Redundant integer representations and fast exponentiation

Dieter Gollmann, Yongfei Han, and Chris J. Mitchell

August 25, 1995

Abstract

In this paper two modifications to the standard square and multiply method for exponentiation are discussed. The first, using a signed-digit representation of the exponent, has been examined previously by a number of authors, and we present a new precise and simple mathematical analysis of its performance. The second, a new technique, uses a different redundant representation of the exponent, which we call a *string replacement representation*; the performance of this new method is analysed and compared with previously proposed methods. The techniques considered in this paper have application in the implementation of cryptographic algorithms such as RSA, where modular exponentiations of very large integers need to be calculated.

Index Terms: RSA, modular exponentiation, signed-digit representation.

1 Introduction

Exponentiation of large integers (modulo a large integer) is the basis of several well known cryptographic algorithms such as RSA [16]. The calculations involved are complex, and can be time-consuming especially when performed in software. As a result algorithms which speed up implementations of modular exponentiation are of considerable practical significance; see, for example, Selby and Mitchell [17]. Techniques for speeding up modular exponentiation and, more specifically, modular multiplication have a distinguished history. Norris and Simmons have proposed an algorithm tailored to implementations on fast multiplication devices, which can exploit the rounding inherent to floating point arithmetics [12]. Similarly, Montgomery uses a special representation of modular equivalence classes to reduce modular arithmetics to simple truncating operations [11]. A combination of these two techniques was recently presented in [13]. Other fast multiplication algorithms are based on redundant integer representations. Signed-digit integers, which will be the starting point of our paper, go back to the early days of computer arithmetics, see e.g. [15]. The same holds for carry-save integers, which led to Brickell's delayed-carry algorithm [4].

The generally accepted method for performing modular exponentiation is the 'square-and-multiply' technique; see, for example, Beker and Piper [2], or Knuth [8]. In brief, if one is required to compute $m^e \pmod N$ and e has binary representation

$$e_{s-1}e_{s-2}\ldots e_0$$

where e_{s-1} is the most significant bit, then the 'left-to-right' version of the algorithm works as follows:

```
d := 1;
for i := s − 1 downto 0 do
    begin
        d := d^2  (mod N);
        if (e_i = 1) d := d · m  (mod N)
    end
```

The result will be contained in d.

Note that the number of modular multiplications involved in performing the algorithm is determined by the number of ones in the binary representation of e. The purpose of this paper is to consider the use of alternative representations of e, which, when used in combination with slight variants of the above algorithm, can considerably reduce the number of multiplications involved. This type of approach is not new, and has been previously considered by a number of authors, e.g. [5, 7, 9, 20].

This paper contains the following main parts. In Section 2, use of signed-digit representations of the exponent is considered, and a precise analysis of the complexity of exponentiation using such representations is given. Section 3 contains a description of a new type of redundant integer representation, called a String Representation. A variant of the square-and-multiply algorithm is also introduced, capable of taking advantage of such representations to speed exponentiation. Finally, Section 4 contains an analysis of the performance of the exponentiation algorithm introduced in Section 3, and a comparison is given with other variants of the square-and-multiply method.

2 Signed-digit representations and exponentiation

2.1 Introduction

A signed-digit representation of an integer e, as introduced by Booth [3], is given by

$$e = \sum_{i=0}^{s-1} e_i 2^i \text{ with } e_i \in \{-1, 0, 1\}.$$

Henceforth, we will write $\bar{1}$ for -1. Signed-digit integers can be used in various ways to speed up integer arithmetics. There are algorithms for carry-free addition of signed-digit integers [19] and for modular addition [18]. Standard shift-and-add algorithms for multiplication can be adapted for signed-digit integers [15]. The square-and-multiply algorithm for exponentiation can be treated in a similar way [7, 20]. As an example, consider the following exponentiation algorithm computing $d = m^e$.

```
d := 1;
for i = s − 1 downto 0 do
    begin
        d := d^2;
        d := d · m^{e_i};
    end
```

The assignment $d := d \cdot m^{e_i}$ can be implemented as a case statement. For $e_i = 1$ we have a multiplication by m, for $e_i = \bar{1}$ we have a multiplication by m^{-1} (division by m), and for $e_i = 0$ no action has to be taken.

Note that e may have more than one signed-digit representation. For example, the decimal number 15 can be represented as 1111 or as $1000\bar{1}$. If we can find a signed-digit representation of e with the minimal number of non-zeros, then the amount of multiplications will be minimal. In general, we will need less multiplications than with the standard binary representation of e.

