Modular arithmetic and the Montgomery multiplication

by Theo Kortekaas

Introduction

Modular arithmetic is arithmetic with residues. If we make a flight of 31 hours and we leave today at 12:00 we will be tomorrow 19:00 at our destination; 7 hours later than the departure time today.
We calculate this by dividing duration of the journey by 24; the number of hours in a day. The rest we add to the time of departure and so we arrive at 19:00. (Here we must of course also take into account the time difference between the location of departure and the location of arrival).

This is a simple example of arithmetic with residues.
In number theory modular arithmetic occurs frequently. In practice it plays an important role; in particular, with the calculation of large prime numbers which are used in cryptography. And cryptographic applications we use almost daily; usually unconsciously: for example, for online banking or credit cards.

The calculation of residues for division by large numbers takes a lot of processing power of a computer, and thus a lot of time. Therefore, all kinds of methods have been developed to accelerate modular arithmetic with large numbers. One such method was developed by Peter Montgomery in 1985 and is named Montgomery Reduction and Montgomery Multiplication[1].
In addition, there are other techniques to speed up the calculating with residues in certain situations. On the internet you can find lots of information on modular arithmetic, but this information is not always accessible; often the information contains complex formulas or the information is not complete. In this paper, we try to give the simplest possible explanation of the Montgomery multiplication and other methods used for modular arithmetic with huge numbers.

Rehearsal

It is almost unavoidable to use some mathematical concepts and simple formulas in this document. Therefore, a small repetition of mathematical concepts, symbols and formulas is here included. Furthermore, a small introduction to the processing and storage of numbers in a binary computer is included.

Modulus

The residue of a number *a* that is divided by a number *m* will be represented mathematically as follows:
$a \pmod{m}$. Number *m* is called the modulus. The result will be preceded by the symbol ≡, this in contrast to the equal (=) sign at the result of a sum without modulus.
So the remainder *r* of the division a/m is: $a \pmod{m} \equiv r$. In general, in modular arithmetic, only natural numbers are involved; so no decimals or fractions. All numbers that can form a residue at a particular modulus *m* is called the modulus domain *m*. All integers from zero up to and including $m-1$ belong to that domain. If a number falls outside this domain we can bring the number within the domain by subtracting the modulus one or more times. If a number is negative, we can bring the number within the domain by adding the modulus one or more times.
In principle negative numbers do not belong to the domain of the modulus. There are some exceptions where negative numbers are used in modular arithmetic, such as with the NTT (Number-theoretic Transform) [2].

Powers

If *a* is a natural number greater than 1, then:
$a^0 = 1$
$a^1 = a$
$a^3 \times a^2 = a^{(3+2)} = a^5$
$a^3 : a^2 = a^{(3-2)} = a^1$
The inverse of $a^1 = a^{-1}$; So, this is one divided by *a*.
Thus, $a^1 \times a^{-1} = a^{(1-1)} = a^0 = 1$

The binary computer

In the binary world of the computer, binary numbers are typically stored in the form of a row of bits. In the modern computer is that a row (register) of 32 or 64 bits. This is called an integer (register) and forms the unit for making calculations in a computer. Each bit can have a value of 0 or 1 and represents a power of two: the bitnummering for a 32-bit integer is from 0 to 31.

The first bit (bit 0) represents the value 2^0 when the bit it is set to 1, otherwise it represents the value 0. The following bit (bit 1) represents the value 2^1 when it is set to 1; otherwise the value is 0. Then follows 2^2, 2^3, 2^4 ... etc. until the value 2^{31} for bit bit 31 when it is set to 1. The total value of an integer is the value of all the bits added together.

Graphically a 32-bit integer register can be viewed as a row of 32 boxes where box 0 is at the right side and box 31 is at the left side. Artificially we can make a greater integer by coupling multiple 32- or 64-bit integers together.

We can multiply the contents of an integer by two, by shifting the content of each of the bits one place to the left. What was at first the content of bit number 30, now goes to bit number 31; what stood at bit 29 goes to bit 30 ... etc. Bit number 0 gets a 0 as content. The content of what was in bit number 31 disappears (or comes into a kind of overflow bit, also referred to as carry-flag).

