# A survey of fast exponentiation methods

Daniel M. Gordon
Center for Communications Research
4320 Westerra Court
San Diego, CA 92121

December 30, 1997

**Abstract**

Public-key cryptographic systems often involve raising elements of some group (e.g. $GF(2^n)$, $\mathbf{Z}/N\mathbf{Z}$, or elliptic curves) to large powers. An important question is how fast this exponentiation can be done, which often determines whether a given system is practical. The best method for exponentiation depends strongly on the group being used, the hardware the system is implemented on, and whether one element is being raised repeatedly to different powers, different elements are raised to a fixed power, or both powers and group elements vary.

This problem has received much attention, but the results are scattered through the literature. In this paper we survey the known methods for fast exponentiation, examining their relative strengths and weaknesses.

## 1 Introduction

Exponentiation is a fundamental operation in computational number theory. For example, primality tests based on Fermat's Little Theorem that

$$a^{p-1} \equiv 1 \pmod{p}$$

for $p$ prime and $a$ relatively prime to $p$ are implemented in most computer algebra systems [23].

Another application in which exponentiation is heavily used is cryptography. In the RSA cryptosystem [25], encryption and decryption are accomplished by exponentiation in $\mathbf{Z}/N\mathbf{Z}$, for $N = pq$ the product of two large primes. For Diffie-Hellman key exchange [9], exponentiation is done modulo a prime $p$. Its difficulty is based on exponentiation being easy, and its inverse, the discrete logarithm problem, being difficult.

Exponentiation can be time-consuming, and is often the dominant part of algorithms for key exchange, electronic signatures, and authentication. A natural question is: how fast can exponentiation be done?

The answer is dependent on the algorithm being used and the implementation. For example, in Diffie-Hellman key exchange, a fixed number is raised to different powers, so precomputing some powers can save time, at the expense of more storage. In other systems, such as RSA, different numbers may be raised to a fixed power, so more work might be spent on finding a good addition chain for that power. If the group being used is $GF(2^n)$, instead of $\mathbf{Z}/N\mathbf{Z}$, squaring can be done cheaply, which reduces the work greatly.

Because of these variations, many papers concentrate on one method, and give a good algorithm for one situation. A person trying to pick the best method for a particular situation has to sift through a large number of choices, none of which may be ideal for the given problem.

In this paper we attempt to list all the known methods for speeding exponentiation, and which situations they are applicable to. We will always use $N$ as the order of the group being used, and $n = \lceil \log N \rceil$, where log denotes the base 2 logarithm. Unless otherwise noted, we will assume that exponents may be any positive integer less than $N$. Following standard practice, we will talk about the general groups multiplicatively, but elliptic curve groups will be written additively. The two viewpoints are equivalent, and hopefully will not cause too much confusion.

We will not deal here with the time required to perform individual multiplications. Alternative representations of integers modulo $N$ can often result in significant improvements. One well-known technique is Montgomery reduction [20], which is often used in practice. Hong, Oh and Yoon [12] recently gave algorithms which run faster than Montgomery's. Bernstein [4] has suggested using an explicit form of the Chinese Remainder Theorem to represent numbers modulo $N$ as a set of single-precision numbers.

## 1.1 Addition Chains

The basic question is: what is the fewest number of multiplications necessary to compute $g^r$, given that the only operation permitted is multiplying two already-computed powers? This is equivalent to the question: what is the length of the shortest addition chain for $r$?

An *addition chain for* $r$ is a list of positive integers

$$a_1 = 1, a_2, \ldots, a_l = r,$$

such that for each $i > 1$, there is some $j$ and $k$ with $1 \le j \le k < i$ and $a_i = a_j + a_k$. A short addition chain for $r$ gives a fast algorithm for computing $g^r$: compute $g^{a_2}, g^{a_3}, \ldots, g^{a_{l-1}}, g^r$. See Knuth [13] for an excellent introduction to addition chains.

Let $l(r)$ be the length of the shortest addition chain for $r$. The exact value of $l(r)$ is known only for relatively small values of $r$. It is known that, for $r$ large,

$$l(r) = \log r + (1 + o(1)) \frac{\log r}{\log \log r}. \tag{1}$$

The lower bound was shown by Erdős [11] using a counting argument, and the upper bound is given by the $m$-ary method below.

Finding the best addition chain is impractical, but we can find near-optimal ones. We will give several efficient algorithms in the next section which produce reasonably good addition chains.

## 1.2 Addition-Subtraction Chains

One way to reduce the length of an addition chain is to allow other operations, such as subtraction. For example, the shortest addition chain for 31 is:

$$1, 2, 3, 5, 10, 11, 21, 31,$$

but if subtraction is allowed we get the shorter chain:

$$1, 2, 4, 8, 16, 32, 31.$$

The idea of addition-subtraction chains has been around for a long time, but they did not seem practical for exponentiation, since division is generally more expensive to implement than multiplication.

Morain and Olivos [21] observed that addition-subtraction chains can be very useful for elliptic curves, on which the inverse of a point can be computed for free. For curves $y^2 = x^3 + Ax + B$ over $GF(p)$ with $p > 3$, the inverse of $(x, y)$ is $(x, -y)$. For $y^2 + xy = x^3 + Ax^2 + B$ over $GF(2^n)$, the inverse is $(x, x + y)$. Most addition chain algorithms, such as the binary method and window methods given in later sections, can be generalized to addition-subtraction chains with some savings.

