Fast Exponentiation with Precomputation: Algorithms and Lower Bounds * †

Ernest F. Brickell, Daniel M. Gordon‡,
Kevin S. McCurley, and David B. Wilson§
Sandia National Laboratories
Organization 1423
Albuquerque, NM 87185

August 3, 1992

Abstract:

In several cryptographic systems, a fixed element g of a group (generally $\mathbb{Z}/q\mathbb{Z}$) is repeatedly raised to many different powers. In this paper we present a practical method of speeding up such systems, using precomputed values to reduce the number of multiplications needed. In practice this provides a substantial improvement over the level of performance that can be obtained using addition chains, and allows the computation of g^n for $n < N$ in time $O(\log N/\log\log N)$. We show that this method is asymptotically optimal, and for specific cases, within a small factor of optimal. We also show how these methods can be parallelized, to compute powers in time $O(\log\log N)$ with $O(\log N/\log\log N)$ processors.

Keywords: Exponentiation, Cryptography.

AMS (MOS) subject classifications: 11Y16, 68Q25.

*This research was supported in part by the U.S. Department of Energy under contract number DE-AC04-76DP00789, in part by the Defense Advanced Research Projects Agency under Grant N00014-91-J-1698, and in part by an ONR-NDSEG fellowship.

†PATENT CAUTION: this document may reveal patentable subject matter.

‡Current address: Department of Computer Science, University of Georgia, Athens, GA 30602

§Current address: Department of Mathematics, M.I.T., Cambridge, MA 02139

1 Introduction

The problem of efficiently evaluating powers has been studied by many people (see [10, section 4.6.4] for an extensive survey). One standard method is to define an *addition chain*: a sequence of integers

$$1 = a_0, a_1, \ldots, a_l = n$$

such that for each $i = 1, \ldots, l$, $a_i = a_j + a_k$, for some j and k less than i. Then x^n may be computed by starting with $x^{a_0} = x$ and computing $x^{a_1}, x^{a_2}, \ldots, x^{a_i} = x^{a_j} \cdot x^{a_k}, \ldots, x^{a_l} = x^n$. As an example, the "square-and-multiply" method of exponentiation (also known as the left-to-right binary method, see [10, page 441]) can be viewed as an addition chain of the form

$$1, 2d_0, 2d_0 + d_1, 2(2d_0 + d_1), 2(2d_0 + d_1) + d_2, \ldots, n,$$

where n is written in binary as $\sum_{i=0}^{m} d_i 2^{m-i}$. This clearly takes at most $\lfloor \log n \rfloor + \nu(n) - 1$ multiplications, where $\log n$ is the base 2 logarithm, and $\nu(n)$ is the number of 1's in the binary representation of n.

The amount of storage necessary for computing an addition chain depends on the number of intermediate a_j's that need to be stored at any one time. It is fairly obvious that the "square-and-multiply" method only requires storing two values, and the amount of storage required by an addition chain is bounded above by the length of the chain.

Let $l(n)$ be the smallest l possible for a given n. Despite dozens of papers on the subject, the behavior of $l(n)$ is not completely understood. It is known that

$$\lceil \log n \rceil \leq l(n) \leq \lfloor \log n \rfloor + \nu(n) - 1, \tag{1.1}$$

where logs are to base 2 and $\nu(n)$ is the number of ones in the binary representation of n. Let $\lambda(n) = \lfloor \log n \rfloor$. Asymptotically, it is known that

$$l(n) \leq \lambda(n) + \lambda(n)/\lambda\lambda(n) + O(\lambda(n)\lambda\lambda\lambda(n)/\lambda\lambda(n)^2). \tag{1.2}$$

Addition chains can be used to great advantage when the exponent n is fixed (as in the RSA cryptosystem), and the goal is to quickly compute x^n for randomly chosen bases x. For a randomly chosen exponent between 0 and $2^m - 1$, we expect the binary algorithm to take $1.5m - 3 + 3/2^m$

multiplications on average, so for a 512-bit exponent we get an average of about 765 multiplications. Results in [3] report that addition chains of length around 605 are relatively easy to compute, resulting in a 21% improvement. Note that it follows from (1.1) that no addition chain could do better than 512 multiplications. In $GF(2^m)$, squarings can be done essentially for free using a normal basis, so powers may be calculated faster than by addition chains (see [2], [16]).

We shall consider a slightly different problem in this paper, for which it is actually possible to break the barrier of 512 multiplications for a 512-bit exponent. For many cryptosystems (e.g. [4],[5],[15],[1]), the dominating computation is to compute for a fixed base g the power g^n for a randomly chosen exponent n. For this problem, we achieve a substantial improvement over addition chains by storing a set of precomputed values.

Unless otherwise noted, we will assume that g is an element of $\mathbb{Z}/q\mathbb{Z}$, where q is a large integer (say 512 bits), and we need to repeatedly calculate powers of g up to g^{N-1}, where N is also large. The Schnorr scheme [15] uses N of 140 bits, the DSS scheme [1] uses N of 160 bits, and the Brickell-McCurley scheme [5] has N of 512 bits. We will assume that operations other than multiplications mod q will use negligible time.

As an example of the practicality of the schemes that we present, consider the exponentiation required for Diffie-Hellman key exchange using a 512-bit prime and 512-bit random exponent. Using the square-and-multiply scheme, we would expect to perform $765+3/2^{512}$ modular multiplications on average, using storage of at least 64 bytes. For one of the schemes that we present here, we expect to perform fewer than 105 modular multiplications on average and 106 in the worst case, using storage of 23168 bytes. This gives better than a seven-fold speedup on average, and a ten-fold speedup in the worst case. Furthermore, we show that for this amount of storage, at least 70 multiplications are required in the worst case. This shows that (for this example) our method is at most 50% from optimality in a model of computation in which the only way to compute g^x $(x \neq 0, 1)$ is to multiply g^{x_1} by g^{x_2} where $x_1 + x_2 = x$.

