



WIKIPEDIA  
The Free Encyclopedia

Main page  
Contents  
Featured content  
Current events  
Random article  
Donate to Wikipedia  
Wikipedia store

Interaction  
Help  
About Wikipedia  
Community portal  
Recent changes  
Contact page

Tools  
What links here  
Related changes  
Upload file  
Special pages  
Permanent link  
Page information  
Wikidata item  
Cite this page

Print/export  
Create a book  
Download as PDF  
Printable version

Languages  
Français  
日本語  
Русский  
中文

Edit links

Create account Log in

Article **Talk**

Read **Edit** View history

# Montgomery modular multiplication

From Wikipedia, the free encyclopedia  
(Redirected from [Montgomery reduction](#))

In modular arithmetic computation, **Montgomery modular multiplication**, more commonly referred to as **Montgomery multiplication**, is a method for performing fast modular multiplication, introduced in 1985 by the American mathematician [Peter L. Montgomery](#).<sup>[1]</sup><sup>[2]</sup> Given two numbers *a* and *b* modulo a positive integer *N*, Montgomery multiplication computes

$$ab \bmod N.$$

Montgomery multiplication requires converting *a* and *b* into a special representation called Montgomery form. Because of the overhead involved in the conversion, computing a single product by Montgomery multiplication is slower than computing the product in the integers and performing a modular reduction by division or [Barrett reduction](#). However, when many products are required, as in [modular exponentiation](#), the conversion to Montgomery form becomes a negligible fraction of the time of the computation, and performing the computation by Montgomery multiplication is faster than the available alternatives. Many important cryptosystems such as [RSA](#) and [Diffie–Hellman key exchange](#) are based on arithmetic operations modulo a large number, and for these cryptosystems, the increased speed afforded by Montgomery multiplication can be important in practice.<sup>[3]</sup>

## Contents [\[hide\]](#)

- 1 Modular arithmetic and Montgomery form
- 2 The REDC algorithm
- 3 Arithmetic in Montgomery form
- 4 Montgomery arithmetic on multiprecision integers
- 5 Side channel attacks
- 6 References

## Modular arithmetic and Montgomery form [\[edit\]](#)

Let *N* denote a positive integer modulus. The [ring](#) **Z**/*N***Z** consists of residue classes modulo *N*, that is, sets of the form:

$$\{a + kN : k \in \mathbf{Z}\},$$

where *a* is some fixed integer. Each residue class is a set of integers such that the difference of any two integers in the set is divisible by *N* (and the residue class is maximal with respect to that property; integers aren't left out of the residue class unless they would violate the divisibility condition). The residue class corresponding to *a* is denoted  $\bar{a}$ . Equality of residue classes is called congruence and is denoted:

$$\bar{a} \equiv \bar{b} \pmod{N}.$$

Storing an entire residue class on a computer is impossible because the residue class has infinitely many elements. Instead, residue classes are stored as representatives. Conventionally, these representatives are the integers *a* for which  $0 \leq a \leq N - 1$ . If *a* is an integer, then the representative of  $\bar{a}$  is written *a* mod *N*. When writing congruences, it's common to identify an integer with the residue class it represents. With this convention, the above equality is written  $a \equiv b \pmod{N}$ .

Arithmetic on residue classes is done by first performing integer arithmetic on their representatives. The output of the integer operation determines a residue class, and the output of the modular operation is determined by computing the residue class's representative. For example, if *N* = 17, then the sum of the residue classes  $\bar{7}$  and  $\bar{15}$  is computed by finding the integer sum  $7 + 15 = 22$ , then determining  $22 \bmod 17$ , the integer between 0 and 16 whose difference with 22 is a multiple of 17. In this case, that integer is 5, so  $\bar{7} + \bar{15} \equiv \bar{5} \pmod{17}$ .