## 1.3 Addition Sequences and Vector Addition Chains

There are two generalizations of addition chains which have important applications, and turn out to be closely related.

An *addition sequence for* $r_1, r_2, \ldots, r_t$ is an addition chain

$$a_1 = 1, a_2, \ldots, a_l$$

which contains $r_1, \ldots r_t$.

Addition sequences are used when one $g$ is to be raised to multiple powers. They can also be used to speed methods such as the window methods given in Section 3. In those methods, a number of powers $g^{r_1}, \ldots g^{r_t}$ are computed first. If they are all small, then just computing $g^2, g^3, \ldots g^{r_t}$ may be fast enough, but if the $r_i$ are spaced far apart, an addition sequence can be much faster.

Yao [32] showed that the minimal length $l(r_1, \ldots, r_t)$ of an addition sequence for $r_1, \ldots, r_t$ is

$$l(r_1, \ldots r_t) = \log r + (t + o(1)) \frac{\log r}{\log \log r}, \tag{2}$$

where $r = \max\{r_1, \ldots, r_t\}$. Bos and Coster [5] give some heuristics for constructing good addition sequences.

A *vector addition chain* is a sequence of elements $v_i$ in $\mathbf{N}^t$ such that $v_i = e_i$ for $1 \leq i \leq t$, and $v_i = v_j + v_k$ for $j \leq k < i$. For example, a vector addition chain for [7,15,23] is:

$$\begin{aligned} &[0,0,1],[0,1,0],[1,0,0],[0,1,1],[1,1,1],[0,1,2],[1,2,3], \\ &\quad [1,3,5],[2,4,6],[3,7,11],[4,8,12],[7,15,23]. \end{aligned} \tag{3}$$

Vector addition chains may be used to compute multinomial powers $g_1^{r_1} g_2^{r_2} \cdots g_t^{r_t}$. Let $l([r_1, \ldots, r_t])$ be the shortest vector addition chain for $[r_1, \ldots, r_t]$. Olivos [22] showed that problems of finding good vector addition chains and addition sequences are equivalent:

**Theorem 1**

$$l([r_1, \ldots, r_t]) = l(r_1, \ldots, r_t) + (t-1).$$

He does this by giving mappings from addition sequences to vector addition chains, and vice versa. For example, the addition sequence he gets from (3) is

$$1, 2, 4, 6, 8, \underline{7}, \underline{15}, \underline{23},$$

while the sequence that maps to (3) is

$$1, 2, 3, 4, \underline{7}, 8, \underline{15}, \underline{23},$$

Doney, Leong and Sethi [10] showed that the problem of finding the shortest addition sequence is NP-complete.

# 2 Basic Methods

## 2.1 Binary Method

This method is also known as the "square and multiply" method. It is over 2000 years old; Knuth [13] discusses its history and gives references. The basic idea is to compute $g^r$ using the binary expansion of $r$. Let

$$r = \sum_{i=0}^{l} c_i 2^i.$$

Then the following algorithm will compute $g^r$:

$a \leftarrow 1$

`for` $d = l$ `to` $0$ `by` $-1$

  $a \leftarrow a * a$

  `if` $c_d = 1$ `then` $a \leftarrow a * g$

`return` $a$.

At each step of the `for` loop $a$ is equal to $g^s$, where the binary representation of $s$ is a prefix of the binary representation of $r$. Squaring $a$ has the effect of doubling $s$, and multiplying by $g$ puts a one in the last digit, if the corresponding bit $c_i$ is one. Knuth [13] gives a right-to-left version of the algorithm, which has the advantage of not needing to know $l$ ahead of time.

This algorithm takes $2\lfloor \log r \rfloor$ multiplies in worst case, and $3\lfloor \log r \rfloor/2$ on average. Since $\lfloor \log r \rfloor$ is a lower bound for the number of multiplies needed to do a single exponentiation in a general group, this method is often good enough. The rest of the paper will be concerned with improving the worst-case and average-case constant factors, and taking advantage of special conditions to get past the $\lfloor \log r \rfloor$ barrier.

## 2.2 $m$-ary method

The above method has an obvious generalization: use a base larger than two. Let

$$r = \sum_{i=0}^{l} c_i m^i.$$

The $m$-ary method computes $g^r$ using this representation:

> Compute $g^2, g^3, \ldots g^{m-1}$.
>
> $a \leftarrow 1$
>
> `for` $d = l$ `to` $0$ `by` $-1$
>
> $\quad a \leftarrow a^m$
>
> $\quad a \leftarrow a * g^{c_d}$
>
> `return` $a$.

This method is particularly attractive if $m = 2^k$, so that raising $a$ to the $m$th power only involves $k$ squarings. In that case, the number of multiplies is at most $2^k - 2 + (1 + 1/k)\lfloor \log r \rfloor$: $2^k - 2$ multiplies for the precomputation, $\lfloor \log r \rfloor$ squarings, and at most $\lfloor \log r \rfloor / k$ multiplies (on average fewer, since some of the $c_i$ will be zero).