The method works for any group, and in Section 5 we discuss its operation in $GF(p^m)$ where p is a small prime and m is large. If a normal basis is used, then precomputation does not have as big of an effect since p'th powers can be computed by performing a shift and are thus almost free. Stinson

[16] has shown that exponentiation in $\mathrm{GF}(2^{593})$ can be computed using 129 multiplications. Using our method, we reduce this to 124 multiplications if the inverse of g is not precomputed, and 113 multiplications if the inverse is precomputed. More stored values can reduce the number of multiplications further.

For the rest of this paper, it will be assumed that g is fixed, and n is uniformly distributed on $\{0, \ldots, N-1\}$.

2 Basic strategies

Using the square-and-multiply method, g^n may be computed using at most $2\lceil \log N \rceil$ multiplications, and on average $\leq 3\lceil \log N \rceil / 2$ multiplications. By storing a set of precomputed values, we want to reduce the number of multiplications to compute g^n.

One simple method (see [7]) is to precompute the set

$$S = \left\{ g^{2^i} \mid i = 1, \ldots, \lceil \log N \rceil - 1 \right\}.$$

Then g^n may be computed in $\nu(n) - 1$ multiplications, or at most $\lceil \log N \rceil / 2 - 1$ on average, with $\lceil \log N \rceil$ storage by multiplying together the powers corresponding to nonzero digits in the binary representation of n.

There is no reason that powers of 2 have to be stored. Suppose we precompute and store $g^{x_0}, g^{x_1}, \ldots, g^{x_{m-1}}$ for some integers $x_0, \ldots, x_{m-1}$. If we are then able to find a decomposition

$$n = \sum_{i=0}^{m-1} a_i x_i, \tag{2.1}$$

where $0 \leq a_i \leq h$ for $0 \leq i < m$, then we can compute

$$g^n = \prod_{d=1}^{h} c_d^d, \tag{2.2}$$

where

$$c_d = \prod_{a_i = d} g^{x_i}. \tag{2.3}$$

If (2.2) were computed using optimal addition chains for $1, 2, \ldots, h$, the total number of multiplications to compute g^n would be about $m + O(h \log h)$.

However, (2.2) can be computed much more efficiently, as the following result shows.

THEOREM 1. *Suppose* $n = \sum_{i=0}^{m-1} a_i x_i$, *where* $0 \le a_i \le h$. *If* g^{x_i} *is precomputed for each* $0 \le i < m$, *and if* $m + h \ge 2$, *then* g^n *can be computed with* $m + h - 2$ *multiplications.*

PROOF: The following is an algorithm to compute g^n.

$b \leftarrow 1$

$a \leftarrow 1$

`for` $d = h$ `to` 1 `by` -1

    `for each` i `such that` $a_i = d$

        $b \leftarrow b * g^{x_i}$

    $a \leftarrow a * b$.

`return` a.

It is easy to prove by induction that, after going through the loop i times, we have $b = c_h c_{h-1} \cdots c_{h-i+1}$ and $a = c_h^i c_{h-1}^{i-1} \cdots c_{h-i+1}$. After traversing the loop h times, it follows that $a = \prod_{d=1}^{h} c_d^d$.

It remains to count the number of multiplications performed by the algorithm. We shall count only those multiplications where both multiplicands are unequal to 1, since the others can be accomplished simply by assignments. We may assume $n \neq 0$. There are m digits, so the $b \leftarrow b * g^{x_i}$ line gets executed at most m times. The $a \leftarrow a * b$ line gets executed h times. Finally, at least two of these multiplications are free since a and b are initially 1. □

Embodied in the algorithm is a method for computing the product $\prod_{d=1}^{h} c_d^d$ in at most $2h - 2$ multiplications. We can argue that, in the absence of any relations between the c_d's, this is optimal. Notice that if we take any algorithm to compute $\prod_{d=1}^{k} c_d^d$ and remove multiplications involving c_k, we have computed $\prod_{d=1}^{k-1} c_d^d$, which takes $2k - 4$ multiplications by our induction hypothesis. There cannot be only one multiplication by c_k, since then c_k would be raised to the same power as whatever it was multiplied by. Therefore at least two extra multiplications are needed.

The most obvious use for (2.2) is to represent the exponent in base b, using at most $m = \lceil \log_b N \rceil$ digits to do so, and precompute g^{b^k}, for $k = 1, \ldots, \lceil \log_b N \rceil - 1$. Using this algorithm with a base b representation for n, Theorem 1 shows that g^n can be computed in at most $\lceil \log_b N \rceil + b - 3$ multiplications. For a randomly chosen exponent n, we expect that a digit will be zero about $1/b$ of the time, so that on average we expect the $b \leftarrow b * g^{x_i}$ line to be executed $\frac{b-1}{b}\lceil \log_b N \rceil$ times, giving an expected number of multiplications that is at most $\frac{b-1}{b}\lceil \log_b N \rceil + b - 3$. For a 512-bit exponent, the optimal value of b is 26. This method requires at most 127.8 multiplications on average, 132 multiplications in the worst case, and requires 109 stored values.

Note that some minimal effort may be required to convert the exponent from binary to base b, but this is probably negligible compared to the modular multiplications (certainly this is the case for exponentiation in $\mathbb{Z}/q\mathbb{Z}$). Even if this is not the case, then we can simply use base 32, which allows us to compute the digits for the exponent by extracting 5 bits at a time. Using this choice, the scheme will require at most 128.8 multiplications on average, 132 multiplications in the worst case, and 103 stored values.

3 Other number systems

The major problem with the general approach described in the previous section is that we must be able to compute a representation of the form (2.1). Subject to this constraint, the goal is to choose the parameters to optimize the necessary number of multiplications for a given amount of storage. In this section we shall explore some approaches to this problem.

