Comparison of three modular reduction functions

Antoon Bosselaers, René Govaerts and Joos Vandewalle

Katholieke Universiteit Leuven, Dept. Electrical Engineering-ESAT
Kardinaal Mercierlaan 94, B-3001 Heverlee, Belgium

antoon.bosselaers@esat.kuleuven.ac.be

25 October 1993

Abstract. Three modular reduction algorithms for large integers are compared with respect to their performance in portable software: the classical algorithm, Barrett's algorithm and Montgomery's algorithm. These algorithms are a time critical step in the implementation of the modular exponentiation operation. For each of these algorithms their application in the modular exponentiation operation is considered. Modular exponentiation constitutes the basis of many well known and widely used public key cryptosystems. A fast and portable modular exponentiation will considerably enhance the speed and applicability of these systems.

1 Introduction

The widely claimed poor performance of public key cryptosystems in portable software usually results in faster, but non-portable assembly language implementations. Although they always will remain faster than their portable counterparts, their major drawback is the fact that their applicability is restricted to a limited number of computers. This means that the development effort has to be repeated for a different processor. A way out is to develop portable software that approaches the speed of an assembly language implementation as closely as possible. A primary candidate for the high level language is the versatile and standardized C language.

A basic operation in public key cryptosystems is the modular reduction of large numbers. An efficient implementation of this operation is the key to high performance. Three well known algorithms are considered and evaluated with respect to their software performance. It will be shown that they all have their specific behavior resulting in a specific field of application. No single algorithm is able to meet all demands. However a good implementation will leave minor differences in performance between the three algorithms.

In Section 2 the representation of large numbers in our implementation is discussed. The three reduction algorithms are described and evaluated in Section 3 and their behavior with respect to their argument is considered in Section 4. Section 5 looks at their use in the modular exponentiation operation. Finally, the conclusion is formulated in Section 6.

2 Representation of numbers

The three algorithms for modular reduction are described for use with large nonnegative integers expressed in radix b notation, where b can be any integer ≥ 2. Although the descriptions are quite general and unrelated to any particular computer, the best choice for b will of course be determined by the computer and the programming language used for the implementation of these algorithms. In particular, b should be chosen such that multiplications with, divisions by, and reductions modulo b^k $(k > 0)$ are easy. The most obvious choice for b will therefore be one of the programming language's available integer types, in which case these three operations are reduced to respectively shifting to the left over k digits, shifting to the right over k digits (i.e., discarding the least significant k digits) and discarding all but the least significant k digits. Moreover the larger b is, the smaller the number of radix b operations to perform the same operation, and hence the faster it will be. On the other hand all multiprecision operations are performed using a number of primitive single precision operations, one of which is the multiplication of two one-digit integers giving a two-digit answer. This means that besides a basic integer type that can represent the values 0 through $b - 1$, we need an integer type that is able to represent the values 0 through $(b - 1)^2$. Since we normally want the ability to add and multiply concurrently [5, Algorithm 4.3.1M], we need an integer type that is able to represent the values 0 through $b^2 - 1$, i.e., a type which is at least twice as long as the basic type.

In the sequel let m be the modulus

$$m = \sum_{i=0}^{k-1} m_i b^i \,, \quad 0 < m_{k-1} < b \text{ and } 0 \leq m_i < b \,, \text{ for } i = 0, 1, \ldots, k-2 \,,$$

and $x \geq m$ be the number to be reduced modulo m

$$x = \sum_{i=0}^{l-1} x_i b^i \,, \quad 0 < x_{l-1} < b \text{ and } 0 \leq x_i < b \,, \text{ for } i = 0, 1, \ldots, l-2 \,,$$

both expressed in radix b notation.

3 Comparative Descriptions and Evaluation

The three algorithms to compute $x \bmod m$ are stated in terms of addition, subtraction and multiplication of both single and multiple precision integers, as well as single precision division, division by a power of b and reduction modulo a power of b. All algorithms require a precalculation, that depends only on the modulus, and hence has to be performed once for a given modulus m. Barrett's and Montgomery's methods require that the argument x is smaller than respectively b^{2k} and mb^k, where $k = \lfloor \log_b m \rfloor + 1$. If, as is mostly the case, these algorithms are used to reduce the product of two integers smaller than the modulus, this restriction will have no impact on their applicability, for then $x < m^2 < mb^k < b^{2k}$. The classical algorithm on the other hand imposes no restriction on the size of x and can easily be adapted to a general purpose division algorithm giving both quotient and remainder.

