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

November 30, 1995

Abstract:

In several cryptographic systems, a fixed element g of a group of order N 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 $O(\log N/\log\log N)$ multiplications. We show that this method is asymptotically optimal given polynomial storage, 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 $O(\log N/\log^2\log N)$ processors.

Keywords: Exponentiation, Cryptography.

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

---

*This research was supported by the U.S. Department of Energy under contract number DE-AC04-76DP00789.

†Current address: Center for Communications Research, San Diego, CA 92121

‡Supported in part by an ONR-NDSEG fellowship.

§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 [11, 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 [11, 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. Any addition chain will require at least $\lceil \log n \rceil$ multiplications, since this many doublings are needed to get to a number of size n.

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 512-bit exponent, we expect the binary algorithm to take an average of about 765 multiplications. Results in [4] report that addition chains of length around 605 are relatively easy to compute, resulting in a 21% improvement. Note that no addition chain could do better than 512 multiplications.

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. [1],[5],[6],[17]), 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 [17] uses N of 140 bits, the DSS scheme [1] uses N of 160 bits, and the Brickell-McCurley scheme [6] has N of 512 bits. We will assume that operations other than multiplications mod q will use negligible time.

One simple method (see [8]) 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 by multiplying together the powers corresponding to nonzero digits in the binary representation of n. This reduces the work to $\lceil \log N \rceil - 1$ multiplications in worst case, and $\lceil \log N \rceil / 2 - 1$ on average, at a cost of storing $\lceil \log N \rceil$ powers. In Section 3 we show that we can do much better:

THEOREM 1. *If $O(\log N / \log\log N)$ powers are precomputed, then we may compute g^n $(0 \le n < N)$ using only $(1 + o(1)) \log N / \log\log N$ multiplications.*

The method works for any group, and in Section 4 we discuss its use in $GF(p^m)$, where p is a small prime and m is large. In Section 5 we discuss parallelizing the method. Finally, in Section 6 we give lower bounds which show that Theorem 1 is asymptotically optimal:

THEOREM 2. *If the number of stored values is bounded by a polynomial in $\log N$, then the number of multiplications required to compute a power is $\Omega(\log N / \log\log N)$. If the number of stored values is $O(\log N / \log\log N)$, then $(1 + o(1))(\log N / \log\log N)$ multiplications are required.*

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 $\le 3\lceil \log N \rceil / 2$ multiplications. The simple method mentioned in the introduction of storing powers g^{2^i} reduces this to $\lceil \log N \rceil - 1$ in worst case and $\lceil \log N \rceil / 2 - 1$ on average, at a cost of storing $\lceil \log N \rceil$ powers.

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

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

Typically the x_i will be powers of a base b, and (2.1) will be the base b representation of n. 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.

Lemma 1. *Suppose $n = \sum_{i=0}^{m-1} a_i x_i$, where $0 \leq a_i \leq h$, and g^{x_i} has been precomputed for each $0 \leq i < m$. If $n \neq 0$ 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. There are m terms in the decomposition $n = \sum_i a_i x_i$, 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. (If $n = 0$ then $h = m = 0$, and 0 rather than -2 multiplications are required. But $n \neq 0$ implies $h \neq 0$ and $m \neq 0$, and the free multiplications do exist.) $\square$

Embodied in the algorithm is a method for computing the product $\prod_{d=1}^{h} c_d^d$ in at most $2h - 2$ multiplications. Given the c_d's, we can argue that in the absence of any relations between them, 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.

Proof of Theorem 1: For any $b > 1$, we may represent N base b with at most $m = \lceil \log_b N \rceil$ digits. Precompute g^{b^k}, for $k = 1, \ldots, \lceil \log_b N \rceil - 1$. Using Lemma 1, we may then compute g^n in at most $\lceil \log_b N \rceil + b - 3$ multiplications. For large N, the optimal value of b is about $\log N / \log^2 \log N$, which gives us Theorem 1. $\square$

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 127.9 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

In the last section, (2.1) was only used as a base b representation. There are many other number systems that could be used, some of which give better results in practice (although they are the same asymptotically). In this section we will explore some of these alternative number systems.