As our first example of this approach, if b is an integer, then every integer n such that $|n| \leq (b^m - 1)/2$ may be represented as $\sum_{i=0}^{m-1} a_i b^i$, where each $a_i \in (-\lceil (b-1)/2 \rceil, \lceil (b-1)/2 \rceil]$ (see Theorem 2 below). If the powers $g^{\pm 1}, g^{\pm b}, \ldots, g^{\pm b^{m-1}}$ are precomputed, then we compute

$$c_d = \prod_{|a_j|=d} g^{sign(a_j) b^j}.$$

In this case, $m = \lceil \log_b(2N-1) \rceil$, $h = \lceil (b-1)/2 \rceil$, and the worst case number of multiplications required is $\lceil \log_b(2N-1) \rceil + \lceil (b-1)/2 \rceil - 2$. Moreover, since the probability that a digit is nonzero is again at most

$(b-1)/b$, the average number of multiplications required is bounded above by $\lceil \log_b(2N-1) \rceil (b-1)/b + \lceil (b-1)/2 \rceil - 2$. The storage required is for $2\lceil \log_b(2N-1) \rceil$ values. For a 512-bit exponent, a good base is 45, resulting in 111.91 multiplications on average and 114 multiplications in the worst case, using 188 stored values.

At one extreme, if we take $h = 1$, we can completely bypass the computation of $\prod_{d=1}^{h} c_d^d$ by storing instead all values g^{db^i}, $1 \le d < b$, $0 \le i \le \lceil \log_b N \rceil - 1$, and perform at most $\lceil \log_b N \rceil (b-1)/b - 1$ multiplications on average, and $\lceil \log_b N \rceil - 1$ multiplications in the worst case. For example, with $N = 2^{512}$ we might take $b = 256$ and $h = 1$ to derive a method that takes 62.75 multiplications on average, and 63 multiplications in the worst case. The problem with this method is that it requires $(b-1)\lceil \log_b N \rceil$ stored values. For this case that is 16320 stored values, or 1,044,480 bytes of storage.

By slightly increasing the value of h, we can reduce either the storage or time required. For instance, taking $h = 2$, let

$$M_2 = \{d \mid 1 \le d < b \ , \ \omega_2(d) \equiv 0 \pmod 2\}, \tag{3.1}$$

where $\omega_p(d)$ is the largest power of p that divides d, i.e., $k = \omega_p(d)$ if and only if $p^k \parallel d$. It suffices to store the values, $\left\{g^{db^i} \mid d \in M_2\right\}$. Then for $1 \le a_i < b$, $g^{a_i b^i} = g^{db^i}$ or g^{2db^i} for some $d \in M_2$. Using the same base, this only increases the time by one multiplication, but reduces the storage substantially. For example, with $b = 256$, we achieve an average of at most 63.75 multiplications but reduce the storage to 10880 values, or 696,320 bytes.

Increasing h or decreasing b further increases the time and lowers the storage. Continuing this line of reasoning, we can take

$$M_3 = \{d \mid 1 \le d < b, \omega_2(d) + \omega_3(d) \equiv 0 \pmod 2\}. \tag{3.2}$$

For example, if we take a base of $b = 128$ for a 512 bit exponent, then we arrive at a method that requires an average of at most 74.42 multiplications, using storage for 5624 values.

In the remainder of this section we shall describe a method that allows us to reduce the amount of computation using a much smaller amount of storage.

Call a set of integers D a *basic digit set* for base b if any integer can be represented as

$$a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b + a_0, \tag{3.3}$$

for some k where each $a_i \in D$. This definition differs from that in [12] in that we allow redundancy; there may be more than b numbers in D, and so the representation may not be unique.

Before we examine the problem of finding basic digit sets for our problem, we should first remark that the difficulty of finding a representation of the form (3.3) is almost exactly the same difficulty as finding the (ordinary) base b representation. The algorithm for finding such a representation was published by Matula [12], and a particularly simple description was later given in [10, Exercise 4.1.19].

In searching for good basic digit sets, we can make use of the following result of Matula [12], which provides a very efficient algorithm for determining if a set is basic.

THEOREM 2. *Suppose that D contains a representative of each residue class modulo b. Let $d_{\min} = \min\{s \mid s \in D\}$ and $d_{\max} = \max\{s \mid s \in D\}$. Then D is a basic digit set for base b if there are representations* (3.3) *for each i with*

$$\frac{-d_{\max}}{b-1} \le i \le \frac{-d_{\min}}{b-1}.$$

In the method that we consider now, we shall store powers g^{mb^j}, for $j \le \lceil \log_b N \rceil$ and m in a set M of multipliers. We need to choose M and h for which

$$D(M,h) = \{km \mid m \in M, 0 \le k \le h\}$$

is a basic digit set. Given a representation $n = \sum_{i=0}^{m-1} d_i b^i$ in terms of this basic digit set, we can represent $d_i = m_i k_i$ and compute

$$g^n = \prod_{k=1}^{h} \left(\prod_{k_i = k} g^{m_i b^i} \right)^k = \prod_{k=1}^{h} c_k^k \tag{3.4}$$

In this notation, the methods mentioned previously have parameters

$$\begin{aligned}
&M = \{1\},\ h = b-1,\\
&M = \{\pm 1\},\ h = \lceil (b-1)/2 \rceil,\\
&M = M_3,\ h = 3,\\
&M = M_2,\ h = 2,\\
&M = \{1, \ldots, b-1\},\ h = 1.
\end{aligned}$$

For every case except $M = \{\pm 1\}$, the representation of a number is exactly the same length as the ordinary base b representation.

Another class of multiplier sets is provided by the following:

THEOREM 3. *Suppose b is odd. Let $M = \{\pm 1, \pm 2\}$ and $h = \lfloor b/3 \rfloor$. Then $D(M,h)$ is a basic digit set.*

PROOF: It is easily checked that the set $D(M,h)$ includes at least one representative of each congruence class mod b. Then Theorem 2 applies trivially, since $d_{\min} = -2\lfloor b/3 \rfloor$ and $d_{\max} = 2\lfloor b/3 \rfloor$. □

The following result shows that the number of digits in a representation using a basic digit set $D(M,h)$ for a base b has at most one more digit than the standard base b representation.

THEOREM 4. *Let $D(M,h)$ be a basic digit set modulo b such that $\max_{a \in M} |a| \leq b/2$, and such that $\{-1, 1\} \subset M$. If $b > h \geq 1$ then for every $n < b^m$, there exists a sequence of digits $d_i \in D$ for $0 \leq i \leq m$ such that $n = \sum_{i=0}^{m} d_i b^i$.*

PROOF: We define sequences n_i and d_i inductively by setting $n_0 = n$, and for $j = 0 \ldots m$, choosing $d_j \in D$ such that $d_j \equiv n_j \bmod b$, and $n_{j+1} = (n_j - d_j)/b$.