We can divide the contents of an integer by two, by shifting the content of each of the bits one place to the right. What was at first the content of bit number two, now goes to bit number one; what was on bit three goes to bit two ... etc. What was at bit 31 goes to bit 30, and a zero is put in bit 31 (for a 32-bit integer). The content of bit o disappears.

In most computers, special instructions are available for shifting the contents of an integer register: instruction SHR stand for Shift Right, and instruction SHL stands for Shift Left.

Special moduli

There are some formats of moduli that simplifies modulair computations. Especially in computerprograms it becomes simpler with such moduli[3]. Calculations with such moduli can also be implemented in computer hardware. Here are three such moduli:

mod (2^n)

With mod (2^n), we determine the rest of a number that we divide by a power of two. If this number is stored in a binary computer in the form of one or more coupled integers, then we can divide the contents of the integer (s) by a power n of two (2^n) by shifting the contents of the bits n places to the the right. In this case, the content of n bits to the right disappears. We can see the bits that are shifted out (shifted to the right) as a separate integer. We can, then, in that separate integer register, put the bits in the right place, so that the bit which is shifted out first, at the location of bit 0 comes, etc. Then, the contents of this separate integer register forms the remainder of the division by (the power of) two. Thus, we can quite easily determine the value of a number mod (2^n).

mod $(2^n + 1)$

In number theory modulair calculations with modulus (2^n+1) occurs frequently. The modulus (2^n+1) is also known as "Fermat modulus", as a Fermat number has the same shape: 2^n+1. The difference is that with a Fermat *number*, the exponent n itself has to be a power of two, while with a Fermat *modulus* this is not a requirement.
The Fermat modulus is used in the "Strassen Schönhage Algorithm" for the multiplication of large numbers [4].

In calculations of numbers modulo (2^n+1), the following trick can be used: First divide the number by (2^n). We then get a quotient q and a remainder r. We had to divide by (2^n+1); so we must reduce the remainder by q. If r then becomes negative, we must add the modulus (2^n+1) again to the remainder r. As we have seen in a binary computer, the division of a binary number by (a power n of) 2 is accomplished by means of a shift to the right over n bits.

The bits that remain form the quotient and the *n* bits that are shifted out (the *n* of the modulus) form the remainder, after the quotient is subtracted from the remainder [5].

mod (2^n - 1)

This modulus is referred to as the Mersenne modulus, because it has the same shape as a Mersenne number. This form of modulus is for instance used in the Lucas Lehmer test for Mersenne primes [6].
In the calculation of numbers modulo a Mersenne number the following trick can be used: First divide the number by 2^n. We then get a quotient q and a remainder r. We had to divide by (2^n-1); so we divided by a number too large. As a result, the remainder becomes too small. We correct that by increasing the remainder r with quotient q. When the remainder r then is greater than the modulus, we have to subtract the modulus from the remainder r.
The division by 2^n is accomplished by means of a shift to the right by n bits. The bits that remain form the quotient q and the n bits that are shifted out (the n of the modulus) form the remainder r. We must now add the quotient q to the remainder r, and check whether the result is greater than the modulus. If the result is greater than the modulus we reduce the remainder r by the modulus.

Montgomery multiplication

The principle of the Montgomery multiplication is to carry out simultaneously the modulus operation with the multiplication. So the multiplication of number *a* by number *b,* followed by the mod operation *mod N* occurs in one operation: *a* x *b (mod N)*. The main difference with a "normal" mod operation is that with the Montgomery operation no division is involved; the only calculations are multiplications, additions and shifts. Divison operations are very expensive in terms of compute-power[7]. By avoiding division-operations and replacing them by multiplication-and shift-operations Montgomery multiplications can be much faster then multiplications followed by an ordinary mod operation.