Definition 2.1 *The* weight *of a signed-digit integer $e_{s-1}e_{s-2}\ldots e_0$ is the number of non-zero coefficients e_i. The signed-digit integer $e_{s-1}e_{s-2}\ldots e_0$ is called a* minimal *representation of the integer e, if there is no other signed-digit representation of e of lesser weight.*

Definition 2.2 *A signed-digit integer is called* sparse *if it has no adjacent non-zero coefficients.*

Lemma 2.3 *[7] Sparse signed-digit integers are minimal. A sparse signed-digit representation of e is unique.*

Various investigations on the average weight of signed-digit representations have been conducted. In [9], non-minimal representations are examined, which are derived by treating all 1-runs individually, replacing strings $01\ldots1$ by strings $10\ldots0\bar{1}$ (of the same length). This algorithm does not make use of the fact that such a replacement may create a new 1-run. The authors use the enumeration of all 1-runs in the binary numbers of a given length n to deduce the precise average weight of these representations which is approximately $\frac{3}{8}n$.

In [10], Markov chains are employed to show that the average weight of the minimal signed-digit representation of binary numbers of length n is approximately $\frac{n}{3}$. The same result is obtained in [20], using enumeration to bound the average weight of minimal signed-digit representations. Neither of the two papers gives a precise value for the average weight. Minimal signed digit representations with radix r have been examined in [1], where a Markov chain argument shows that a minimal signed digit representation of an n-digit radix r integer has approximately $\frac{2n}{r+1}$ zeros.

The precise average weight of minimal signed-digit integers of length n, as opposed to the average weight of minimal signed-digit representations of binary numbers of length n, is given in [5]. This result is derived from a recursive representation of the formal language of all minimal signed-digit numbers. In this paper, we will again use recursion to derive explicitly the average weight of the minimal signed-digit representation of binary numbers of length n.

2.2 Minimisation Algorithms

Algorithms for finding minimal signed-digit representations have been suggested by several authors [7, 15, 20]. We will give a short sketch of the algorithm given in [7], which generates the unique (minimal) sparse signed-digit representation of an integer. Starting from an n-bit binary integer $e_{n-1}\ldots e_0$, the algorithm starts at the least significant bit and searches for the next string $01\ldots1$ of length $k+1$, $k \geq 2$, and replaces this string by $10\ldots\bar{1}$. This step is then repeated until no such string can be found. This algorithm may convert the n-bit binary integer into a signed-digit integer of length $n+1$. As an example, the binary integer 11011 is converted successively into $1110\bar{1}$ and then $100\bar{1}0\bar{1}$. In the remainder of this section we will always use sparse signed-digit representations.

2.3 Main Result

Let $T(n)$ be the conversion table for binary integers of length n to sparse signed-digit representations. Let $s(n)$ be the total number of bits 'saved' by converting to a signed-digit representation. These savings can be computed recursively. We will consider the upper and lower half of the table $T(n+1)$ separately.

- The entries in the upper half give the representation of n-bit integers so the savings are the same as in $T(n)$.
- The entries in the lower half are the representations of integers $2^n + x$, where x is an n-bit integer.

Consider now the savings in the lower half. Let x denote a binary integer and y a signed-digit integer of length n. By writing $x \rightarrow 0y$ (or $x \rightarrow 1y$) we indicate that the binary number x has the minimal signed-digit representation $0y$ (or $1y$).

- For $x \rightarrow 0y$ and $y_{n-1} = 0$, we have $2^n + x \rightarrow 1y$ and the savings are the same as in $T(n)$.
- For $x \rightarrow 0y$ and $y_{n-1} = 1$, we have $2^n + x \rightarrow 10\bar{1}y_{n-2} \ldots y_0$ and the savings are the same as in $T(n)$.
- For $x \rightarrow 1y$, we have $2^n + x \rightarrow 10y$ and the savings are one more than the savings in $T(n)$.

To determine the exact number of bits saved we have to count the number of instances $x \rightarrow 1y$ in $T(n)$.

Lemma 2.4 *The largest integer x_n with $x \rightarrow 0y$ is $\frac{c}{3} \cdot (4^{\lfloor \frac{n+1}{2} \rfloor} - 1)$ where $c = 1$ for n odd and $c = 2$ for n even. The smallest integer with $x \rightarrow 1y$ is $x_n + 1$.*

Proof As we use sparse representations, the largest integer x_n with $x \rightarrow 0y$ has the $(n+1)$-digit representation $0101 \ldots .01(0)$. Thus

$$x_n = c \cdot \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor} 4^i = \frac{c}{3} \cdot (4^{\lfloor \frac{n+1}{2} \rfloor} - 1).$$

The smallest integer with $x \rightarrow 1y$ has the representation $10\bar{1}0 \ldots \bar{1}0(\bar{1})$ and the value $x_n + 1$. □

We now have

Theorem 2.5 *The savings $s(n+1)$ are given by*

$$\begin{aligned} s(n+1) &= 2 \cdot s(n) + 2^n - (x_n + 1) \\ &= 2 \cdot s(n) + 2^n - 1 - \frac{c}{3} \cdot (4^{\lfloor \frac{n+1}{2} \rfloor} - 1). \end{aligned}$$

where $c = 1$ for n odd and $c = 2$ for n even.