It is easy to verify that $n = n_{j+1} b^{j+1} + \sum_{i=0}^{j} d_i b^i$ for $0 \leq j \leq m$. We will finish the proof by showing that d_m can be chosen to force $n_{m+1} = 0$.

Let $n = \sum_{i=0}^{m-1} a_i b^i$, where $0 \leq a_i \leq b-1$ for $0 \leq i \leq m-1$. We shall prove by induction on j that $n_j = c_j + \sum_{i=0}^{m-1-j} a_{i+j} b^i$ for some c_j with $|c_j| \leq h$ for $0 \leq j \leq m$. Clearly we have $c_0 = 0$. Note that

$$\begin{aligned} n_{j+1} &= (n_j - d_j)/b \\ &= (c_j + a_j - d_j)/b + \sum_{i=0}^{m-j-2} a_{i+j+1} b^i. \end{aligned}$$

Let $c_{j+1} = (c_j + a_j - d_j)/b$. Then by the induction hypothesis,

$$|c_{j+1}| \leq (h + b - 1 + hb/2)/b \leq h + 1/2.$$

Since c_{j+1} is an integer, it follows that $|c_{j+1}| \leq h$. By defining $d_m = n_m$, we achieve $n_{m+1} = 0$. □

It is clear from the proof that the high order digit in a base b representation using digits from $D(M,h)$ is bounded in absolute value by h. Consequently, the only values of k for which g^{kb^m} needs to be stored are for $k = \pm 1$.

Note also that slight improvements are possible in Theorem 4 for certain cases. In particular, for b odd, $M = \{-1, 1\}$ and $h = (b-1)/2$, we have already observed that it is possible to represent every number $n < N$ using at most $\lceil \log_b(2N-1) \rceil$ digits rather than the bound $\lceil \log_b N \rceil + 1$ given by Theorem 4.

Tables 1 and 2 summarize the effects of the various methods presented above on the storage and complexity of the parameters that might be used for the DSS and Brickell-McCurley schemes, namely 160 and 512 bit exponents respectively. The larger sets of multipliers were found by a computer search. Large sets of good multipliers become harder to find, and use increasing amounts of storage for progressively smaller reductions in computation.

In each table, most of the expected times are actually upper bounds for the expected time. The upper bounds were calculated using the assumption that the probability of a digit being zero for any base b basic digit set is $1/b$. We have not proven this in general, but it's a reasonable heuristic that matches empirical results. In either case, the numbers given as upper bounds for the expected times are off by at most 1 (the bounds given might be this far off in the case where $\log_b N$ is only slightly larger than an integer).

A lower bound for the amount of computation required using this method is given by the fact that for a fixed value of $|M|$, we require $h|M| \geq b-1$ in order for the set $D(M,h)$ to represent every value modulo b. Hence for a given base b, the worst case number of multiplications is bounded below by

$$\lceil \log_b N \rceil + h - 2 \geq \lceil \log_b N \rceil + \lceil (b-1)/|M| \rceil - 2. \tag{3.5}$$

For example, using a set M with 2 elements and a 512 bit exponent, we can do no better than 114 multiplications in the worst case, and the entry given in Table 2 achieves this (although the storage might be reduced to as little as 176 values from our 188). Similarly, using a set M with 8 elements, $b = 72$, and a 512 bit exponent, we can do no better than 90 multiplications in the worst case, and no matter what b is, we cannot do better than 88 multiplications. The entry in Table 2 for $|M| = 8$ achieves 93 multiplications in the worst case.

While the preceding argument shows that there is not much room for improvement using (3.4), slight improvements may be achieved by modifying the product $\prod_{k=1}^{h} c_k^k$. For example, it can easily be verified that the set

$$D = \{km \mid k \in K, m \in M\},$$

where

$$\begin{aligned} K &= \{0,1,2,3,4,6,8,11\} \\ M &= \{\pm 1, \pm 12\}. \end{aligned}$$

is a basic digit set for the base 29. Thus it suffices to store powers g^{m29^i}, $m \in M$, and compute a product of the form $c_1 c_2^2 c_3^3 c_4^4 c_5^6 c_6^8 c_7^{11}$. It can be shown that the latter product can be computed in only 12 multiplications, and the average number of multiplications required for a 160-bit exponent is at most 37.83. This does better than the entries in Table 1, but we do not include it in the table because it does not fit the schemes using (3.4).

4 Parallelizing the algorithm

The first method for computing a power g^n that we presented in section 2 consisted of three main steps:

1. Determine a representation $n = a_0 + a_1 b + \ldots + a_{m-1} b^{m-1}$.
2. Calculate $c_d = \prod_{a_j = d} g^{b^j}$ for $d = 1, \ldots, h$.
3. Calculate $g^n = \prod_{d=1}^{h} c_d^d$.

As we mentioned previously, the algorithm of Matula makes the first step easy, even with a large set of multipliers. Most time is spent in the second and third steps. Both of these may be parallelized. Suppose we have h processors. Then for step 2, each processor can calculate its c_d separately. The time needed to calculate c_d depends on the number of a_j's equal to d. Thus the time for step 2 will be the d with the largest number of a's equal to it.

