumed to be supplied along with M. One hopes that these sophisticated algorithms someday will become useful. The main obstacle in using them in the RSA cryptosystem seems to be that the time required for the computation of $M^{-1} \pmod{n}$ exceeds the time gained by the use of the recoding technique.

An analysis of the canonical recoding m-ary method has been performed in [12]. It is shown that the average number of squarings plus multiplications for the recoding binary

($d=1$), the recoding quaternary ($d=2$), and the recoding octal ($d=3$) methods are equal to

$$T_r(k,1)=\frac{4}{3}k-\frac{4}{3}\ , \quad T_r(k,2)=\frac{4}{3}k-\frac{2}{3}\ , \quad T_r(k,3)=\frac{23}{18}k+\frac{75}{18}\ ,$$

respectively. In comparison, the standard binary, quaternary, and octal methods respectively require

$$T_s(k,1)=\frac{3}{2}k-\frac{3}{2}\ , \quad T_s(k,2)=\frac{11}{8}k-\frac{3}{4}\ , \quad T_s(k,3)=\frac{31}{24}k-\frac{17}{8}$$

multiplications in the average. Furthermore, the average number of squarings plus multiplications for the canonical recoding m-ary method for $m=2^d$ is equal to

$$T_r(k,d)=k-d+\left(1-\frac{1}{3\ 2^{d-2}}\right)\left(\frac{k}{d}-1\right)+\frac{1}{3}\left[2^{d+2}+(-1)^{d+1}\right]-3\ .$$

For large k and fixed d, the behavior of $T_r(k,d)$ and $T_s(k,d)$ of the standard m-ary method is governed by the coefficient of k. In the following table we compare the values $T_r(k,d)/k$ and $T_s(k,d)/k$ for large k.

$d=\log_2 m$	1	2	3	4	5	6	7	8
$T_s(k,d)/k$	1.5000	1.3750	1.2917	1.2344	1.1938	1.1641	1.1417	1.1245
$T_r(k,d)/k$	1.3333	1.3333	1.2778	1.2292	1.1917	1.1632	1.1414	1.1244

We can compute directly from the expressions that for constant d

$$\lim_{k\to\infty}\frac{T_r(k,d)}{T_s(k,d)} \ = \ \frac{(d+1)2^d-\frac{4}{3}}{(d+1)2^d-1} \ < \ 1\ .$$

However, it is interesting to note that if we consider the *optimal* values d_s and d_r of d (which depend on k) which minimize the average number of multiplications required by the standard and the recoded m-ary methods, respectively, then

$$\frac{T_r(k,d_r)}{T_s(k,d_s)} \ > \ 1$$

for large k. It is shown in [12] that

$$\frac{T_r(k,d_r)}{T_s(k,d_s)}\approx\frac{1+\frac{1}{d_r}}{1+\frac{1}{d_s}}$$

for large k, which implies $T_r(k,d_r)>T_s(k,d_s)$. Exact values of d_s and d_r for a given k can be obtained by enumeration. These optimal values of d_s and d_r are given in the following table together with the corresponding values of T_s and T_r for each $k=128,256,\ldots,2048$.

k	d_s	$T_s(k, d_s)$	d_r	$T_r(k, d_r)$
128	4	168	3	168
256	4	326	4	328
512	5	636	4	643
1024	5	1247	5	1255
2048	6	2440	6	2458

In the following figure, we plot the average number of multiplications required by the standard and canonical recoding m-ary methods as a function of d and k.

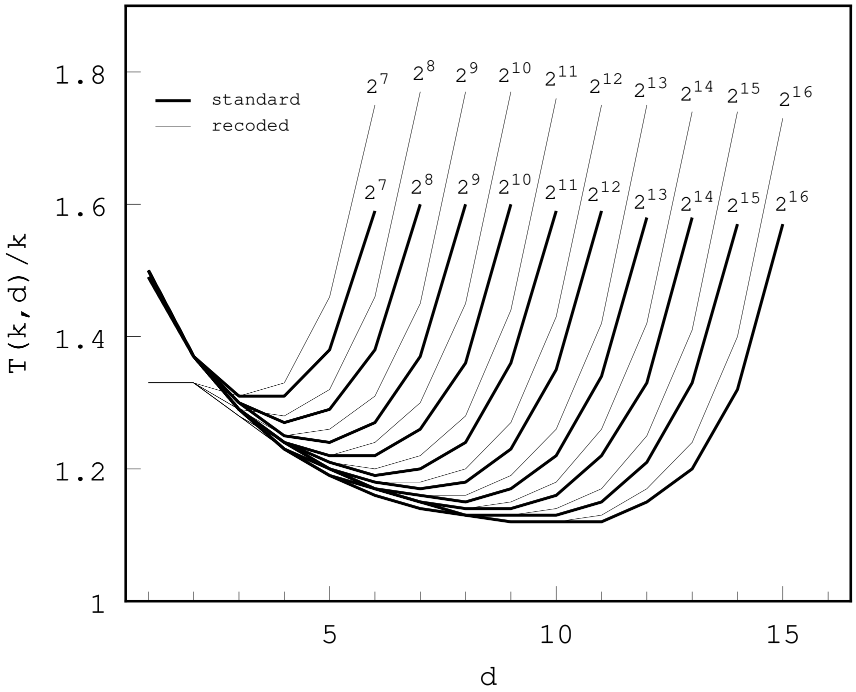

Figure 2.3: The standard versus recoding m-ary methods.

This figure and the previous analysis suggest that the recoding m-ary method may not be as useful as the straightforward m-ary method. A discussion of the usefulness of the recoding exponentiation techniques is found in the following section.

2.10.4 Recoding Methods for Cryptographic Algorithms

The recoding exponentiation methods can perhaps be useful if M^{-1} can be supplied without too much extra cost. Even though the inverse $M^{-1} \pmod{n}$ can easily be computed using the extended Euclidean algorithm, the cost of this computation far exceeds the time gained by the use of the recoding technique in exponentiation. Thus, at this time the recoding techniques do not seem to be particularly applicable to the RSA cryptosystem. In some contexts, where the plaintext M as well as its inverse M^{-1} modulo n are available for some reason, these algorithms can be quite useful since they offer significant savings in terms of the number multiplications, especially in the binary case. For example, the recoding binary method requires $1.33k$ multiplications while the nonrecoding binary method requires $1.5k$ multiplications. Also, Kaliski [18] has recently shown that if one computes the *Montgomery inverse* instead of the inverse, certain savings can be achieved by making use of the right-shifting binary algorithm. Thus, Kaliski's approach can be utilized for fast computation of the inverse, which opens up new avenues in speeding modular exponentiation computations using the recoding techniques.

On the other hand, the recoding techniques are shown to be useful for computations on elliptic curves over finite fields since in this case the inverse is available at no additional cost [33, 20]. In this context, one computes $e \cdot M$ where e is a large integer and M is a point on the elliptic curve. The multiplication operator is determined by the group law of the elliptic curve. An algorithm for computing M^e is easily converted to an algorithm for computing $e \cdot M$, where we replace multiplication by addition and division (multiplication with the inverse) by subtraction.

Chapter 3

Modular Multiplication

The modular exponentiation algorithms perform modular squaring and multiplication operations at each step of the exponentiation. In order to compute $M^e \pmod{n}$ we need to implement a modular multiplication routine. In this section we will study algorithms for computing

$$R \;:=\; a \cdot b \pmod{n} \;,$$

where a, b, and n are k-bit integers. Since k is often more than 256, we need to build data structures in order to deal with these large numbers. Assuming the word-size of the computer is w (usually $w = 16$ or 32), we break the k-bit number into s words such that $(s-1)w < k \leq sw$. The temporary results may take longer than s words, and thus, they need to be accommodated as well.

3.1 Modular Multiplication

In this report, we consider the following three methods for computing of $R = a \cdot b \pmod{n}$.

- Multiply and then Reduce:

  First Multiply $t := a \cdot b$. Here t is a $2k$-bit or $2s$-word number.

  Then Reduce: $R := t \bmod n$. The result u is a k-bit or s-word number.

  The reduction is accomplished by dividing t by n, however, we are not interested in the quotient; we only need the remainder. The steps of the division algorithm can be somewhat simplified in order to speed up the process.

- Blakley's method:

  The multiplication steps are interleaved with the reduction steps.

- Montgomery's method:

  This algorithm rearranges the residue class modulo n, and uses modulo 2^j arithmetic.

3.2 Standard Multiplication Algorithm

Let a and b be two s-digit (s-word) numbers expressed in radix W as:

$$\begin{aligned} a &= (a_{s-1}a_{s-2}\cdots a_0) = \sum_{j=0}^{s-1} a_i W^i \ , \\ b &= (b_{s-1}b_{s-2}\cdots b_0) = \sum_{j=0}^{s-1} b_i W^i \ , \end{aligned}$$

where the digits of a and b are in the range $[0, W-1]$. In general W can be any positive number. For computer implementations, we often select $W = 2^w$ where w is the word-size of the computer, e.g., $w = 32$. The standard (pencil-and-paper) algorithm for multiplying a and b produces the partial products by multiplying a digit of the multiplier (b) by the entire number a, and then summing these partial products to obtain the final number $2s$-word number t. Let t_{ij} denote the (Carry,Sum) pair produced from the product $a_i \cdot b_j$. For example, when $W = 10$, and $a_i = 7$ and $b_j = 8$, then $t_{ij} = (5, 6)$. The t_{ij} pairs can be arranged in a table as

$$\begin{array}{cccccccc} & & & & a_3 & a_2 & a_1 & a_0 \\ \times & & & & b_3 & b_2 & b_1 & b_0 \\ \hline & & & & t_{03} & t_{02} & t_{01} & t_{00} \\ & & & t_{13} & t_{12} & t_{11} & t_{10} & \\ & & t_{23} & t_{22} & t_{21} & t_{20} & & \\ + & t_{33} & t_{32} & t_{31} & t_{30} & & & \\ \hline t_7 & t_6 & t_5 & t_4 & t_3 & t_2 & t_1 & t_0 \end{array}$$

The last row denotes the total sum of the partial products, and represents the product as an $2s$-word number. The standard algorithm for multiplication essentially performs the above digit-by-digit multiplications and additions. In order to save space, a single partial product variable t is being used. The initial value of the partial product is equal to zero; we then take a digit of b and multiply by the entire number a, and add it to the partial product t. The partial product variable t contains the final product $a \cdot b$ at the end of the computation. The standard algorithm for computing the product $a \cdot b$ is given below:

The Standard Multiplication Algorithm
Input: a, b
Output: $t = a \cdot b$
0. Initially $t_i := 0$ for all $i = 0, 1, \ldots, 2s-1$.
1. **for** $i = 0$ **to** $s - 1$
2. $\quad C := 0$
3. $\quad$ **for** $j = 0$ **to** $s - 1$
4. $\quad\quad (C, S) := t_{i+j} + a_j \cdot b_i + C$
5. $\quad\quad t_{i+j} := S$
6. $\quad t_{i+s} := C$
7. **return** $(t_{2s-1}t_{2s-2}\cdots t_0)$

In the following, we show the steps of the computation of $a \cdot b = 348 \cdot 857$ using the standard algorithm.

