# Exponentiation using Division Chains

Colin D. Walter, *Member, IEEE*

Computation Department, U.M.I.S.T.,
PO Box 88, Sackville Street, Manchester M60 1QD, U.K.
C.Walter@umist.ac.uk http://www.co.umist.ac.uk/

## Abstract

*Exponentiation may be performed faster than the traditional square and multiply method by iteratively reducing the exponent modulo numbers which, as exponents themselves, require few multiplications. This mainly includes those with few non-zero bits. For a suitable choice of such divisors, the resulting mixed basis representation of the exponent reduces the expected number of non-squaring multiplications by over half at the cost of a single extra register. Pre-processing effort depends entirely on the exponent and can be kept down to the work saved in a single exponentiation. Moreover, no pre-computed look-up tables are required, so the method is especially applicable where space is at a premium. In particular, it out performs the instance of the m-ary method which uses the same space. However, for 512-bit exponents, it beats every instance of the m-ary method, achieving well under 635 multiplications on average. Both hardware and software implementations of the RSA crypto-system can benefit from this algorithm.*

**Key Words** Modular exponentiation, bit recoding, RSA cryptosystem, addition chains, *m*-ary method, mixed basis arithmetic, radix representation.

## 1 Introduction

Fast exponentiation is becoming increasingly important with the widening use of encryption. Whereas the most startling improvements in speed are achieved through the use of dedicated hardware for multiplication, some small gains can also be made through a good choice of algorithm for organising the order of multiplications at either the hardware or software level.

The expected number of multiplications (including squarings) to compute $A^e$ in the traditional way, using square and multiply, is approximately $\frac{3}{2}\log_2 e$ [14], [15]. The pre-calculation or intermediate calculation of auxiliary powers of $A$ enables some unnecessary repetition of work to be avoided. There are several methods described below for reducing the average number of operations to about $\frac{4}{3}\log_2 e$ using storage for only one or two extra powers of $A$, but the variance of the distribution is high: there is a very real chance of still requiring $\frac{3}{2}\log_2 e$ or even more operations. With more storage, the coefficient $\frac{4}{3}$ can be further reduced [22], but the lower limit is still above 1 [11], [14] since $\lfloor \log_2 e \rfloor$ squarings are certainly necessary to generate a number as large as $A^e$. However, fast registers are normally very limited in number, so the use of too much more storage is actually likely to *retard* the exponentiation: the cost of communication with slower memory may easily outweigh any minimal reduction in the coefficient.

This paper describes another technique which uses only *one* register more than the minimum for the standard square and multiply method. It achieves an average coefficient below $\frac{5}{4}$, and is *consistently* close to its average. Some pre-computation is required, but only enough to establish a suitable sequence of squarings and multiplications. Thus the pre-computation need not be repeated in a sequence of exponentiations using the same exponent. A simple version of the technique achieves $\frac{5}{4}$ with a pre-computation effort equivalent to only a few multiplications of integers the size of the exponent. So the overall gain is small for a single exponentiation, but when the same key is used repeatedly as in the RSA cryptosystem, the pre-computation cost can be ignored and the full reduction claimed. The method is very adaptable, with better results easily obtained from greater effort or from allowing more register space. It also works equally well for any group, so that the speed-ups work as well for an elliptic curve group as for an integer residue class group.

## 2 Notation and Literature Review

On a sequential machine, any exponentiation by $e$ can be described by an *addition chain* $a_0, a_1, a_2, a_3,..., a_n$, where $a_0 = 1$, $a_n = e$ and, for each $i > 0$, $a_i = a_j + a_k$ for some $j, k < i$ [14]. The $i$th multiplication performed is $A^{a_i} = A^{a_j} \times A^{a_k}$ and exponentiation by $e$ takes $n$ multiplications. Storage requirements for any chain can be worked out easily from the sequence itself, although if there is a choice of $j$ and $k$ for any $i$ then the minimum storage might not be clear. For given small exponents $e$, the minimal number of multiplications can be found by a search of all addition chains for $e$. As an NP-hard problem [8], however, such a search is impractical for the typical decryption keys $e$ found in RSA cryptography. Nevertheless, here we present a technique which is easily tailored to the available time and space resources and ultimately would yield a best addition chain.

Suppose $e = \sum_{j=0}^{n} r_j 2^j$ is the binary representation of $e$. The standard method of square and multiply can be performed by scanning the bits of $e$ in either direction. First, a Horner-style evaluation

$$A^e = ((...((A^{r_n})^2 A^{r_{n-1}})^2 ...)^2 A^{r_1})^2 A^{r_0}$$

corresponds to an addition chain with $a_{i+1} = 2a_i$ (square) or $a_{i+1} = a_i + a_0$ (multiply). This requires just 2 storage registers, containing $A = A^{a_0}$ and the partial result $A^{a_i}$ respectively. Alternatively, and dually, rather than squaring the partial result and multiplying in $A$ as required, $A$ can be repeatedly squared and the resulting power multiplied into the partial result when needed:

$$A^e = (A^{2^0})^{r_0} (A^{2^1})^{r_1} (A^{2^2})^{r_2} ... (A^{2^n})^{r_n}$$

Here the first of the two registers now contains $A^{2^i}$ for $i = 0,1,..., n$. For natural number arithmetic this requires larger registers than Horner's method, but there is no difference for finite rings, such as the residues of RSA cryptosystems, or for real approximations. The number of multiplications, excluding squarings, is one less than the Hamming weight of $e$, i.e. one less than the number of non-zero bits in $e$; on average $\lfloor \log_2 e \rfloor / 2$.

By expressing the exponent using the radix $m$ instead of 2 and pre-computing the powers $A^i$ for $i = 1, 2, ..., m-1$ we obtain the $m$-ary method [14]. This follows the Horner style evaluation above, requiring repeatedly raising the partial result to the $m$th power and then multiplying by an $A^i$. It is usually convenient to pick $m$ as a power of 2. Then the number of squarings is roughly the same as before, but the number of other multiplications excluding pre-computations reduces to $\lfloor \log_m e \rfloor (m-1)/m$ on average. This is good for larger $m$, but storage requirements quickly become prohibitive for RSA applications. Moreover, the hardware to select one of $m$ registers on each iteration has a critical path length of order $O(\log_2 m)$. Thus, without considerable care, the complexity of the non-squaring multiplications for this method might still be essentially $O(\log_2 e)$.