If *a* and *b* are integers in the range  $[0, N - 1]$ , then their sum is in the range  $[0, 2N - 2]$  and their difference is in the range  $[-N + 1, N - 1]$ , so determining the representative in  $[0, N - 1]$  requires at most one subtraction or addition (respectively) of *N*. However, the product *ab* is in the range  $[0, N^2 - 2N + 1]$ . Storing the intermediate integer product *ab* requires twice as many bits as either *a* or *b*, and efficiently determining the representative in  $[0, N - 1]$  requires division. Mathematically, the integer between 0 and *N* − 1 that is congruent to *ab* can be

expressed by applying the [division algorithm](#):

$$ab = qN + r,$$

where  $q$  is the quotient  $\lfloor ab/N \rfloor$  and  $r$ , the remainder, is in the interval  $[0, N - 1]$ . The remainder  $r$  is  $ab \bmod N$ . Determining  $r$  can be done by computing  $q$ , then subtracting  $qN$  from  $ab$ . For example, the product  $7 \cdot 15$  is determined by computing  $7 \cdot 15 = 105$ , dividing  $\lfloor 105/17 \rfloor = 6$ , and subtracting  $105 - 6 \cdot 17 = 105 - 102 = 3$ .

Because the computation of  $q$  requires division, it is undesirably expensive on most computer hardware. Montgomery form is a different way of expressing the elements of the ring in which modular products can be computed without expensive divisions. While divisions are still necessary, they can be done with respect to a different divisor  $R$ . This divisor can be chosen to be a whole number of machine words, making division and reduction much cheaper.

The only mathematical requirement on the auxiliary modulus  $R$  is that it be a positive integer such that  $\gcd(N, R) = 1$ . For computational purposes it is also necessary that division and reduction modulo  $R$  be inexpensive, and the modulus is not useful for modular multiplication unless  $R > N$ . The **Montgomery form** or **Montgomery representation** of the residue class  $\bar{a}$  with respect to  $R$  is  $aR \bmod N$ , that is, it is the representative of the residue class  $\overline{aR}$ . For example, suppose that  $N = 17$  and that  $R = 100$ . The Montgomery forms of 3, 5, 7, and 15 are  $300 \bmod 17 = 11$ ,  $500 \bmod 17 = 7$ ,  $700 \bmod 17 = 3$ , and  $1500 \bmod 17 = 4$ .

Addition and subtraction in Montgomery form are the same as ordinary modular addition and subtraction because of the distributive law:

$$\begin{aligned} aR + bR &= (a + b)R, \\ aR - bR &= (a - b)R. \end{aligned}$$

This is a consequence of the fact that, because  $\gcd(R, N) = 1$ , multiplication by  $R$  is an [isomorphism](#) on the additive group  $\mathbb{Z}/N\mathbb{Z}$ . For example,  $(7 + 15) \bmod 17 = 5$ , which in Montgomery form becomes  $(3 + 4) \bmod 17 = 7$ .

Multiplication in Montgomery form, however, is seemingly more complicated. The usual product of  $aR$  and  $bR$  does not represent the product of  $a$  and  $b$  because it has an extra factor of  $R$ :

$$(aR \bmod N)(bR \bmod N) \bmod N = (abR) \bmod N.$$

Computing products in Montgomery form requires removing the extra factor of  $R$ . While division by  $R$  is cheap, the intermediate product  $(aR \bmod N)(bR \bmod N)$  is not divisible by  $R$  because the modulo operation has destroyed that property. So for instance, the product of the Montgomery forms of 7 and 15 modulo 17 is the product of 3 and 4, which is 12. Since 12 is not divisible by 100, additional effort is required to remove the extra factor of  $R$ .

Removing the extra factor of  $R$  can be done by multiplying by an integer  $R'$  such that  $RR' \equiv 1 \pmod{N}$ , that is, by an  $R'$  whose residue class is the [modular inverse](#) of  $R \bmod N$ . Then, working modulo  $N$ ,

$$(aR \bmod N)(bR \bmod N)R' \equiv (aR)(bR)R^{-1} \equiv (ab)R \pmod{N}.$$

The integer  $R'$  exists because of the assumption that  $R$  and  $N$  are coprime. It can be constructed using the [extended Euclidean algorithm](#). The extended Euclidean algorithm efficiently determines integers  $R'$  and  $N'$  that satisfy [Bézout's identity](#):  $0 < R' < N$ ,  $0 < N' < R$ , and:

$$RR' - NN' = 1.$$

This shows that it is possible to do multiplication in Montgomery form. A straightforward algorithm to multiply numbers in Montgomery form is therefore to multiply  $aR \bmod N$ ,  $bR \bmod N$ , and  $R'$  as integers and reduce modulo  $N$ .