This is equivalent to the maximum bucket occupancy problem: given m balls randomly distributed in h buckets, what is the expected maximum

b	M	h	expected time	worst-case time	storage
12	$\{1\}$	11	50.25	54	45
19	$\{\pm 1\}$	9	43.00	45	76
29	$\{\pm 1, \pm 2\}$	9	39.83	41	134
29	$\{\pm 1, -2, 9, 10\}$	8	38.83	40	167
29	$\{\pm 1, \pm 2, \pm 9\}$	7	37.83	39	200
36	$\{\pm 1, 9, \pm 14, \pm 17\}$	7	36.11	37	219
36	$\{\pm 1, \pm 3, \pm 11, \pm 13\}$	6	35.11	36	250
36	$\{\pm 1, \pm 3, \pm 6 \pm 11, \pm 13\}$	5	34.11	35	312
36	M_3	3	31.14	32	620
53	M_3	3	28.47	29	840
64	M_3	3	27.58	28	972
64	M_2	2	26.58	27	1134
102	M_3	3	24.76	25	1440
128	M_3	3	23.82	24	1748
128	M_2	2	22.82	23	1955
155	M_2	2	21.86	22	2244
256	M_2	2	20.92	21	2751

Table 1: Parameters for a 160-bit exponent ($N = 2^{160}$). By comparison, the binary method requires $237 + 3/2^{160}$ multiplications on average, and 318 multiplications in the worst case.

b	M	h	expected time	worst-case time	storage
26	$\{1\}$	25	127.81	132	109
45	$\{\pm 1\}$	22	111.91	114	188
53	$\{\pm 1, \pm 2\}$	17	104.28	106	362
53	$\{\pm 1, -9, 18, 27\}$	16	103.28	105	452
67	$\{\pm 1, \pm 2, \pm 23\}$	16	98.72	100	512
75	$\{\pm 1, 5, 14, -16, 29, -31\}$	15	95.88	97	583
81	$\{\pm 1, \pm 3, \pm 26, \pm 28\}$	13	91.99	93	650
72	$\{\pm 1, \pm 3, \pm 4, \pm 23, \pm 25\}$	11	91.84	93	832
64	M_3	3	85.66	87	3096
122	M_3	3	74.39	75	5402
256	M_2	2	63.75	64	10880
256	$\{1, 2, \ldots, 255\}$	1	62.75	63	16320

Table 2: Parameters for a 512-bit exponent ($N = 2^{512}$). By comparison, the binary method requires $765 + 3/2^{512}$ multiplications on average and 1022 in the worst case.

bucket occupancy? This is discussed in [17], in connection with analysis of hashing algorithms. Taking b and h to be $O(\log N / \log\log N)$, so $m/h = \Theta(1)$, the expected value is

$$\frac{\log h}{\log\log h} = O\left(\frac{\log\log N}{\log\log\log N}\right).$$

For step 3, each processor can compute c_d^d for one d using a standard addition chain method, taking at most $2\log h$ multiplications. Then the c_d^d's may be combined by multiplying them together in pairs repeatedly to form g^n (this is referred to as *binary fan-in multiplication* in [16]). This takes $\log h$ rounds.

Therefore, taking $h = O(\log N / \log\log N)$, we may calculate powers in $O(\log\log N)$ rounds with $O(\log N / \log\log N)$ processors. For example, storing only powers of b, we may compute powers for a 160-bit exponent in 13 rounds using 15 processors, taking $b = 16$ and $M = \{1\}$. For a 512-bit exponent, we can compute powers with 27 processors in 17 rounds, using $b = 28$.

The disadvantage to this method is that each processor needs access to each of the powers g^{b^i}, so we either need a shared memory or every power stored at every processor. An alternative approach allows us to store only one power at each processor.

For this method, we will have m processors, each of which computes one $g^{a_i b^i}$ using a stored value and an addition chain for a_i. This will take at most $2 \log h$ rounds. Then the processors multiply together their results using binary fan-in multiplication to get g^n. The total time spent is at most $2 \log h + \log m$, which is again $O(\log \log N)$ rounds with $O(\log N / \log \log N)$ processors.

If the number of processors is not a concern, then the optimal choice of base is $b = 2$, for which we need $\log N$ processors and $\log \log N$ rounds. We could compute powers for a 512-bit exponent with 512 processors in 9 rounds, and for a 160-bit exponent with 160 processors in 8 rounds. Taking a larger base reduces the number of processors, but increases the time.

5 Exponentiation in $GF(p^n)$

The above methods work for any group, but for special cases we may take advantage of special structure. Suppose that g is in $GF(p^n)$, where p is a small prime ($p = 2$ is the most-studied case, and has been proposed for use in cryptographic systems [13]). A normal basis has the form $\{\beta, \beta^p, \beta^{p^2} \ldots, \beta^{p^{n-1}}\}$ (see, for example, [11] for details). Using a normal basis representation, the pth power of an element is simply a cyclic shift, and so is almost free compared to a standard multiplication.

This fact may be used to eliminate the extra storage. If the base b is chosen to be p^k, then the powers g^{b^j} may be calculated rapidly by cyclic shifts, and the exponentiation may be done as before, storing only the powers g^m for each m in the set of multipliers. This is a generalization of the methods given in [2], [8], and [16].

It has been shown [16] that exponentiation in $GF(2^n)$ can be done in $\lceil n/k \rceil + 2^{k-1} - 2$ multiplications. Taking $N = 2^n$ and $b = 2^k$ in (2.2), and using no precomputation, (*i.e.* starting with g and computing g^{2^j} using cyclic shifts), the algorithm of Theorem 1 takes $\lceil n/k \rceil + 2^k - 3$ multiplications. However, a slight variation of the algorithm takes the same number of multiplications as [16]. Given the conditions of Theorem 1, and h odd, the

following algorithm computes g^n with $m + (h+1)/2 - 2$ multiplications and at most m cyclic shifts.

$b \leftarrow 1$

$a \leftarrow 1$

`for` $d = h$ `to` 1 `by` -2

  `for each` i `such that` $a_i = d * 2^{j_i}$

    $b \leftarrow b * (g^{x_i})^{2^{j_i}}$

  $a \leftarrow a * b$.

`return` a.

If we precompute and store only g^{-1}, then this algorithm takes only $\lceil n/k \rceil + 2^{k-2} - 2$ multiplications. But Agnew, Mullin, and Vanstone [2] show that g^{-1} can be computed in $\lfloor \log_2 n \rfloor + \nu(n) - 2$ multiplications. Thus, without any precomputation, we require only $\lceil n/k \rceil + 2^{k-2} + \lfloor \log_2 n \rfloor + \nu(n) - 4$ multiplications. For $GF(2^{593})$, this improves Stinson's results from 129 multiplications to 124 multiplications.