With the Montgomery multiplication the factors *a* and *b* are first brought into a *"Montgomery form"*. This form consists of the number multiplied by a factor *"R",* and application of *(mod N)*. More later on this factor *"R"*.
So we get *a'* = *a* x *R (mod N)* and *b'* = *b* x *R (mod N)*. Multiplication of *a'* with *b'*, then produces the intermediate result: *T* = *a* x *R* x *b* x *R (mod N)* = *ab* x R^2 *(mod N)*. By dividing *T* by *R* we get *ab* x *R (mod N)*. The result is once again a number in the *Montgomery form*. In this way we can continue multiplication with other numbers that also are in the *Montgomery form.*
The division by *R* is made possible by adding the modulus to the intermediate result *T* that many times as is needed to make this intermediate result *T* divisible by *R* (adding the modulus does not change the modular result). The factor *R* is so determined that division by *R* may be carried out easily (in a decimal environment *R* can be set to a certain power of 10. In a binary environment, a power of 2 is taken). For *R* we take the smallest power of 2 (or the smallest power of 10) that is greater than the modulus.

Example:
Modulus *N = 843*
R = 1,000
T = intermediate result; *T* is limited to the modulus, but during calculations *T* will temporarily have a content that exceeds the modulus.
In this example we want to make the intermediate result *T* divisible by R

(1,000), so for readability, we leave out all the thousands of *T*.
We want to find a factor *m* such that *m* times *N,* added to *T,* produces an intermediate result *T* of ...000; so divisible by *R*.
We can calculate *m* by taking the last three digits of *T* and multiply this by a factor *x*. (In a formula: $m = T\ (mod\ R) * x$). The factor *x* in this case is 293.
For example:
If $T = ..001$ then $m = 001 * 293$ and $m*N = 246,999$; added to *T* gives *...000*.
If $T = ..123$ then $m = 123 * 293 = 36,039$ and $m*N = 30,380,877$ added to *T* gives *...000*. Etc.
For every value of *T(mod R)* there can be found a factor *m,* such that $m*N$ added to *T* produces a thousand-fold.
How do we get the factor *x*?
The factor *x* we can calculate with the *extended Euclidean algorithm* [8]. It is beyond the scope of this document to go deeper into this algorithm.
As input for this algorithm we use *R* and *N*. In our example, $R = 1000$ and $N = 843$. The result is $x = 293$. There always can be found a factor *x* provided that *R* and *N* has no common factors; so $GCD\ (R, N) = 1$.
It is easy to meet this requirement if *R* is a power of two and *N* is an odd number. If *R* is a power of 10 (as in this example) the requirement is met if *N* is odd and does not end on a five.

After the Montgomery multiplication the result is still in the Montgomery-form: $T \times R\ (mod\ N)$. This must be brought back into normal form by dividing *T* by *R (mod N)*. This is best done by multiplying the last intermediate result via a montgomery multiplication with a normal *1*.

That division by *R* can also be done in stages; when $R=10,000$ you can also divide four times by 10. We'll see later on how that goes.

Example
We want to calculate: *a* x *b (mod N)* ≡ *c* with the the Montgomery method.
Number *a = 1,234*; number *b = 5,678*; the modulus *N = 6,543.*
First we bring both numbers in the Montgomery form. Because the numbers exist of four decimal figures, we use an "R" of 10^4; So we multiply both numbers by 10^4 and divide them by the modulus with the remainder being the result as follows:
a' = 1,234 x 10,000 (mod 6,543) ≡ 6,445
b' = 5,678 x 10,000 (mod 6,543) ≡ 6,389
In the example we multiply *6,445 x 6,389* in the traditional way but insert after each line a "Montgomery" line to bring the result in a 10-fold. We could have chosen to add once a "Montgomery" line; thus bring the result directly to a 10,000-fold. We do not do that because it is more complex and does not lead to higher efficiency, as we will see later.
First we have to calculate factor *x* via the extended euclidean algorithm. As input for this calculation we take *R = 10* and for *N* we take the last digit of the modulus; so *N = 3*. The factor *x* is then *3*. The last decimal digit of each intermediate result T is multiplied by *x (mod 10)* to get the factor *m*. Then *m* is multiplied by the modulus and added to the intermediate result.

```
          multiplicand  a':      6389    factor x = 3
          multiplier    b':      6445 x           ↓
          modulus:             ------
 5 x 6389                       31945    m =(5 x 3 mod 10) = 5
          6543 x 5              32715
                                6466
 4 x 6389                      25556
                               32022     m =(2 x 3 mod 10) = 6
          6543 x 6             39258
                               7128
 4 x 6389                     25556
                              32684      m =(4 x 3 mod 10) = 2
          6543 x 2            13086
                              4577
 6 x 6389                    38334
                             42911       m =(1 x 3 mod 10) = 3
          6543 x 3           19629
                        c':  6254
```