The *classical algorithm* is a formalization of the ordinary $l-k$ step pencil-and-paper method, each step of which is the division of a $(k+1)$-digit number z by the k-digit divisor m, yielding the one-digit quotient q and the k-digit remainder r. Each remainder r is less than m, so that it can be combined with the next digit of the dividend into the $(k+1)$-digit number $rb+$ (next digit of dividend) to be used as the new z in the next step.

The formalization by *D. Knuth* [5, Algorithm 4.3.1A] consists in estimating the quotient digit q as accurately as possible. Dividing the two most significant digits of z by m_{k-1} will result in an estimate that is never too small and, if $m_{k-1} \geq \lfloor \frac{b}{2} \rfloor$, at most two in error. Using an additional digit of both z and m (i.e., using the three most significant digits of z and the two most significant digits of m) this estimate can be made almost always correct, and at most one in error (an event occurring with probability $\approx 2/b$). The pseudocode of this algorithm is given in Algorithm 1.

```
if (x > mb^(l-k)) then
   x = x - mb^(l-k);
for (i = l - 1; i > k - 1; i--) do {
   if (x_i == m_(k-1)) then
      q = b - 1;
   else
      q = (x_i b + x_(i-1)) div m_(k-1);
   while (q(m_(k-1) b + m_(k-2)) > x_i b^2 + x_(i-1) b + x_(i-2)) do
      q = q - 1;
   x = x - qmb^(i-k);
   if (x < 0) then
      x = x + mb^(i-k);
}
```

Algorithm 1. Classical Algorithm ($m_{k-1} \geq \lfloor \frac{b}{2} \rfloor$)

In general the normalization $m^* = \lfloor \frac{b}{m_{k-1}} \rfloor m$ will ensure that $m^*_{k-1} \geq \lfloor \frac{b}{2} \rfloor$. On a binary computer b will be a power of 2, and hence the normalization process can be implemented more efficiently as a shift over so many bits to the left as is necessary to make the most significant bit of the most significant digit of m equal to 1. At the end the correct remainder r is obtained by applying to it the inverse of the normalization on m, i.e., by dividing it by $\lfloor \frac{b}{m_{k-1}} \rfloor$ or by shifting it to the right over the same number of bits as m was shifted over to the left during normalization.

A slightly more involved kind of normalization [7, 10] fixes one or more of the modulus' most significant digits in such a way that the most significant digit of z can be used as a first estimate for q, resulting in a faster reduction. However this normalization will increase the length of a general modulus by at least one digit, and hence all intermediate results of a modular exponentiation as well. First experiments seem to indicate that what is saved during a modular exponentiation in the modular reductions, is lost again in additional multiplications. It is as yet unclear whether further optimalization will result in a faster modular exponentiation.

P. Barrett [1] introduced the idea of estimating the quotient x div m with operations that either are less expensive in time than a multiprecision division by m (viz., 2 divisions by a power of b and a partial multiprecision multiplication), or can be done as a precalculation for a given m (viz., $\mu = b^{2k}$ div m, i.e., μ is a scaled estimate of the modulus' reciprocal). The estimate $\hat{q}$ of x div m is obtained by replacing the floating point divisions in $q = \lfloor (x/b^{2k-t})(b^{2k}/m)/b^t \rfloor$ by integer divisions:

$$\hat{q} = \left((x \text{ div } b^{2k-t})\mu\right) \text{ div } b^t \,.$$

This estimate will never be too large and, if $k < t \leq 2k$, the error is at most two:

$$x \text{ div } m - 2 \leq \hat{q} \leq x \text{ div } m \,, \quad \text{for } k < t \leq 2k \,.$$