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

$$a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b + a_0, \tag{3.1}$$

for some k where each $a_i \in D$. This definition differs from that in [13] 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 (3.1) is almost exactly the same difficulty as finding the (ordinary) base b representation. The algorithm for finding such a representation was published by Matula [13], and a particularly simple description was later given in [11, Exercise 4.1.19].

In searching for good basic digit sets, we can make use of the following result of Matula [13], which provides a very efficient algorithm for determining if a set is basic.

THEOREM 3. *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* (3.1) *for each i with*

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

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

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

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

$$g^n = \prod_{k=1}^{h} \left(\prod_{i:k_i=k} g^{m_i b^i} \right)^k = \prod_{k=1}^{h} c_k^k \tag{3.2}$$

In this notation, the method of the last section has parameters $M = \{1\}$, $h = b - 1$.

The next four theorems give other examples of basic digit sets which give efficient exponentiation schemes. As our first example of this approach, if b is an integer, then every integer n such that $|n| \le (b^m - 1)/2$ may be represented as $\sum_{i=0}^{m-1} a_i b^i$, where each $a_i \in (-\lceil (b-1)/2 \rceil, \lceil (b-1)/2 \rceil]$, by Theorem 3. If the powers $g^{\pm 1}, g^{\pm b}, \ldots, g^{\pm b^{m-1}}$ are precomputed, then we compute

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

THEOREM 4. $M = \{\pm 1\}$, $h = \lceil (b-1)/2 \rceil$, *is a basic digit set.*

This digit set is particularly useful when inverses are easy to compute, as for elliptic curves (see [14]). When $b = 2$, there is a unique representation with no two adjacent nonzeros, which reduces the worst case number of multiplications to $\lceil \log N \rceil / 2$ and the average to $\lceil \log N \rceil / 3$ (see [3]).

By taking a slightly larger multiplier set, we may further reduce h.

THEOREM 5. *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 3 applies trivially, since $d_{\min} = -2\lfloor b/3 \rfloor$ and $d_{\max} = 2\lfloor b/3 \rfloor$. □

An alternative approach is to take large multiplier sets and small values of h. For instance, taking $h = 2$, let

$$M_2 = \{d \mid 1 \le d < b\ ,\ \omega_2(d) \equiv 0 \pmod 2\}, \tag{3.3}$$

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 $d \in M_2$. This shows:

THEOREM 6. *$M = M_2$, $h = 2$, is a basic digit set.*

Continuing this line of reasoning, we can take

$$M_3 = \{d \mid 1 \le d < b,\, \omega_2(d) + \omega_3(d) \equiv 0 \pmod{2}\}. \tag{3.4}$$

Each integer d between 1 and $b-1$ is in M_3, $2M_3$, or $3M_3$.

THEOREM 7. *$M = M_3$, $h = 3$, is a basic digit set.*

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 8. *Let $D = D(M, h)$ be a basic digit set modulo b such that $\max_{a \in M} |a| \le b - 1$, and such that $\{-1, 1\} \subset M$. For every $n < b^m$, we can find a sequence of digits $d_i \in D$ for $0 \le i \le m$ 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 $n_{j+1} = (n_j - d_j)/b$. If $n_j \bmod b$ is a random number modulo b, independent of the less significant base-b digits of n, then $d_j = 0$ with probability $1/b$, independent of the previous digits.

It is easy to verify that $n = n_{j+1} b^{j+1} + \sum_{i=0}^{j} d_i b^i$ for $0 \le j \le m$. 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 $0 \le a_i < b$ for $0 \le i \le m-1$. We shall prove by induction on j that $n_j = c_j + \sum_{i=j}^{m-1} a_i b^{i-j}$ for some c_j with $|c_j| \le h$ for $0 \le j \le m$. Clearly we have $c_0 = 0$. Note that

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

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

$$|c_{j+1}| < (h + b + h(b-1))/b = h + 1.$$

Since c_{j+1} is an integer, it follows that $|c_{j+1}| \le h$. 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 $k = \pm 1$.

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.