Taking

$$k = \log \log r - 2 \log \log \log r,$$

gives the upper bound in (1).

## 2.3 Redundant Number Systems

As mentioned in Section 1.2, inverses can be computed for free on elliptic curves, and so addition-subtraction chains can be used. This suggests using a representation allowing negative digits. Consider representations

$$x = \sum_{i=0}^{\infty} c_i 2^i \tag{4}$$

with $c_i \in \{-1, 0, 1\}$ for all $i$. Let the *weight* of a representation be the number of nonzero $c_i$, and $w(x)$ be the minimum weight of any such representation of $x$. A *Nonadjacent Form (NAF)* is a representation with $c_i c_{i+1} = 0$ for all $i \geq 0$. The following theorem, which has been redisovered many times, is also useful in the theory of arithmetic codes [28]:

**Theorem 2** *Every integer $x$ has exactly one NAF. The number of nonzeros in the NAF is $w(x)$.*

The advantage of using the NAF is that it in general has fewer nonzeros than the binary representation, reducing the number of multiplies. Morain and Olivos [21] showed that the expected number of nonzeros in a length $l$ NAF is $l/3$ (see [3] for a different proof).

The $m$-ary method may of course also be generalized to allow negative digits (for example, see the balanced ternary system in [13], or generalizations of NAFs to other bases in [28]). However, the savings quickly go down, since the average number of nonzeros in an $l$-digit generalized NAF is $l(m-1)/(m+1)$ (see [3]), which is not much better than the $l(m-1)/m$ in the base $m$ representation for large $m$.

# 3 Window Methods

The $2^k$-ary method may be thought of as taking $k$-bit *windows* in the binary representation of $r$, calculating the powers in the windows one by one, squaring them $k$ times to shift them over, and then multiplying by the power in the next window.

This leads to several different generalizations. One obvious one is that there is no reason to force the windows to be next to each other. Strings of zeros do not need to be calculated, and may be skipped. Moreover, only odd powers of $g$ need to be computed in the first step.

The example $r = 26235947428953663183191$ is given in [5]. Its binary representation is:

$$101100011100100000011101001010011101010000001011110000011111001100101010111.$$

The optimal choice for the $m$-ary method for this 75-bit number is $m = 8$, which takes 102 multiplications. For the window method, with windows of length up to 4, the number of multiplies is only 93: 8 multiplies to compute the odd powers up to 15, 71 squarings, and 14 multiplies for the intermediate values:

$$\underset{11}{\underline{1011}}000\underset{7}{\underline{111}}00\underset{1}{\underline{1}}000000\underset{7}{\underline{111}}0\underset{9}{\underline{1001}}0\underset{9}{\underline{1001}}\,\underset{13}{\underline{1101}}0\underset{1}{\underline{1}}000000\underset{11}{\underline{1011}}\,\underset{3}{\underline{11}}00000\underset{15}{\underline{1111}}\,\underset{9}{\underline{1001}}\,\underset{9}{\underline{1001}}0\underset{5}{\underline{101}}0\underset{7}{\underline{111}}$$

Bos and Coster [5] suggest using larger windows. Instead of constructing a table of all odd numbers less than $m$, they use an addition sequence to compute all the intermediate values needed for this particular exponentiation. Using large windows can reduce the number of multiplies to 89:

$$\underset{5689}{\underline{1011000111001}}000000\underset{933}{\underline{1110100101}}00\underset{117}{\underline{1110101}}000000\underset{47}{\underline{101111}}00000\underset{499}{\underline{111110011}}00\underset{343}{\underline{101010111}}$$

They use 62 squarings, 5 multiplies of intermediate values, and 22 multiplies to compute the addition sequence

$$1, 2, 4, 8, 10, 11, 18, 36, \underline{47}, 55, 91, 109, \underline{117}, 226, \underline{343}, 434, 489, \underline{499}, \underline{933}, \\ 1422, 2844, 5688, \underline{5689}.$$

Bos and Coster give a heuristic algorithm to compute an addition sequence for a given set of numbers. Note that we are reducing the number of multiplies needed at the expense of more work in the preparation phase: deciding what windows to use, finding a good addition chain for the values in the windows, and a more complicated algorithm to combine these values. This is all right as long as this work is cheap compared to the work of multiplying. That assumption can break down if multiplications are not that expensive (for small moduli, or in $GF(2^n)$).

For elliptic curve systems, Koyama and Tsuruoka [15] combine the window method with a redundant number system to get further gains. Using the NAF of $r$ instead of the binary representation will increase the number of runs of zeros, for further savings. For the number used above, we get:

$$\underbrace{\underline{10\bar{1}0\bar{1}}}_{11}00\underbrace{\underline{100\bar{1}001}}_{57}00000\underbrace{\underline{100\bar{1}01}}_{29}00\underbrace{\underline{10101}}_{21}00\underbrace{\underline{\bar{1}0101}}_{-11}00000\underbrace{\underline{10\bar{1}000\bar{1}}}_{47}0000\underbrace{\underline{10000\bar{1}}}_{31}0\underbrace{\underline{10\bar{1}010\bar{1}}}_{51}0\underbrace{\underline{\bar{1}0\bar{1}00\bar{1}}}_{-41}$$

where $\bar{1}$ denotes $-1$. This can be evaluated with 90 multiplications. Using large windows can save a few more multiplies.