Half the memory can be saved by pre-computing only $A^i$ for odd $i < m$ [22]. The exponent is recoded as

$$e = r_0 + m_0(r_1 + m_1(r_2 + m_2(...(r_{n-1} + m_{n-1} r_n)...))) \quad (1)$$

where $m_i = m$ is chosen whenever it makes $r_i$ odd, and otherwise $m_i = 2$ is chosen with $r_i = 0$. Then $A^e$ is evaluated starting at the innermost bracketed expression:

$$A^e = ((...((A^{r_n})^{m_{n-1}} A^{r_{n-1}})^{m_{n-2}} ...)^{m_1} A^{r_1})^{m_0} A^{r_0} \quad (2)$$

This requires $m/2$ registers besides the partial result, about $\lfloor \log_2 e \rfloor - \frac{1}{2}\log_2 m + 1$ squarings if $m$ is a power of 2, and about $\lfloor \log_2 e \rfloor / (\log_2 m + 1)$ other multiplications on average besides any used for the pre-computations. (For $i < n$ there are on average as many cases of $m_i = m$ as $m_i = 2$.) Taking $m = 2$ gives the binary method above with coefficient $\frac{3}{2}$ for $\log_2 e$ in the total number of multiplications. Taking $m = 4$ means a third register, which holds $A^3$, and this reduces the coefficient to $\frac{4}{3}$.

If the inverse $A^{-1}$ exists and can be calculated cheaply, then any sequence of 1s can be replaced by the sequence $10...0\overline{1}$ as follows. Starting from the least significant bit of $e$, whenever two adjacent 1s are encountered with no carry from lower down, the lower 1 is replaced by $\overline{1}$ and a carry of 1 generated to the upper 1. Similarly, for a carry of 1 into 10, the lower 0 is replaced by $\overline{1}$ with a carry up to the 1. Otherwise carries propagate up as usual. On average the carry is 1 in 50% of cases, 2/3rds of these are into a following 1. Similarly, a carry of 0 is to a digit 0 in 2/3rds of cases. Thus, after this modified Booth recoding, a 0 digit occurs on average in 2/3rds of all cases. So again about $\frac{4}{3}\log_2 e$ squarings or multiplications by $A$ or $A^{-1}$ are required, with one multiplication for each non-zero digit [26]. Storage is 3 registers, one for each of $A$, $A^{-1}$ and the partial result. This method can be combined with the $m$-ary method [9], [10], [15]. Further similar methods include [12].

The $m$-ary method has been developed in several important ways. One of these uses the ideas of data compression: common subsequences of bits are identified and the corresponding powers of $A$ computed once and stored for use each time the bit sequence is encountered. In effect, this windowing technique corresponds to allowing the $m_i$ in (1) to be any power of 2 [1], [2], [16]. Another method employs *vector* addition chains so that work involved in computing several different powers of $A$ can be shared wherever possible [2],[18],[19],[20],[23]. Similar sharing is also considered in [17], [24], [25].

In many of these methods, the trade-off between space and time is poor when $e$ is static and $A$ variable. In hardware implementations, a speed-up of less than two-fold, as here, should require less than twice the area. Since the total hardware requirement for a digit parallel modular multiplier is equivalent to only a small number of registers [21], few of these techniques are likely to be of much value to hardware designers. However, we will produce more than half of the maximum possible savings by using only 1 more register than for the square and multiply method.

## 3 Mixed Basis Representations.

In a *mixed basis* number representation, an exponent $e$ is written as a digit linear combination

$$e = r_0 b_0 + r_1 b_1 + \ldots + r_n b_n$$

of basis elements $b_i$. Normally, $b_i$ is a multiple of $b_{i-1}$, with $b_i = m^i$ in a standard radix $m$ representation. The divisibility property enables the nested representation of $e$ necessary in the $m$-ary method. In the variation where half the memory is saved, the basis elements are $b_i = m_0 m_1 \ldots m_{i-1} = b_{i-1} m_{i-1}$ with $m_{i-1} = 2$ or $m$. Dimitrov *et al* [6] consider the particular case of this representation with $m = 3$, illustrating the value of $m$ not being a power of 2. There the digits $r_i$ are 0 or 1 and basis elements $b_i$ are of the form $2^j 3^k$. This leads to $1.42361 \times \log_2 e$ multiplications on average. They suggest that each $m_i$ could be any prime up to some bound $m$. Then the powers $A^i$ could be pre-computed for $i < m$ and $A^e$ calculated following the $m$-ary method, that is, with Horner style evaluation. In fact, the algorithm presented in [6] for $m = 3$ processes the digits in the opposite order; the cost is the same for that case. If many of the possible $b_i$th powers of $A$ are precomputed, then this latter order can lead to considerable time savings when a number of exponentiations with fixed $A$ and variable $e$ are involved [4], [7]. However, this is a different problem and the space cost may be large.

Here we retain the key divisibility property of the basis elements, namely $b_i = b_{i-1} m_{i-1}$, and develop algorithms for making good choices of the $m_i$. The divisibility property enables the digits to be processed in either order. Where space is expensive, using Horner's order of evaluation without precomputed powers leads to duplication of effort when digits are equal since then the $r_i$th power of $A$ must be re-calculated. We concentrate on scanning the digits in the reverse order: the $b_i$th power of $A$ will be computed dynamically from the $b_{i-1}$st and held in a single register, just as the $2^i$th power is held in the corresponding binary method. At the same time, the $r_{i-1} b_{i-1}$st power will be formed and multiplied into the partial result.

## 4 The Division Chain Method

The new means of reducing the number of multiplications used in exponentiation arises from the iterative application of a decomposition $e = me' + r$ where $r$ is usually the least non-negative residue of $e$ modulo $m$. At each repetition the divisor $m$ is selected by reference to a pre-determined set of inexpensive pairs $(m,r)$ and the powers $A^m$ and $A^r$ are computed. Since $A^e$ satisfies the relationship

$$A^e = (A^m)^{e'} A^r \qquad (3)$$

it suffices to multiply $A^r$ into a partial product register and re-apply the process to the remaining problem of raising $A^m$ to the power $e'$. Including all residues for some divisor such as 2, 3 or 5 guarantees each step is possible and that termination occurs. When $e' = 0$ has been processed the partial product register contains the required output. So, if $m_0, m_1, \ldots, m_n$ is the list of divisors which this generates, and $r_0, r_1, \ldots, r_n$ are the associated remainders then, as in (2),

$$A^e = A^{r_0 + m_0 (r_1 + m_1 (\ldots + m_{n-1} (r_{n-1} + m_{n-1} r_n) \ldots))}$$

and evaluation is performed by processing the exponent expression from left to right. Note that the divisors for the algorithm can be chosen on the fly as the exponentiation is performed or in advance whenever the exponent is known.