These tables also give an upper bound on the expected number of multiplications, based on the assumption that n is a random number satisfying $0 \leq n < N$. When N is a power of b, it follows from the first paragraph of the proof of Theorem 8 that each digit of n has probability $1/b$ of being zero, so on average fewer than the worst-case number of multiplications are required. Even when N is not a power of b, it can be shown that the expected number of zero digits is at least

$$\frac{\lfloor \log_b N \rfloor}{b} - \frac{1/2}{b-1}.$$

For redundant digit sets there are often other algorithms which have a higher probability of zeros [3].

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 (3.2), 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 37.9. 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 (3.2).

4 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 [15]). A normal basis has the form $\{\beta, \beta^p, \beta^{p^2} \ldots, \beta^{p^{n-1}}\}$ (see, for example, [12] 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], [9], and [18].

It has been shown [18] 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), our system takes the same number of multiplications (using shifts to do the squarings in Lemma 1). If we precompute and store only g^{-1}, then

				time		
b	M	h	storage	expected	worst-case	lower bnd
12	$\{1\}$	11	45	50.35	54	31
19	$\{\pm 1\}$	9	76	43.05	45	30
29	$\{\pm 1, \pm 2\}$	9	134	39.90	41	27
29	$\{\pm 1, -2, 9, 10\}$	8	167	38.90	40	26
29	$\{\pm 1, \pm 2, \pm 9\}$	7	200	37.90	39	26
36	$\{\pm 1, 9, \pm 14, \pm 17\}$	7	219	36.17	37	25
36	$\{\pm 1, \pm 3, \pm 11, \pm 13\}$	6	250	35.17	36	25
36	$\{\pm 1, \pm 3, \pm 6, \pm 11, \pm 13\}$	5	312	34.17	35	24
36	M_3	3	620	31.17	32	21
53	M_3	3	840	28.49	29	20
64	M_3	3	972	27.59	28	20
64	M_2	2	1134	26.59	27	19
102	M_3	3	1440	24.77	25	19
128	M_3	3	1748	23.83	24	18
128	M_2	2	1955	22.83	23	18
155	M_2	2	2244	21.86	22	17
256	M_2	2	2751	20.92	21	17

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. Time is measured in multiplications, and storage is measured in group elements. The entries under "expected time" are rigorous upper bounds on the expected time given random inputs. Included also are lower bounds for the worst-case number of multiplications given the amount of storage used. The worst case lower bounds, derived in Section 6, are not much larger than average case lower bounds.

				time		
b	M	h	storage	expected	worst-case	lower bnd
26	$\{1\}$	25	109	127.85	132	78
45	$\{\pm1\}$	22	188	111.94	114	75
53	$\{\pm1, \pm2\}$	17	362	104.32	106	70
53	$\{\pm1, -9, 18, 27\}$	16	452	103.32	105	68
67	$\{\pm1, \pm2, \pm23\}$	16	512	98.75	100	67
75	$\{\pm1, 5, 14, -16, 29, -31\}$	15	583	95.91	97	66
81	$\{\pm1, \pm3, \pm26, \pm28\}$	13	650	92.01	93	66
72	$\{\pm1, \pm3, \pm4, \pm23, \pm25\}$	11	832	91.86	93	64
64	M_3	3	3096	85.67	87	54
122	M_3	3	5402	74.40	75	51
256	M_2	2	10880	63.75	64	47
256	$\{1, 2, \ldots, 255\}$	1	16320	62.75	63	44

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.

our system 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, our system requires 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 [18].

For example, consider computations in $GF(2^{593})$. In [18], 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.

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 $GF(2^{593})$.

5 Parallelizing the algorithm

We give two parallel algorithms for computing g^n. They both run in $O(\log\log N)$ time. The first is randomized and uses $O(\log N/\log^2\log N)$ processors, while the second is deterministic and uses $O(\log N/\log\log N)$ processors.

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 $c_d = \prod_{j:a_j=d} g^{b^j}$ 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.