Koyama and Tsuruoka also point out that the NAF is not necessarily the optimal representation to use. It does have minimal weight, but allowing a few adjacent nonzeros may increase the length of zero runs, reducing the total number of multiplies. They give a new method of computing a representation, which improves their "signed binary window method" in practice.

These methods are all heuristic, in that no good bounds on their performance or proofs of their superiority are known. However, they do appear to give significant speedups, and should be considered when picking an exponentiation method. To determine their usefulness for a particular problem, it is usually necessary to do simulation runs to determine the best choice of representation, window size, and addition sequence algorithm to use.

# 4 Special Groups

## 4.1 Normal Bases

Some groups have added structure that allow much faster exponentiation. In $GF(p^n)$, normal bases allow $p$th powers to be calculated with just a cyclic shift, greatly speeding the $p$-ary method. See [2], [27], [29] for some algorithms for this situation.

The most common use of this is in $GF(2^n)$, where the use of a normal basis allows squarings to be done with just a shift. The $2^k$-ary method then takes only $\lceil n/k \rceil + 2^{k-1} - 2$ multiplications, since only odd powers up to $2^k - 1$ need to be computed.

> Compute $g^3, g^5, \ldots g^{2^k-1}$.
>
> $a \leftarrow 1$
>
> `for` $d = 2^k - 1$ `to` $1$ `by` $-2$
>
>     `for each` $i$ `such that` $c_i = d * 2^{j_i}$

$$a \leftarrow a * (g^d)^{2^{ki+j_i}}$$

`return` $a$.

The savings is dramatic; exponentiation in $GF(2^{593})$ takes only 129 multiplies with this algorithm [27]. Use of a window method can further reduce the work.

## 4.2 Elliptic Curves

One family of groups that are often proposed for cryptosystems are elliptic curves. Because there is no index calculus method known for them, much smaller key lengths seem to be secure. Their main drawback is that adding two points on an elliptic curve involves several multiplies. The exact number depends on the parameterization of the curve. See [18] for information on elliptic curves and their use in cryptography.

Certain special types of elliptic curves allow for faster addition of points. Supersingular curves were suggested by several authors for use in cryptosystems, but it was discovered by Menezes, Okamoto and Vanstone [19] that the discrete logarithm problem on supersingular curves could be reduced to the discrete logarithm problem in an extension field.

Koblitz [14] suggested an alternative, which he called *anomalous* curves. These are the curves

$$E_1: \quad y^2 + xy = x^3 + x^2 + 1$$

and

$$E_2: \quad y^2 + xy = x^3 + 1$$

over $GF(2^n)$. These curves have complex multiplication by $K = \mathbf{Q}(\sqrt{-7})$. Their Frobenius automorphisms $\varphi$, which correspond to multiplication by $\tau = (1 + \sqrt{-7})/2$ and $-\bar{\tau} = (-1 + \sqrt{-7})/2$, respectively, can be computed very cheaply: $\varphi(x, y) = (x^2, y^2)$. Using a normal basis, this requires just two cyclic shifts.

Koblitz noted two possible ways to take advantage of this mapping. $\varphi$ satisfies the relation $T - T^2 = 2$, which does not help with doubling, but iterating the relation gives: $4 = -T^3 - T^2$, $8 = -T^3 + T^5$, and $16 = T^4 - T^8$ for $E_1$, and similar formulas for $E_2$. Using this, the 16-ary method can be applied with a savings of $3n/4$ additions.

Another method uses the base-$\tau$ expansion of $r$. Any integer $r$ has a representation as

$$r = \sum_{i=0}^{\infty} c_i \tau^i$$

for $c_i \in \{0, 1\}$, since $\tau$ is an element of norm 2 in the Euclidean domain $\mathcal{O}_K = \mathbf{Z}[\tau]$. To show that such an expansion can yield an efficient algorithm for exponentiation, we will need a few theorems, similar to those used in Section 2.3.

For any $r \in \mathcal{O}_K$ a representation

$$r = \sum_{i=0}^{\infty} c_i \tau^i \tag{5}$$

is called an *NAF* if $c_i \in \{0, \pm 1\}$ and $c_i c_{i+1} = 0$ for all $i \geq 0$. Let $w(r)$ be the minimal number of nonzero $c_i$'s in any representation (5) with $c_i \in \{0, \pm 1\}$.

**Theorem 3** *Every $r \in \mathcal{O}_K$ has a unique NAF, which has weight $w(r)$.*

PROOF: We will give a proof similar to the proof of Theorem 10.2.3 in [28], by changing an arbitrary representation into an NAF without increasing its weight.