We assume that it is known how best to calculate $A^m$ and $A^r$. So divisor / residue pairs $(m,r)$ must be selected from a set for which this information is known. Then at each step the cheapest such decomposition can be chosen from this fixed set of pairs. A sequence of pairs $(m,r)$ used to direct an exponentiation in this way will be called a *division chain* by analogy with the addition chain description. Although it is usually the case, we see later that the residue need not always be the least non-negative one. However, if no alternative choice of residue is ever permitted, then the sequence of divisors suffices to determine the division chain.

When $m = 2$ at each step, this technique is just the standard square and multiply method described above. If $r = 0$ at each step it becomes the *factor* method [14]. So the division chain method here generalises both of these. Moreover, if $m_i = m$ always, it is the dual of the $m$-ary method, processing digits in the opposite order. Bos and Coster [2] provide some heuristics for their MakeSequence method of constructing addition chains. This method shows how these can be realised.

If $F(e)$ is the minimum number of multiplications used to compute $A^e$ by any method, then the decomposition (3) shows $F(e) \le F(e')+F(m)+F(r)+1$. Inequality will arise if either $r$ is 0 or some multiplications which are used to form $A^m$ may also be used in constructing $A^r$. However, $F(e) \approx F(e')+F(m)$ because $e \approx me'$ and $F(x)$ is always of order $\log(x)$. So the main step of the algorithm can only be useful if $A^m$ and $A^r$ are both cheap to compute, with much of this work in common. In particular, this is the case when $m = 2^n+1$ and $r = 0$, $2^{n-1}+1$ or $2^i$ for some $i \le n$. Then the formation of $A^r$ is a by-product of computing $A^m$. In general, useful pairs $(m,r)$ normally have the property that $r$ lies in an addition chain for $m$ of minimal length or, at worst, $r$ is the sum of two or three members of such a chain. Consequently, if $f(e)$ is the number of multiplications this method yields, and there are sufficient such pairs $(m,r)$, then we will have $f(e) \approx f(e')+f(m)$ at each step and can expect to obtain a reasonably efficient scheme for exponentiation.

With suitable restrictions on the divisors $m$ and residues $r$, only three storage registers are needed, although more can be used. One register holds the partial result which is the product of all the $A^r$ from previous steps. The other two are used to form $A^m$ using an addition chain that requires only two values to be kept. When the components required for $A^r$ are formed, they are multiplied into the partial result register and so do not interfere with the calculation of $A^m$.

As an illustration, we show how to compute $A^{349}$ starting with the divisor 17. First, $A^{349} = (A^{17})^{20}A^9$ requires computing $B = A^{17}$ and $A^9$. These are found in 5 multiplications using the addition chain (1, 2, 4, 8, 9, 17), and $A^9$ is placed in the partial product register. This leaves $B^{20}$ to be calculated and multiplied into the accumulating product. Using the divisor 4 next, $B^{20} = (B^4)^5$. We obtain $C = B^4$ with 2 squarings and must then compute $C^5$ for multiplying into the partial product. Using the divisor 4 again, $C^5 = (C^4)^1C^1$. The $C^1$ is multiplied into the partial product, then $C^4$ is computed with two squarings and the result finally multiplied into the partial product to yield $A^{349}$. So exponentiation by 349 can be done this way with 11 multiplications rather than the 13 required by the binary and $A$, $A^3$ methods.

## 5 Calculation of the Coefficient

We will demonstrate the value of the method with a small set of divisors. Suppose the strategy is as follows:

```
If      e ≡ 0 mod 2        then m = 2
elseif  e ≡ 0 mod 3        then m = 3
elseif  e ≡ 1,2,5,8 mod 9  then m = 9
else   {e ≡ 4,7 mod 9}     then m = 3
```

with $r$ chosen as the least non-negative residue modulo $m$. Let $f(e) = c\log_2 e$ be the average number of squarings plus multiplications expected for a random exponent $e$. Then $f(e)$ can be expressed fairly accurately as a sum of terms $p\times(v+f(e/m))$, one for each case $(m,r)$, where $p$ is the probability of the case arising, $m$ is the associated divisor, and $v$ is the number of multiplications required to form $A^m$ and $A^r$ and multiply $A^r$ into the result. For the case $e \equiv 5 \bmod 9$, we can use the addition chain (1, 2, 4, 5, 9) to show that 4 multiplications yield both $A^m$ and $A^r$, giving $v = 5$. The other values of $v$ are clear.

| e mod 18 | m | Rel freq | Distrib of new e mod 6 | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 | 4 | 5 |
| 0,6,12 | 2 | 4 | 2 | - | - | 2 | - | - |
| 2,8,14 | 2 | 10 | - | 5 | - | - | 5 | - |
| 4,10,16 | 2 | 10 | - | - | 5 | - | - | 5 |
| 3 | 3 | ⎫ ⎧ | - | 2 | - | - | - | - |
| 9 | 3 | ⎬6⎨ | - | - | - | 2 | - | - |
| 15 | 3 | ⎭ ⎩ | - | - | - | - | - | 2 |
| 1 | 9 | ⎫ ⎧ | 1 | - | 1 | - | 1 | - |
| 7 | 3 | ⎬9⎨ | - | - | 3 | - | - | - |
| 13 | 3 | ⎭ ⎩ | - | - | - | - | 3 | - |
| 5 | 9 | ⎫ ⎧ | 1 | - | 1 | - | 1 | - |
| 11 | 9 | ⎬9⎨ | - | 1 | - | 1 | - | 1 |
| 17 | 9 | ⎭ ⎩ | - | 1 | - | 1 | - | 1 |
| Freq sums: 48 | | | 4 | 9 | 10 | 6 | 10 | 9 |

Table 1. Frequency counts for the Markov process.

The values of the probabilities $p$ are less obvious since application of the method causes the residue classes to be no longer uniformly distributed. Action depends entirely on the residue of $e$ modulo the lowest common multiple of the divisors, namely 18, and not on any previous exponent. The sequence of residues for successive exponents forms a Markov chain for which the stochastic transition matrix $P = (p_{ij})$ is easy to construct: $p_{ij}$ is the probability of obtaining the new exponent $e' \equiv j \bmod 18$ from an exponent $e \equiv i \bmod 18$. For example, $e = 18k+7$ uses $m = 3$, yielding $e' = 6k+2 \equiv 2$, 8 or 14 mod 18. So $p_{7,2} = p_{7,8} = p_{7,14} = 1/3$. The probabilities $p_i$ of exponents which are $i$ mod 18 eventually stabilise: the row vector $\boldsymbol{p} = (p_i)$ satisfies $\boldsymbol{p} = \boldsymbol{p}P$ and can be calculated easily. It turns out that $p_i$ only depends on $i$ mod 6, with $p_0 = p_6 = p_{12} = 1/36$, etc. The relative frequencies of the 18 classes are given in Table 1, which can readily be used to check that these do indeed yield the equilibrium values.

