Fast Exponentiation with Precomputation * (Extended Abstract)**

Ernest F. Brickell[1], Daniel M. Gordon[2],
Kevin S. McCurley[1], and David B. Wilson[3]

[1] Division 1423, Sandia National Laboratories, Albuquerque, NM 87185
[2] Department of Computer Science, University of Georgia, Athens, GA 30602
[3] Department of Mathematics, M.I.T., Cambridge, MA 02139

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 $O(\log N/\log\log N)$ group multiplications. We also show how these methods can be parallelized, to compute powers in $O(\log\log N)$ group multiplications with $O(\log N/\log\log N)$ processors.

1 Introduction

The problem of efficiently evaluating powers has been studied by many people (see [6, Sect. 4.6.4] for an extensive survey). One standard method is to define an *addition chain*. Let $l(n)$ denote the length of the shortest addition chain for an exponent n (and the smallest number of multiplications possible by this approach). Then it is known that

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

where logs are to base 2 and $\nu(n)$ is the number of ones in the binary representation of 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. We shall consider a slightly different problem in this paper. For many cryptosystems (e.g. [3],[4],[8],[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.

* 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.

We will assume that g is an element of a group such as $\mathbb{Z}/q\mathbb{Z}$, where q is a large integer (say 512 bits). We address the problem of repeatedly calculating powers of g up to g^N, where N is also large. In the Schnorr scheme [8], N is about 140 bits, the DSS scheme uses N of 160 bits, and the Brickell-McCurley scheme [4] uses N of 512 bits. Our results will apply to any group, so that the speedups work as well in an elliptic curve group as for modular exponentiation. Specialized results can also be derived from this approach for the special case of $GF(p^k)$, but we will defer these to the full paper. We will assume that operations other than multiplications in the group 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 (see [6], page 442), we would expect to perform $765 + 3/2^{512}$ modular multiplications on average and 1022 multiplications in the worst case, using storage of at least 64 bytes. Results in [2] report that addition chains of length around 608 can be computed, resulting in a 21% improvement over the average number for the binary method. It follows from (1) that addition chains cannot do better than 512 multiplications for a 512 bit exponent. 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 about a ten-fold speedup in the worst case over the square and multiply method. Even very small amounts of storage can produce dramatic speedups.

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 $2\lceil \log N \rceil - 2$ 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 [5]) is to precompute the set $\{g^{2^i} | i = 1, \ldots, \lceil \log N \rceil - 1\}$. Then g^n may be computed in $\nu(n) - 1$ multiplications, using $\lceil \log N \rceil$ 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 instead precompute and store $g^{x_0}, \ldots, g^{x_{m-1}}$ for some integers $x_0, \ldots, x_{m-1}$. If we are then able to find a decomposition

$$n = \sum_{i=0}^{m-1} a_i x_i, \tag{2}$$

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

$$g^n = \prod_{d=1}^{h} c_d^d, \tag{3}$$

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

If (3) 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, (3) 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 ← 1
a ← 1
for d = h to 1 by −1
      for each i such that a_i = d
            b ← b * g^{x_i}
      a ← 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 $n \ne 0$. 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 (3) 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 109 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). 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 $b > 1$, 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

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

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}, $1 \leq d < b$, $0 \leq i \leq \lceil \log_b N \rceil - 1$, 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 $M_2 = \{d \mid 1 \leq d < b \ , \ \omega_2(d) \equiv 0$

(mod 2)}, 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, $\{g^{db^i} | d \in M_2\}$. Then for $1 \leq a_i < b$, $g^{a_i b^i} = g^{db^i}$ 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 $M_3 = \{d \mid 1 \leq d < b,\ \omega_2(d) + \omega_3(d) \equiv 0 \pmod 2\}$. 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 without such a huge increase in the amount of storage. Call a set of integers D a *basic digit set* for base b if any integer can be represented in base b using digits selected from the set D.

Before we examine the problem of finding basic digit sets for our problem, we should first remark that the difficulty of finding a representation using digits from D is almost exactly the same difficulty as finding the (ordinary) base b representation. The algorithm for finding such a representation was published by Matula [7], and a particularly simple description was later given in [6, Exercise 4.1.19].

In searching for good basic digit sets, we can make use of the following result of Matula [7], which provides a very efficient algorithm for determining if a set is basic.