Proof As observed above, the savings in $T(n)$ are certain to be repeated both in the upper and lower half of $T(n+1)$. In addition, we will save one further digit in the top $2^n - (x_n + 1)$ rows of $T(n+1)$, where x_n is defined as in the proof of Lemma 2.4. We thus get

$$s(n+1) = 2 \cdot s(n) + 2^n - (x_n + 1),$$

and the Theorem follows with Lemma 2.4. □

Corollary 2.6 *For $n \geq 1$, $s(n)$ is given explicitly by*

$$s(n) = \frac{n}{3} \cdot 2^{n-1} - \frac{4}{9} \cdot 2^n + \frac{4}{9} \quad \textit{for } n \textit{ even,}$$

$$s(n) = \frac{n}{3} \cdot 2^{n-1} - \frac{4}{9} \cdot 2^n + \frac{5}{9} \quad \textit{for } n \textit{ odd.}$$

Proof The theorem holds for $n = 1$. For $n > 1$, we can proceed by induction using the equality from Theorem 2.5. □

The relative savings are $s(n)/(n \cdot 2^{n-1})$ and we have, as in [9, 20],

$$\lim_{n \to \infty} \frac{s(n)}{n \cdot 2^{n-1}} = \frac{1}{3} .$$

3 The SR algorithm

As we have seen, use of a signed-digit representation of the exponent can significantly reduce the number of multiplications involved in computing a modular exponentiation. However, one important limitation of this approach is that it requires the precomputation of m^{-1}, which in some environments can be relatively time-consuming.

We now present a new type of alternative exponent representation which can also help to reduce the number of multiplications required to compute a modular exponentiation. This approach has certain practical advantages, including the fact that it does not require the pre-computation of m^{-1}. We first present the method and then provide a mathematical analysis of its performance using techniques analogous to those used above. Finally, a comparison of previously proposed schemes for alternative exponent representations with the scheme presented in this paper can be found in Section 4.4.

3.1 A string replacement representation

The modified algorithm first requires the selection of a small positive integer k, the choice of which is considered below. Now suppose that, in a binary representation of a number, entries of $2^i - 1$ are allowed for every i $(2 \leq i \leq k)$ (in addition to 0 and 1). Numbers represented in this alternative form have the same positional interpretation as in a normal binary representation.

This alternative representation has the effect of allowing the replacement of any string of i consecutive ones in the binary representation of a number by a string of $i - 1$ zeros followed by the value $2^i - 1$ (for any i satisfying $2 \leq i \leq k$). Hence we call such a representation a k-ary String Replacement representation (or a k–SR representation).

A number may have many such representations. For example, the number 7 has unique binary representation

$$111$$

but has two other 2–SR representations, namely:

$$103 \quad \text{and} \quad 031.$$

Finally observe that a 1–SR representation is the same as a binary representation.

3.2 The SR algorithm itself

Now suppose that e has a k–SR representation

$$f_{r-1}f_{r-2}\ldots f_0$$

where $f_{r-1} \neq 0$ is the most significant coefficient, and hence

$$e = \sum_{i=0}^{r-1} f_i 2^i.$$

Definition 3.1 *The* weight *of a k–SR representation $f_{r-1}f_{r-2}\ldots f_0$ is the number of non-zero coefficients f_i.*

A modified 'left-to-right' version of the square-and-multiply algorithm, which we call the *SR algorithm*, works as follows:

```
pre-compute m^3, m^7, ..., m^(2^k - 1) (modulo N);
d := m^(f_{r-1})  (mod N);
for i := r - 2 downto 0 do
    begin
        d := d^2  (mod N);
        d := d · m^(f_i)  (mod N);
    end
```

Note that we assume that if $f_i = 0$ then no multiplication is performed.

Checking the validity of the modified algorithm is straightforward. It should now be clear that the number of modular multiplications in the revised algorithm depends on the number of non-zero entries in the k–SR representation of e which we are using, i.e. on the weight of the representation of e. This number can be significantly smaller than the weight of the standard binary representation of e, offering considerable performance advantages, provided that the pre-computation of $m^3, m^7, \ldots, m^{2^k-1}$ can be performed efficiently.