Let $i$ be the minimal value such that $c_{i+1}$ and $c_i$ are both nonzero. Then we may apply one of the following transformations or their negatives to make $c_{i+1}$ zero:

$$\tau + 1 \longrightarrow -\tau^3 - 1 \tag{6}$$

or

$$\tau - 1 \longrightarrow \tau^2 + 1. \tag{7}$$

These maps add $\delta_i = \pm 1$ to $c_{i+2}$ or $c_{i+3}$. If that coefficient was zero, then there is no net change in the weight. If adding $\delta_i$ cancels it out, then the weight decreases. Otherwise, we end up with a coefficient equal to 2, which can be eliminated with the map

$$2 \longrightarrow -\tau^3 - \tau. \tag{8}$$

The combination of (8) with (6) or (7) also leaves the weight unchanged. If some larger coefficients are nonzero, further applications of (8) may be needed, but the weight will never increase.

To prove uniqueness, suppose that some $r$ has two representations $\sum c_i\tau^i$ and $\sum c_i'\tau^i$. Without loss of generality we may assume that $c_0 \neq c_0'$. Neither of them may be zero, since $r$ is either divisible by $\tau$ or not, so we may take $c_0 = 1$ and $c_0' = -1$, and $\tau$ does not divide $r$. Since the representations are NAFs, $c_1 = c_1' = 0$. But then adding the two representations we have $\tau^2|2r$, which is a contradiction.

$\square$

The algorithm given for computing the NAF in Theorem 3 was useful for showing that the NAF has minimal weight, but may not be the best method to use in practice. Reiter and Solinas [26] first showed the existence of the NAF using an algorithm that computes the NAF directly. If $\tau|r$, then $c_0 = 0$. Otherwise, $\tau^2$ divides either $r+1$ or $r-1$ (since $\tau|2$), and the NAF ends in $(0,-1)$ or $(0,1)$, respectively. Then $r$ is replaced by $r/\tau$, $(r+1)/\tau^2$, or $(r-1)/\tau^2$, and the process continued.

The problem with the NAF, as noted in [17], is that the NAF of $r$ will in general be twice as long as the binary representation of $r$, since the norm of $\tau$ is two, and the norm of $r$ is $r^2$. However, $\varphi^n = 1$ in $GF(2^n)$ (since $\varphi^n \cdot (x,y) = (x^{2^n}, y^{2^n}) = (x,y)$), so any two representations which agree modulo $\tau^n - 1$ will yield the same endomorphism on the curve. Using this, Meier and Staffelbach showed:

**Theorem 4** *Every $r \in \mathcal{O}_K$ has a representation*

$$r \equiv \sum_{i=0}^{n-1} c_i\tau^i \pmod{\tau^n - 1},$$

*with $c_i \in \{0, \pm 1\}$.*

Meier and Staffelbach conjecture, based on empirical evidence, that on average half of the $c_i$ will be nonzero. If slightly more digits are allowed, this density of nonzeros can be reduced to 1/3.

**Theorem 5** *Every $r \in \mathcal{O}_K$ has an NAF representation*

$$r \equiv \sum_{i=0}^{n+1} c_i\tau^i \pmod{\tau^n - 1},$$

*with* $c_i \in \{0, \pm 1\}$.

PROOF: Apply the method for constructing an NAF in Theorem 3 to the representation of $r$ given in Theorem 4. This will turn it into an NAF, with the final map possibly extending to $c_{n+1}$. Usually such overflow digits can be wrapped around, using $\tau^n \equiv 1 \pmod{\tau^n - 1}$, but this will not always terminate. For example, in $GF(2^3)$, we have:

$$\begin{aligned}
6 &\equiv \tau^2 + \tau \pmod{\tau^3 - 1} \\
&= -\tau^4 - \tau \quad \text{(using (6))} \\
&\equiv -2\tau \\
&= \tau^4 + \tau^2 \quad \text{(using (8))} \\
&\equiv \tau^2 + \tau
\end{aligned}$$

This example also demonstrates that the NAF modulo $\tau^n - 1$ is not unique. □

As mentioned in Section 2.3, the average number of nonzeros in an NAF of length $n$ is $n/3$. Bjorn Poonen [24] has pointed out that we can prove the same bound for the NAFs of rational integers modulo $\tau^n - 1$. Let $l = \mathrm{norm}(\tau^n - 1)$, the order of the curve.

**Theorem 6** *As* $n \to \infty$*, the NAFs of* $\{1, 2, \ldots l - 1\}$ *given by Theorem 5 have average weight* $(1 + o(1))n/3$.

PROOF: The reductions of $L = \{0, 1, \ldots l - 1\}$ cover all congruence classes modulo $\tau^n - 1$. The set of points $r/(\tau^n - 1) \in \mathbf{C}$ for $r \in L$ are equivalent modulo the lattice $\mathbf{Z}[\tau]$ to points $\overline{r}$ in the Voronoi region of the lattice (a hexagon centered at the origin), and so we may take the reductions of $L$ modulo $\tau^n - 1$ to be lattice points $a + b\tau$ in the Voronoi region of $(\tau^n - 1)\mathbf{Z}[\tau]$.

Calculate the NAF of each such residue $\overline{r}$. The coefficients $c_0, c_1, \ldots, c_j$ are determined by the residue of $\overline{r}$ modulo $\tau^{j+2}$. For each $j$, these residues are almost perfectly uniformly distributed for $r$ within the Voronoi region until $j$ is close to $n$. □

Using the above theorems, we may multiply points on anomalous curves using the NAF expansion. The $\tau$-ary method will take $n/3$ multiplies on average, by Theorem 6. Using windows will further reduce the work, and as in [15], it is possible to use a representation with some adjacent nonzeros to increase the length of the runs of zeros. See [26] for details.