i	j	Step	(C,S)	Partial t
0	0	$t_0 + a_0 b_0 + C$	$(0,*)$	000000
		$0 + 8 \cdot 7 + 0$	$(5,6)$	00000**6**
	1	$t_1 + a_1 b_0 + C$		
		$0 + 4 \cdot 7 + 5$	$(3,3)$	0000**3**6
	2	$t_2 + a_2 b_0 + C$		
		$0 + 3 \cdot 7 + 3$	$(2,4)$	000**4**36
				00**2**436
1	0	$t_1 + a_0 b_1 + C$	$(0,*)$	
		$3 + 8 \cdot 5 + 0$	$(4,3)$	0024**3**6
	1	$t_2 + a_1 b_1 + C$		
		$4 + 4 \cdot 5 + 4$	$(2,8)$	002**8**36
	2	$t_3 + a_2 b_1 + C$		
		$2 + 3 \cdot 5 + 2$	$(1,9)$	00**9**836
				0**1**9836
2	0	$t_2 + a_0 b_2 + C$	$(0,*)$	
		$8 + 8 \cdot 8 + 0$	$(7,2)$	019**2**36
	1	$t_3 + a_1 b_2 + C$		
		$9 + 4 \cdot 8 + 7$	$(4,8)$	01**8**236
	2	$t_4 + a_2 b_2 + C$		
		$1 + 3 \cdot 8 + 4$	$(2,9)$	0**9**8236
				298236

In order to implement this algorithm, we need to be able to execute Step 4:

$$(C,S) := t_{i+j} + a_j \cdot b_i + C \ ,$$

where the variables t_{i+j}, a_j, b_i, C, and S each hold a single-word, or a W-bit number. This step is termed as an inner-product operation which is common in many of the arithmetic and number-theoretic calculations. The inner-product operation above requires that we multiply two W-bit numbers and add this product to previous 'carry' which is also a W-bit number and then add this result to the running partial product word t_{i+j}. From these three operations we obtain a $2W$-bit number since the maximum value is

$$2^W - 1 + (2^W - 1)(2^W - 1) + 2^W - 1 = 2^{2W} - 1 \ .$$

Also, since the inner-product step is within the innermost loop, it needs to run as fast as possible. Of course, the best thing is to have a single microprocessor instruction for this computation; unfortunately, none of the currently available microprocessors and signal processors offers such a luxury. A brief inspection of the steps of this algorithm reveals that the total number of inner-product steps is equal to s^2. Since $s = k/w$ and w is a constant on a given computer, the standard multiplication algorithm requires $O(k^2)$ bit operations in order to multiply two k-bit numbers. This algorithm is asymptotically slower than the Karatsuba algorithm and the FFT-based algorithm which are to be studied next. However, it is simpler to implement and, for small numbers, gives better performance than these asymptotically faster algorithms.

3.3 Karatsuba-Ofman Algorithm

We now describe a recursive algorithm which requires asymptotically fewer than $O(k^2)$ bit operations to multiply two k-bit numbers. The algorithm was introduced by two Russian mathematicians Karatsuba and Ofman in 1962. The details of the Karatsuba-Ofman algorithm can be found in Knuth's book [19]. The following is a brief explanation of the algorithm. First, decompose a and b into two equal-size parts:

$$\begin{aligned} a &:= 2^h a_1 + a_0 \ , \\ b &:= 2^h b_1 + b_0 \ , \end{aligned}$$

i.e., a_1 is higher order h bits of a and a_0 is the lower h bits of a, assuming k is even and $2h = k$. Since we will be worried only about the asymptotics of the algorithm, let us assume that k is a power of 2. The algorithm breaks the multiplication of a and b into multiplication of the parts a_0, a_1, b_0, and b_1. Since

$$\begin{aligned} t &:= a \cdot b \\ &:= (2^h a_1 + a_0)(2^h b_1 + b_0) \\ &:= 2^{2h}(a_1 b_1) + 2^h(a_1 b_0 + a_0 b_1) + a_0 b_0 \\ &:= 2^{2h} t_2 + 2^h t_1 + t_0 \ , \end{aligned}$$

the multiplication of two $2h$-bit numbers seems to require the multiplication of four h-bit numbers. This formulation yields a recursive algorithm which we will call the standard recursive multiplication algorithm (SRMA).

function SRMA(a, b)
$t_0 :=$ SRMA(a_0, b_0)
$t_2 :=$ SRMA(a_1, b_1)
$u_0 :=$ SRMA(a_0, b_1)
$u_1 :=$ SRMA(a_1, b_0)
$t_1 := u_0 + u_1$
return $(2^{2h} t_2 + 2^h t_1 + t_0)$

Let $T(k)$ denote the number of bit operations required to multiply two k-bit numbers. Then the standard recursive multiplication algorithm implies that

$$T(k) = 4T(\frac{k}{2}) + \alpha k \ ,$$

where αk denotes the number of bit operations required to compute the addition and shift operations in the above algorithm (α is a constant). Solving this recursion with the initial condition $T(1) = 1$, we find that the standard recursive multiplication algorithm requires $O(k^2)$ bit operations to multiply two k-bit numbers.

The Karatsuba-Ofman algorithm is based on the following observation that, in fact, three half-size multiplications suffice to achieve the same purpose:

$$\begin{aligned} t_0 &:= a_0 \cdot b_0 \ , \\ t_2 &:= a_1 \cdot b_1 \ , \\ t_1 &:= (a_0 + a_1) \cdot (b_0 + b_1) - t_0 - t_2 = a_0 \cdot b_1 + a_1 \cdot b_0 \ . \end{aligned}$$

This yields the Karatsuba-Ofman recursive multiplication algorithm (KORMA) which is illustrated below:

function KORMA(a, b)
$t_0 :=$ KORMA(a_0, b_0)
$t_2 :=$ KORMA(a_1, b_1)
$u_0 :=$ KORMA$(a_1 + a_0, b_1 + b_0)$
$t_1 := u_0 - t_0 - t_2$
return $(2^{2h} t_2 + 2^h t_1 + t_0)$

Let $T(k)$ denote the number of bit operations required to multiply two k-bit numbers using the Karatsuba-Ofman algorithm. Then,

$$T(k) = 2T(\frac{k}{2}) + T(\frac{k}{2} + 1) + \beta k \approx 3T(\frac{k}{2}) + \beta k \ .$$

Similarly, βk represents the contribution of the addition, subtraction, and shift operations required in the recursive Karatsuba-Ofman algorithm. Using the initial condition $T(1) = 1$, we solve this recursion and obtain that the Karatsuba-Ofman algorithm requires

$$O(k^{\log_2 3}) = O(k^{1.58})$$

bit operations in order to multiply two k-bit numbers. Thus, the Karatsuba-Ofman algorithm is asymptotically faster than the standard (recursive as well as nonrecursive) algorithm which requires $O(k^2)$ bit operations. However, due to the recursive nature of the algorithm, there is some overhead involved. For this reason, Karatsuba-Ofman algorithm starts paying off as k gets larger. Current implementations indicate that after about $k = 250$, it starts being faster than the standard nonrecursive multiplication algorithm. Also note that since $a_0 + a_1$ is one bit larger, thus, some implementation difficulties may arise. However, we also have the option of stopping at any point during the recursion. For example, we may apply one level of recursion and then compute the required three multiplications using the standard nonrecursive multiplication algorithm.

3.4 FFT-based Multiplication Algorithm

The fastest multiplication algorithms use the fast Fourier transform. Although the fast Fourier transform was originally developed for convolution of sequences, which amounts to multiplication of polynomials, it can also be used for multiplication of long integers. In the standard algorithm, the integers are represented by the familiar positional notation. This is equivalent to polynomials to be evaluated at the radix; for example, $348 = 3x^2 + 4x + 8$ at $x = 10$. Similarly, $857 = 8x^2 + 5x + 7$ at $x = 10$. In order to multiply 348 by 857, we can first multiply the polynomials

$$(3x^2 + 4x + 8)(8x^2 + 5x + 7) = 24x^4 + 47x^3 + 105x^2 + 68x + 56 \ ,$$

then evaluate the resulting polynomial

$$24(10)^4 + 47(10)^3 + 105(10)^2 + 68(10) + 56 = 298236$$

at 10 to obtain the product $348 \cdot 857 = 298236$. Therefore, if we can multiply polynomials quickly, then we can multiply large integers quickly. In order to multiply two polynomials, we utilize the discrete Fourier transform. This is achieved by evaluating these polynomials at the roots of unity, then multiplying these values pointwise, and finally interpolating these values to obtain the coefficients of the product polynomial. The fast Fourier transform algorithm allows us to evaluate a given polynomial of degree $s-1$ at the s roots of unity using $O(s \log s)$ arithmetic operations. Similarly, the interpolation step is performed in $O(s \log s)$ time.

A polynomial is determined by its coefficients. Moreover, there exists a unique polynomial of degree $s - 1$ which 'visits' s points on the plane provided that the axes of these points are distinct. These s pairs of points can also be used to uniquely represent the polynomial of degree $s - 1$. Let $A(x)$ be a polynomial of degree $l - 1$, i.e.,

$$A(x) \ = \ \sum_{i=0}^{l-1} A_i x^i \ .$$

Also, let ω be the primitive lth root of unity. Then the fast Fourier transform algorithm can be used to evaluate this polynomial at $\{1, \omega, \omega^2, \ldots, \omega^{l-1}\}$ using $O(l \log l)$ arithmetic operations [31]. In other words, the fast Fourier transform algorithm computes the matrix vector product

$$\begin{bmatrix} A(1) \\ A(\omega) \\ \vdots \\ A(\omega^{l-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{l-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{l-1} & \cdots & \omega^{(l-1)(l-1)} \end{bmatrix} \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{l-1} \end{bmatrix} ,$$

in order to obtain the polynomial values $A(\omega^i)$ for $i = 0, 1, \ldots, l-1$. These polynomial values also uniquely define the polynomial $A(x)$. Given these polynomial values, the coefficients A_i

for $i = 0, 1, \ldots, l-1$ can be obtained by the use of the 'inverse' Fourier transform:

$$\begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{l-1} \end{bmatrix} = l^{-1} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \cdots & \omega^{-(l-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(l-1)} & \cdots & \omega^{-(l-1)(l-1)} \end{bmatrix} \begin{bmatrix} A(1) \\ A(\omega) \\ \vdots \\ A(\omega^{l-1}) \end{bmatrix} ,$$

where l^{-1} and ω^{-1} are the inverses of l and ω, respectively. The polynomial multiplication algorithm utilizes these subroutines. Let the polynomials $a(x)$ and $b(x)$

$$a(x) = \sum_{i=0}^{s-1} a_i x^i \ , \quad b(x) = \sum_{i=0}^{s-1} b_i x^i$$

denote the multiprecision numbers $a = (a_{s-1}a_{s-2}\cdots a_0)$ $b = (b_{s-1}b_{s-2}\cdots b_0)$ represented in radix W where a_i and b_i are the 'digits' with the property $0 \le a_i, b_i \le W-1$. Let the integer $l = 2s$ be a power of 2. Given the primitive lth root of unity ω, the following algorithm computes the product $t = (t_{l-1}t_{l-2}\cdots t_0)$.