It can be shown that for about 90% of the values of $x < m^2$ and m the initial value of $\hat{q}$ will be equal to x div m and only in 1% of cases $\hat{q}$ will be two in error. The only influence of the t least significant digits of the product $(x \text{ div } b^{2k-t})\mu$ on the most significant part of this product is the carry from position t to position $t+1$. This carry can be accurately estimated by only calculating the digits at position $t-1$ and t, which has the advantage that the calculation of the $t-2$ least significant digits of the product is avoided. The resulting quotient is never too large and almost always the same as $\hat{q}$, and, if $b > l-k$, at most one in error. Moreover the number of single precision multiplications and the resulting error are more or less independent of t. The best choice for t, resulting in the least single precision multiplications and the smallest maximal error, is $k+1$, which also was Barrett's original choice. The calculation of $\hat{q}$ can be speeded up even slightly more by normalizing m, such that $m_{k-1} \geq \lfloor \frac{b}{2} \rfloor$. This way $l-k+1$ single precision multiplications can be transformed into as many additions.

An estimate $\hat{r}$ for x mod m is then given by $\hat{r} = x - \hat{q}m$, or, as $\hat{r} < b^{k+1}$ (if $b > 2$), by

$$\hat{r} = (x \bmod b^{k+1} - (\hat{q}m) \bmod b^{k+1}) \bmod b^{k+1} \,,$$

which means that once again only a partial multiprecision multiplication is needed. At most two further subtractions of m are required to obtain the correct remainder. Barrett's algorithm can therefore be implemented according to the pseudocode of Algorithm 2.

```
q = ((x div b^{k-1})μ) div b^{k+1};
x = x mod b^{k+1} - (qm) mod b^{k+1};
if (x < 0) then
   x = x + b^{k+1};
while (x ≥ m) do
   x = x - m;
```

Algorithm 2. Barrett's Algorithm ($\mu = b^{2k}$ div m)

By representing the residue classes modulo m in a nonstandard way, *Montgomery's method* [6] replaces a division by m with a multiplication followed by a division by a power of b. This operation will be called Montgomery reduction.

Let $R > m$ be an integer relatively prime to m such that computations modulo R are easy to process: $R = b^k$. Notice that the condition $\gcd(m, b) = 1$ means that this method can not be used for all moduli. In case b is a power of 2, it simply means that m should be odd. The m-residue with respect to R of an integer $x < m$ is defined as $xR \bmod m$. The set $\{xR \bmod m \,|\, 0 \leq x < m\}$ clearly forms a complete residue system. The Montgomery reduction of x is defined as $xR^{-1} \bmod m$, where R^{-1} is the inverse of R modulo m, and is the inverse operation of the m-residue transformation. It can be shown that the multiplication of two m-residues followed by Montgomery reduction is isomorphic to the ordinary modular multiplication.

The rationale behind the m-residue transformation is the ability to perform a Montgomery reduction $xR^{-1} \bmod m$ for $0 \leq x < Rm$ in almost the same time as a multiplication. This is based on the following theorem:

Theorem 1 P. Montgomery. *Let* $m' = -m^{-1} \bmod R$. *If* $\gcd(m, R) = 1$, *then for all integers* x, $(x + tm)/R$ *is an integer satisfying*

$$\frac{x + tm}{R} \equiv xR^{-1} \pmod{m}$$

where $t = xm' \bmod R$.

It can easily be verified that the estimate $\hat{x} = (x + tm)/R$ for $xR^{-1} \bmod m$ is never too small and the error is at most one. This means that a Montgomery reduction is not more expensive than two multiplications, and one can do even better: almost twice as fast. Hereto, it is sufficient to observe [2] that the basic idea of Montgomery's Theorem is to make x a multiple of R by adding multiples of m. Instead of computing all of t at once, one can compute one digit t_i at a time, add $t_i m b^i$ to x, and repeat. This change allows to compute $m'_0 = -m_0^{-1} \bmod b$ instead of m'. It turns out to be a generalization of Hensel's odd division for computing inverses of "2-adic" numbers (introduced by *K. Hensel* around the turn of the century, see e.g., [3]) to a representation using b-ary numbers that have $\gcd(m_0, b) = 1$ [9].