So the 4 cases of the algorithm have probabilities 1/2, 1/8, 1/4 and 1/8 respectively. In consequence,

$$f(e) = (1/2)(1+f(e/2)) + (1/8)(2+f(e/3)) + (1/4)(5+f(e/9)) + (1/8)(3+f(e/3))$$

Then using $f(e/m) \approx f(e)–f(m) \approx f(e)–c\log_2 m$ we obtain

$$0 = (1/2)(1–c\log_2 2) + (1/8)(2–c\log_2 3) + (1/4)(5–2c\log_2 3) + (1/8)(3–c\log_2 3)$$

So $c \approx 1.4064$ (cf. [6] with $c \approx 1.42361$ for a slightly simpler example).

Better values for the coefficient $c$ arise from the use of more divisors. Consider:

```
If      e ≡ 0 mod 33      then m = 33
elseif  e ≡ 0 mod 17      then m = 17
elseif  e ≡ 0 mod 2       then m =  2
elseif  e ≡ 0 mod 5       then m =  5
elseif  e ≡ 0 mod 3       then m =  3
elseif  e ≡ 1,2,4,8,16,17,32 mod 33
                          then m = 33
elseif  e ≡ 1,2,4,8,9,16 mod 17
                          then m = 17
else case e mod 90 of
 1,17,77,29,43,47,67,83      : m = 3
 7,11,13,31,41,49,61,71,79,89: m = 2
 19,23,37,53,59,73           : m = 9
end.
```

Again, choose $r$ minimally. The more complex final case is engineered to enable useful factors such as 2, 3, 5 or 9 to be picked up on the next iteration. As before the residue classes modulo 90×11×17 are not equi-probable. The transition matrix $P$ has grown to around $2^{28}$ entries and made the direct solution of $\boldsymbol{p} = \boldsymbol{p}P$ infeasible. However, if $\boldsymbol{i}$ is the suitably scaled all 1s row vector then $\boldsymbol{p} = \lim_{n\to\infty} \boldsymbol{i}P^n$. In practice convergence is fairly rapid. The Euclidean distance between successive values of $\boldsymbol{p}$ is almost halved at each iteration, and so another decimal place of $c$ is established on every fourth iteration. After just 7 iterations we obtain the correctly rounded $c = 1.3566$. When the divisor 65 is included in a similar fashion to 33, $c$ is reduced to 1.343.

The time complexity of finding $\boldsymbol{p}$ is driven by the number of non-zero entries in $P$. The matrix has size $l\times l$ where $l$ is the lowest common multiple of the moduli used, and it has not far short of $ln$ non-zero entries where $n$ is the number of different divisor/residue pairs considered. Thus, including more than a few small divisors makes the vector $\boldsymbol{p}$ too big and the computation too time consuming for a direct solution, and it forces a statistical approach to estimating $c$.

To achieve a coefficient noticeably less than the $\frac{4}{3}$ of the $A$, $A^3$ method, take the following divisors:

$$2,\ 3,\ 5,\ 17,\ 33,\ 49,\ 65,\ 97,\ 129,\ 257,\ 513,\ 1025$$

and allow residues which are either 0, or are in a minimal addition chain which uses just two memory fields, or are the sum of two residues in such a chain. (Details are contained in the Appendix.) At each iteration simply choose the divisor $m$ for which the ratio $v/\log_2 m$ is least, where $v$ is the number of multiplications associated with the residue which occurs. Then, on average, random 512-bit integers require 672.15 multiplications – almost 1.5% fewer than the average of 682 for the $A$, $A^3$ method.

How much work is involved in choosing the division chain for a given exponent and a given such strategy? A small, variable but bounded number of modular divisions are performed on the exponent at each step to decide the next divisor. Each of these divisions requires at most a small constant times more work to perform than the division which is actually chosen. Thus choosing the divisor $m$ at each step requires on average a constant, say $c'$, times more work than simply dividing the remaining exponent by $m$. Overall, $c'$ times more work is done to find the division chain than is used in dividing the original exponent by the chain of divisors. However, dividing the exponent by the chain of divisors requires the same order of work as finding the product of two integers the same size as the exponent. So generating the division chain has the same time complexity as forming the product of two numbers with the size of the exponent.

In the RSA cryptosystem the exponent $e$ for decrypting is usually of the same length as the message $A$ requiring decryption. So choosing the division chain will then be equivalent to the cost of a fixed number of multiplications, say $c''$. The total work of exponentiation will then be about $c''+c\log_2 e$ multiplications. Thus, for a sequence of exponentiations with the same key $e$, this is cheaper than the $A$, $A^3$ method as long as $c < \frac{4}{3}$ and the exponent $e$ is large enough.

## 6 Choice and Ordering of Divisors

From the last section the formula for $c$ is

$$c = \frac{\sum_i p_i v_i}{\sum_i p_i \log_2 m_i}$$

where the sum extends over cases $i$, which occur with probability $p_i$, have divisor $m_i$ and require $v_i$ multiplications. Thus, if the relative probabilities of these cases remain essentially unchanged, it is worth adding any new pair $(m,r)$ with a ratio $v/\log_2 m$ which is better than the current value of $c$. In particular, in addition to using numbers with the form $2^n+1$ (2 bit numbers), we might also consider divisors $m$ of Hamming weight 3 (i.e. 3 non-zero bits) with residues $r$ which are 0 or are generated en route to $m$. Then a minimal addition chain using just two registers will require $n+2$ multiplications where the highest power of 2 in $m$ has order $n$. Useful examples therefore include 49 and 97, both with ratios under $\frac{5}{4}$ for these residues.

It is a matter of only a few minutes computing to generate all optimal and near optimal addition chains for all potentially useful divisors up to 10 or so bits in length, say (see Appendix), and hence establish the smallest number of extra additions which will generate each residue. Here we should relax the apparent restriction $r < m$ since, to minimise multiplications, it may be necessary to go beyond the least non-negative residue. Thus 11 mod 13 is only obtained with a minimal number of multiplications within the suggested memory restrictions by allowing $r = 12+12$. However, if both $r$ and $r+m$ are equally cheap to compute, it may make sense to select the one which makes the next exponent $e'$ even so that the beneficial (2,0) could then be applied.