FFT-based Integer Multiplication Algorithm

Step 1. Evaluate $a(\omega^i)$ and $b(\omega^i)$ for $i = 0, 1, \ldots, l-1$ by calling the fast Fourier transform procedure.

Step 2. Multiply pointwise to obtain

$$\{a(1)b(1), a(\omega)b(\omega), \ldots, a(\omega^{l-1})b(\omega^{l-1})\} \ .$$

Step 3. Interpolate $t(x) = \sum_{i=0}^{l-1} t_i x^i$ by evaluating

$$l^{-1} \sum_{i=0}^{l-1} a(\omega^i) b(\omega^i) x^i$$

on $\{1, \omega^{-1}, \ldots, \omega^{-(l-1)}\}$ using the fast Fourier transform procedure.

Step 4. Return the coefficients $(t_{l-1}, t_{l-2}, \ldots, t_0)$.

The above fast integer multiplication algorithm works over an arbitrary field in which l^{-1} and a primitive lth root of unity exist. Here, the most important question is which field to use. The fast Fourier transform was originally developed for the field of complex numbers in which the familiar lth root of unity $e^{2\pi j/l}$ makes this field the natural choice (here, $j = \sqrt{-1}$). However, there are computational difficulties in the use of complex numbers. Since computers can only perform finite precision arithmetic, we may not be able perform arithmetic with quantities such as $e^{2\pi j/l}$ because these numbers may be irrational.

In 1971, Pollard [36] showed that any field can be used provided that l^{-1} and a primitive lth root of unity are available. We are especially interested in finite fields, since our computers perform finite precision arithmetic. The field of choice is the Galois field of p elements where p is a prime and l divides $p-1$. This is due to the theorem which states that if p be prime and l divides $p-1$, then l^{-1} is in $GF(p)$ and $GF(p)$ has a primitive lth root of unity. Fortunately, such primes p are not hard to find. Primes of the form $2^r s+1$, where s is odd, have been listed in books, e.g, in [39]. Their primitive roots are readily located by successively testing. There exist an abundance of primes in the arithmetic progression $2^r s+1$, and primitive roots make up more than 3 out of every π^2 elements in the range from 2 to $p-1$ [31, 7]. For example, there are approximately 180 primes $p = 2^r s + 1 < 2^{31}$ with $r \geq 20$. Any such prime can be used to compute the fast Fourier transform of size 2^{20} [31]. Their primitive roots may also be found in a reasonable amount of time. The following list are the 10 largest primes of the form $p = 2^r s + 1 \leq 2^{31} - 1$ with $r > 20$ and their least primitive roots α.

p	r	α
2130706433	24	3
2114977793	20	3
2113929217	25	5
2099249153	21	3
2095054849	21	11
2088763393	23	5
2077229057	20	3
2070937601	20	6
2047868929	20	13
2035286017	20	10

The primite lth root of unity can easily be computed from α using $\alpha^{(p-1)/l}$. Thus, mod p FFT computations are viable. There are many Fourier primes, i.e., primes p for which FFTs in modulo p arithmetic exist. Moreover, there exists a reasonably efficient algorithm for determining such primes along with their primitive elements [31]. From these primitive elements, the required primitive roots of unity can be efficiently computed. This method for multiplication of long integers using the fast Fourier transform over finite fields was discovered by Schönhage and Strassen [45]. It is described in detail by Knuth [19]. A careful analysis of the algorithm shows that the product of two k-bit numbers can be performed using $O(k \log k \log\log k)$ bit operations. However, the constant in front of the order function is high. The break-even point is much higher than that of Karatsuba-Ofman algorithm. It starts paying off for numbers with several thousand bits. Thus, they are not very suitable for performing RSA operations.

3.5 Squaring is Easier

Squaring is an easier operation than multiplication since half of the single-precision multiplications can be skipped. This is due to the fact that $t_{ij} = a_i \cdot a_j = t_{ji}$.

$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
\times & & & & a_3 & a_2 & a_1 & a_0 \\
\hline
 & & & & t_{03} & t_{02} & t_{01} & t_{00} \\
 & & & t_{13} & t_{12} & t_{11} & t_{01} & \\
 & & t_{23} & t_{22} & t_{12} & t_{02} & & \\
+ & t_{33} & t_{23} & t_{13} & t_{03} & & & \\
\hline
 & & & & 2t_{03} & 2t_{02} & 2t_{01} & t_{00} \\
 & & & 2t_{13} & 2t_{12} & t_{11} & & \\
 & & 2t_{23} & t_{22} & & & & \\
+ & t_{33} & & & & & & \\
\hline
t_7 & t_6 & t_5 & t_4 & t_3 & t_2 & t_1 & t_0
\end{array}
$$

Thus, we can modify the standard multiplication procedure to take advantage of this property of the squaring operation.

The Standard Squaring Algorithm
Input: a
Output: $t = a \cdot a$
0. Initially $t_i := 0$ for all $i = 0, 1, \ldots, 2s - 1$.
1. **for** $i = 0$ **to** $s - 1$
2. $\quad (C, S) := t_{i+i} + a_i \cdot a_i$
3. $\quad$ **for** $j = i + 1$ **to** $s - 1$
4. $\qquad (C, S) := t_{i+j} + 2 \cdot a_j \cdot a_i + C$
5. $\qquad t_{i+j} := S$
6. $\quad t_{i+s} := C$
7. **return** $(t_{2s-1} t_{2s-2} \cdots t_0)$

However, we warn the reader that the carry-sum pair produced by operation

$$(C, S) := t_{i+j} + 2 \cdot a_j \cdot a_i + C$$

in Step 4 may be 1 bit longer than a single-precision number which requires w bits. Since

$$(2^w - 1) + 2(2^w - 1)(2^w - 1) + (2^w - 1) = 2^{2w+1} - 2^{w+1}$$

and

$$2^{2w} - 1 < 2^{2w+1} - 2^{w+1} < 2^{2w+1} - 1 \ ,$$

the carry-sum pair requires $2w + 1$ bits instead of $2w$ bits for its representation. Thus, we need to accommodate this 'extra' bit during the execution of the operations in Steps 4, 5, and 6. The resolution of this carry may depend on the way the carry bits are handled by the particular processor's architecture. This issue, being rather implementation-dependent, will not be discussed here.

3.6 Computation of the Remainder

The multiply-and-reduce modular multiplication algorithm first computes the product $a \cdot b$ (or, $a \cdot a$) using one of the multiplication algorithms given above. The multiplication step is then followed by a division algorithm in order to compute the remainder. However, as we have noted in Section 3.1, we are not interested in the quotient; we only need the remainder. Therefore, the steps of the division algorithm can somewhat be simplified in order to speed up the process. The reduction step can be achieved by making one of the well-known sequential division algorithms. In the following sections, we describe the restoring and the nonrestoring division algorithms for computing the remainder of t when divided by n.

Division is the most complex of the four basic arithmetic operations. First of all, it has two results: the quotient and the remainder. Given a dividend t and a divisor n, a quotient Q and a remainder R have to be calculated in order to satisfy

$$t = Q \cdot n + R \text{ with } \quad R < n \ .$$

If t and n are positive, then the quotient Q and the remainder R will be positive. The sequential division algorithm successively shifts and subtracts n from t until a remainder R with the property $0 \leq R < n$ is found. However, after a subtraction we may obtain a negative remainder. The restoring and nonrestoring algorithms take different actions when a negative remainder is obtained.

3.6.1 Restoring Division Algorithm

Let R_i be the remainder obtained during the ith step of the division algorithm. Since we are not interested in the quotient, we ignore the generation of the bits of the quotient in the following algorithm. The procedure given below first left-aligns the operands t and n. Since t is $2k$-bit number and n is a k-bit number, the left alignment implies that n is shifted k bits to the left, i.e., we start with $2^k n$. Furthermore, the initial value of R is taken to be t, i.e., $R_0 = t$. We then subtract the shifted n from t to obtain R_1; if R_1 is positive or zero, we continue to the next step. If it is negative the remainder is restored to its previous value.

The Restoring Division Algorithm
Input: t, n
Output: $R = a \bmod n$
1. $R_0 := t$
2. $n := 2^k n$
3. **for** $i = 1$ **to** k
4. $\quad R_i := R_{i-1} - n$
5. $\quad$ **if** $R_i < 0$ **then** $R_i := R_{i-1}$
6. $\quad n := n/2$
7. **return** R_k

In Step 5 of the algorithm, we check the sign of the remainder; if it is negative, the previous remainder is taken to be the new remainder, i.e., a restore operation is performed. If the remainder R_i is positive, it remains as the new remainder, i.e., we do not restore. The restoring division algorithm performs k subtractions in order to reduce the $2k$-bit number t modulo the k-bit number n. Thus, it takes much longer than the standard multiplication algorithm which requires $s = k/w$ inner-product steps, where w is the word-size of the computer.

In the following, we give an example of the restoring division algorithm for computing $3019 \bmod 53$, where $3019 = (101111001011)_2$ and $53 = (110101)_2$. The result is $51 = (110011)_2$.

R_0		101111	001011	t
n		110101		subtract
	$-$	000110		negative remainder
R_1		101111	001011	restore
$n/2$		11010	1	shift and subtract
	$+$	10100	1	positive remainder
R_2		10100	101011	not restore
$n/2$		1101	01	shift and subtract
	$+$	0111	01	positive remainder
R_3		0111	011011	not restore
$n/2$		110	101	shift and subtract
	$+$	000	110	positive remainder
R_4		000	110011	not restore
$n/2$		11	0101	shift
$n/2$		1	10101	shift
$n/2$			110101	shift and subtract
	$+$		000010	negative remainder
R_5			110011	restore
R			110011	final remainder

Also, before subtracting, we may check if the most significant bit of the remainder is 1. In this case, we perform a subtraction. If it is zero, there is no need to subtract since $n > R_i$. We shift n until it is aligned with a nonzero most significant bit of R_i. This way we are able to skip several subtract/restore cycles. In the average, $k/2$ subtractions are performed.

3.6.2 Nonrestoring Division Algorithm

The nonrestoring division algorithm allows a negative remainder. In order to correct the remainder, a subtraction or an addition is performed during the next cycle, depending on the whether the sign of the remainder is positive or negative, respectively. This is based on the following observation: Suppose $R_i = R_{i-1} - n < 0$, then the restoring algorithm assigns

$R_i := R_{i-1}$ and performs a subtraction with the shifted n, obtaining

$$R_{i+1} = R_i - n/2 = R_{i-1} - n/2 \ .$$

However, if $R_i = R_{i-1} - n < 0$, then one can instead let R_i remain negative and add the shifted n in the following cycle. Thus, one obtains

$$R_{i+1} = R_i + n/2 = (R_{i-1} - n) + n/2 = R_{i-1} - n/2 \ ,$$

which would be the same value. The steps of the nonrestoring algorithm, which implements this observation, are given below:

The Nonrestoring Division Algorithm
Input: t, n
Output: $R = t \bmod n$
1. $R_0 := t$
2. $n := 2^k n$
3. **for** $i = 1$ **to** k
4. **if** $R_{i-1} > 0$ **then** $R_i := R_{i-1} - n$
5. **else** $R_i := R_{i-1} + n$
6. $n := n/2$
7. **if** $R_k < 0$ **then** $R := R + n$
8. **return** R_k

Note that the nonrestoring division algorithm requires a final restoration cycle in which a negative remainder is corrected by adding the last value of n back to it. In the following we compute $51 = 3019 \bmod 53$ using the nonrestoring division algorithm. Since the remainder is allowed to stay negative, we use 2's complement coding to represent such numbers.