A Montgomery modular reduction can be implemented according to the pseudocode of Algorithm 3. If x is the product of two m-residues, the result is the m-residue of the remainder, and the remainder itself is obtained by applying one additional Montgomery reduction. However both the initial m-residue transformation of the argument(s) and the final inverse transformation (Montgomery reduction) are only necessary at the beginning, respectively the end of an operation using Montgomery reduction (e.g., a modular exponentiation).

```
for (i = 0; i < k; i++) do {
    t_i = (x_i · m'_0) mod b;
    x = x + t_i m b^i;
}
x = x div b^k;
if (x ≥ m) then
    x = x − m;
```

Algorithm 3. Montgomery's Algorithm ($m'_0 = -m_0^{-1} \bmod b$, Hensel's b-ary division)

An indication of the attainable performance of the different algorithms will be given by the number of single precision multiplications and divisions necessary to reduce an argument twice as long as the modulus ($l = 2k$). This approach is justified by the fact that a multiplication and a division are the most time consuming operations in the inner loops of all three algorithms, with respect to which the others are negligible. The number of multiplications and divisions in Table 1 are only for the reduction operation, i.e., they do not include the multiplications and divisions of the precalculation, the argument transformation, and the postcalculation. Our reference operation is the multiplication of two k-digit numbers.

Table 1 indicates that if only the reduction operation is considered (i.e., without the precalculations, argument transformations, and postcalculations) and for arguments twice the length of the modulus, Montgomery's algorithm (only for moduli m for which $\gcd(m_0, b) = 1$) is clearly faster than both Barrett's and the classical one and almost as fast as a multiplication. Barrett's and the classical algorithm will be almost equally fast, with a slight advantage for Barrett.

These observations are confirmed by a software implementation of these algorithms, see Table 2. The implementation is written in ANSI C [4] and hence should be portable to any computer for which an implementation of the ANSI C standard exists. All figures in this article are obtained on a 33 MHz 80386 based

Table 1. Complexity of the three reduction algorithms in reducing a $2k$-digit number x modulo a k-digit modulus m.

Algorithm	Classical	Barrett	Montgomery	Multiplication
Multiplications	$k(k+2.5)$	$k(k+4)$	$k(k+1)$	k^2
Divisions	k	0	0	0
Precalculation	Normalization	b^{2k} div m	$-m_0^{-1}$ mod b	None
Arg. transformation	None	None	m-residue	None
Postcalculation	Unnormalization	None	Reduction	None
Restrictions	None	$x < b^{2k}$	$x < mb^k$	None

Table 2. Execution times for the reduction of a $2k$-digit number modulo a k-digit modulus m for the three reduction algorithms compared to the execution time of a $k \times k$-digit multiplication ($b = 2^{16}$, on a 33 MHz 80386 based PC with WATCOM C/386 9.0).

	Length of	Times in mseconds			
k	m in bits	Classical	Barrett	Montgomery	Multiplication
8	128	0.278	0.312	0.205	0.182
16	256	0.870	0.871	0.668	0.632
32	512	3.05	2.84	2.43	2.36
48	768	6.56	5.96	5.33	5.19
64	1024	11.39	10.23	9.33	9.12

PC using the 32-bit compiler WATCOM C/386 9.0. The radix b is equal to 2^{16}, which means that Montgomery's algorithm is only applicable to odd moduli.

However an operation using Barrett's or Montgomery's modular reduction methods will only be faster than the same operation using the classical modular reduction if the pre- and postcalculations and the m-residue transformation (only for Montgomery) are subsequently compensated for by enough (faster) modular reductions. An example of such an operation is modular exponentiation. This also means that for a single modular reduction the classical algorithm is the obvious choice, as the pre- and postcalculation only involve a very fast and straightforward normalization process.

4 Behavior w.r.t. argument

The execution time for the three reduction functions depends in a different way on the length of the argument. The time for a reduction using the classical algorithm or Barrett's method will vary linearly between their maximum value (for an argument twice as long as the modulus) and almost zero (for an argument as long as the modulus). For arguments smaller than the modulus no reduction takes place, as they are already reduced. On the other hand, the time for a reduction using Montgomery's method will be independent of the length of the argument. This is a consequence of the fact that in all cases, whatever the value of the argument, a modular multiplication by R^{-1} takes place. This means that both the classical algorithm and Barrett's method will be faster than Montgomery's method below a certain length of the argument. This is illustrated in Figure 1 for a 512-bit modulus. However in most cases the argument will be close to twice the length of the modulus, as it normally is the product of two values close in length to that of the modulus.