The result c' is still in the Montgomery form: c' = 6,254. We can continue

Montgomery multiplications with this number if necessary. (Not now). To bring the number back into the "normal" form we have to multiply, still in the Montgomery way, the number 6,254 with a normal 1:

```
          multiplicand                1    factor x  = 3
          multiplier      c':      6254 x          ↓
          modulus:               ------
 4 x    1                             4    m =(4 x 3 mod 10) = 2
          6543 x 2                13086
                                  1309
 5 x    1                            5
                                  1314     m =(4 x 3 mod 10) = 2
          6543 x 2               13086
                                 1440
 2 x    1                           2
                                 1442      m =(2 x 3 mod 10) = 6
          6543 x 6              39258
                                4070
 6 x    1                          6
                                4076       m =(6 x 3 mod 10) = 8
          6543 x 8             52344
                       c:      5642
```

The result c = 5,642 and is exactly equal to the result of *c = a* x *b (mod N) = 1,234* x *5,678 (mod 6,543) ≡ 5,642.*

We see that we do really have to carry out a lot of extra work for a relatively simple modular multiplication. Later we will see that there are situations where this method just saves a lot of work. In the example above we have the division by 10,000 executed in four steps of 10. We could also do this in one step of 10,000. In that case, the number of "core multiplications" [9] remains the same, but the number of additions and shifts becomes less. However, the calculation of the factor *m* with the extended Euclidean algorithm is for numbers *R = 10,000* and *N = 6,543* much more complex than for numbers *R = 10* and *N = 3*.

All in all, it is easier to keep the factors *N* and *R* small in the Montgomery calculation; thus in a decimal environment (such as in our example) set *R* on *10* and set *N* to the last digit of the modulus. In a binary environment (a computer program) *R* becomes the value of 2^{32} or 2^{64}. The *N* factor then gets the value of the last 32 or 64 bits of the modulus.

Modular exponentiation

In number theory, formulas in the form a^p *(mod N)* occur frequently. The base number *(a)* is usually a small number, but the exponent *p* can be very large (hundreds of decimal digits, or in binary form thousands of bits). It is clear that *a* x *a* x *a* x ... and that 10^p times (where *p* is expressed in decimal digits) or 2^p (where p is expressed in bits) multiplication by *a* is an impossible task. That would take too much time and would create too large a number, even if we have the smallest possible value > 1 for *a* ; *a = 2.*
This happens when we wait with the (*mod N)* operation until the exponentiation is completely finished.
We may also perform the *(mod N)* operation after each multiplication with *a*. The result remains the same as the result when we do the *(mod N)* operation at the very end.
We get ((*a* x *a (mod N)* x *a (mod N))*... etc. The size of the intermediate number remains limited to a maximum of *N*.
The impossibly large number *(2^p)* of multiplications and *(mod N)* operations remains a substantial limitation unless we can reduce this number. This can be done as follows: We can divide the exponent *p* in pieces that each are easy to calculate. On those pieces we apply the *(mod N)*. Thereafter we can multiply pieces together to obtain the final result; condition is that the number of pieces is much smaller than p.
We can write the exponent *p* as a binary number; each bit then represents a number which is a power of two. The bits set to 1 represents the values of 2^i, for i = 0, 1, 2 .. for resp. bit 0, 1, 2 ...

Binary ladders [10]

The algorithm that describes exponentiation with an exponent expressed as a binary number has got the name "binary ladder exponentiation".
Take the example of the formula a^p, where we give the exponent p the value of 37, and p expressed as a binary number. We do this by dividing p in parts of powers of two, as follows: $37 = 32 + 4 + 1 = 2^5 + 2^2 + 2^0$. So p, expressed as a binary number: $= 32 + 4 + 1 = 37 = 1\ 0\ 0\ 1\ 0\ 1$

```
Bitnumber:                    5    4    3    2    1    0
                              2^5  2^4  2^3  2^2  2^1  2^0
Value per bit:               32   16    8    4    2    1
Binary number p               1    0    0    1    0    1
Value of p                   32 + 0 + 0 + 4 + 0 + 1   = 37
As exponent of a:             a^32          a^4      a^1  = a^37
```

Where in the binary number there is a '0' bit, this bit has the value $a^0 = 1$. The total value of a^p can now be obtained by multiplying all the terms of the binary field with each other. By continue squaring a and the subsequent results we can find the values of a^1, a^2, a^4, a^8, a^{16}, a^{32}. By multiplying the values of the '1' bits with each other in a separate resultfield, we can find the value of a^{37}.