R_0	0101111	001011	t
n	0110101		subtract
R_1	1111010		negative remainder
$n/2$	011010	1	add
R_2	010100	1	positive remainder
$n/2$	01101	01	subtract
R_3	00111	01	positive remainder
$n/2$	0110	101	subtract
R_4	0000	110	positive remainder
$n/2$	011	0101	
$n/2$	01	10101	
$n/2$	0	110101	subtract
R_5	1	111110	negative remainder
n	0	110101	add (final restore)
R	0	110011	Final remainder

3.7 Blakley's Method

Blakley's method [2, 47] directly computes $a \cdot b \bmod n$ by interleaving the shift-add steps of the multiplication and the shift-subtract steps of the division. Since the division algorithm proceeds bit-by-bit, the steps of the multiplication algorithm must also follow this process. This implies that we use a bit-by-bit multiplication algorithm rather than a word-by-word multiplication algorithm which would be much quicker. However, the bit-by-bit multiplication algorithms can be made run faster by employing bit-recoding techniques. Furthermore, the m-ary segmentation of the operands and canonical recoding of the multiplier allows much faster implementations [27]. In the following we describe the steps of Blakley's algorithm. Let a_i and b_i represent the bits of the k-bit numbers a and b, respectively. Then, the product t which is a $2k$-bit number can be written as

$$t = a \cdot b = \left(\sum_{i=0}^{k-1} a_i 2^i \right) \cdot b = \sum_{i=0}^{k-1} (a_i \cdot b) 2^i \ .$$

Blakley's algorithm is based on the above formulation of the product t, however, at each step, we perform a reduction in order to make sure that the remainder is less than n. The reduction step may involve several subtractions.

The Blakley Algorithm
Input: a, b, n
Output: $R = a \cdot b \bmod n$
1. $R := 0$
2. **for** $i = 0$ **to** $k - 1$
3. $R := 2R + a_{k-1-i} \cdot b$
4. $R := R \bmod n$
5. **return** R

At Step 3, the partial remainder is shifted one bit to the right and the product $a_{k-1-i}b$ is added to the result. This is a step of the right-to-left multiplication algorithm. Let us assume that $0 \leq a, b, R \leq n-1$. Then the new R will be in the range $0 \leq R \leq 3n - 3$ since Step 3 of the algorithm implies

$$R := 2R + a_j \cdot b \leq 2(n-1) + (n-1) = 3n - 3 \ ,$$

i.e., at most 2 subtractions will be needed to bring the new R to the range $[0, n-1]$. Thus, Step 4 of the algorithm can be expanded as:

4.1 If $R \geq n$ then $R := R - n$
4.2 If $R \geq n$ then $R := R - n$

This algorithm computes the remainder R in k steps, where at each step one left shift, one addition, and at most two subtractions are performed; the operands involved in these computations are k-bit binary numbers.

3.8 Montgomery's Method

In 1985, P. L. Montgomery introduced an efficient algorithm [32] for computing $R = a \cdot b \bmod n$ where a, b, and n are k-bit binary numbers. The algorithm is particularly suitable for implementation on general-purpose computers (signal processors or microprocessors) which are capable of performing fast arithmetic modulo a power of 2. The Montgomery reduction algorithm computes the resulting k-bit number R without performing a division by the modulus n. Via an ingenious representation of the residue class modulo n, this algorithm replaces division by n operation with division by a power of 2. This operation is easily accomplished on a computer since the numbers are represented in binary form. Assuming the modulus n is a k-bit number, i.e., $2^{k-1} \leq n < 2^k$, let r be 2^k. The Montgomery reduction algorithm requires that r and n be relatively prime, i.e., $\gcd(r, n) = \gcd(2^k, n) = 1$. This requirement is satisfied if n is odd. In the following we summarize the basic idea behind the Montgomery reduction algorithm.

Given an integer $a < n$, we define its n-residue with respect to r as

$$\bar{a} = a \cdot r \bmod n \ .$$

It is straightforward to show that the set

$$\{ \ i \cdot r \bmod n \mid 0 \leq i \leq n-1 \ \}$$

is a complete residue system, i.e., it contains all numbers between 0 and $n-1$. Thus, there is a one-to-one correspondence between the numbers in the range 0 and $n-1$ and the numbers in the above set. The Montgomery reduction algorithm exploits this property by introducing a much faster multiplication routine which computes the n-residue of the product of the two integers whose n-residues are given. Given two n-residues $\bar{a}$ and $\bar{b}$, the *Montgomery product* is defined as the n-residue

$$\bar{R} = \bar{a} \cdot \bar{b} \cdot r^{-1} \bmod n$$

where r^{-1} is the inverse of r modulo n, i.e., it is the number with the property

$$r^{-1} \cdot r = 1 \bmod n \ .$$

The resulting number $\bar{R}$ is indeed the n-residue of the product

$$R = a \cdot b \bmod n$$

since

$$\begin{aligned} \bar{R} &= \bar{a} \cdot \bar{b} \cdot r^{-1} \bmod n \\ &= a \cdot r \cdot b \cdot r \cdot r^{-1} \bmod n \\ &= a \cdot b \cdot r \bmod n \ . \end{aligned}$$

In order to describe the Montgomery reduction algorithm, we need an additional quantity, n', which is the integer with the property

$$r \cdot r^{-1} - n \cdot n' = 1 \ .$$

The integers r^{-1} and n' can both be computed by the extended Euclidean algorithm [19]. The Montgomery product algorithm, which computes

$$u = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}$$

given $\bar{a}$ and $\bar{b}$, is given below:

function MonPro($\bar{a}, \bar{b}$)
Step 1. $t := \bar{a} \cdot \bar{b}$
Step 2. $m := t \cdot n' \bmod r$
Step 3. $u := (t + m \cdot n)/r$
Step 4. **if** $u \geq n$ **then return** $u - n$
else return u

The most important feature of the Montgomery product algorithm is that the operations involved are multiplications modulo r and divisions by r, both of which are intrinsically fast operations since r is a power 2. The MonPro algorithm can be used to compute the product of a and b modulo n, provided that n is odd.

function ModMul(a, b, n) { n is an odd number }
Step 1. Compute n' using the extended Euclidean algorithm.
Step 2. $\bar{a} := a \cdot r \bmod n$
Step 3. $\bar{b} := b \cdot r \bmod n$
Step 4. $\bar{x} :=$ MonPro($\bar{a}, \bar{b}$)
Step 5. $x :=$ MonPro($\bar{x}, 1$)
Step 6. **return** x

A better algorithm can be given by observing the property

$$\text{MonPro}(\bar{a}, b) = (a \cdot r) \cdot b \cdot r^{-1} = a \cdot b \pmod{n} \ ,$$

which modifies the above algorithm as

function ModMul(a, b, n) { n is an odd number }
Step 1. Compute n' using the extended Euclidean algorithm.
Step 2. $\bar{a} := a \cdot r \bmod n$
Step 3. $x :=$ MonPro($\bar{a}, b$)
Step 4. **return** x

However, the preprocessing operations, especially the computation of n', are rather time-consuming. Thus, it is not a good idea to use the Montgomery product computation algorithm when a single modular multiplication is to be performed.

3.8.1 Montgomery Exponentiation

The Montgomery product algorithm is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute a modular exponentiation, i.e., the computation of M^e mod n. Using one of the addition chain algorithms given in Chapter 2, we replace the exponentiation operation by a series of square and multiplication operations modulo n. This is where the Montgomery product operation finds its best use. In the following we summarize the modular exponentiation operation which makes use of the Montgomery product function MonPro. The exponentiation algorithm uses the binary method.

function ModExp(M, e, n) { n is an odd number }
Step 1. Compute n' using the extended Euclidean algorithm.
Step 2. $\bar{M} := M \cdot r \bmod n$
Step 3. $\bar{x} := 1 \cdot r \bmod n$
Step 4. **for** $i = k - 1$ **down to** 0 **do**
Step 5. $\quad \bar{x} := \text{MonPro}(\bar{x}, \bar{x})$
Step 6. $\quad$ **if** $e_i = 1$ **then** $\bar{x} := \text{MonPro}(\bar{M}, \bar{x})$
Step 7. $x := \text{MonPro}(\bar{x}, 1)$
Step 8. **return** x

Thus, we start with the ordinary residue M and obtain its n-residue $\bar{M}$ using a division-like operation, which can be achieved, for example, by a series of shift and subtract operations. Additionally, Steps 2 and 3 require divisions. However, once the preprocessing has been completed, the inner-loop of the binary exponentiation method uses the Montgomery product operations which performs only multiplications modulo 2^k and divisions by 2^k. When the binary method finishes, we obtain the n-residue $\bar{x}$ of the quantity $x = M^e \bmod n$. The ordinary residue number is obtained from the n-residue by executing the MonPro function with arguments $\bar{x}$ and 1. This is easily shown to be correct since

$$\bar{x} = x \cdot r \bmod n$$

immediately implies that

$$x = \bar{x} \cdot r^{-1} \bmod n \ = \ \bar{x} \cdot 1 \cdot r^{-1} \bmod n \ := \ \text{MonPro}(\bar{x}, 1) \ .$$

The resulting algorithm is quite fast as was demonstrated by many researchers and engineers who have implemented it, for example, see [10, 30]. However, this algorithm can be refined and made more efficient, particularly when the numbers involved are multi-precision integers. For example, Dussé and Kaliski [10] gave improved algorithms, including a simple and efficient method for computing n'. We wil describe these methods in Section 4.2.

3.8.2 An Example of Exponentiation

Here we show how to compute $x = 7^{10} \bmod 13$ using the Montgomery exponentiation algorithm.

- Since $n = 13$, we take $r = 2^4 = 16 > n$.

- Computation of n':

  Using the extended Euclidean algorithm, we determine that $16 \cdot 9 - 13 \cdot 11 = 1$, thus, $r^{-1} = 9$ and $n' = 11$.

- Computation of $\bar{M}$:

  Since $M = 7$, we have $\bar{M} := M \cdot r \pmod{n} = 7 \cdot 16 \pmod{13} = 8$.

- Computation of $\bar{x}$ for $x = 1$:

  We have $\bar{x} := x \cdot r \pmod{n} = 1 \cdot 16 \pmod{13} = 3$.