Using all $m$ up to over $3\times2^{10}$, optimal division chains were constructed sequentially for $e$ up to over $2^{20}$. The frequencies of pairs $(m,r)$ were recorded, and this confirmed the relative uselessness of pairs with a poor ratio $v/\log_2 m$ unless $m$ was small. It also showed that composite $m$ in this range are often superfluous since frequently no more multiplications are required than when their factors are used as divisors instead.

The obvious order in which to prioritise the divisor/residue pairs for choice is in increasing cost per bit, i.e. following the order of the ratios $v/\log_2 m$. However, larger divisors have a longer lasting effect than smaller ones, and so they are better (resp. worse) choices if they have similar but below (resp. above) average ratios. To be more precise, if $m$ is applied to the exponent $e$, then we can expect $v + c\log_2(e/m) = v - c\log_2(m) + c\log_2(e)$ multiplications on average. This is minimised if a divisor is chosen for which cost function $v - c\log_2 m$ is minimal.

Thus, the best order for selecting pairs $(m,r)$ should be close to that determined by the values of

$$v - c\log_2 m$$

not that of the ratios $v/\log_2 m$. These criteria are called the *difference* and *ratio* tests respectively. The former, while slightly better, does need a good approximation to $c$ in advance, whereas the latter requires no such information and so could be used to generate the approximation to $c$ for the difference test.

Using the better difference test explains the order in the second example of Section 5 where (2,0) is not the best first choice, as one might expect. Note that in situations such as the last case of that example, any divisor that will be picked up automatically for the next iteration needs to be taken into account when using either cost function to assess the relative merits of different divisors.

## 7 Algorithms for Large Exponents

Most of the well-known algorithms described in Section 2, such as square and multiply, prescribe a single course of action, as do the strategies described here so far for division chains. In all cases the high variance means frequent poor results. However, for division chains considerable choice is possible. Fixing a specific order for trying divisors usually yields a sub-optimal method. A better coefficient $c$ is obtained when the best chain is selected from all those generated by extending partially constructed chains in every possible way. This also reduces the variance and hence gives a good value more reliably. Unfortunately, it is not feasible for a large exponent – there are too many combinations to evaluate all possibilities. The search space must be reduced.

Suppose the division chain $(m_0,r_0)$, $(m_1,r_1)$, $(m_2,r_2)$, ..., $(m_{k-1},r_{k-1})$ reduces the exponentiation problem from the power $e$ to the power $e'$. Then $e = me' + r$ where $m = m_0m_1m_2...m_{k-1}$ and (normally) $r < m$. The number of multiplications $v$ associated with the chain is the sum of the numbers $v_i$ associated with each pair $(m_i,r_i)$, namely the number of multiplications required to form the $m_i$ and $r_i$th powers and multiply the $r_i$th power into the existing partial product. If $e'$ is still large compared to $m$ then $e$ is essentially reduced by a factor $m$ for the cost of $v$ multiplications. Thus, given a number of such chains, the best one to choose is the one for which $v/\log_2 m$ or, better, $v - c\log_2 m$ is minimal. This process can be repeated to reduce $e'$ in its turn. Eventually $e'$ becomes small enough for the value of $r$ to affect the choice of chain noticeably (say $e' < m$), and then a table might be used to complete the division chain optimally.

This yields an algorithm for large exponents. Fix a suitable length $k$ for the division chain segments to be considered. The larger the choice of $k$, the longer the algorithm takes to complete, but the better the result. Now repeatedly perform the following:

i) generate all reasonably priced chains of $k$ divisors to reduce the current value of the exponent;

ii) select the cheapest one under the chosen criterion;

iii) apply this sub-chain to reduce the exponent.

The iterative procedure should terminate when the current value of the exponent becomes less than the upper limit of a pre-calculated table of optimal chains for small exponents. Of course, if in the last few iterations $k$ divisors were to reduce the exponent to close to 0 so that the costing criterion becomes inaccurate, then the offending chains can be curtailed earlier, say at the point where the exponent falls into the range of the table. Finally, with the exponent in the range of the table, the best way of completing the chain can be looked up.

To cost this, suppose $S$ is the sum over all divisors of the probability of the divisor giving an acceptable residue. So $S$ will be at most the number of divisors being used. For a better estimate, assuming a close to uniform distribution on residues of exponents modulo each divisor, $S$ is approximately $\sum_i n_i / m_i$ where $n_i$ is the number of residues associated with the divisor $m_i$. Then, to extend any division chain by one divisor, there are roughly $S$ possible choices (assuming successive choices are mostly independent).

Next suppose that $L$ is the result of averaging the logarithms of the divisors used in an optimal sub-chain, weighted according to their frequencies, i.e. the average number of bits by which a divisor in such a chain reduces the exponent. Assume finally that $T$ is the logarithm of the maximum exponent in the look-up table. Then, for exponents of $N$ bits, the above algorithm will require about $(N–T)/Lk$ iterations, each of which generates about $S^k$ sub-chains. So the work involved in choosing a division chain is roughly proportional to

$$S^k(N–T)D/Lk$$

where $D$ is the effort required to perform a single division of the exponent by a divisor. This shows that selecting a division chain for very large exponents is not much more difficult to deal with than one for smaller exponents; the work is proportional to the square of the number of bits.

For good performance, $k$ must not be too small since then the sub-chains would not be particularly good on average. Hence the best savings in time are made by reducing $S$ through a sensible choice of acceptable pairs. An obvious choice is the set of pairs used most frequently over a large range of optimal chains. Furthermore, picking large $k$ may be a waste of effort since the average reduction in numbers of multiplications declines exponentially with $k$ (see Section 9).

Many variations in the algorithm are clearly possible. For example, the subchains could be bounded by limiting the divisor product rather than by $k$, or $k$ might be varied when there is no good subchain of a specific length at some point. Also, only the initial few divisors from optimal sub-chains might be selected at each iteration and the test for optimality (which contains an approximation to $c$) might be varied dynamically. If a good chain is still not obtained, choosing a different first divisor or using all initial subchains is also possible.

## 8 Test Results

We next consider test results from a specific implementation of the algorithm in order to establish that a coefficient of $\frac{5}{4}$ is generally obtainable for exponents of the size used in, say, the RSA cryptosystem. None of the improvements suggested in the last paragraph were included for the tabulated results.

First, for divisors $m$ up to $2^8$ the minimal addition chains using only two registers were generated and a table was constructed of optimal division chains for exponents up to over $2^{20}$. In any subrange the method was found to use fewer than $\frac{5}{4}\log_2 e$ multiplications and squarings on average. In particular, between $2^{15}$ and $2^{16}$, apart from a few short chains such as those for $2^{15}$ itself and $2^{15}+2^a$ with $a < 15$, almost 90% require 19 or 20 multiplications, under 1.5% require 21, and none require more. Hence the variation in numbers of multiplications is very small, unlike for the binary or $A$, $A^3$ methods. Essentially no exponents require too many multiplications, but there is an increasing tail of exponents which can be dealt with using fewer multiplications than expected. It is this that makes searches for better division chains worthwhile.