# 5 Precomputation

## 5.1 The BGMW method

In cases such as Diffie-Hellman key exchange, where a fixed number is raised repeatedly to different powers, precomputing some of the powers is an option to speed up exponentation. This was first suggested by Brickell, Gordon, McCurley and Wilson [7].

The simplest example of the BGMW method is to store $g^{2^k}$ for $k = 1, 2, \ldots$, and then use the binary method without having to do any squarings. This gives essentially the same results as for normal bases in the previous section. The disadvantage is the extra space needed to store the extra numbers, but different schemes can be used according to how much storage is available.

For a precomputation version of the $m$-ary method, it is clear that one wants to store $g^{m^k}$. However, an observation in [7] is that more time may be saved by multiplying together powers with like coefficients, and then raising the subproducts to powers step by step.

Suppose $r = \sum_{i=0}^{l-1} a_i x_i$, where $0 \leq a_i \leq h$. Then

$$g^r = \prod_{d=1}^{h} c_d^d, \tag{9}$$

where

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

The key point is that (9) may be computed efficiently, using

$$\prod_{d=1}^{h} c_d^d = c_h(c_h c_{h-1})(c_h c_{h-1} c_{h-2}) \cdots (c_h c_{h-1} \cdots c_1) \tag{10}$$

The following theorem is Lemma 1 from [7]:

**Theorem 7** *Suppose* $r = \sum_{i=0}^{l-1} a_i x_i$, *where* $0 \leq a_i \leq h$, *and* $g^{x_i}$ *has been precomputed for each* $0 \leq i < l$. *The following algorithm computes* $g^r$ *in* $l + h - 2$ *multiplications.*

$b \leftarrow 1$

| $h$ | $\{x_i\}$ | comment |
|---|---|---|
| $m-1$ | $\{m^j\}$ | $m$-ary method |
| $\lceil (m-1)/2 \rceil$ | $\{\pm m^j\}$ | NAF for $m=2$ |
| $\lfloor m/3 \rfloor$ | $\{\pm m^j, \pm 2m^j\}$ | |
| 2 | $M_2(m)\cdot\{m^j\}$ | These methods use more storage |
| 3 | $M_3(m)\cdot\{m^j\}$ | |

Table 1: Some BGMW number systems

$a \leftarrow 1$

`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` $a$.

Taking $x_i = b^i$ with $b = \lfloor \log r / \log^2 \log r \rfloor$, this algorithm will compute $g^r$ in $(1+o(1))\log r/\log\log r$ multiplications with $O(\log r/\log\log r)$ precomputed powers.

Note that this algorithm is more general than the $m$-ary method. Any set of $x_i$'s which allow representations of all integers in the desired range will work. In [7] a number of schemes are suggested, some of which are shown in Table 1. In the table,

$$M_2(m) = \{d | 1 \le d < m, \omega_2(d) \equiv 0 \pmod 2\},$$

where $\omega_p(d)$ is the exponent of the largest power of $p$ dividing $d$, and

$$M_3(m) = \{d | 1 \le d < m, \omega_2(d) + \omega_3(d) \equiv 0 \pmod 2\}.$$

See [7] for a number of other number systems, and how many multiplies they require for 512-bit and 160-bit exponentiations. For a particular exponent size and amount of memory, it is often possible to find a sporadic number system which outperforms one of the general classes in the table. One such example in [7] has $\{x_i\} = \{\pm 1, -2, 9, 10\} \cdot \{29^j\}$ and $h = 8$.

## 5.2 Precomputation with Vector Addition Chains

Two 1994 papers ([8], [16]) independently made the observation that the BGMW method tends to use too much memory. It works best when $h$ is small compared to $l$, so that (9) does not take too long to compute, and most of the $c_d$'s are nontrivial. But taking $h$ small forces more storage.

Suppose we take $h$ large. Then many of the possible digits between 0 and $h-1$ will not be used, and (10) becomes a less attractive method. Instead, we may use a vector addition chain to compute

$$c_{d_1}^{d_1} \cdot c_{d_2}^{d_2} \cdots c_{d_t}^{d_t}$$

for the digits that do occur. Instead of taking time $l+h-2$, Theorem 1 and (2) imply that a good vector addition chain will take

$$l + \log h + (t + 1 + o(1)) \log h / \log \log h$$

multiplies. This lets us take $m$ and $h$ reasonably large, decreasing the storage requirements without increasing computation time. This is analogous to the large window techniques of Section 3, where we used an addition sequence to avoid computing many small powers.

DeRooij [8] tried various algorithms for constructing good vector addition chains, gaining significant improvements over [7]. We will concentrate on the method Lim and Lee [16] proposed, since it includes a specific vector addition chain algorithm which is easy to implement and has good performance.