- Steps 5 and 6 of the ModExp routine:

e_i	Step 5	Step 6
1	$\text{MonPro}(3,3) = 3$	$\text{MonPro}(8,3) = 8$
0	$\text{MonPro}(8,8) = 4$	
1	$\text{MonPro}(4,4) = 1$	$\text{MonPro}(8,1) = 7$
0	$\text{MonPro}(7,7) = 12$	

  - Computation of $\text{MonPro}(3,3) = 3$:
    $t := 3 \cdot 3 = 9$
    $m := 9 \cdot 11 \pmod{16} = 3$
    $u := (9 + 3 \cdot 13)/16 = 48/16 = 3$

  - Computation of $\text{MonPro}(8,3) = 8$:
    $t := 8 \cdot 3 = 24$
    $m := 24 \cdot 11 \pmod{16} = 8$
    $u := (24 + 8 \cdot 13)/16 = 128/16 = 8$

  - Computation of $\text{MonPro}(8,8) = 4$:
    $t := 8 \cdot 8 = 64$
    $m := 64 \cdot 11 \pmod{16} = 0$
    $u := (64 + 0 \cdot 13)/16 = 64/16 = 4$

  - Computation of $\text{MonPro}(4,4) = 1$:
    $t := 4 \cdot 4 = 16$
    $m := 16 \cdot 11 \pmod{16} = 0$
    $u := (16 + 0 \cdot 13)/16 = 16/16 = 1$

  - Computation of $\text{MonPro}(8,1) = 7$:
    $t := 8 \cdot 1 = 8$
    $m := 8 \cdot 11 \pmod{16} = 8$
    $u := (8 + 8 \cdot 13)/16 = 112/16 = 7$

  - Computation of $\text{MonPro}(7,7) = 12$:
    $t := 7 \cdot 7 = 49$
    $m := 49 \cdot 11 \pmod{16} = 11$
    $u := (49 + 11 \cdot 13)/16 = 192/16 = 12$

- Step 7 of the ModExp routine: $x = \text{MonPro}(12,1) = 4$
  $t := 12 \cdot 1 = 12$
  $m := 12 \cdot 11 \pmod{16} = 4$
  $u := (12 + 4 \cdot 13)/16 = 64/16 = 4$

Thus, we obtain $x = 4$ as the result of the operation $7^{10} \bmod 13$.

3.8.3 The Case of Even Modulus

Since the existence of r^{-1} and n' requires that n and r be relatively prime, we cannot use the Montgomery product algorithm when this rule is not satisfied. We take $r = 2^k$ since arithmetic operations are based on binary arithmetic modulo 2^w where w is the word-size of the computer. In case of single-precision integers, we take $k = w$. However, when the numbers are large, we choose k to be an integer multiple of w. Since $r = 2^k$, the Montgomery modular exponentiation algorithm requires that

$$\gcd(r, n) = \gcd(2^k, n) = 1$$

which is satisfied if and only if n is odd. We now describe a simple technique [22] which can be used whenever one needs to compute modular exponentiation with respect to an even modulus. Let n be factored such that

$$n = q \cdot 2^j$$

where q is an odd integer. This can easily be accomplished by shifting the even number n to the right until its least-significant bit becomes one. Then, by the application of the Chinese remainder theorem, the computation of

$$x = a^e \bmod n$$

is broken into two independent parts such that

$$\begin{aligned} x_1 &= a^e \bmod q \ , \\ x_2 &= a^e \bmod 2^j \ . \end{aligned}$$

The final result x has the property

$$\begin{aligned} x &= x_1 \bmod q \ , \\ x &= x_2 \bmod 2^j \ , \end{aligned}$$

and can be found using one of the Chinese remainder algorithms: The single-radix conversion algorithm or the mixed-radix conversion algorithm [49, 19, 31]. The computation of x_1 can be performed using the ModExp algorithm since q is odd. Meanwhile the computation of x_2 can be performed even more easily since it involves arithmetic modulo 2^j. There is however some overhead involved due to the introduction of the Chinese remainder theorem. According to the mixed-radix conversion algorithm, the number whose residues are x_1 and x_2 modulo q and 2^j, respectively, is equal to

$$x = x_1 + q \cdot y$$

where

$$y = (x_2 - x_1) \cdot q^{-1} \bmod 2^j \ .$$

The inverse $q^{-1} \bmod 2^j$ exists since q is odd. It can be computed using the simple algorithm given in Section 4.2. We thus have the following algorithm:

function EvenModExp(a, e, n) { n is an even number }
1. Shift n to the right obtain the factorization $n = q \cdot 2^j$.
2. Compute $x_1 := a^e \bmod q$ using ModExp routine above.
3. Compute $x_2 := a^e \bmod 2^j$ using the binary method and modulo 2^j arithmetic.
4. Compute $q^{-1} \bmod 2^j$ and $y := (x_2 - x_1) \cdot q^{-1} \bmod 2^j$.
5. Compute $x := x_1 + q \cdot y$ and **return** x.

3.8.4 An Example of Even Modulus Case

The computation of $a^e \bmod n$ for $a = 375$, $e = 249$, and $n = 388$ is illustrated below.

Step 1. $n = 388 = (110000100)_2 = (11000001)_2 \times 2^2 = 97 \times 2^2$. Thus, $q = 97$ and $j = 2$.

Step 2. Compute $x_1 = a^e \bmod q$ by calling ModExp with parameters $a = 375$, $e = 249$, and $q = 97$. We must remark, however, that we can reduce a and e modulo q and $\phi(q)$, respectively. The latter is possible if we know the factorization of q. Such knowledge is not necessary but would further decrease the computation time of the ModExp routine. Assuming we do not know the factorization of q, we only reduce a to obtain

$$a \bmod q \quad = 375 \bmod 97 \quad = 84$$

and call the ModExp routine with parameters $(84, 249, 97)$. Since q is odd, the ModExp routine successfully computes the result as $x_1 = 78$.

Step 3. Compute $x_2 = a^e \bmod 2^j$ by calling an exponentiation routine based on the binary method and modulo 2^j arithmetic. Before calling such routine we should reduce the parameters as

$$\begin{array}{lll} a \bmod 2^j & = 375 \bmod 4 & = 3 \\ e \bmod \phi(2^j) & = 249 \bmod 2 & = 1 \end{array}$$

In this case, we are able to reduce the exponent since we know that $\phi(2^j) = 2^{j-1}$. Thus, we call the exponentiation routine with the parameters $(3, 1, 4)$. The routine computes the result as $x_2 = 3$.

Step 4. Using the extended Euclidean algorithm, compute

$$q^{-1} \bmod 2^j = 97^{-1} \bmod 4 = 1 \ .$$

Now compute

$$\begin{aligned} y &= (x_2 - x_1) \cdot q^{-1} \bmod 2^j \\ &= (3 - 78) \cdot 1 \bmod 4 \\ &= 1 \ . \end{aligned}$$

Step 5. Compute and return the final result

$$x = x_1 + q \cdot y = 78 + 97 \cdot 1 = 175 \ .$$

Chapter 4

Further Improvements and Performance Analysis

4.1 Fast Decryption using the CRT

The RSA decryption and signing operation, i.e., given C, the computation of

$$M := C^d \pmod{n} \ ,$$

can be performed faster using the Chinese remainder theorem (CRT) since the user knows the factors of the modulus: $n = p{\cdot}q$. This method was proposed by Quisquater and Couvreur [37], and is based on the Chinese remainder theorem, another number theory gem, like the binary method, coming to us from antiquity. Let p_i for $i = 1, 2, \ldots, k$ be pairwise relatively prime integers, i.e.,

$$\gcd(p_i, p_j) = 1 \text{ for } \quad i \neq j \ .$$

Given $u_i \in [0, p_i - 1]$ for $i = 1, 2, \ldots, k$, the Chinese remainder theorem states that there exists a unique integer u in the range $[0, P - 1]$ where $P = p_1 p_2 \cdots p_k$ such that

$$u = u_i \pmod{p_i} \ .$$

The Chinese remainder theorem tells us that the computation of

$$M := C^d \pmod{p \cdot q} \ ,$$

can be broken into two parts as

$$\begin{aligned} M_1 &:= C^d \pmod{p} \ , \\ M_2 &:= C^d \pmod{q} \ , \end{aligned}$$

after which the final value of M is computed (lifted) by the application of a Chinese remainder algorithm. There are two algorithms for this computation: The single-radix conversion

(SRC) algorithm and the mixed-radix conversion (MRC) algorithm. Here, we briefly describe these algorithms, details of which can be found in [14, 49, 19, 31]. Going back to the general example, we observe that the SRC or the MRC algorithm computes u given $u_1, u_2, \ldots, u_k$ and $p_1, p_2, \ldots, p_k$. The SRC algorithm computes u using the summation

$$u = \sum_{i=1}^{k} u_i c_i P_i \pmod{P} ,$$

where

$$P_i = p_1 p_2 \cdots p_{i-1} p_{i+1} \cdots p_k = \frac{P}{p_i} ,$$

and c_i is the multiplicative inverse of P_i modulo p_i, i.e.,

$$c_i P_i = 1 \pmod{p_i} .$$

Thus, applying the SRC algorithm to the RSA decryption, we first compute

$$\begin{aligned} M_1 &:= C^d \pmod{p} , \\ M_2 &:= C^d \pmod{q} , \end{aligned}$$

However, applying Fermat's theorem to the exponents, we only need to compute

$$\begin{aligned} M_1 &:= C^{d_1} \pmod{p} , \\ M_2 &:= C^{d_2} \pmod{q} , \end{aligned}$$

where

$$\begin{aligned} d_1 &:= d \bmod (p-1) , \\ d_2 &:= d \bmod (q-1) . \end{aligned}$$

This provides some savings since $d_1, d_2 < d$; in fact, the sizes of d_1 and d_2 are about half of the size of d. Proceeding with the SRC algorithm, we compute M using the sum

$$M = M_1 c_1 \frac{pq}{p} + M_2 c_2 \frac{pq}{q} \pmod{n} = M_1 c_1 q + M_2 c_2 p \pmod{n} ,$$

where $c_1 = q^{-1} \pmod{p}$ and $c_2 = p^{-1} \pmod{q}$. This gives

$$M = M_1 (q^{-1} \bmod p) q + M_2 (p^{-1} \bmod q) p \pmod{n} .$$

In order to prove this, we simply show that

$$\begin{aligned} M \pmod{p} &= M_1 \cdot 1 + 0 = M_1 , \\ M \pmod{q} &= 0 + M_2 \cdot 1 = M_2 . \end{aligned}$$

The MRC algorithm, on the other hand, computes the final number u by first computing a triangular table of values:

$$\begin{array}{lllll} u_{11} & & & & \\ u_{21} & u_{22} & & & \\ u_{31} & u_{32} & u_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ u_{k1} & u_{k2} & \cdots & \cdots & u_{k,k} \end{array}$$

where the first column of the values u_{i1} are the given values of u_i, i.e., $u_{i1} = u_i$. The values in the remaining columns are computed sequentially using the values from the previous column according to the recursion

$$u_{i,j+1} = (u_{ij} - u_{jj})c_{ji} \pmod{p_i} ,$$

where c_{ji} is the multiplicative inverse of p_j modulo p_i, i.e.,

$$c_{ji}p_j = 1 \pmod{p_i} .$$

For example, u_{32} is computed as

$$u_{32} = (u_{31} - u_{11})c_{13} \pmod{p_3} ,$$

where c_{13} is the inverse of p_1 modulo p_3. The final value of u is computed using the summation

$$u = u_{11} + u_{22}p_1 + u_{33}p_1p_2 + \cdots + u_{kk}p_1p_2 \cdots p_{k-1}$$

which does not require a final modulo P reduction. Applying the MRC algorithm to the RSA decryption, we first compute

$$\begin{array}{rcll} M_1 & := & C^{d_1} & \pmod{p} , \\ M_2 & := & C^{d_2} & \pmod{q} , \end{array}$$

where d_1 and d_2 are the same as before. The triangular table in this case is rather small, and consists of

$$\begin{array}{ll} M_{11} & \\ M_{21} & M_{22} \end{array}$$

where $M_{11} = M_1$, $M_{21} = M_2$, and

$$M_{22} = (M_{21} - M_{11})(p^{-1} \bmod q) \pmod{q} .$$

Therefore, M is computed using

$$M := M_1 + [(M_2 - M_1) \cdot (p^{-1} \bmod q) \bmod q] \cdot p .$$

This expression is correct since

$$\begin{array}{llcl} M & \pmod{p} & = & M_1 + 0 \;=\; M_1 , \\ M & \pmod{q} & = & M_1 + (M_2 - M_1) \cdot 1 \;=\; M_2 . \end{array}$$

The MRC algorithm is more advantageous than the SRC algorithm for two reasons:

- It requires a single inverse computation: $p^{-1} \bmod q$.
- It does not require the final modulo n reduction.