4 Algorithm performance

4.1 Preliminary remarks

First note that the algorithm requires $r-1$ modular squaring operations and $w-1$ modular multiplication operations, where w is the weight of the k–SR representation in use. Hence

to evaluate the performance of the SR algorithm we need to consider the weights of k–SR representations, with the objective of calculating the expected value of w.

Clearly, if we are to minimise the number of modular multiplications required to compute m^e, then we need to minimise the weight of the k–SR representation that we are using. We therefore now present a simple method for generating k–SR representations which appears to produce representations of near-minimal weight. The following algorithm takes as input an n-bit binary representation of an integer.

for $i := k$ to 2 step -1 do

> starting from the most significant digit replace every string of i consecutive ones with a string of $i-1$ zeros in the $(i-1)$ most significant positions and $2^i - 1$ in the least significant position;

We call the k–SR representation generated by the above algorithm the *Canonical* k–SR representation of an integer.

Note that, in general, minimal weight k–SR representations are not unique, as shown by the example above of the two minimum weight 2–SR representations of the number 7. Moreover, the following example shows that canonical k–SR representations do not always have the minimum possible weight. Providing an algorithm for generating minimum weight k–SR representations for every integer and every $k > 1$ remains an area for future research.

Example 4.1 *The integer 21 has canonical k–SR representation 10101 (of weight 3) for every $k \geq 1$. However it has a 2–SR representation as 333 (weight 3) and a 3–SR representation as 77 (weight 2).*

4.2 Expected weight of canonical k–SR representations

We first need the following notation. If n and k are positive integers then let $w(n,k)$ be the sum of the weights of the canonical k–SR representations of all the integers with n-bit binary representations, i.e. of all non-negative integers less than or equal to $2^n - 1$. If $\boldsymbol{s}$ is a string of i bits $(i \leq n)$ then $w_{\mathbf{s}}(n,k)$ denotes the sum of the weights of the canonical k–SR representations of all the integers with n-bit binary representations for which the first i bits are equal to $\boldsymbol{s}$.

Example 4.2 *The following table gives the canonical i–SR representation for $i = 2,3,4$ for every 4-bit integer. Note that F is used to denote the value $2^4 - 1$.*

Binary	*2–SR*	*3–SR*	*4–SR*	*Binary*	*2–SR*	*3–SR*	*4–SR*
0000	*0000*	*0000*	*0000*	*1000*	*1000*	*1000*	*1000*
0001	*0001*	*0001*	*0001*	*1001*	*1001*	*1001*	*1001*
0010	*0010*	*0010*	*0010*	*1010*	*1010*	*1010*	*1010*
0011	*0003*	*0003*	*0003*	*1011*	*1003*	*1003*	*1003*
0100	*0100*	*0100*	*0100*	*1100*	*0300*	*0300*	*0300*
0101	*0101*	*0101*	*0101*	*1101*	*0301*	*0301*	*0301*
0110	*0030*	*0030*	*0030*	*1110*	*0310*	*0070*	*0070*
0111	*0031*	*0007*	*0007*	*1111*	*0303*	*0071*	*000F*

Table 1: String representations for 4-bit integers

By summing the weights of the entries in various columns of the table it should, for example, be clear that $w(4,2) = 23$, $w(4,3) = 21$, $w(4,4) = 20$, $w_{\mathbf{0}}(4,2) = 9$, $w_{\mathbf{0}}(4,3) = w_{\mathbf{0}}(4,4) = 8$, $w_{\mathbf{11}}(4,2) = 7$, $w_{\mathbf{11}}(4,3) = 6$ and $w_{\mathbf{11}}(4,4) = 5$.

We now have:

Lemma 4.3 *If $n \geq 1$ and $k \geq 1$ then*

$$w(n,k) = \sum_{i=1}^{k} w(n-i,k) + w(n-k,k) + 2^{n-1}$$

where, as throughout, $w(i,k) = 0$ for $i \leq 0$.

Proof First observe that

$$w(n,k) = w_{\mathbf{0}}(n,k) + w_{\mathbf{10}}(n,k) + w_{\mathbf{110}}(n,k) + \cdots + w_{\mathbf{1_{k-1}0}}(n,k) + w_{\mathbf{1_k}}(n,k) \qquad (1)$$

where $\boldsymbol{1_t}$ denotes a string of t ones. It should be clear that

$$w_{\mathbf{0}}(n,k) = w(n-1,k). \qquad (2)$$

Next observe that, if $2 \leq i \leq k$, then

$$w_{\mathbf{1_{i-1}0}}(n,k) = w(n-i,k) + 2^{n-i}. \qquad (3)$$

This follows since the canonical k–SR representation of the binary string $\boldsymbol{1_{i-1}0u}$ (where $\boldsymbol{u}$ is an $(n-i)$-bit string) has weight one more than the canonical k–SR representation of the binary string $\boldsymbol{u}$. In a similar way observe that

$$w_{\mathbf{1_k}}(n,k) = w(n-k,k) + 2^{n-k}. \qquad (4)$$

Hence, combining equations 1, 2, 3 and 4, we have

$$\begin{aligned} w(n,k) &= w(n-1,k) + \sum_{i=2}^{k} (w(n-i,k) + 2^{n-i}) + w(n-k,k) + 2^{n-k} \\ &= \sum_{i=1}^{k} w(n-i,k) + w(n-k,k) + \sum_{i=2}^{k} 2^{n-i} + 2^{n-k} \end{aligned}$$

and the result follows. □

This immediately leads to the following:

Lemma 4.4 *If $n \geq 1$ and $k \geq 1$ then*

$$\sum_{i=0}^{k-1} w(n-i,k) = n2^{n-1}.$$

Proof First let

$$S(n,k) = \sum_{i=0}^{k-1} w(n-i,k).$$

We prove the lemma by induction on n. First note that

$$S(1,k) = w(1,k) = 1$$

for every $k \geq 1$, and hence the lemma is true for $n = 1$. Now suppose the lemma holds for every $n < N$, for some $N > 1$. By definition

$$\begin{aligned} S(N,k) - 2S(N-1,k) &= w(N,k) - \sum_{i=1}^{k} w(N-i,k) - w(N-k,k) \\ &= 2^{N-1} \text{ (by Lemma 4.3).} \end{aligned}$$

But $S(N-1,k) = (N-1)2^{N-2}$ by the inductive hypothesis, and hence

$$\begin{aligned} S(N,k) &= 2^{N-1} + 2(N-1)2^{N-2} \\ &= N2^{N-1} \end{aligned}$$

and the result follows. □

This then gives us:

Lemma 4.5 *Suppose $n \geq 1$, $k \geq 1$ and*

$$t = \left\lfloor \frac{n-1}{k} \right\rfloor,$$

where $\lfloor x \rfloor$ denotes the largest integer less than or equal to x. Then

$$w(n,k) = \sum_{i=0}^{t} (n+1-ki)2^{n-2-ki}.$$

Proof We again prove this result by induction on n. If $1 \leq n \leq k$ then $t = 0$, and now, by Lemma 4.3,

$$\begin{aligned} w(n,k) &= \sum_{i=1}^{k} w(n-i,k) + w(n-k,k) + 2^{n-1} \\ &= \sum_{i=1}^{k} w(n-i,k) + 2^{n-1} \text{ (since } k \geq n) \\ &= (n-1)2^{n-2} + 2^{n-1} \text{ (by Lemma 4.4)} \\ &= \sum_{i=0}^{t} (n+1-ki)2^{n-2-ki} \text{ (since } t = 0), \end{aligned}$$

and the result holds. Now suppose the lemma holds for every $n < N$, for some $N > k$. Then

$$\begin{aligned} w(N,k) - w(N-k,k) &= \sum_{i=0}^{k-1} w(N-1-i,k) + 2^{N-1} \text{ (by Lemma 4.3)} \\ &= (N-1)2^{N-2} + 2^{N-1} \text{ (by Lemma 4.4)} \\ &= (N+1)2^{N-2}. \end{aligned}$$

Now, by the inductive hypothesis (noting that $1 \leq N-k$),

$$w(N-k,k) = \sum_{i=1}^{t} (N+1-ki)2^{N-2-ki}$$

and hence

$$\begin{aligned} w(N,k) &= (N+1)2^{N-2} + \sum_{i=1}^{t} (N+1-ki)2^{N-2-ki} \\ &= \sum_{i=0}^{t} (N+1-ki)2^{N-2-ki} \end{aligned}$$

as required. □

We can now state the main result of this part of the paper.

Theorem 4.6 *Suppose $n \geq 1$, $k \geq 1$ and s is defined by $s \equiv n \pmod{k}$ and $1 \leq s \leq k$. Then*

$$w(n,k) = \frac{n2^{n+k-2}}{(2^k-1)} + \frac{(2^k-k-1)2^{n+k-2}}{(2^k-1)^2} + \frac{((k-s-1)2^{k+s-2}+(s+1)2^{s-2})}{(2^k-1)^2}.$$