Theorem 2. *Suppose that D is a complete residue system modulo b. Let $d_{\min} = \min\{s|s \in D\}$ and $d_{\max} = \max\{s|s \in D\}$. Then D is a basic digit set for base b if there are representations for each i with*

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

using digits from D.

In the method 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

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

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

$$g^n = \prod_{k=1}^{h} \left(\prod_{k_i = k} g^{m_i b^i} \right)^k = \prod_{k=1}^{h} c_k^k \ . \tag{4}$$

Another class of multiplier sets is provided by the following:

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

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.

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

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
36	$\{\pm 1, 9, \pm 14, \pm 17\}$	7	36.11	37	219
36	M_3	3	31.14	32	620
64	M_2	2	26.58	27	1134
128	M_3	3	23.82	24	1748
256	M_2	2	20.92	21	2751

Table 2. Selected parameters for a 512-bit exponent ($N = 2^{512}$). By comparison, the binary method requires about 765 multiplications on average and 1022 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
67	$\{\pm 1, \pm 2, \pm 23\}$	16	98.72	100	512
64	M_3	3	85.66	87	3096
122	M_3	3	74.39	75	5402
256	M_2	2	63.75	64	10880

For some of the lines of each table, the expected times are actually upper bounds for the expected time. For the others, the expected times 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.

For a given N and amount of storage, it seems difficult to prove that a scheme is optimal. However, we can show that the above schemes are asymptotically optimal. 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 we cannot do better with a reasonable amount of storage.

Theorem 4. *If the number of stored values is less than* $\log^k N$ *for* $k \geq 1$*, then the number of multiplications required is at least* $(1/k + o(1))(\log N/\log\log N)$*.*

4 Parallelizing the algorithm

The first method for computing a power g^n that we presented in Sect. 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 $c_d = \prod_{\substack{j=0 \\ a_j = d}}^{m-1} g^{b^j}$ 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 $k+1$ balls randomly distributed in h buckets, what is the expected maximum bucket occupancy? This is discussed in [10], in connection with analysis of hashing algorithms. Taking b and h to be $O(\log N/\log\log N)$, so $(k+1)/h = \Theta(1)$, the expected value is

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

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 [9]). This takes $\log h$ time.

Therefore, taking $h = O(\log N/\log\log N)$, we may calculate powers in time $O(\log\log N)$ with $O(\log N/\log\log N)$ processors. For example, storing only powers of b, we may compute powers for a 140-bit exponent in the time necessary for 13 multiplications using 15 processors, taking $b = 16$ and $M = \{1\}$. For a 512-bit exponent, we can compute powers with 27 processors in the time for 17 multiplications, 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 k 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 $2 \log h$ time. Then the processors multiply together their results using binary fan-in multiplication to get g^n. The total time spent is at most $2 \log h + \log k$, which is again $O(\log \log N)$ time 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 $\log \log N$ time. We could compute powers for a 512-bit exponent with 512 processors in the time for 9 multiplications, and for a 140-bit exponent with 140 processors in the time for 8 multiplications. Taking a larger base reduces the number of processors, but increases the time.

Acknowledgment. We would like to thank Professor Tsutomu Matsumoto of Yokohama National University for informing us of reference [5], 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. 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.
3. W. Diffie and M. Hellman, New Directions in Cryptography, *IEEE Transactions on Information Theory* **22** (1976), 472-492.
4. E.F. Brickell and K.S. McCurley, An Interactive Identification Scheme Based on Discrete Logarithms and Factoring, to appear in *Journal of Cryptology*.
5. Ryo Fuji-Hara, Cipher Algorithms and Computational Complexity, Bit **17** (1985), 954–959 (in Japanese).
6. D.E. Knuth, *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*, Second Edition, Addison-Wesley, Massachusetts, 1981.
7. D.W. Matula, Basic digit sets for radix representation, *Journal of the ACM*, **29** (1982), pp. 1131-1143.
8. C.P. Schnorr, Efficient signature generation by smart cards, to appear in *Journal of Cryptology*.
9. D.R. Stinson, Some observations on parallel algorithms for fast exponentiation in $GF(2^n)$, *Siam. J. Comput.*, **19**, (1990), pp. 711-717.
10. 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.