For these fields, large speedups can be obtained with small amounts of precomputation and storage. Suppose we take $N = 2^n$ and $b = 2^k$ as above, but use a multiplier set $M = \{1, 2, \ldots, 2^k - 1\}$. Then any power g^{mb^j} for $m \in M$ can be calculated by shifting g^m by the appropriate amount, so we only need $\lceil n/k \rceil - 1$ multiplications to combine these terms. With multiple processors, this may easily be parallelized using binary fan-in multiplication as in [16].

For example, consider computations in $GF(2^{593})$. In [16], Stinson shows that exponentiation can be done in at most 129 multiplications with one processor, 77 rounds with four processors, and 11 rounds with 32 processors. These numbers can be improved significantly, as shown in Table 3.

6 A randomized approach

The above methods are designed to deal with the situation where a random exponent n is provided as input, and we wish to compute g^n, but for

processors	storage	worst-case time
1	32	98
1	64	84
1	128	74
2	32	49
4	32	26
8	32	15
16	32	10
32	32	8

Table 3: Parameters for $GF(2^{593})$.

cryptographic applications we might instead incorporate the random number generator directly into exponentiation process. For this section only we shall assume that g generates a group of order N. We shall appeal to the following theorem, which is a variant of a theorem by Erdős and Rényi in [6]. Let $a_1, \ldots, a_k$ be chosen uniformly from $\mathbb{Z}/N\mathbb{Z}$, and for any $b \in \mathbb{Z}/N\mathbb{Z}$, let $V(b) = V_{k,n}(b)$ be the number of representations of b of the form

$$\sum_{i=1}^{k} \epsilon_i a_i \equiv b \pmod{N},$$

where $\epsilon_i \in \{0, 1\}$, and $\sum_{i=1}^{k} \epsilon_i = m$. Let ϵ and δ be arbitrarily small positive numbers.

THEOREM 5. *If* $\binom{k}{m} > N^2(1/\varepsilon)^2(1/\delta)$, *then*

$$\text{Prob}\left(\max_{b<N} \left| V(b) - \binom{k}{m}/N \right| \leq \varepsilon \binom{k}{m}/N \right) > 1 - \delta.$$

In other words, when $\binom{k}{m}$ is big enough, the distribution can be made arbitrarily close to uniform with probability arbitrarily close to one. If $\binom{k}{m}$ is smaller than N^2 the distribution is likely to be nonuniform. The proof of the theorem is essentially the same as in [6].

Thus we can compute random powers of a fixed base g by the following method: a user stores random numbers $\{a_1, \ldots, a_k\} \in \mathbb{Z}/N\mathbb{Z}$, and generates

a random m-subset $a_{i_1}, \ldots, a_{i_m}$, where k and m are chosen to satisfy Theorem 5. Then the product $g^{a_{i_1}} \cdots g^{a_{i_m}}$ is approximately uniformly distributed in the group generated by g. By choosing δ and ε appropriately, the probability of the distribution being nonuniform enough to give an adversary any information can be made arbitrarily small.

Unfortunately, the size of k and m in Theorem 5 is too large to be useful. For instance, to achieve a uniform distribution in 45 multiplications for a 160-bit exponent, we would need to store at least 2,500 random values, as opposed to 76 in Table 1. In order to be competitive with the above methods, we need $\binom{k}{m}$ to be smaller, say $O(N)$ or $O(N \log N)$. This means that the distribution of the powers of g will be nonuniform. It's not clear what useful information an adversary could obtain from this distribution, but we have not been able to prove that it doesn't weaken the security of the system.

7 Lower Bounds

There are many approaches to the problem of calculating g^n efficiently, of which the schemes given in the previous sections are only a small part. Other number systems could be used, such as the Fibonacci number system (see [9, exercise 1.2.8.34]), where a number is represented as the sum of Fibonacci numbers. Other possibilities include representing numbers by sums of terms of other recurrent sequences, binomial coefficients, or arbitrary sets that happen to work well. These, and a number of other number systems, are given in [10]. For a given N and amount of storage, it seems difficult to prove that a scheme is optimal.

In this section we derive lower bounds for the number of multiplications required for a given value of N and a given amount of storage. We assume a model of computation in which the only way to compute g^x $(x \neq 0, 1)$ is to multiply g^{x_1} by g^{x_2} where $x_1 + x_2 = x$. Note that this is a fairly strong assumption, and in particular our lower bounds neglect any improvement that might result from knowledge of the order of the cyclic group generated by g.

Storing powers g^{rb^j} for r in a fixed set of multipliers, the optimal value of b is about $\log N/(\log\log N)^2$, for which $(1 + o(1)) \log N/\log\log N$ multiplications and $O(\log N/\log\log N)$ stored values are needed. The following theorem shows that asymptotically we cannot do better with a reasonable

amount of storage.

THEOREM 6. *If the number of stored values s satisfies $s \geq e \log N / \log s$, then more than $\log N / \log s - 3$ multiplications are required in the worst case.*

COROLLARY 1. *If the number of stored values is less than $\log^k N$ for a fixed $k \geq 1$, then the number of multiplications required is at least $(1/k + o(1))(\log N / \log \log N)$.*

COROLLARY 2. *If the number of stored values is $O(\log N / \log \log N)$, then the number of multiplications required is at least $(1+o(1))(\log N / \log \log N)$.*

The same methods used to prove this result will also be used to derive concrete lower bounds for a given amount of storage.

In our model, an exponentiation corresponds to a *vector addition chain*. A scalar addition chain is a sequence of numbers $1 = a_0, \ldots, a_l$ so that for each $0 < k \leq l$ there are indices $i, j < k$ with $a_k = a_i + a_j$. A vector addition chain generalizes scalar addition chains. Let the s unit vectors of $\mathbb{N}^s$ $(s \geq 0)$ be denoted $\mathbf{e}_0, \ldots, \mathbf{e}_{s-1}$. A vector addition chain is a sequence of vectors $\mathbf{e}_{s-1} = \mathbf{a}_{-s+1}, \ldots, \mathbf{e}_0 = \mathbf{a}_0, \ldots, \mathbf{a}_l$ so that for each $0 < k \leq l$ there are indices $i, j < k$ with $\mathbf{a}_k = \mathbf{a}_i + \mathbf{a}_j$. The length of the chain is l.