Proof We first state the following well known identity (which can easily be proved by induction). If $x \neq 1$ then

$$\sum_{i=0}^{m} ix^i = \frac{x\left((mx-m-1)x^m+1\right)}{(x-1)^2}. \tag{5}$$

As before let

$$t = \left\lfloor \frac{n-1}{k} \right\rfloor,$$

and we now have

$$\begin{aligned}
w(n,k) &= \sum_{i=0}^{t}(n+1-ki)2^{n-2-ki} \text{ (by Lemma 4.5)} \\
&= 2^{n-2}\left((n+1)\sum_{i=0}^{t}(2^{-k})^i - k\sum_{i=0}^{t} i(2^{-k})^i\right) \\
&= 2^{n-2}\left(\frac{(n+1)(1-2^{-k(t+1)})}{(1-2^{-k})} - \frac{k2^{-k}\left((t2^{-k}-t-1)2^{-kt}+1\right)}{(2^{-k}-1)^2}\right) \text{ (by (5))} \\
&= 2^{n-kt-2}\left(\frac{(n+1)(2^{k(t+1)}-1)}{(2^k-1)} - \frac{k\left((t-2^k(t+1))+2^{k(t+1)}\right)}{(2^k-1)^2}\right).
\end{aligned}$$

¿From the definitions of s and t it should be clear that $t = (n-s)/k$, i.e. $kt = n-s$. The theorem then follows from

$$\begin{aligned}
w(n,k) &= 2^{s-2}\left(\frac{(n+1)(2^{n-s+k}-1)}{(2^k-1)} - \frac{k\left((\frac{n-s}{k}-2^k(\frac{n-s}{k}+1))+2^{n-s+k}\right)}{(2^k-1)^2}\right) \\
&= 2^{s-2}\left(\frac{n2^{n-s+k}+2^{n-s+k}-n-1}{(2^k-1)} - \frac{(n-s-n2^k+s2^k-k2^k+k2^{n-s+k})}{(2^k-1)^2}\right) \\
&= \frac{n2^{n+k-2}}{(2^k-1)} + \frac{(2^k-k-1)2^{n+k-2}}{(2^k-1)^2} + \frac{(k-s-1)2^{k+s-2}+(s+1)2^{s-2}}{(2^k-1)^2},
\end{aligned}$$

□

4.3 Overall algorithm complexity

We next consider the precise computational complexity of the SR algorithm when using canonical k–SR representations. First observe that, in the description of the SR algorithm given in Section 3.2 above, we assumed that e (the exponent) has a k–SR representation

$$f_{r-1}f_{r-2}\ldots f_0$$

where f_{r-1} is the most significant value, and $f_{r-1} \neq 0$. We call r the *length* of the k–SR representation. The number of modular squaring operations involved in performing the algorithm is then one less than the length of the k–SR representation in use (as noted

in Section 4.1 above). Hence, in order to derive the precise algorithm complexity for canonical representations, we first need to examine the expected value of the length of a canonical k–SR representation.

Let $v(n,k)$ denote the sum of the lengths of canonical k–SR representations of all possible n-bit integers (i.e. all integers in the range $[0, 2^n - 1]$). We then have:

Lemma 4.7 *Suppose $n \geq 1$. Then:*

(i) $v(n,1) = (n-1)2^n + 1$, *and*

(ii) $v(n,k+1) = v(n,k) - 2^{n-k} + 1$, *for every k satisfying $1 \leq k < n$.*

Proof We prove (i) by induction on n. First note that $v(1,1) = 1$, and hence (i) holds for $n = 1$. Next suppose that (i) holds for every $n < N$ for some $N > 1$. Now it is straightforward to observe that

$$\begin{aligned} v(N,1) &= 2^{N-1}N + v(N-1,1) \\ &= 2^{N-1}N + (N-2)2^{N-1} + 1 \quad \text{(by the inductive hypothesis)} \\ &= (N-1)2^N + 1 \end{aligned}$$

and (i) follows.

To establish (ii) we consider which n-bit integers will have different lengths for canonical k–SR representations and canonical $(k+1)$–SR representations. It should be clear that there will be a difference in the lengths if and only if the n-bit integer has a binary representation starting with an arbitrary number of zeros followed by a string of $k+1$ ones. Moreover, for every such integer the difference in lengths will be precisely one. Hence $v(n,k) - v(n,k+1)$ will be equal to the number of n-bit integers of the above form. It is not hard to see that there are exactly

$$2^{n-(k+1)} + 2^{n-(k+2)} + \cdots + 2^1 + 2^0 = 2^{n-k} - 1$$

such integers, and (ii) follows. □

This then immediately leads to:

Theorem 4.8 *Suppose $n \geq k \geq 1$. Then*

$$v(n,k) = (n-2)2^n + (2^{n-k+1} + k).$$

Proof We prove this theorem by induction on k. Choose some $n \geq 1$ and first note that, by Lemma 4.7(i), $v(n,1) = (n-2)2^n + (2^n + 1)$, and hence the theorem holds for $k = 1$. Next suppose that the theorem holds for every $k < K$ for some K satisfying $n \geq K > 1$. Now, by Lemma 4.7(ii), it follows immediately that

$$\begin{aligned} v(n,K) &= v(n,K-1) - 2^{n-K+1} + 1 \\ &= (n-2)2^n + (2^{n-K+2} + K - 1) - 2^{n-K+1} + 1 \quad \text{(by the inductive hypothesis)} \\ &= (n-2)2^n + (2^{n-K+1} + K) \end{aligned}$$

and the result follows. □

We now have all the information we need to compute the complexity of a single modular exponentiation using the SR algorithm (assuming the use of a canonical k–SR representation).