A simple version of the Lim-Lee method would be to compute a $n$-bit exponent $g^r$ by writing the binary representation of $r$ in two rows, writing $r = 2^{n/2} r_1 + r_0$. We will precompute values corresponding to the powers represented by any column: $G[00] = 1$, $G[01] = g$, $G[10] = g^{2^{n/2}}$, and $G[11] = g^{2^{n/2}+1}$. Then $g^r = G[10]^{r_1} G[01]^{r_0}$ may be computed similarly to the binary method with at most $n$ multiplies, at each step squaring the intermediate value and multiplying by some $G[e_0 e_1]$, corresponding to the bits of $r_0$ and $r_1$ in that column. Figure 1 illustrates this idea.

For the general method, we may break $r$ into $h$ rows instead of two, and group the columns together into $b$-bit blocks to get more speedup with extra precomputation. For an $n$-bit exponent $r$, let $vb$ be the number of columns, and $h = \lceil n/vb \rceil$ be the number of rows.

As in the simple example above, we will precompute powers of $g$ corresponding to all possible column vectors. For any column vector $\bar{e} =$

$n/2$

| | | | | | |
|---|---|---|---|---|---|
| $r_0$ | 1 | 1 | 0 | $\cdots$ | 1 |
| $r_1$ | 1 | 0 | 1 | $\cdots$ | 0 |

Figure 1: A simple example of the Lim-Lee method. In this case, $g^r = G[01]^{r_0} G[10]^{r_1} = ((((G[11])^2 \cdot G[01])^2 \cdot G[10])^2 \cdots G[01])$

$e_0, e_1, \ldots, e_{h-1}$, we will precompute the power of $g$ corresponding to that vector:

$$G[\bar{e}] = \prod_{i=0}^{h-1} g^{e_i 2^{ivb}}.$$

To be able to handle blocks together, we will also precompute

$$G[j, \bar{e}] = G[\bar{e}]^{jb}$$

for $j = 0, 1, \ldots, v-1$. Let $e[i]$ denote the $i$th column vector. Now we have

$$g^r = \prod_{k=0}^{b-1} \left( \prod_{j=0}^{v-1} G[j, e[k+jb]] \right)^{2^k}.$$

Then the Lim-Lee algorithm becomes:

$z \leftarrow 1$

`for` $k = b-1$ `to` $0$ `by` $-1$

  $z \leftarrow z * z$

  `for` $j = v-1$ `to` $0$ `by` $-1$

    $z \leftarrow z * G[j, e[k+jb]]$

`return` $z$.

See [16] for the number of multiplies required for various amounts of precomputation for 160-bit and 512-bit exponents.

# 6 Parallel Algorithms

In contrast to the serial case, the parallel complexity of exponentiation is not well understood. The basic question of whether modular exponentiation is in **NC**, i.e. can be solved by Boolean circuits with polynomial size ($O(n^k)$ multiplications, for some $k$) and polylog depth ($O(\log^l n)$ time, for some $l$), is unknown. Adleman and Kompella [1] showed that powers modulo an $n$-bit number could be computed with a circuit of depth $O(\log^3 n)$ and size $O(e^{c\sqrt{n \log n}})$.

If all the prime factors of $N$ are less than a bound $s$, von zur Gathen [30] showed that exponentiation modulo $N$ can be done by circuits with depth $O(\log^2 s \log\log s)$ and polynomial size for log-space uniform families, and depth $O(\log s)$ for $P$-uniform families. Stinson [27] showed that in $GF(2^n)$ free squaring could be used to exponentiate using $\log n$ time and $O(n/\log n)$ processors. In [31], von zur Gathen extended the method to $GF(q^n)$.

The precomputation methods lend themselves to parallel implementations. Lim and Lee [16] show that by having one processor handle each of the $v$ column blocks, and then having the $v$ processors multiply their results together in $\log v$ time, they can compute powers modulo an $n$-bit number in $O(\log n)$ time using $O(n/\log n)$ processors. Each processor needs to store only a constant number of precomputed values.

In [6], an unpublished extended version of [7], two parallel versions of the BGMW algorithm are given. Both run in $O(\log n)$ time. One is similar to the Lim-Lee method; each processor computes $g^{a_i b^i}$ using the precomputed value $g^{b^i}$ and an addition chain for $a_i$, and then the results are multiplied together. This also takes $O(n/\log n)$ processors with a constant amount of memory per processor.

A second way of parallelizing the BGMW algorithm is to have $h$ processors compute $c_d^d = \prod_{a_i=d} g^{a_i b^i}$, and then multiply the results together. This requires only $O(n/\log^2 n)$ processors, and takes expected time $O(\log n)$. Some powers would take longer, say if all the $a_i$'s are equal, but this could be dealt with by having idle processors help out busy ones. A more serious problem is that each processor needs to store all $O(n/\log n)$ precomputed values.

In [6] it is shown that any exponentiation algorithm using a polylog number of precomputed values requires at least $O(n/\log n)$ multiplications. Thus for any parallel algorithm running in time $O(\log n)$, we will need at least $O(n/\log^2 n)$ processors. It is an open problem to find such an algorithm which uses a constant number of stored values per processor.

## 7 Conclusions

There are too many possible choices among the above methods to have one clear winner. A good general strategy for a particular implementation is to decide on which general method best fits the available computational power and storage, and then experiment with the parameters to optimize performance. There are a few general principles that can help to pick the best exponentiation method:

1. If a special group such as $GF(2^n)$ or an anomalous elliptic curve can be used without affecting security, the immediate gain from the free operations described in Section 4 overwhelms the advantages of any other scheme.