The inverse value ($p^{-1} \bmod q$) can be precomputed and saved. Here, we note that the order of p and q in the summation in the proposed public-key cryptography standard PKCS # 1 is the reverse of our notation. The data structure [43] holding the values of user's private key has the variables:

```
exponent1 INTEGER, -- d mod (p-1)
exponent2 INTEGER, -- d mod (q-1)
coefficient INTEGER, -- (inverse of q) mod p
```

Thus, it uses ($q^{-1} \bmod p$) instead of ($p^{-1} \bmod q$). Let M_1 and M_2 be defined as before. By reversing p, q and M_1, M_2 in the summation, we obtain

$$M := M_2 + [(M_1 - M_2) \cdot (q^{-1} \bmod p) \bmod p] \cdot q \ .$$

This summation is also correct since

$$\begin{aligned} M \pmod q &= M_2 + 0 \ = \ M_2 \ , \\ M \pmod p &= M_2 + (M_1 - M_2) \cdot 1 \ = \ M_1 \ , \end{aligned}$$

as required. Assuming p and q are $(k/2)$-bit binary numbers, and d is as large as n which is a k-bit integer, we now calculate the total number of bit operations for the RSA decryption using the MRC algorithm. Assuming d_1, d_2, ($p^{-1} \bmod q$) are precomputed, and that the exponentiation algorithm is the binary method, we calculate the required number of multiplications as

- Computation of M_1: $\frac{3}{2}(k/2)$ $(k/2)$-bit multiplications.
- Computation of M_2: $\frac{3}{2}(k/2)$ $(k/2)$-bit multiplications.
- Computation of M: One $(k/2)$-bit subtraction, two $(k/2)$-bit multiplications, and one k-bit addition.

Also assuming multiplications are of order k^2, and subtractions are of order k, we calculate the total number of bit operations as

$$2 \ \frac{3k}{4}(k/2)^2 + 2(k/2)^2 + (k/2) + k \ = \ \frac{3k^3}{8} + \frac{k^2 + 3k}{2} \ .$$

On the other hand, the algorithm without the CRT would compute $M = C^d \pmod n$ directly, using $(3/2)k$ k-bit multiplications which require $3k^3/2$ bit operations. Thus, considering the high-order terms, we conclude that the CRT based algorithm will be approximately 4 times faster.

4.2 Improving Montgomery's Method

The Montgomery method uses the Montgomery multiplication algorithm in order to compute multiplications and squarings required during the exponentiation process. One drawback of the algorithm is that it requires the computation of n' which has the property

$$r \cdot r^{-1} - n \cdot n' = 1 \ ,$$

where $r = 2^k$ and the k-bit number n is the RSA modulus. In this section, we show how to speed up the computation of n' within the MonPro routine. Our first observation is that we do not need the entire value of n'. We repeat the MonPro routine from Section 3.8 in order to explain this observation:

function MonPro$(\bar{a}, \bar{b})$
Step 1. $t := \bar{a} \cdot \bar{b}$
Step 2. $m := t \cdot n' \bmod r$
Step 3. $u := (t + m \cdot n)/r$
Step 4. **if** $u \geq n$ **then return** $u - n$
else return u

The multiplication of these multi-precision numbers are performed by breaking them into words, as shown in Section 3.2. Let w be the wordsize of the computer. Then, these large numbers can be thought of integers represented in radix $W = 2^w$. Assuming, these numbers require s words in their radix W representation, we can take $r = 2^{sw}$. The multiplication routine, then, accomplishes its task by computing a series of inner-product operations. For example, the multiplication of $\bar{a}$ and $\bar{b}$ in Step 1 is performed using:

1. **for** $i = 0$ **to** $s - 1$
2. $C := 0$
3. **for** $j = 0$ **to** $s - 1$
4. $(C, S) := t_{i+j} + \bar{a}_j \cdot \bar{b}_i + C$
5. $t_{i+j} := S$
6. $t_{i+s} := C$

When $\bar{a} = \bar{b}$, we can use the squaring algorithm given in Section 3.5. This will provide about 50 % savings in the time spent in Step 1 of the MonPro routine. The final value obtained is the $2s$-precision integer $(t_{2s-1}t_{2s-2} \cdots t_0)$. The computation of m and u in Steps 2 and 3 of the MonPro routine can be interleaved. We first take $u = t$, and then add $m \cdot n$ to it using the standard multiplication routine, and finally divide it by 2^{sw} which is accomplished using a shift operation (or, we just ignore the lower sw bits of u). Since $m = t \cdot n' \bmod r$ and the interleaving process proceeds word by word, we can use $n'_0 = n' \bmod 2^w$ instead of n'. This observation was made by Dussé and Kaliski [10], and used in their RSA implementation for the Motorola DSP 56000.

Thus, after t is computed by multiplying $\bar{a}$ and $\bar{b}$ using the above code, we proceed with the following code which updates t in order to compute $t + m \cdot n$.

7. **for** $i = 0$ **to** $s - 1$
8. $\quad C := 0$
9. $\quad m := t_i \cdot n'_0 \bmod 2^w$
10. $\quad$ **for** $j = 0$ **to** $s - 1$
11. $\quad\quad (C, S) := t_{i+j} + m \cdot n_j + C$
12. $\quad\quad t_{i+j} := S$
13. $\quad$ **for** $j = i + s$ **to** $2s - 1$
14. $\quad\quad (C, S) := t_j + C$
15. $\quad\quad t_j := S$
16. $t_{2s} := C$

In Step 9, we multiply t_i by n'_0 modulo 2^w to compute m. This value of m is then used in the inner-product step. Steps 13, 14, and 15 are needed to take of the carry propagating to the last word of t. We did not need these steps in multiplying $\bar{a}$ and $\bar{b}$ (Steps 1–6) since the initial value of t was zero. In Step 16, we save the last carry out of the operation in Step 14. Thus, the length of the variable t becomes $2s + 1$ due to this carry. After Step 16, we divide t by r, i.e., simply ignore the lower half of t. The resulting value is u which is then compared to n; if it is larger than n, we subtract n from it and return this value. These steps of the MonPro routine are given below:

17. **for** $j = 0$ **to** s
18. $\quad u_j := t_{j+s}$
19. $B = 0$
20. **for** $j = 0$ **to** s
21. $\quad (B, D) := u_j - n_j - B$
22. $\quad v_j := D$
23. **if** $B = 0$ **then return** $(v_{s-1} v_{s-2} \cdots v_0)$
$\qquad\qquad$ **else return** $(u_{s-1} u_{s-2} \cdots u_0)$

Thus, we have greatly simplified the MonPro routine by avoiding the full computation of n', and by using only single-precision multiplication to multiply t and n'. In the following, we will give an efficient algorithm for computing n'_0. However, before that, we give an example in which the computations performed in the MonPro routine are summarized. In this example, we will use decimal arithmetic for simplicity of the illustration. Let $n = 311$ and $r = 1000$. It is easy to show that the inverse of r is

$$r^{-1} = 65 \pmod{n} \ ,$$

and also that

$$n' = \frac{r \cdot r^{-1} - 1}{n} = \frac{1000 \cdot 65 - 1}{311} = 209 \ ,$$

and thus, $n'_0 = 9$. We will compute the Montgomery product of 216 and 123, which is equal to 248 since

$$\text{MonPro}(216, 123) = 216 \cdot 123 \cdot r^{-1} \ = \ 248 \pmod{n} \ .$$

The first step of the algorithm is to compute the product $216 \cdot 123$, accomplished in Steps 1–6. The initial value of t is zero, i.e., $t = 000\ 000$.

i	j	(C, S)	$t = 000\ 000$
0	0	$0 + 6 \cdot 3 + 0 = 18$	000 008
	1	$0 + 1 \cdot 3 + 1 = 04$	000 048
	2	$0 + 2 \cdot 3 + 0 = 06$	000 648
1	0	$4 + 6 \cdot 2 + 0 = 16$	000 668
	1	$6 + 1 \cdot 2 + 1 = 09$	000 968
	2	$0 + 2 \cdot 2 + 0 = 04$	004 968
2	0	$9 + 6 \cdot 1 + 0 = 15$	004 568
	1	$4 + 1 \cdot 1 + 1 = 06$	006 568
	2	$0 + 2 \cdot 1 + 0 = 02$	026 568

Then, we execute Steps 7 through 16, in order to compute $(t + m \cdot n)$ using the value of $n'_0 = 9$. The initial value of $t = 026\ 568$ comes from the previous step. Steps 7 through 16 are illustrated below:

i	$m \bmod 10$	j	(C, S)	$t = 026\ 568$
0	$8 \cdot 9 = 2$	0	$8 + 2 \cdot 1 + 0 = 10$	026 560
		1	$6 + 2 \cdot 1 + 1 = 09$	026 590
		2	$5 + 2 \cdot 3 + 0 = 11$	026 190
		3	$6 + 1 = 07$	027 190
		4	$2 + 0 = 02$	027 190
		5	$0 + 0 = 00$	027 190
1	$9 \cdot 9 = 1$	0	$9 + 1 \cdot 1 + 0 = 10$	027 100
		1	$1 + 1 \cdot 1 + 1 = 03$	027 300
		2	$7 + 1 \cdot 3 + 0 = 10$	020 300
		4	$2 + 1 = 03$	030 300
		5	$0 + 0 = 00$	030 300
2	$3 \cdot 9 = 7$	0	$3 + 7 \cdot 1 + 0 = 10$	030 000
		1	$0 + 7 \cdot 1 + 1 = 08$	038 000
		2	$3 + 7 \cdot 3 + 0 = 24$	048 000
		5	$0 + 2 = 02$	0 248 000

After Step 15, we divide t by r by shifting it s words to the right. Thus, we obtain the value of u as 248. Then, subtraction is performed to check if $u \geq n$; if it is, $u - n$ is returned as the final product value. Since in our example $248 < 311$, we return 248 as the result of the routine MonPro$(126, 123)$, which is the correct value.