Theorem 4.9 *Suppose e is randomly chosen from the range $[0, 2^n - 1]$ from some $n \geq 1$. Then the expected number of computations required to compute $m^e \bmod N$ using the SR algorithm with a canonical k–SR representation is*

$$\frac{n2^{k-2}}{(2^k-1)} + \frac{(2^k-k-1)2^{k-2}}{(2^k-1)^2} + \frac{(k-s-1)2^{k+s-n-2} + (s+1)2^{s-n-2}}{(2^k-1)^2} + k - 2$$

multiplication operations (modulo N), where $s \equiv n \pmod{k}$, and

$$n + k - 4 + 2^{1-k} + k2^{-n}$$

squaring operations (modulo N).

Proof To compute the complexity of the SR algorithm we first need to examine the work involved in pre-computing $m^3, m^7, \ldots, m^{2^k-1}$ (all modulo N). These values can be successively computed by a single squaring operation followed by a single multiplication by m (both mod N). Hence the pre-computation for the SR algorithm amounts to $k-1$ modular squaring operations together with $k-1$ modular multiplication operations.

As observed at the beginning of Section 3, the expected number of modular multiplication operations is one less than the expected weight of a canonical k–SR representation, added to the pre-computation overhead. This immediately implies that the expected number of modular multiplications is equal to

$$\frac{w(n,k)}{2^n} + k - 2$$

and the desired result follows from Theorem 4.6.

The expected number of modular squaring operations can also be simply calculated as one less than the expected length of the k–SR representation in use, added to the pre-computation overhead. Since we assume the use of the canonical k–SR representation, this means that the expected number of modular squaring operations is given by

$$\frac{v(n,k)}{2^n} + k - 2$$

and the desired result follows from Theorem 4.8. □

Since n is always likely to be fairly large (for RSA n is likely to be between 512 and 1024), the following simple corollary is useful.

Corollary 4.10 *Suppose e is randomly chosen from the range $[0, 2^n - 1]$ for some large $n \geq 1$. Then the expected number of computations required to compute $m^e \bmod N$ using the SR algorithm with a canonical k–SR representation approximates to*

$$\frac{n2^{k-2}}{(2^k-1)} + \frac{(2^k-k-1)2^{k-2}}{(2^k-1)^2} + k - 2$$

multiplication operations (modulo N) and

$$n + k - 4 + 2^{1-k}$$

squaring operations (modulo N).

We now tabulate the overall complexity of the SR algorithm for small values of k and for three 'typical' values of n using the above results. The complexity is quoted as the sum of two numbers, the first representing the expected number of modular squaring operations required and the second the number of modular multiplications required. Note that, for comparison purposes, the values in the table for $k = 1$ correspond precisely to the complexity of the conventional square-and-multiply algorithm presented in Section 1.

n	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=10$
512	510+255	511+171	511+147	512+139	513+135	514+134	515+134	518+136
768	766+383	767+256	767+221	768+207	769+201	770+199	771+199	774+200
1024	1022+511	1023+341	1023+294	1024+275	1025+267	1026+264	1027+263	1030+264

Table 2: Complexity of SR algorithm

4.4 Comparison with other techniques

We now compare the SR algorithm with other variants of the square-and-multiply exponentiation algorithm as given by Jedwab and Mitchell, [17], Koç, [9], Eğecioğlu and Koç, [5] and Zhang, [20]. Note first that the SR algorithm has the potential to reduce the number of modular multiplications in exponentiation with an n-bit exponent from $n/2$ to a little over $n/4$, whilst leaving the number of modular squaring operations essentially unchanged at n. Note also that, using techniques such as those described by Knuth, [8], modular squaring can be made to run perhaps twice as fast as general modular multiplication, making the reduction more significant than it would otherwise appear.

Probably the most simple of these other methods is to use a signed-digit representation of the exponent, as described in Section 2 above. As discussed there, this has the potential to reduce the number of modular multiplications to $n/3$ (again leaving the number of modular squaring operations unchanged), at the cost of pre-computing m^{-1}. Given that $n/3$ is significantly greater than $n/4$, and also observing that pre-computing m^{-1} can be a non-trivial operation, the SR algorithm presented above offers clear advantages.