To simplify the run-time analysis, let's take the multiplier set M to be $\{1\}$ and $h = b - 1$. Then the digits a_j of n will be approximately uniformly distributed, so that the time for step 2 is equivalent to the maximum bucket occupancy problem: given m balls randomly distributed in h buckets, what is the expected maximum bucket occupancy? This is discussed in [19], in connection with analysis of hashing algorithms. Taking b to be $O(\log N/\log^2\log N)$, so $m/h = \Theta(\log\log N)$, the expected maximum value is $O(\log\log N)$.

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 [18]). This takes $\log h$ rounds.

Therefore, taking $h = O(\log N/\log^2\log N)$, we may calculate powers in $O(\log\log N)$ expected rounds with $O(\log N/\log^2\log N)$ processors, given random inputs. If we wish the algorithm to have no bad inputs, then on input n randomly choose $x < n$, compute g^x and g^{n-x}, and multiply them.

THEOREM 9. *With $O(\log N/\log^2\log N)$ processors, we may compute powers in $O(\log\log N)$ expected time.*

One 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 use distributed memory, and is deterministic.

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 gives

THEOREM 10. *With $O(\log N/\log\log N)$ processors, we may compute powers in $O(\log\log N)$ time.*

6 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 [10, 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 [11]. 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.

In Section 2 we saw how to compute powers using $(1+o(1)) \log N / \log\log N$ multiplications when $O(\log N / \log\log N)$ values have been precomputed. Theorem 2 asserted that asymptotically we cannot do better with a reasonable amount of storage. The proof of this theorem follows directly from the following lemma.

LEMMA 2. *If the number of stored values s is not too small (specifically $s \geq e \log N / \log s$, where $e = 2.718\ldots$), then more than $\log N / \log s - 3$ multiplications are required in the worst case.*

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 Lemma 2, 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 3. *For every $s, l \geq 0$,*

$$R(s,l) = \sum_{s'=0}^{s} \binom{s}{s'} P(s',l) \tag{6.1}$$

PROOF: Every vector in $\mathcal{R}(s,l)$ has some number s' of nonzero entries. For each $s' \leq s$, 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 [16]:

LEMMA 4. *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

$$P(s,l) \leq C(l-s+1)(l-s+2)^s, \tag{6.2}$$

since from Lemma 4 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 (6.1) and (6.2) 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.

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 $0 \leq y_i \leq z_i < i$. Thus, the addition chain corresponds to a set of l unordered pairs:

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

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 $C(l)$.

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 5.

$$F(m, l) \leq \sum_{j=0}^{l-(m-1)} \binom{m}{j} F(m-1, l-j). \tag{6.3}$$

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 5 gives us a good bound in practice, we need a simple closed-form upper bound to prove Lemma 2.

LEMMA 6.

$$F(m,l) \le (m+1)^l$$

PROOF: The base case of $m = 0$ is trivial. Suppose the lemma is true for $m-1$. We may assume $l \ge m$. From Lemma 5 we have

$$F(m,l) \le \sum_{j\ge 0} \binom{m}{j} m^{l-j} = m^l(1+1/m)^m \le (m+1)^l$$

□

From this we get $C(l) \le (l+1)^l$. We are now ready to put these lemmas together and prove Lemma 2.

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

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

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

$$R(s,l) \le (l+2)^{l+3} \frac{s^{l+1}}{(l+1)^{l+1}}$$

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

$\square$

We can prove better bounds for particular values of N and s. We bound $F(m,l)$ by using (6.3), which with (6.1) and (6.2) give a bound for $R(s,l)$. To improve the bounds for $R(s,l)$, we computed $P(s,l)$ exactly for $s+5 \ge l$ using Lemma 4, and calculated $F(m,l)$ exactly for $l \le 15$ by a depth-first search. Our results are summarized included in Tables 3 and 3. 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 [8], 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] S. Arno and F. S. Wheeler, Signed digit representations of minimal hamming weight, IEEE Transactions on Computers, **42** (1993), pp. 1007–1010.

[4] 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.

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

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

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

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

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

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

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

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

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

[14] F. Morain and J. Olivos, Speeding up the computations on an elliptic curve using addition-subtraction chains, *Inform. Theor. Appl.*, **24** (1990), pp. 531–543.

[15] 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.

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

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

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

[19] 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.