For example, to multiply 7 and 15 modulo 17 in Montgomery form, compute the product of 3 and 4 to get 12 as above. The extended Euclidean algorithm implies that  $8 \cdot 100 - 47 \cdot 17 = 1$ , so  $R' = 8$ . Multiply 12 by 8 to get 96 and reduce modulo 17 to get 11. This is the Montgomery form of 3, as expected.

## The REDC algorithm [\[edit\]](#)

While the above algorithm is correct, it is slower than multiplication in the standard representation because of the need to multiply by  $R'$  and divide by  $N$ . **Montgomery reduction**, also known as REDC, is an algorithm that simultaneously computes the product by  $R'$  and reduces modulo  $N$  more quickly than the naive method. The speed is because all computations are done using only reduction and divisions with respect to  $R$ , not  $N$ :

**function** REDC is

```

input: Integers  $R$  and  $N$  with  $\gcd(R, N) = 1$ ,
        Integer  $N'$  in  $[0, R - 1]$  such that  $NN' \equiv -1 \pmod{R}$ ,
        Integer  $T$  in the range  $[0, RN - 1]$ 
output: Integer  $S$  in the range  $[0, N - 1]$  such that  $S \equiv TR^{-1} \pmod{N}$ 

 $m \leftarrow ((T \bmod R)N') \bmod R$ 
 $t \leftarrow (T + mN) / R$ 
if  $t \geq N$  then
    return  $t - N$ 
else
    return  $t$ 
end if
end function

```

To see that this algorithm is correct, first observe that  $m$  is chosen precisely so that  $T + mN$  is divisible by  $R$ . A number is divisible by  $R$  if and only if it is congruent to zero mod  $R$ , and we have:

$$T + mN \equiv T + (((T \bmod R)N') \bmod R)N \equiv T + TN'N \equiv T - T \equiv 0 \pmod{R}.$$

Therefore  $t$  is an integer. Second, the output is either  $t$  or  $t - N$ , both of which are congruent to  $t \bmod N$ , so prove that the output is congruent to  $TR^{-1} \bmod N$ , it suffices to prove that  $t$  is. Modulo  $N$ ,  $t$  satisfies:

$$t \equiv (T + mN)R^{-1} \equiv TR^{-1} + (mR^{-1})N \equiv TR^{-1} \pmod{N}.$$

Therefore the output has the correct residue class. Third,  $m$  is in  $[0, R - 1]$ , and therefore  $T + mN$  is between 0 and  $(RN - 1) + (R - 1)N < 2RN$ . Hence  $t$  is less than  $2N$ , and because it's an integer, this puts  $t$  in the range  $[0, 2N - 1]$ . Therefore reducing  $t$  into the desired range requires at most a single subtraction, so the algorithm's output lies in the correct range.

To use REDC to compute the product of 7 and 15 modulo 17, first convert to Montgomery form and multiply as integers to get 12 as above. Then apply REDC with  $R = 100$ ,  $N = 17$ ,  $N' = 47$ , and  $T = 12$ . The first step sets  $m$  to  $12 \cdot 47 \bmod 100 = 64$ . The second step sets  $t$  to  $(12 + 64 \cdot 17) / 100$ . Notice that  $12 + 64 \cdot 17$  is 1100, a multiple of 100 as expected.  $t$  is set to 11, which is less than 17, so the final result is 11, which agrees with the computation of the previous section.

As another example, consider the product  $7 \cdot 15 \bmod 17$  but with  $R = 10$ . Using the extended Euclidean algorithm, compute  $-5 \cdot 10 + 3 \cdot 17 = 1$ , so  $N'$  will be  $-3 \bmod 10 = 7$ . The Montgomery forms of 7 and 15 are  $70 \bmod 17 = 2$  and  $150 \bmod 17 = 14$ , respectively. Their product 28 is the input  $T$  to REDC, and since  $28 < RN = 170$ , the assumptions of REDC are satisfied. To run REDC, set  $m$  to  $(28 \bmod 10) \cdot 32 \bmod 10 = 256 \bmod 10 = 6$ . Then  $28 + 6 \cdot 17 = 130$ , so  $t = 13$ . Because  $30 \bmod 17 = 13$ , this is the Montgomery form of  $3 = 7 \cdot 15 \bmod 17$ .