Hui and Lam [6] avoid pre-computing m^{-1} and use instead all the 2^{k-1} odd numbers of length k or less. In a binary integer representation, all strings corresponding to a pre-computed odd number are then replaced by a single digit. The average weight of such a representation converges to $\frac{n}{k+1}$ for large n.

Koç [9], and Eğecioğlu and Koç [5], also describe several other square-and-multiply variants involving processing the exponent more than one bit at a time. Certain of these algorithms offer the possibility of reducing the expected number of modular multiplications a little below $n/4$, although they still require approximately n modular squaring operations. However, as we see in the next section, although more powerful in theory, they do not fit well with other possible improvements in the exponentiation calculations.

4.5 Combination with other algorithms

We conclude this paper by considering how the SR algorithm might be used in conjunction with certain other types of fast exponentiation techniques, and in particular the schemes of Quisquater and Couvreur [14] and Selby and Mitchell [17]. Both these papers describe methods which take advantage of the fact that the basic left-to-right version of the square-and-multiply algorithm (as given in Section 1 above) involves repeated multiplication by $m \pmod N$. It is therefore worth investing a certain amount of time before commencing the square-and-multiply procedure in calculating tables which can speed up modular multiplication by m.

However, if the modified version of square-and-multiply to be used involves multiplying by a large range of different powers of m (as with the algorithms of Koç and Eğecioğlu)

then pre-computing tables is no longer worthwhile. But this is not the case with the SR algorithm, at least when small values of k are employed. For example, if $k = 2$ then the SR version of the exponentiation algorithm requires repeated multiplication by either m or m^3 and it will almost certainly still be worth pre-computing tables (one for multiplication by m and one for m^3). As should be clear from Table 2, small values of k realise a large part of the potential gains of the SR algorithm, and hence we have the basis for a fruitful combination of techniques which have the potential to offer significant advantages over other methods for modular exponentiation.

Acknowledgement

We have to thank an anonymous referee for considerably simplifying the derivation of Theorem 2.5.

References

[1] S. Arno and F.S. Wheeler. Signed digit representations of minimal Hamming weight. *IEEE Transactions on Computers*, **42**:1007–1010, 1993.

[2] H.J. Beker and F.C. Piper. *Cipher Systems.* Van Nostrand Reinhold, 1982.

[3] A.D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4:236–240, 1951.

[4] E.F.Brickell. A fast modular multiplication algorithm with application to two key cryptography. In *Advances in Cryptology, Proceedings of Crypto'82*: 51–60, 1982

[5] O. Eğecioğlu and C.K. Koç. Fast modular exponentiation. In *Proceedings of 1990 Bilkent International Conference on New Trends in Communication, Control, and Signal Processing*, pages 188–194, Ankara Turkey, July 1990. Elsevier Science Publishing Co.

[6] L.C.K. Hui and K.Y. Lam. Fast Square-and-Multiply Exponentiation for RSA. *Electronics Letters*, **30**:1396–1397, 1994.

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

[8] D.E. Knuth. *The art of computer programming, Volume 2: seminumerical algorithms.* Addison-Wesley, Reading, Mass., 2nd edition, 1981.

[9] C.K. Koç. High-radix and bit recoding techniques for modular exponentiation. *International Journal of Computer Mathematics*, **40**:139–156, 1991.

[10] C.K. Koç and C.-Y. Hung. Adaptive m-ary segmentation and canonical recoding algorithms for multiplication of large binary numbers. *Computers Math. Applic.*, **24**:3–12, 1992.

[11] P.L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, **44**: 519–521, 1985.

[12] M.J. Norris and G.J. Simmons. Algorithms for high-speed modular arithmetics. *Congressus Numerantium*, **31**:151–163, 1981.

[13] K.C. Posch and R. Posch. Modulo Reduction in Residue Number Systems. *IEEE Transactions on Parallel and Distributed Systems*, **6**:449–454, 1995.

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

[15] G.W. Reitwiesner. Binary arithmetic. In F.L. Alt, editor, *Advances in Computers 1*, pages 232–308. Academic, New York, 1960.

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

[17] A. Selby and C.J. Mitchell. Algorithms for software implementations of RSA. *Proceedings of the IEE, Part E*, **136**:166–170, 1989.

[18] N. Takagi and S. Yajima. Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem. *IEEE Transactions on Computers*, **41**:887–891, 1992.

[19] N. Takagi, H. Yasuura, and S. Yajima. High-speed VLSI multiplication algorithm with a redundant binary addition tree. *IEEE Transactions on Computers*, **C-34**:789–796, 1985.

[20] C.N. Zhang. An improved binary algorithm for RSA. *Computers Math. Applic.*, **25**:15–24, 1993.