As we have pointed out earlier, there is an efficient algorithm for computing the single precision integer n'_0. The computation of n'_0 can be performed by a specialized Euclidean algorithm instead of the general extended Euclidean algorithm. Since $r = 2^{sw}$ and

$$r \cdot r^{-1} - n \cdot n' = 1 \ ,$$

we take modulo 2^w of the both sides, and obtain

$$-n \cdot n' = 1 \pmod{2^w} \ ,$$

or, in other words,

$$n'_0 = -n_0^{-1} \pmod{2^w} \ ,$$

where n'_0 and n_0^{-1} are the least significant words (the least significant w bits) of n' and n^{-1}, respectively. In order to compute $-n_0^{-1} \pmod{2^w}$, we use the algorithm given below which computes $x^{-1} \pmod{2^w}$ for a given odd x.

function ModInverse$(x, 2^w)$ { x is odd }
1. $y_1 := 1$
2. **for** $i = 2$ **to** w
3. **if** $2^{i-1} < x \cdot y_{i-1} \pmod{2^i}$
then $y_i := y_{i-1} + 2^{i-1}$
else $y_i := y_{i-1}$
4. **return** y_w

The correctness of the algorithm follows from the observation that, at every step i, we have

$$x \cdot y_i = 1 \pmod{2^i} \ .$$

This algorithm is very efficient, and uses single precision addition and multiplications in order to compute x^{-1}. As an example, we compute $23^{-1} \pmod{64}$ using the above algorithm. Here we have $x = 23$, $w = 6$. The steps of the algorithm, the temporary values, and the final inverse are shown below:

i	2^i	y_{i-1}	$x \cdot y_{i-1} \pmod{2^i}$	2^{i-1}	y_i
2	4	1	$23 \cdot 1 = 3$	2	$1 + 2 = 3$
3	8	3	$23 \cdot 3 = 5$	4	$3 + 4 = 7$
4	16	7	$23 \cdot 7 = 1$	8	7
5	32	7	$23 \cdot 7 = 1$	16	7
6	64	7	$23 \cdot 7 = 33$	32	$7 + 32 = 39$

Thus, we compute $23^{-1} = 39 \pmod{64}$. This is indeed the correct value since

$$23 \cdot 39 = 14 \cdot 64 + 1 = 1 \pmod{64} \ .$$

Also, at every step i, we have $x \cdot y_i = 1 \pmod{2^i}$, as shown below:

i	$x \cdot y_i \bmod 2^i$
1	$23 \cdot 1 = 1 \bmod 2$
2	$23 \cdot 3 = 1 \bmod 4$
3	$23 \cdot 7 = 1 \bmod 8$
4	$23 \cdot 7 = 1 \bmod 16$
5	$23 \cdot 7 = 1 \bmod 32$
6	$23 \cdot 39 = 1 \bmod 64$

4.3 Performance Analysis

In this section, we give timing analyses of the RSA encryption and decryption operations. This analysis can be used to estimate the performance of the RSA encryption and decryption operations on a given computer system. The analysis is based on the following assumptions.

Algorithmic Issues:

1. The exponentiation algorithm is the binary method.
2. The Montgomery reduction algorithm is used for the modular multiplications.
3. The improvements on the Montgomery method are taken into account.

Data Size:

1. The size of n is equal to s words.
2. The sizes of p and q are $s/2$ words.
3. The sizes of M and C are s words.
4. The size of e is k_e bits.
5. The Hamming weight of e is equal to h_e, where $1 < h_e \leq k_e$.
6. The size of d is k_d bits.
7. The Hamming weight of d is equal to h_d, where $1 < h_d \leq k_d$.

Precomputed Values:

1. The private exponents d_1 and d_2 are precomputed and available.
2. The coefficient $(p^{-1} \bmod q)$ or $(q^{-1} \bmod p)$ is precomputed and available.

Computer Platform:

1. The wordsize of the computer is w bits.
2. The addition of two single-precision integers requires A cycles.
3. The multiplication of two single-precision integers requires P cycles.
4. The inner-product operation requires $2A + P$ cycles.

In the following sections, we will analyze the performance of the RSA encryption and decryptions operations separately based on the preceding assumptions.

4.3.1 RSA Encryption

The encryption operation using the Montgomery product first computes n'_0, which requires

$$\sum_{j=2}^{w}(P+A) = (w-1)(P+A) \tag{4.1}$$

cycles. It then proceeds to compute $\bar{M} = M \cdot r \pmod{n}$ and $\bar{C} = 1 \cdot r \pmod{n}$. The computation of $\bar{M}$ requires sw s-precision subtractions. The computation of $\bar{C}$, on the other hand, may require up to w s-precision subtractions. Thus, these operations together require

$$sw(sA) + w(sA) = (s^2+s)wA \tag{4.2}$$

cycles. We then start the exponentiation algorithm which requires $(k_e - 1)$ Montgomery square and $(h_e - 1)$ Montgomery product operations. The Montgomery product operation first computes the product $\bar{a} \cdot \bar{b}$ which requires

$$\sum_{i=0}^{s-1}\sum_{j=0}^{s-1}(P+2A) = s^2(P+2A)$$

cycles. Then, Steps 7 through 15 are followed, requiring

$$\sum_{i=0}^{s-1}\left[P + \sum_{j=0}^{s-1}(P+2A) + \sum_{j=i+s}^{2s-1} A\right] = sP + s^2(P+2A) + \frac{s^2+s}{2}A = (s^2+s)P + \frac{5s^2+s}{2}A$$

cycles. The s-precision subtraction operation which is performed in Steps 18–21 requires a total of s single-precision subtractions. Thus, Steps 7 through 22 require a total of

$$(s^2+s)P + \frac{5s^2+s}{2}A + sA = (s^2+s)P + \frac{5s^2+3s}{2}A$$

Thus, we calculate the total number of cycles required by the Montgomery product routine as

$$s^2(P+2A) + (s^2+s)P + \frac{5s^2+3s}{2}A = (2s^2+s)P + \frac{9s^2+3s}{2}A\ . \tag{4.3}$$

The Montgomery square routine uses the optimized squaring algorithm of Section 3.5 in order to compute $\bar{a} \cdot \bar{a}$. This step requires

$$\frac{s(s-1)}{2}(P+2A)$$

cycles. The remainder of the Montgomery square algorithm is the same as the Montgomery product algorithm. Thus, the Montgomery square routine requires a total of

$$\frac{s(s-1)}{2}(P+2A) + (s^2+s)P + \frac{5s^2+3s}{2}A = \frac{3s^2+s}{2}P + \frac{7s^2+s}{2}A \tag{4.4}$$

cycles. The total number of cycles required by the RSA encryption operation is then found by adding the number of cycles for computing n_0' given by Equation (4.1), the number of cycles required by computing $\bar{M}$ and $\bar{C}$ given by Equation (4.2), $(k_e - 1)$ times the number of cycles required by the Montgomery square operation given by Equation (4.4), and $(h_e - 1)$ times the number cycles required by the Montgomery product operation given by Equation (4.3). The total number of cycles is found as

$$\begin{aligned} T_1(s, k_e, h_e, w, P, A) &= (w-1)(P+A) + (s^2+s)wA + (k_e-1)\left[\frac{3s^2+s}{2}P + \frac{7s^2+s}{2}A\right] \\ &+ (h_e-1)\left[(2s^2+s)P + \frac{9s^2+3s}{2}A\right] \ . \end{aligned} \tag{4.5}$$

4.3.2 RSA Decryption without the CRT

The RSA decryption operation without the Chinese remainder theorem by disregarding the knowledge of the factors of the user's modulus is the same operation as the RSA encryption. Thus, the total number of cycles required by the RSA decryption operation is the same as the one given in Equation (4.5), except that k_e and h_e are replaced by k_d and h_d, respectively.

$$\begin{aligned} T_1(s, k_d, h_d, w, P, A) &= (w-1)(P+A) + (s^2+s)wA + (k_d-1)\left[\frac{3s^2+s}{2}P + \frac{7s^2+s}{2}A\right] \\ &+ (h_d-1)\left[(2s^2+s)P + \frac{9s^2+3s}{2}A\right] \ . \end{aligned} \tag{4.6}$$

4.3.3 RSA Decryption with the CRT

The RSA decryption operation using the Chinese remainder theorem first computes M_1 and M_2 using

$$\begin{aligned} M_1 &:= C^{d_1} \pmod{p} \ , \\ M_2 &:= C^{d_2} \pmod{q} \ . \end{aligned}$$

The computation of M_1 is equivalent to the RSA encryption with the exponent d_1 and modulus p. Assuming the number of words required to represent p is equal to $s/2$, we find the number of cycles required in computing M_1 as

$$T_1(\frac{s}{2}, k_{d_1}, h_{d_1}, w, P, A) \ ,$$

where k_{d_1} and h_{d_1} is the bit size and Hamming weight of d_1, respectively. Similarly the computation of M_2 requires

$$T_1(\frac{s}{2}, k_{d_2}, h_{d_2}, w, P, A)$$

cycles. Then, the mixed-radix conversion algorithm computes M using

$$M := M_1 + (M_2 - M_1) \cdot (p^{-1} \bmod q) \cdot p \ ,$$

which requires one $s/2$-precision subtraction, two s-precision multiplications, and one s-precision addition. This requires a total of

$$\frac{s}{2}A + 2s^2(P + 2A) + sA = 2s^2P + (4s^2 + \frac{3s}{2})A$$

cycles assuming the coefficient $(p^{-1} \bmod q)$ is available. Therefore, we compute the total number of cycles required by the RSA decryption operation with the CRT as

$$\begin{aligned} T_2(s, k_{d_1}, h_{d_1}, k_{d_2}, h_{d_2}, w, P, A) &= T_1(\frac{s}{2}, k_{d_1}, h_{d_1}, w, P, A) + T_1(\frac{s}{2}, k_{d_2}, h_{d_2}, w, P, A) \\ &\quad + 2s^2P + (4s^2 + \frac{3s}{2})A \ . \end{aligned} \tag{4.7}$$

4.3.4 Simplified Analysis

In this section, we will consider three cases in order to simplify the performance analysis of the RSA encryption and decryption operations.