Any exponentiation operation which uses s stored values and l multiplications forms a vector addition chain, with $\mathbf{a}_{-s+1}, \ldots, \mathbf{a}_0$ representing the s stored values, and $\mathbf{a}_i$ for $i = 1, \ldots, l$ representing the multiplication of either stored values or the results of earlier multiplications.

Let $\mathcal{R}(s, l)$ be the set of vectors $\mathbf{v} \in \mathbb{N}^s$ contained in some vector addition chain of length l, together with $\mathbf{0}$, and let $R(s, l) = |\mathcal{R}(s, l)|$. To prove Theorem 6, we will show that $R(s, l) < N$ when $l \leq \log N / \log s - 3$. This will imply that for any set of s stored values, there is some power that cannot be computed with l multiplications.

Let $\mathcal{P}(s, l)$ be the vectors in $\mathcal{R}(s, l)$ for which all s entries are nonzero (*i.e.* all s stored values were used in the computation), and $P(s, l) = |\mathcal{P}(s, l)|$. For convenience, define $P(0, l) = 1$. Clearly $P(s, l) = 0$ for $s > l + 1$.

LEMMA 1. *For every $s, l \geq 0$,*

$$R(s, l) = \sum_{s'=0}^{s} \binom{s}{s'} P(s', l) \tag{7.1}$$

PROOF: Every vector in $\mathcal{R}(s,l)$ has some number s' of nonzero entries. For each $s' \le s$, there are $\binom{s}{s'}$ ways to choose which entries those are. □

Next we need a bound for $P(s,l)$. Let $\mathcal{C}(l)$ denote the set of scalar addition chains of length l, where the numbers in each chain are arranged in strictly increasing order, and $C(l) = |\mathcal{C}(l)|$.

The following lemma is from [14]:

LEMMA 2. *A vector* $\mathbf{v} = (v_1, \ldots, v_s)$ *is in* $\mathcal{P}(s,l)$ *if and only if there is a scalar addition chain of length* $l - s + 1$ *such that each* v_i *is in the chain.*

From this we have

$$P(s,l) \le C(l-s+1)(l-s+2)^s, \tag{7.2}$$

since from Lemma 2 any vector $\mathbf{v}$ in $\mathcal{P}(s,l)$ may be formed by taking a chain in $\mathcal{C}(l-s+1)$ and choosing any of the numbers $a_0, a_1, \ldots, a_{l-s+1}$ for each of the s entries of $\mathbf{v}$.

We now require an upper bound for $C(l)$, which with (7.1) and (7.2) will give a bound for $R(s,l)$. The first few values of $C(l)$ can be calculated by brute force, but the numbers grow very quickly.

l	$C(l)$	l	$C(l)$
0	1	8	73191
1	1	9	833597
2	2	10	10917343
3	6	11	162402263
4	25	12	2715430931
5	135	13	50576761471
6	913	14	1041203832858
7	7499	15	23529845888598

Table 4: The first few values of $C(l)$.

Suppose that $a_0, a_1, \ldots, a_l$ is an addition chain. Each a_i for positive i may be written as $a_{y_i} + a_{z_i}$, where $0 \le y_i \le z_i < i$. Thus, the addition chain corresponds to a set of l unordered pairs:

$$\{(y_i, z_i) \mid i = 1, \ldots, l \text{ and } 0 \le y_i \le z_i < i\}.$$

Any set of l such pairs corresponds to at most one addition chain with strictly increasing entries. If we take l pairs and arrange them to form a representation for an addition chain, at the ith step there may be several pairs (y, z) which have y and z less than i (if there are none, then the set does not correspond to any chain). If there is more than one such pair, we must choose the smallest one first, in order to make the numbers in the chain strictly increasing.

Some chains will correspond to more than one set of pairs, if some a_i may be formed as the sum of earlier a's in more than one way. Let the *canonical* representation be the unique set where, if there is more than one choice for an (y_i, z_i), the pair with the minimal y_i is chosen.

Let $\mathcal{F}(m, l)$ be the set of chains in $\mathcal{C}(l)$ where each y_i and z_i is less than m, and $F(m, l) = |\mathcal{F}(m, l)|$. Then $F(0, 0) = 1$, $F(0, l) = 0$ for $l > 0$, and $F(m, l) = F(l, l) = C(l)$ for $m > l$.

LEMMA 3.

$$F(m, l) \leq \sum_{j=0}^{l-(m-1)} \binom{m}{j} F(m-1, l-j). \tag{7.3}$$

PROOF: Let j be the number of pairs in the chain with $z_i = m - 1$. If these pairs are removed, the remaining pairs will determine a chain in $\mathcal{F}(m-1, l-j)$ (the remaining pairs still form a canonical sequence). In each of the j removed pairs, y_i is between 0 and $m - 1$. Each of the y_i's must be different, so there are $\binom{m}{j}$ possibilities. □

While Lemma 3 gives us a good bound in practice, we need a simple closed-form upper bound to prove Theorem 6.

LEMMA 4.

$$F(m, l) \leq (m + 1)^{l+1}$$

PROOF: The base case of $m = 0$ is trivial. Suppose the lemma is true for $m - 1$. We may assume $l \geq m$. From Lemma 3 we have

$$F(m, l) \leq \sum_{j \geq 0} \frac{m^j}{j!} m^{l+1-j} \leq e m^{l+1} < (m + 1)^{l+1}$$

□

From this we get $C(l) \leq (l+1)^{l+1}$. We are now ready to put these lemmas together and prove Theorem 6.