Next, over ranges of at least $2^{17}$ exponents above $2^{20}$ and divisors up to $3\times2^9$, the average value of $v/\log_2 e$ was calculated for optimal chains and found to be about 2% under $\frac{5}{4}$. Indeed, successive ranges show this ratio has a tendency to decrease as the exponents increase. This is because once the reduction operations have made the exponent less than the maximum divisor, the choice of future divisors becomes progressively more limited. This leads to a poorer ratio for smaller exponents.

The coefficient for the whole of a large exponent is essentially of the form $\left(\sum_i v_i\right)/\left(\sum_i \log_2 m_i\right)$ where each $v_i/\log_2 m_i$ comes from an optimal subchain. From the above experimental data each such term should be less than $\frac{5}{4}$ on average since each $m_i$ will fall within the range investigated (unless $k$ is large). So the expected coefficient ought to be better than $\frac{5}{4}$. Although a given sub-chain being optimal at one step undoubtedly confers some properties on the next exponent, the division process seems to minimise most of these effects. So the relevant multiplicative properties of successive exponents appear to be mostly independent for successive steps of the algorithm. Thus, individual steps resulting in a large number of multiplications should not affect the rest of the algorithm. Indeed, for larger exponents which require a greater number of steps, the variance in the distribution of the number of multiplications should be smaller. Thus extreme cases will be evened out and this virtually guarantees finding a sequence for $e$ with under $\frac{5}{4}\log_2 e$ multiplications.

To test 512-bit exponents and investigate how small a coefficient $c$ might be possible, $2^{12}$ of the most frequently used divisor/residue pairs were chosen from optimal chains of exponents up to over $2^{20}$. They used 225 different divisors up to about $3\times2^{10}$, as listed in the Appendix. Choosing $k$ = 4, an average of under 635 multiplications was achieved for randomly generated 512-bit exponents, with a standard deviation of under 4 (see Table 2). This clearly improves on the coefficient $\frac{5}{4}$. Assuming the distribution is close to normal, a total of at most 640 multiplications or squarings can be effectively guaranteed for over 91% of all 512-bit exponents even before any of the enhancements described in Section 7. In comparison, the square & multiply and $A$, $A^3$ methods average about $\frac{3}{2}\times511$ = 766.5 and 2+ $\frac{4}{3}\times510$ = 682 multiplications with standard deviations of about 11.3 and 6.2 respectively. Interestingly, the *best* choice of radix $m$ for 512-bit exponents in the standard $m$-ary method requires space for 32 powers to be held and still only achieves a worse average of 635 multiplications (Table 2 of [15]).

| $k$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Av. No. Mults. | 650.36 | 641.7 | 637.33 | 634.67 |
| Stand. Dev. | 3.45 | 3.3 | 3.4 | 3.9 |

Table 2. Multiplication numbers for $2^{12}$ ($m$,$r$) pairs.

For comparison, Table 3 contains results for the 12 divisors used in Section 5, namely 2, 3, 5, 17, 33, 49, 65, 97, 129, 257, 513 and 1025, for which the 165 most frequently occurring divisor/residue pairs were selected and the difference test applied.

| $k$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Av. No. Mults. | 668.6 | 659.7 | 654.7 | 651.2 | 648.55 |
| Stand. Dev. | 4.75 | 4.35 | 3.8 | 3.54 | 3.6 |
| Diff. Test $c$ | 1.30 | 1.29 | 1.278 | 1.27 | 1.265 |
| Av. No. Divs. | 168 | 181 | 184 | 188 | 192 |

Table 3. Multiplication numbers for the 12 listed divisors.

The cost of this particular scheme for $k$ = 5 and a random exponent of $N$ = 512 bits is easy to estimate. The sum in Section 7 for approximating $S$ shows there are about 4.86 ways of extending any subchain by one more pair, and, if 192 divisors are used on average, the typical divisor has $N$/192 bits. So $L \approx 2.7$. Averaging over a few sub-chains would also give $L$. If no look-up table is used, there are around $192/k \approx 38$ iterations of the algorithm in which $S^k \approx 2620$. So a complete chain is obtained with an effort of roughly 2620×38 divisions of 256-bit numbers by numbers averaging 6 or 7 bits.

As this is the equivalent of about $2^9$ multiplications of pairs of 512-bit numbers, the effort is only justified in RSA cryptography if the key is re-used a number of times. However, for $k$ = 1 or 2 the effort is reduced to around 5 or 14 such multiplications respectively. This leads to nearly the same computational effort as the $A$, $A^3$ method if searching for a multiplication scheme must be done for every iteration.

| $k$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Av. No. Mults. | 656.33 | 649.46 | 644.64 | 641.44 | 639.28 |
| Stand. Dev. | 4.0 | 3.6 | 3.6 | 4.0 | 4.7 |
| Av. No. Divs. | 139 | 157 | 160 | 164 | 169 |

Table 4. Statistics for the 29 listed divisors.

Only a few divisors are actually required in order to achieve an average of under $\frac{5}{4}$. Twenty-nine divisors with low Hamming weight were considered: the twelve above together with 9, 41, 43, 81, 83, 161, 163, 193, 321, 323, 385, 641, 643, 769, 1281, 1283 and 1537. Here numbers of the forms $5\times2^n+1$ and $5\times2^n+3$ have a minimal addition chain of length $n$+3 by using the sequence 1, 2, 3, 5, 10, 20, ..., $5\times2^n$, $5\times2^n+1$ or $5\times2^n+3$. About $2^9$ of the most frequently occurring divisor/residue pairs were selected. From Table 4, the choice $k$ = 5 can be seen to yield a constant of under $\frac{5}{4}$ without having to use as many divisors as in Table 2. The work involved in obtaining a division chain here can be estimated from the table using $S \approx 7.12$: about 7 and 27 512-bit multiplications' worth of effort for $k$ = 1 and 2 respectively. Overall there is again little difference with the $A$, $A^3$ method unless the same RSA key $e$ is re-used.