Short Exponent RSA Encryption: We will take the public exponent as $e = 2^{16} + 1$. Thus, $k_e = 17$ and $h_e = 2$. This gives the total number of cycles as

$$\begin{aligned} T_{es}(s, w, P, A) &= \left[Aw + 26P + \frac{121A}{2}\right] s^2 + \left[Aw + 9P + \frac{19A}{2}\right] s + \\ &\quad + (w-1)(P + A) \ . \end{aligned} \tag{4.8}$$

Long Exponent RSA Encryption: We will assume that the public exponent has exactly k bits (i.e., the number of bits in n), and its Hamming weight is equal to $k/2$. Thus, $k_e = k = sw$ and $h_e = k/2 = sw/2$. This case is also equivalent to the RSA decryption without the CRT in terms of the number of cycles required to perform the operation. This gives the total number of cycles as

$$\begin{aligned} T_{el}(s, w, P, A) &= \left[\frac{5Pw}{2} + \frac{23Aw}{4}\right] s^3 + \left[Pw + \frac{9Aw}{4} - \frac{7P}{2} - 8A\right] s^2 + \\ &\quad + \left[Aw - \frac{3P}{2} - 2A\right] s + (w-1)(P + A) \ . \end{aligned} \tag{4.9}$$

RSA Decryption with CRT: The number of bits and the Hamming weights of d_1 and d_2 are assumed to be given as $k_{d_1} = k_{d_2} = k/2 = sw/2$ and $h_{d_1} = h_{d_2} = k/4 = sw/4$. Since $k_{d_1} = k_{d_2}$ and $h_{d_1} = h_{d_2}$, we have

$$T_{dl}(s, w, P, A) = 2T_1(\frac{s}{2}, \frac{sw}{2}, \frac{sw}{4}, w, P, A) + 2s^2P + (4s^2 + \frac{3s}{2})A \ .$$

Substituting $k_{d_1} = sw/2$ and $h_{d_1} = sw/4$, we obtain

$$T_{dl}(s,w,P,A) = \left[\frac{5Pw}{8} + \frac{23Aw}{16}\right]s^3 + \left[\frac{Pw}{2} + \frac{9Aw}{8} + \frac{P}{4}\right]s^2 + \\ + \left[Aw - \frac{3P}{2} - \frac{A}{2}\right]s + 2(w-1)(P+A) \ . \quad (4.10)$$

4.3.5 An Example

In a given computer implementation, the values of w, P, and A are fixed. Thus, the number of cycles required is a function of s, i.e., the word-length of the modulus. In this section, we will apply the above analysis to the Analog Devices Signal Processor ADSP 2105. This signal processor has a data path of $w = 16$ bits, and runs with a clock speed of 10 MHz. Furthermore, examining the arithmetic instructions, we have determined that the ADSP 2105 signal processor adds or multiplies two single-precision numbers in a single clock cycle. Considering the read and write times, we take $A = 3$ and $P = 3$. The simplified expressions for T_{es}, T_{el}, and T_{dl} are given below:

$$\begin{aligned} T_{es} &= \frac{615}{2}s^2 + \frac{207}{2}s + 90 \ , \\ T_{el} &= 396s^3 + \frac{243}{2}s^2 + \frac{75}{2}s + 90 \ , \\ T_{dl} &= 99s^3 + \frac{315}{4}s^2 + 42s + 180 \ . \end{aligned}$$

Using the clock cycle time of the ADSP 2105 as 100 ns, we tabulate the encryption and decryption times for the values of $k = 128, 256, 384, \ldots, 1024$, corresponding to the values of $s = 8, 16, 24, \ldots, 64$, respectively. The following table summarizes the times (in milliseconds) of the short exponent RSA encryption (T_{es}), the long exponent RSA encryption (T_{el}), and the RSA decryption with the CRT (T_{dl}).

k	T_{es}	T_{el}	T_{dl}
128	3	21	6
256	8	165	43
384	18	555	142
512	32	1,310	333
640	50	2,554	646
768	71	4,408	1,113
896	97	6,993	1,764
1024	127	10,431	2,628

Our experiments with the ADSP simulator validated these estimated values. However, we note that the values of P and A must be carefully determined for a reliable estimation of the timings of the RSA encryption and decryption operations.

Bibliography

[1] G. B. Arfken, D. F. Griffing, D. C. Kelly, and J. Priest. *University Physics*. San Diego, CA: Harcourt Brace Jovanovich Publishers, 1989.

[2] G. R. Blakley. A computer algorithm for the product AB modulo M. *IEEE Transactions on Computers*, 32(5):497–500, May 1983.

[3] A. D. Booth. A signed binary multiplication technique. *Q. J. Mech. Appl. Math.*, 4(2):236–240, 1951. (Also reprinted in [48], pp. 100–104).

[4] J. Bos and M. Coster. Addition chain heuristics. In G. Brassard, editor, *Advances in Cryptology — CRYPTO 89, Proceedings*, Lecture Notes in Computer Science, No. 435, pages 400–407. New York, NY: Springer-Verlag, 1989.

[5] E. F. Brickell. A survey of hardware implementations of RSA. In G. Brassard, editor, *Advances in Cryptology — CRYPTO 89, Proceedings*, Lecture Notes in Computer Science, No. 435, pages 368–370. New York, NY: Springer-Verlag, 1989.

[6] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT 92*, Lecture Notes in Computer Science, No. 658, pages 200–207. New York, NY: Springer-Verlag, 1992.

[7] P. Chiu. Transforms, finite fields, and fast multiplication. *Mathematics Magazine*, 63(5):330–336, December 1990.

[8] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

[9] P. Downey, B. Leony, and R. Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 3:638–696, 1981.

[10] S. R. Dussé and B. S. Kaliski, Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT 90*, Lecture Notes in Computer Science, No. 473, pages 230–244. New York, NY: Springer-Verlag, 1990.

[11] Ö. Eğecioğlu and Ç. K. Koç. Fast modular exponentiation. In E. Arıkan, editor, *Communication, Control, and Signal Processing: Proceedings of 1990 Bilkent International Conference on New Trends in Communication, Control, and Signal Processing*, pages 188–194, Bilkent University, Ankara, Turkey, July 2–5 1990. Amsterdam, Netherland: Elsevier.

[12] Ö. Eğecioğlu and Ç. K. Koç. Exponentiation using canonical recoding. *Theoretical Computer Science*, 129(2):407–417, 1994.

[13] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[14] H. L. Garner. The residue number systems. *IRE Transactions on Electronic Computers*, 8(6):140–147, June 1959.

[15] F. Hoornaert, M. Decroos, J. Vandewalle, and R. Govaerts. Fast RSA-hardware: dream or reality? In C. G. Gunther, editor, *Advances in Cryptology — EUROCRYPT 88*, Lecture Notes in Computer Science, No. 330, pages 257–264. New York, NY: Springer-Verlag, 1988.

[16] K. Hwang. *Computer Arithmetic, Principles, Architecture, and Design.* New York, NY: John Wiley & Sons, 1979.

[17] J. Jedwab and C. J. Mitchell. Minimum weight modified signed-digit representations and fast exponentiation. *Electronics Letters*, 25(17):1171–1172, 17th August 1989.

[18] B. S. Kaliski, Jr. The Z80180 and big-number arithmetic. *Dr. Dobb's Journal*, pages 50–58, September 1993.

[19] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Reading, MA: Addison-Wesley, Second edition, 1981.

[20] N. Koblitz. CM-curves with good cryptographic properties. In J. Feigenbaum, editor, *Advances in Cryptology — CRYPTO 91, Proceedings*, Lecture Notes in Computer Science, No. 576, pages 279–287. New York, NY: Springer-Verlag, 1991.

[21] Ç. K. Koç. High-radix and bit recoding techniques for modular exponentiation. *International Journal of Computer Mathematics*, 40(3+4):139–156, 1991.

[22] Ç. K. Koç. Montgomery reduction with even modulus. *IEE Proceedings: Computers and Digital Techniques*, 141(5):314–316, September 1994.

[23] Ç. K. Koç. Analysis of sliding window techniques for exponentiation. *Computers and Mathematics with Applications*, 30(10):17–24, 1995.

[24] Ç. K. Koç and C. Y. Hung. Carry save adders for computing the product AB modulo N. *Electronics Letters*, 26(13):899–900, 21st June 1990.

[25] Ç. K. Koç and C. Y. Hung. Multi-operand modulo addition using carry save adders. *Electronics Letters*, 26(6):361–363, 15th March 1990.

[26] Ç. K. Koç and C. Y. Hung. Bit-level systolic arrays for modular multiplication. *Journal of VLSI Signal Processing*, 3(3):215–223, 1991.

[27] Ç. K. Koç and C. Y. Hung. Adaptive m-ary segmentation and canonical recoding algorithms for multiplication of large binary numbers. *Computers and Mathematics with Applications*, 24(3):3–12, 1992.

[28] M. Kochanski. Developing an RSA chip. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pages 350–357. New York, NY: Springer-Verlag, 1985.

[29] I. Koren. *Computer Arithmetic Algorithms.* Englewood Cliffs, NJ: Prentice-Hall, 1993.

[30] D. Laurichesse and L. Blain. Optimized implementation of RSA cryptosystem. *Computers & Security*, 10(3):263–267, May 1991.

[31] J. D. Lipson. *Elements of Algebra and Algebraic Computing.* Reading, MA: Addison-Wesley, 1981.

[32] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[33] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. Rapport de Recherche 983, INRIA, March 1989.

[34] National Institute for Standards and Technology. Digital signature standard (DSS). *Federal Register*, 56:169, August 1991.

[35] J. Olivos. On vectorial addition chains. *Journal of Algorithms*, 2(1):13–21, March 1981.

[36] J. M. Pollard. The fast Fourier transform in a finite field. *Mathematics of Computation*, 25:365–374, 1971.

[37] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, October 1982.

[38] G. W. Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:231–308, 1960.

[39] H. Riesel. *Prime Numbers and Computer Methods for Factorization.* Boston, MA: Birkhäuser, 1985.

[40] R. L. Rivest. RSA chips (Past/Present/Future). In T. Beth, N. Cot, and I. Ingemarsson, editors, *Advances in Cryptology, Proceedings of EUROCRYPT 84*, Lecture Notes in Computer Science, No. 209, pages 159–165. New York, NY: Springer-Verlag, 1984.

[41] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[42] RSA Laboratories. Answers to Frequently Asked Questions About Today's Cryptography. RSA Data Security, Inc., October 1993.

[43] RSA Laboratories. The Public-Key Cryptography Standards (PKCS). RSA Data Security, Inc., November 1993.

[44] A. Schönhage. A lower bound for the length of addition chains. *Theoretical Computer Science*, 1:1–12, 1975.

[45] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971.

[46] H. Sedlak. The RSA cryptography processor. In D. Chaum and W. L. Price, editors, *Advances in Cryptology — EUROCRYPT 87*, Lecture Notes in Computer Science, No. 304, pages 95–105. New York, NY: Springer-Verlag, 1987.

[47] K. R. Sloan, Jr. Comments on "A computer algorithm for the product AB modulo M". *IEEE Transactions on Computers*, 34(3):290–292, March 1985.

[48] E. E. Swartzlander, editor. *Computer Arithmetic*, volume I. Los Alamitos, CA: IEEE Computer Society Press, 1990.

[49] N. S. Szabo and R. I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology.* New York, NY: McGraw-Hill, 1967.

[50] C. D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–378, March 1993.

[51] S. Waser and M. J. Flynn. *Introduction to Arithmetic for Digital System Designers.* New York, NY: Holt, Rinehart and Winston, 1982.

[52] Y. Yacobi. Exponentiating faster with addition chains. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT 90*, Lecture Notes in Computer Science, No. 473, pages 222–229. New York, NY: Springer-Verlag, 1990.

[53] A. C.-C. Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5(1):100–103, March 1976.