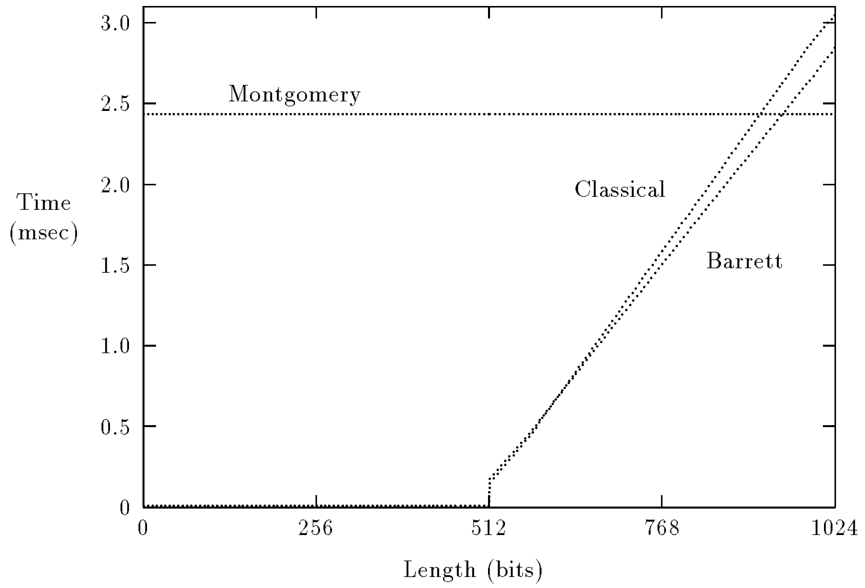

Fig. 1. Typical behavior of the three reduction functions in reducing a number up to twice the length of the modulus ($b = 2^{16}$, length of the modulus = 512 bits, on a 33 MHz 80386 based PC with WATCOM C/386 9.0).

In addition, all the modular reduction functions have, for a given length, input values for which they perform faster than average in reducing them. For some of these inputs the gain in speed can be quite substantial. Since these input values are different for each of the reduction functions, none of the functions is the fastest for all inputs of a given length.

Montgomery's method will be faster than average in reducing m-residues with consecutive zeroes in its *least* significant digit positions. The gain in speed will be directly proportional to the number of zero digits. The same applies to arguments that produce, after n steps $(0 < n < k)$ in Montgomery's algorithm, a number of consecutive zero digits in the intermediate value x. For example, the argument

$$x = hb^k + b^k - (\sum_{i=0}^{n-1} t_i m b^i) \bmod b^k \, ,$$

where

$$\begin{aligned} &0 < h < b^{l-k} \\ &t_i = \big(y_i(-m_0^{-1} \bmod b)\big) \bmod b \, , \quad 0 \le y_i < b \, , \end{aligned}$$

produces after n steps $k - n$ consecutive zeroes, with once again a speed gain directly proportional to the number of consecutive zero digits.

Barrett's method will be faster than average, and possibly faster than Montgomery's method, for an argument x with zero digits among its $k+1$ *most* significant digits or that produces an approximation $\hat{q}$ of x div m containing zero digits. An example of the latter will be encountered in the next paragraph.