## 9 Asymptotic Results

Comparing the contents of Tables 2-4, it is clear that adding further divisors brings diminishing returns. Neglecting the case $k$ = 1, the figures here (and further results not reproduced) all suggest an exponentially declining formula for the average number of multiplications that any choice of divisor/residue set will require. Good least squares approximations were obtained as follows:

Table 3: #mults $\approx$ 641.15 + 27.33 ×( 1.3656 )$^{k-1}$

Table 4: #mults $\approx$ 634.68 + 21.61 ×( 1.4687 )$^{k-1}$

Table 2: #mults $\approx$ 631.51 + 18.74 ×( 1.8074 )$^{k-1}$

Since the best divisors have already been included, these give a good indication that the method of division chains as described here is unlikely to yield an average of, say, 630 or fewer multiplications, although this could be improved a little by some of the enhancements suggested in Section 7.

The work involved in discovering an exponentiation scheme using these methods is related to the *S* of Section 8. For Tables 3, 4 and 2 respectively, S is roughly 4.86, 7.12 and 19.50. Consequently, it is possible to estimate the best divisor set to minimise the work necessary to achieve a given average number of multiplications. The 12 divisor case is best if the number of multiplications need not be under 664.5. Then a sub-chain length *k* of 1 should be chosen. The 29 divisor case becomes more worthwhile once fewer than 664.5 multiplications are required, again taking a subchain length of 1. However, the 225 divisor case would be chosen to reduce the number of multiplications below 650.5, choosing a subchain length according to Table 2. Below 634, say, a variety of independent full length chains can be generated from a number of different initial choices. These should contain a spread of multiplication numbers, some perhaps being sufficiently below average for use.

## 10 Conclusion

A straightforward, cheap algorithm has been described for reducing the number of multiplications normally performed in an exponentiation. Almost no extra memory is required: just one register more than for square and multiply. The average improvement is better than established methods with the same memory requirements, and exceeds that for *any* instance of the standard *m*-ary method for 512 bit exponents. Furthermore, the method is adaptable to a wide range of space and time resources, providing a variable search space from which better evaluation orders can be found. For any exponent *e* and very high probability of success, it has been shown how to find a scheme requiring under $\frac{5}{4}\log_2 e$ multiplications or squarings when only three registers are available.

*Acknowledgement:* The author would like to thank the anonymous referee who corrected the argument in Section 5 and kindly improved the presentation by supplying a table similar to that given.

## Bibliography

[1] I.E. Bocharova & B.D. Kudryashov, "Fast Exponentiation In Cryptography", *Lecture Notes in Computer Science*, Vol. **948**, Springer-Verlag, 1995, pp. 146-157.

[2] J. Bos & M. Coster, "Addition Chain Heuristics", *Proc. Crypto '89, Lecture Notes in Computer Science*, Vol. **435**, Springer-Verlag, 1990, pp. 400-407.

[3] E.F. Brickell, D.M. Gordon, K.S. McCurley and D.B. Wilson, "Fast Exponentiation with Precomputation", *Proc. Eurocrypt '92, Lecture Notes in Computer Science*, Vol. **658**, Springer-Verlag, 1993, pp. 200-207.

[4] C.Y. Chen, C.C. Chang & W.P. Yang, "Hybrid Method for Modular Exponentiation with Precomputation", *Electronics Letters*, Vol. **32**, No. 6, 1996, pp. 540-541.

[5] L.S. Danilchenko, "Efficient Algorithms for Remainder Computation and Exponentiation of Long Numbers", *Cybernetics and Systems Analysis*, Vol. **32**, No. 3, 1996, pp. 437- 441.

[6] V. Dimitrov & T. Cooklev, "Two algorithms for Modular Exponentiation Using Non-standard Arithmetics", *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. **E78-A**, No. 1, Jan 1995, pp. 82-87.

[7] V.S. Dimitrov, G.A. Jullien and W.C. Miller, "Theory and Applications for a Double-Base Number System", *Proc. 13th IEEE Symposium on Computer Arithmetic*, 6th-9th July 1997, Monterey, CA, IEEE, 1997, pp. 44-51.

[8] P. Downey, B. Leong & R. Sethi, "Computing Sequences with Addition Chains", *SIAM J. Comput.* vol. **10** No. 3, 1981, pp. 638-646.

[9] Ö. Eğecioğlu & Ç. K. Koç, "Fast Modular Exponentiation", *Comm., Control & Signal Processing*, ed. E. Arikan, Elsevier Science, 1990, pp. 188-194.

[10] Ö. Eğecioğlu & Ç. K. Koç, "Exponentiation Using Canonical Recoding", *Theoretical Computer Science*, 1994, Vol. **129**, No.2, pp. 407-417.

[11] P. Erdös, "Remarks on Number Theory III: On Addition Chains", *Acta Arithmetica*, Vol. **6**, 1960, pp. 77-81.

[12] L. C. K. Hui & K.-Y. Lam, "Fast Square-and-Multiply Exponentiation for RSA", *Electronics Letters*, vol. **30**, no. 17, Aug 1994, pp. 1396-1397.

[13] S. Kawamura, K. Takabayashi & A. Shimbo, "A Fast Modular Exponentiation Algorithm", *IEICE Trans. Comms, Electronics, Information and Systems*, Vol **E-74**, No. 8, Aug 1991, pp. 2136-2142.

[14] D. E. Knuth, "*The Art of Computer Programming*", vol. **2**, "*Seminumerical Algorithms*", §4.6.3, 2nd Edition, Addison-Wesley, 1981, pp. 441-466.

[15] Ç. K. Koç, "High Radix and Bit Recoding Techniques for Modular Exponentiation", *International J. of Computer Mathematics*, vol. **40**, 1991, No. 3-4, pp. 139-156.

[16] Ç.K. Koç, "Analysis of Sliding Window Techniques for Exponentiation", *Computers & Mathematics with Applications*, Vol. **30**, No.10, 1995, pp. 17-24.

[17] D.C. Lou & C.C. Chang, "Fast Exponentiation Method Obtained By Folding The Exponent In Half", *Electronics Letters*, Vol. **32**, No.11, 1996, pp. 984-985.

[18] J. Olivos, "On Vectorial Addition Chains", *Journal of Algorithms*, Vol. **2**, No. 1, 1981, pp. 13-21.

[19] P. de Rooij, "Efficient Exponentiation using Precomputation and Vector Addition Chains", *Proc Eurocrypt '94*, Lecture Notes in Computer Science, vol. **950**, Springer-Verlag, 1995, pp. 389-399.

[20] Y. Tsuruoka & K. Koyama, "Fast Exponentiation Algorithms based on Batch-Processing and Precomputation", *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. **E80-A**, No.1, Jan. 1997, pp. 34-39.

[21] C. D. Walter, "Space/Time Trade-offs for Higher Radix Modular Multiplication using Repeated Addition", *IEEE Trans. Comp.*, Vol. **46** No. 2, 1997, pp. 139-141.