Two methods of binary ladder exponentiation are known: the first method is known as *RightToLeftBinaryExponentiation*. The second method offers some advantages over the first-mentioned method and has the name of *LeftToRightBinaryExponentiation* [11].

Left To Right Binary exponentiation

This method uses the exponent expressed as a binary number and starts at the left most bit that is set to 1 (i.e., the bit having the greatest value) and goes to the right up to bit number 0.
In the example with exponent $p = 37$, we begin with the leftmost bit (bitnumber 5), and start the intermediate result T with the value a^1. We continue to the right with a bit that is set to 0 and execute the squaring of the value T; which now becomes a^2.

At the next bit again a squaring is performed; $T = a^{4}$. The next bit to the right has a value of 1. We now perform not only a squaring $(T = a^{8})$, but also a multiplication with a; so $T = a^{9}$. Then again a zero-bit: $T = a^{18}$. The last (right most) bit is a 1 so we perform a squaring $(T = a^{36})$, and multiply T again with a $(T = A^{37})$.
When the exponent has a large value, there arises an impossibly large intermediate number T. But often the exponentiation is combined with modular arithmetic. By applying the modulus to the intermediate result T every time the value of T changes, this value can be kept limited to the value of the modulus.

Now we can combine the method of the binary ladder with the Montgomery multiplication, and so we get the “Montgomery exponentiation”. Each squaring of the intermediate result T now happens with the "square Montgomery multiplication". A nice application of the Montgomery exponentiation is in the calculation of Fermat's little theorem.

Fermat's little theorem

Pierre de Fermat lived in early 17th century in France. He has done much research in the field of mathematics and number theory. He is best known for "Fermat's last theorem" but his "Fermat's little theorem" is much more important in practice[12]. This theorem is the basis of many cryptographic applications. This theorem reads:

If p is a prime number, and a is a positive integer > 1, and further have a and p no common divisors, then:

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

In this document we call this theorem further the Fermat test or the little Fermat test. If the result is different from $1 \pmod p$, then p is composite. With this test we can distinguish numbers that are composite and numbers that may be prime. In cryptographic applications large prime numbers are required. They can be obtained by generating random numbers, and test these numbers for primality. For this test we can use the Fermat test. If a number succeeds the little Fermat test, chances are that it is a prime number and additional testing may be executed to obtain a near certainty that the number is prime.

The little Fermat test can be performed just fine with the Montgomery exponentiation. This could be done as follows:

(In the following description, we assume a value of 2 for a. If a different value for a is chosen, then some of the improvements described herein are not applicable).

We want to test a number p with the Fermat test. We choose $a = 2$. For R we choose a power of two, namely a multiple of 32 (we assume a 32-bit processor). First we determine the size of p in number of significant bits. This number is rounded up to a 32-fold and is called q; which then forms the exponent of 2 for R. Thus, $R = 2^q$

For the Montgomery multiplication we will bring a into the Montgomery form: therefore $a' = R$ x a *(mod p)* $= 2^q$ x 2 *(mod p)*. We take the number of significant bits of p (that we have previously determined) but **without** the rounding up to a 32-fold. If that number is not a multiple of 32, we need to calculate the remainder by dividing the number (2^q x 2) by p. This is a computer-intensive calculation.
If the number of significant bits of p indeed is a multiple of 32, the calculation can be much simpler, and goes as follows:
$a' = 2$ x $(R - p)$. If a' then is bigger than p, subtract p from a'.

(For the calculation of prime numbers for cryptographic applications, it often occurs that the number of bits of the prime numbers is a multiple of 32).