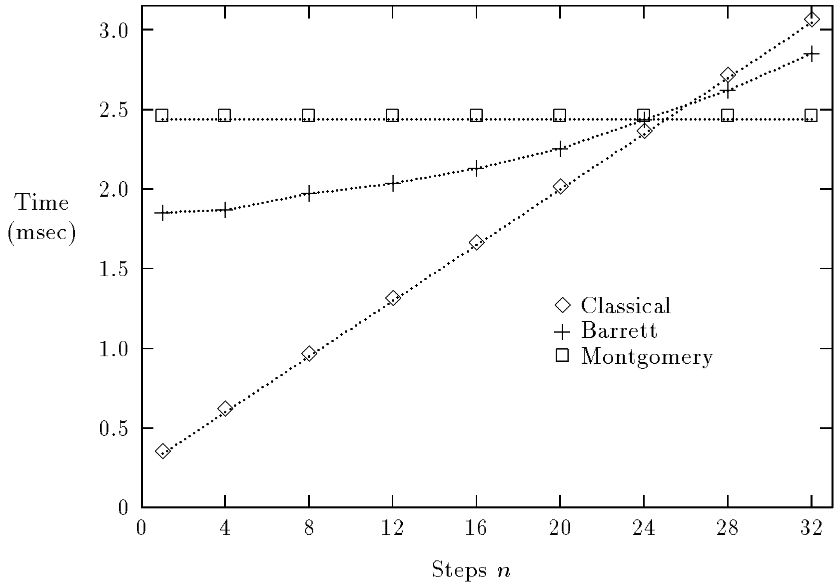

Fig. 2. Behavior of the three reduction functions in reducing the argument $x = gmb^{k-n} + h$, where $0 < n \le k$, $0 < g < b^n$ and $0 \le h < m$ for the case $k = 32$ ($b = 2^{16}$, length of the modulus = 512 bits, on a 33 MHz 80386 based PC with WATCOM C/386 9.0).

The central part of the classical algorithm is the $(l - k)$-fold loop, in each iteration of which a digit of the quotient x div m is determined. Therefore the classical algorithm will be faster than average, and possibly faster than Montgomery's and Barrett's method, for an argument that produces a quotient with a number of zero digits. For example, the argument

$$x = gmb^{l-k-n} + h\,, \quad \begin{array}{l} k < l \leq 2k \\ 0 < n \leq l-k \\ 0 < g < b^n \\ 0 \leq h < m\,, \end{array}$$

produces a quotient $q = gb^{l-k-n}$ containing $l-k-n$ zero digits in its *least* significant positions, and hence only n steps of the central loop will be executed. As the time for a reduction using the classical algorithm is clearly directly proportional to the number of non-void steps in the central loop, the reduction of the above argument will be considerably faster than average. Moreover, since the actual quotient contains $l-k-n$ zero digits, the reduction of this argument using Barrett's method will be faster than average as well: in 90% of the cases the approximation $\hat{q}$ will be equal to q, and hence the multiplication $\hat{q}m \bmod b^{k+1}$ will consist of n steps only instead of the $l-k$ steps in the average case. This means that in this case the classical algorithm will be faster than Barrett's method, which in turn will be faster than Montgomery's method. This situation is illustrated in Figure 2 for the case $l = 2k = 64$.

5 Use in modular exponentiation

The calculation of $a^e \bmod m$ in our implementation uses an (optimized) p-ary generalization of the standard binary square and multiply method, in which a table of small powers of a is used. For $p = 16$ this reduces the mean number of modular multiplications to about $\frac{1}{5}$ the number of bits in e (compared to $\frac{1}{2}$ for binary square and multiply). The number of squarings in both methods is the same and equal to the number of bits in e. Each of the three reduction algorithms can be used in this implementation, resulting in three modular exponentiation functions. The speed differences between the reduction functions will consequently be reflected in speed differences between the exponentiation functions. For a full length exponentiation (length of argument = length of exponent = length of modulus) the Montgomery based exponentiation will be slightly faster than the Barrett based exponentiation, in turn being slightly faster than the classical one, see Table 3

The behavior of the reduction functions with respect to the size of the argument will also be reflected in the behavior of the exponentiation functions. The exponentiation of an argument a smaller in length than the modulus will for the classical and Barrett's algorithm result in a table of small powers of a containing values which are still smaller in length than the modulus. Hence each multiplication by an entry of this table will yield a product that is shorter than twice the length of the modulus. The subsequent reduction will be faster than average, as the execution time of the classical and Barrett's algorithm depends linearly on the length of its argument. For these two algorithms the exponentiation of an argument smaller in length than the modulus will thus be faster than an exponentiation of a full length argument. Moreover for small enough

Table 3. Execution times for a full length modular exponentiation (length of argument = length of exponent = length of modulus, $b = 2^{16}$, on a 33 MHz 80386 based PC with WATCOM C/386 9.0).