[22] Y. Yacobi, "Exponentiating Faster with Addition Chains", *Advances in Cryptology - Eurocrypt 90*, Lecture Notes in Computer Science, vol. **473**, Springer-Verlag, 1991, pp. 222-229.

[23] A. Yao, "On the Evaluation of Powers", *SIAM J. Computing*, vol. **5**, 1976, pp. 100-103.

[24] S. M. Yen, "Improved Common-Multiplicand Multiplication and Fast Exponentiation by Exponent Decomposition", *IEICE Trans. on Fundamentals of Electronics Communications and Computer Sciences*, Vol. **E80-A**, No.6, 1997, pp. 1160-1163.

[25] S.M. Yen & C.-S. Laih, "Common-Multiplicand Multiplication and its Applications to Public Key Cryptography", *Electronics Letters*, Vol. **29**, No.17, 1993, pp. 1583-1584.

[26] C. N. Zhang, H. L. Martin, & D. Y. Y. Yun, "Parallel Algorithms and Systolic Array Designs for RSA Cryptosystem", *Proc. of the International Conference on Systolic Arrays*, K. Bromley, S.Y. Kung & E. Swartzlander (Eds.), Computer Society Press, San Diego, California, May 25-27, 1988, pp. 341-350.

## Appendix

This appendix lists the sets of divisors $m$ used in the examples of Sections 5 and 8, in some cases with i) the length of a minimal addition chain used for exponentiating to the power $m$ (subject to the space restrictions), and ii) lists of acceptable residues $r$ preceded by the number of extra multiplications required to include the $r$th power in the partial result. The residue listed is the best one to use, and so is not always the least non-negative one.

First are the 12 divisors used in Table 3 and Section 5. Using the ratio test to order the different cases (and the +3 lists not used in Section 5) achieves 671.66 multiplications on average for a 512-bit exponent. Applying the difference test instead with 1.3 as the approximation to $c$ yields a much improved 668.63, as in the table.

Divisor 2 Min chain length = 1
+0: 0 ; +1: 1

Divisor 3 Min chain length = 2
+0: 0 ; +1: 1 2

Divisor 5 Min chain length = 3
+0: 0 ; +1: 1 2 3 4

Divisor 17 Min chain length = 5
+0: 0 ; +1: 1 2 4 8 9 16 ; +2: 11 13

Divisor 33 Min chain length = 6
+0: 0 ; +1: 1 2 4 8 16 17 32 ; +2: 19 25

Divisor 49 Min chain length = 7
+0: 0
+1: 2 3 4 6 8 12 16 17 24 25 32 33 48
+2: 23

Divisor 65 Min chain length = 7
+0: 0 ; +1: 2 4 8 16 32 33 64
+2: 24 37 49 56

Divisor 97 Min chain length = 8
+0: 0
+1: 2 3 4 6 8 12 16 24 32 33 48 49 64 65 96
+2: 23 41 53 55 69

Divisor 129 Min chain length = 8
+0: 0 ; +1: 2 4 8 16 32 64 65 128
+2: 67 73 81 96 97 192

Divisor 257 Min chain length = 9
+0: 0 ; +1: 2 4 8 16 32 64 128 129 256
+2: 12 18 20 40 48 66 72 96 131 133 136 137
144 145 160 161 192 193
+3: 139 147 149

Divisor 513 Min chain length = 10
+0: 0
+1: 2 4 8 16 32 64 128 256 257 512
+2: 34 66 72 259 261 265 273 289 385
+3: 269 277 281 293

Divisor 1025 Min chain length = 11
+0: 0
+1: 1 2 4 8 16 32 64 128 256 512 513 1024
+2: 12 24 36 48 515 517 521 529 544 545 576
577 769 1152
+3: 523 531 547 549 561 579 581 585

Table 4, which achieved under $\frac{5}{4}\log_2 e$ multiplications, used the following 29 divisors in 519 pairs:

2 3 5 9 17 33 41 43 49 65
81 83 97 129 161 163 193 257 321 323
385 513 641 643 769 1025 1281 1283 1537

The following 225 divisors were used in constructing Table 2:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 5 | 7 | 9 | 11 | 13 | 14 | 17 | 19 |
| 23 | 26 | 31 | 33 | 37 | 41 | 43 | 49 | 50 | 59 |
| 61 | 65 | 66 | 67 | 73 | 77 | 83 | 97 | 98 | 107 |
| 109 | 113 | 121 | 129 | 131 | 133 | 137 | 145 | 149 | 163 |
| 168 | 193 | 194 | 197 | 199 | 211 | 227 | 229 | 233 | 257 |
| 259 | 261 | 265 | 281 | 289 | 290 | 292 | 293 | 296 | 298 |
| 321 | 371 | 373 | 385 | 386 | 389 | 397 | 409 | 437 | 449 |
| 481 | 483 | 513 | 515 | 517 | 521 | 529 | 540 | 545 | 546 |
| 552 | 553 | 560 | 561 | 562 | 577 | 578 | 580 | 581 | 584 |
| 586 | 588 | 608 | 641 | 642 | 643 | 644 | 646 | 648 | 656 |
| 673 | 677 | 683 | 730 | 739 | 769 | 770 | 773 | 775 | 780 |
| 781 | 785 | 793 | 801 | 803 | 809 | 812 | 813 | 846 | 848 |
| 852 | 869 | 884 | 888 | 897 | 899 | 900 | 901 | 902 | 904 |
| 905 | 906 | 912 | 914 | 920 | 928 | 944 | 961 | 964 | 966 |
| 968 | 972 | 976 | 984 | 1008 | 1025 | 1027 | 1029 | 1033 | 1041 |
| 1057 | 1072 | 1073 | 1097 | 1153 | 1154 | 1156 | 1157 | 1170 | 1200 |
| 1264 | 1281 | 1282 | 1283 | 1348 | 1352 | 1361 | 1364 | 1384 | 1396 |
| 1412 | 1415 | 1422 | 1424 | 1432 | 1444 | 1448 | 1456 | 1464 | 1468 |
| 1470 | 1477 | 1482 | 1488 | 1490 | 1500 | 1504 | 1520 | 1524 | 1528 |
| 1537 | 1538 | 1540 | 1541 | 1543 | 1544 | 1546 | 1550 | 1553 | 1556 |
| 1561 | 1564 | 1569 | 1572 | 1576 | 1585 | 1588 | 1592 | 1593 | 1601 |
| 1603 | 1604 | 1606 | 1608 | 1609 | 1610 | 1612 | 1613 | 1616 | 1617 |
| 1618 | 1624 | 1626 | 1648 | 1671 | | | | | |