Then we express the exponent p in binary form (first reduce p by one) and perform the Montgomery exponentiation: for each bit position of $(p - 1)$, we execute a Montgomery square multiplication, beginning with intermediate result $T = a'$, and starting from the second bit from the left (the bits having the highest values). For each "0" bit we execute the square Montgomery multiplication, and for every "1" bit we do the same and additionally multiply T with a; so with two.
Multiplication by two is accomplished by shifting the intermediate result T one bit to the left. Then we calculate the modular result; we achieve this by subtracting p from T, as far as the intermediate result $T > p$.
We handle all the bits of $(p - 1)$ down to bit 0. We finally have an intermediate result T that we only have to bring back from the Montgomery form to the normal form.
We do this by dividing T by R *(mod p)* = dividing T by 2^q *(mod p)*.

But wait a moment; we have first a (two), $(p-1)$ times multiplied by itself *(mod p)*, and now we have to divide the endresult by 2^q *(mod p)*.
We can do better!

We can reduce the exponent with q, prior to the begin of the exponentiation. The exponent will then be $(p - 1 - q)$. Doiing so, the last step (bringing back the result from the Montgomery form to the normal form) is eliminated.

If the end result not is equal 1, then p is certainly **not** a prime number but p is composite. If the final result is now equal to 1, then p is probably prime. With further testing, we can increase the probability that p is a primenumber. We can, for example, again perform the Fermat test for p, but with a different base a; for example, $a = 3$. The same calculation using the Montgomery squaring can then be carried out, but without the improvements, that belong to $a = 2$, such as the elimination of the final step to bring back the result to the normal form.

If the final result of a Fermat test for a composite number nevertheless ≡ 1, then we call such a number a pseudo-prime number for base a.
If a composite number with the Fermat test yields modular value ≡ 1, for any value of base a ,such a number is referred to as a Carmichael number.

In practice, the Fermat test (if successful) is followed by the Miller-Rabin test[13]. This test looks a bit like the Fermat test; it also made use of the Montgomery exponentiation and the test also uses a base *a.* If this test is successfully carried out ten times with different values for *a,* then the probability that the tested number is a prime number is greater than 99.9999% .

References

1. *Montgomery* (http://en.wikipedia.org/wiki/Peter_Montgomery_(mathematician))
2. Theo Kortekaas. *Multiplying large numbers and the Schönhage-Strassen Algorithm* (http://tonjanee.home.xs4all.nl/SSAdescription.pdf) page 10-12.
3. Richard Crandall en Carl Pomerance. *Prime Numbers – A Computational Perspective,* Second Edition, Springer, 2005. Section 9.2.3: Moduli of special form, Pages 454-457.
4. Pierrick Gaudry, Alexander Kruppa, Paul Zimmermann. *A GMP-based Implementation of Schönhage-Strassen's Large Integer Multiplication Algorithm* (http://www.loria.fr/~gaudry/publis/issac07.pdf)
5. *Shift optimization* (https://en.wikipedia.org/wiki/Schönhage-Strassen_algorithm) Section 1.4
6. *Lucas-Lehmer primality test* (https://en.wikipedia.org/wiki/Lucas-Lehmer_primality_test)
7. Richard Brent, Paul Zimmermann. *Modern Computer Arithmetic* Cambridge University Press, Section 2.1.2: Montgomery's form, page 48.
8. *Extended Euclidean Algorithm* (https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm)
9. Theo Kortekaas. *Multiplying large numbers and the Schönhage-Strassen Algorithm* (http://tonjanee.home.xs4all.nl/SSAdescription.pdf) Core multiplication, page 3.
10. Richard Crandall en Carl Pomerance. *Prime Numbers – A Computational Perspective,* Second Edition, Springer, 2005. Section 9.3.1: Basic binary ladders pages 458-460.
11. Richard Brent, Paul Zimmermann. *Modern Computer Arithmetic* Cambridge University Press, Section 2.6.1: Binary exponentiation, LeftToRightBinaryExponentiation, page 70.
12. Fermats little theorem (https://en.wikipedia.org/wiki/Fermat's_little_theorem)
13. Miller-Rabin test (https://en.wikipedia.org/wiki/Miller-Rabin_primality_test)

Author: T.P.J. Kortekaas
Email: t.kortekaas@xs4all.nl
Web-pages: www.tonjanee.home.xs4all.nl