2. Precomputation can make a large difference as well. Generally using vector addition chains works better than the BGMW methods, but some amount of playing around with the parameter choices will be necessary to get the best results.

3. Without precomputation or special group structure, the differences between the methods is not that great. The 16-ary method works well for a large range of exponent sizes, and is easy to implement. Window methods and redundant number systems can give significant further speedups, without too much added complication.

4. Smart cards have far more limited memory and processing power, so many of these schemes may be impractical. Using $GF(2^n)$ or anomalous elliptic curves with the methods of Section 4 may be the only way to get a method that works reasonably fast without large memory requirements.

## References

[1] Leonard M. Adleman and Kireeti Kompella. Using smoothness to achieve parallelism. In *Proceedings of the 20th ACM Symposium on the Theory of Computing*, pages 528–538, 1988.

[2] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. Fast exponentiation in $GF(2^n)$. In *Advances in Cryptology – Proceedings of Eurocrypt '88*, volume 330, pages 251–255. Springer-Verlag, 1988.

[3] S. Arno and F. S. Wheeler. Signed digit represenations of minimal Hamming weight. *IEEE Trans. Computers*, 42:1007–1010, 1993.

[4] Daniel J. Bernstein. *Detecting perfect powers in essentially linear time, and other studies in computational number theory.* PhD thesis, University of California at Berkeley, 1995.

[5] Jurjen Bos and Matthijs Coster. Addition chain heuristics. In *Advances in Cryptology – Proceedings of Crypto '89*, volume 435, pages 400–407. Springer-Verlag, 1990.

[6] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation: algorithms and lower bounds. preprint, 1995, contact the second author for a copy.

[7] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In *Advances in Cryptology – Proceedings of Eurocrypt '92*, volume 658, pages 200–207. Springer-Verlag, 1992.

[8] Peter de Rooij. Efficient exponentiation using precomputation and vector addition chains. In *Advances in Cryptology – Proceedings of Eurocrypt '94*, volume 950, pages 389–399, 1994.

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

[10] Peter Downey, Benton Leong, and Ravi Sethi. Computing sequences with addition chains. *SIAM J. Comput.*, 10:638–646, 1981.

[11] Paul Erdős. Remarks on number theory III. On addition chains. *Acta Arith.*, pages 77–81, 1960.

[12] Seong-Min Hong, Sang-Yeop Oh, and Hyunsoo Yoon. New modular multiplication algorithms for fast modular exponentiation. In *Advances in Cryptology – Proceedings of Eurocrypt '96*, pages 166–177, 1996.

[13] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Reading, Massachusetts, second edition, 1981.

[14] Neal Koblitz. CM-curves with good cryptographic properties. In J. Feigenbaum, editor, *Advances in Cryptology – Proceedings of Crypto '92*, volume 576, pages pp.279–287, 1992.

[15] Kenji Koyama and Yukio Tsuruoka. Speeding up elliptic cryptosystems by using a signed binary window method. In *Advances in Cryptology – Proceedings of Crypto '92*, volume 740, pages 345–357. Springer-Verlag, 1993.

[16] Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In *Advances in Cryptology – Proceedings of Crypto '94*, volume 839, pages 95–107, 1994.

[17] Willi Meier and Othmar Staffelbach. Efficient multiplication on certain nonsupersingular elliptic curves. In *Advances in Cryptology – Proceedings of Crypto '92*, volume 740, pages 333–344. Springer-Verlag, 1993.

[18] Alfred J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer, 1993.

[19] Alfred J. Menezes, Tatsuaki Okamoto, and Scott A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Trans. Info. Theory*, 39:1639–1646, 1993.

[20] P. Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44:519–521, 1985.

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

[22] Jorge Olivos. On vectorial addition chains. *J. Algorithms*, 2:13–21, 1981.

[23] R. G. E. Pinch. Some primality testing algorithms. *Notices Amer. Math. Soc.*, 40:1203–1210, 1993.

[24] Bjorn Poonen. private communication.

[25] Ronald Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[26] Jerome A. Solinas. An improved algorithm for arithmetic on a family of elliptic curves. In *Advances in Cryptology – Proceedings of Crypto '97*, pages 357–371, 1997.

[27] D. R. Stinson. Some observations on parallel algorithms for fast exponentiation in $GF(2^n)$. *SIAM J. Comput.*, 19:711–717, 1990.

[28] J. H. van Lint. *Introduction to Coding Theory*. Springer-Verlag, 1982.

[29] J. von zur Gathen. Efficient exponentiation in finite fields. In *Proceedings of the 32nd IEEE Symposium on the Foundations of Computer Science*, pages 384–391, 1991.

[30] Joachim von zur Gathen. Computing powers in parallel. *SIAM J. Comput.*, pages 930–945, 1987.

[31] Joachim von zur Gathen. Efficient and optimal exponentiation in finite fields. *Comput. Complexity*, pages 360–394, 1991.

[32] A. C. Yao. On the evaluation of powers. *SIAM J. Comput.*, 5:100–103, 1976.