Length of m in bits	Times in seconds		
	Classical	Barrett	Montgomery
128	0.072	0.078	0.062
256	0.430	0.430	0.366
512	2.95	2.83	2.55
768	9.46	8.90	8.28
1024	21.74	20.30	19.14

arguments the exponentiation using these algorithms will be even faster than the exponentiation using Montgomery's modular reduction, which is explained by the fact that for these arguments not only the products but also some squares will be shorter than twice the length of the modulus. This is illustrated in Figure 3. Barrett based exponentiation is therefore the best choice to perform Rabin primality tests [8] with small bases.

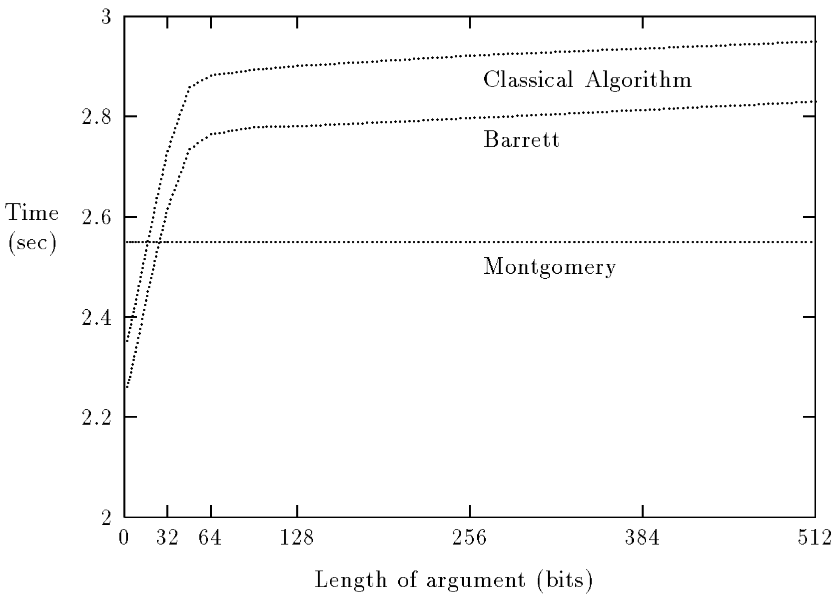

Fig. 3. Typical behavior of the exponentiation functions based on the three reduction functions in exponentiating a number up to the length of the modulus ($b = 2^{16}$, length of modulus and exponent = 512 bits, on a 33 MHz 80386 based PC with WATCOM C/386 9.0).

6 Conclusion

A theoretical and practical comparison has been made of three algorithms for the reduction of large numbers. It has been shown that in a good portable implementation the three algorithms are quite close to each other in performance. The classical algorithm is the best choice for single modular reductions. Modular exponentiation based on Barrett's algorithm is superior to the others for small arguments. For general modular exponentiations the exponentiation based on Montgomery's algorithm has the best performance.

References

1. P.D. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," *Advances in Cryptology, Proc. Crypto'86, LNCS 263*, A.M. Odlyzko, Ed., Springer-Verlag, 1987, pp. 311–323.
2. S.R. Dussé and B.S. Kaliski, "A cryptographic library for the Motorola DSP56000," *Advances in Cryptology, Proc. Eurocrypt'90, LNCS 473*, I.B. Damgård, Ed., Springer-Verlag, 1991, pp. 230–244.
3. K. Hensel, *Theorie der algebraischen Zahlen*, Leipzig, 1908.
4. *"American National Standard for Programming Languages—C,"* ISO/IEC Standard 9899:1990, International Standards Organization, Geneva, 1990.
5. D.E. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, 2nd Edition*, Addison-Wesley, Reading, Mass., 1981.
6. P.L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, Vol. 44, 1985, pp. 519–521.
7. J.-J. Quisquater, presentation at the rump session of Eurocrypt'90.
8. M.O. Rabin, "Probabilistic algorithms for testing primality," *J. of Number Theory*, Vol. 12, 1980, pp. 128–128.
9. M. Shand and J. Vuillemin, "Fast Implementations of RSA cryptography," *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 252–259.
10. C.D. Walter, "Faster modular multiplication by operand scaling," *Advances in Cryptology, Proc. Crypto'91, LNCS 576*, J. Feigenbaum, Ed., Springer-Verlag, 1992, pp. 313–323.