PROOF: Recall that we are assuming that $s \geq e \log N / \log s$. We will show that if $l \leq \log N / \log s - 3$, then $R(s,l) < N$.

$$\begin{aligned}
R(s,l) &= \sum_{s'=0}^{\min(s,l+1)} \binom{s}{s'} P(s',l) \\
&\leq \sum_{s'=0}^{l+1} \frac{s^{s'} e^{s'}}{s'^{s'}} C(l-s'+1)(l-s'+2)^{s'} \\
&\leq \sum_{s'=0}^{l+1} \frac{s^{s'} e^{s'}}{s'^{s'}} (l-s'+2)^{l+2} \\
&\leq \sum_{s'=0}^{l+1} \frac{s^{s'} e^{s'}}{s'^{s'}} (l+2)^{l+2} e^{-s'} \\
R(s,l) &\leq (l+2)^{l+2} \sum_{s'=0}^{l+1} \frac{s^{s'}}{s'^{s'}}
\end{aligned}$$

Because $s > e(l+1)$, the largest term in the summation occurs when $s' = l+1$.

$$\begin{aligned}
R(s,l) &\leq (l+2)^{l+3} \frac{s^{l+1}}{(l+1)^{l+1}} \\
R(s,l) &\leq (l+2)^2 e s^{l+1} \\
R(s,l) &< s^{l+3} \leq s^{\log N / \log s} = N
\end{aligned}$$

□

We can prove better bounds for particular values of N and s. We bound $F(m,l)$ by using (7.3), which with (7.1) and (7.2) give a bound for $R(s,l)$. To improve the bounds for $R(s,l)$, we computed $P(s,l)$ exactly for $s+5 \geq l$ using Lemma 2, and calculated $F(m,l)$ exactly for $l \leq 15$ by a depth-first search. Our results are summarized in Table 5. The quantity $M(N,s)$ represents the

number of multiplications achieved by the methods of this paper, $L(N, s)$ represents a lower bound for the number of multiplications required in the worst case. The worst case lower bounds are not much larger than average case lower bounds.

From these bounds we see that the methods of this paper are reasonably close to optimal for a method in which we compute g^x by multiplying together stored powers of g. For example, for a 512-bit exponent, if we store 650 values, then the method of this paper allows us to compute g^x in at most 93 multiplications, but we prove that any method using this much storage must use at least 66 multiplications in the worst case. Note also that the number of multiplications for the algorithm of Section 3 differ from the lower bounds we give by a factor of between 1.23 and 1.75.

Acknowledgment. We would like to thank Professor Tsutomu Matsumoto of Yokohama National University for informing us of reference [7], and for providing a partial translation.

References

[1] A Proposed Federal Information Processing Standard for Digital Signature Standard, *Federal Register*, Volume 56, No. 169, August 31, 1991, pp. 42980-42982.

[2] G.B. Agnew, R.C. Mullin, and S.A. Vanstone, Fast exponentiation in $GF(2^n)$, in *Advances in Cryptology–Eurocrypt '88, Lecture Notes in Computer Science*, Volume 330, Springer-Verlag, Berlin, 1988, pp. 251–255.

[3] J. Bos and M. Coster, Addition Chain Heuristics, in *Advances in Cryptology - Proceedings of Crypto '89, Lecture Notes in Computer Science*, Volume 435, Springer-Verlag, New York, 1990, pp. 400–407.

[4] W. Diffie and M. Hellman, New Directions in Cryptography, *IEEE Transactions on Information Theory* **22** (1976), pp. 472–492.

[5] E.F. Brickell and K.S. McCurley, An Interactive Identification Scheme Based on Discrete Logarithms and Factoring, to appear in *Journal of Cryptology*.

N	s	$M(N,s)$	$L(N,s)$
2^{512}	109	132	78
2^{512}	188	115	75
2^{512}	362	106	70
2^{512}	452	105	68
2^{512}	512	100	67
2^{512}	583	97	66
2^{512}	650	93	66
2^{512}	832	93	64
2^{512}	3096	87	54
2^{512}	5402	75	51
2^{512}	10880	64	47
2^{512}	16320	63	44
2^{160}	45	54	31
2^{160}	76	46	30
2^{160}	134	41	27
2^{160}	167	40	26
2^{160}	200	39	26
2^{160}	219	37	25
2^{160}	250	36	25
2^{160}	312	35	24
2^{160}	620	32	21
2^{160}	840	29	20
2^{160}	972	28	20
2^{160}	1134	27	19
2^{160}	1440	25	19
2^{160}	1748	24	18
2^{160}	1955	23	18
2^{160}	2244	22	17
2^{160}	2751	21	17

Table 5: Bounds for the number of multiplications required to compute g^x $(0 \leq x < N)$ when $s-1$ powers are precomputed. The algorithms for computing powers of g provide the upper bound $M(N,s)$. A vector addition chain counting argument provides the lower bound $L(N,s)$.

[6] P. Erdős and A. Rényi, Probabilistic methods in group theory, *Journal d'Analyse Math.*, **14** (1965), pp. 127–138.

[7] Ryo Fuji-Hara, Cipher Algorithms and Computational Complexity, Bit **17** (1985), 954–959 (in Japanese).

[8] J. von zur Gathen, Efficient exponentiation in finite fields, *Proceedings of the 32nd IEEE Symposium on the Foundations of Computer Science*, to appear.

[9] D.E. Knuth, *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, Addison-Wesley, Massachusetts, 1981.

[10] D.E. Knuth, *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*, Second Edition, Addison-Wesley, Massachusetts, 1981.

[11] R. Lidl and H. Niederreiter, *Finite Fields*, Cambridge University Press, London, 1987.

[12] D.W. Matula, Basic digit sets for radix representation, *Journal of the ACM*, **29** (1982), pp. 1131–1143.

[13] A. M. Odlyzko, "Discrete logarithms in finite fields and their cryptographic significance," *Advances in Cryptology* (Proceedings of Eurocrypt 84), *Lecture Notes in Computer Science* **209**, Springer-Verlag, NY, pp. 224–314.

[14] Jorge Olivos, "On Vectorial Addition Chains", *Journal of Algorithms*, **2** (1981) pp. 13–21.

[15] C.P. Schnorr, Efficient signature generation by smart cards, to appear in *Journal of Cryptology.*

[16] D.R. Stinson, Some observations on parallel algorithms for fast exponentiation in $GF(2^n)$, *Siam. J. Comput.*, **19**, (1990), pp. 711-717.

[17] J.S. Vitter and P. Flajolet, Average-case analysis of algorithms and data structures, in *Handbook of Theoretical Computer Science*, ed. J. van Leeuwen, Elsevier, Amsterdam, 1990, pp. 431–524.