## Arithmetic in Montgomery form [\[edit\]](#)

Many operations of interest modulo  $N$  can be expressed equally well in Montgomery form. Addition, subtraction, negation, comparison for equality, multiplication by an integer not in Montgomery form, and greatest common divisors with  $N$  may all be done with the standard algorithms. The [Jacobi symbol](#) can be calculated as  $\left(\frac{a}{N}\right) = \left(\frac{aR}{N}\right) / \left(\frac{R}{N}\right)$  as long as  $(R/N)$  is stored.

When  $R > N$ , most other arithmetic operations can be expressed in terms of REDC. This assumption implies that the product of two representatives mod  $N$  is less than  $RN$ , the exact hypothesis necessary for REDC to generate correct output. In particular, the product of  $aR \bmod N$  and  $bR \bmod N$  is  $\text{REDC}((aR \bmod N)(bR \bmod N))$ . The combined operation of multiplication and REDC is often called **Montgomery multiplication**.

Conversion into Montgomery form is done by computing  $\text{REDC}((a \bmod N)(R^2 \bmod N))$ . Conversion out of Montgomery form is done by computing  $\text{REDC}(aR \bmod N)$ . The modular inverse of  $aR \bmod N$  is  $\text{REDC}((aR \bmod N)^{-1}(R^3 \bmod N))$ . Modular exponentiation can be done using [exponentiation by squaring](#) by initializing the initial product to the Montgomery representation of 1, that is, to  $R \bmod N$ , and by replacing the multiply and square steps by Montgomery multiplies.

Performing these operations requires knowing at least  $R^2 \bmod N$ . The constants  $R \bmod N$  and  $R^3 \bmod N$  can be generated as  $\text{REDC}(R^2 \bmod N)$  and as  $\text{REDC}((R^2 \bmod N)(R^2 \bmod N))$ . The fundamental operation is to compute REDC of a product. When standalone REDC is needed, it can be computed as REDC of a product with  $1 \bmod N$ . The only place where a direct reduction modulo  $N$  is necessary is in the precomputation of  $R^2 \bmod N$ .

## Montgomery arithmetic on multiprecision integers [\[edit\]](#)

Most cryptographic applications require numbers that are hundreds or even thousands of bits long. Such numbers are too large to be stored in a single machine word. Typically, the hardware performs multiplication mod some base  $B$ , so performing larger multiplications requires combining several small multiplications. The base  $B$  is typically 2 for microelectronic applications,  $2^8$  for 8-bit firmware,<sup>[4]</sup> or  $2^{32}$  or  $2^{64}$  for software applications.

The REDC algorithm requires products modulo  $R$ , and typically  $R > N$  so that REDC can be used to compute products. However, when  $R$  is a power of  $B$ , there is a variant of REDC which requires products only of machine word sized integers. Suppose that positive multi-precision integers are stored **little endian**, that is,  $x$  is stored as an array  $x[0], \dots, x[\ell - 1]$  such that  $0 \leq x[i] < B$  for all  $i$  and  $x = \sum x[i] B^i$ . The algorithm begins with a multiprecision integer  $T$  and reduces it one word at a time. First an appropriate multiple of  $N$  is added to make  $T$  divisible by  $B$ . Then a multiple of  $N$  is added to make  $T$  divisible by  $B^2$ , and so on. Eventually  $T$  is divisible by  $R$ , and after division by  $R$  the algorithm is in the same place as REDC was after the computation of  $t$ .

```

function MultiPrecisionREDC is
    Input: Integer  $N$  with  $\gcd(B, N) = 1$ , stored as an array of  $p$  integers,
            Integer  $R = B^F$ ,
            Integer  $N'$  in  $[0, B - 1]$  such that  $NN' \equiv -1 \pmod{B}$ ,
            Integer  $T$  in the range  $0 \leq T < RN$ , stored as an array of  $r + p$  integers.

    Output: Integer  $S$  in  $[0, N - 1]$  such that  $TR^{-1} \equiv S \pmod{N}$ , stored as an array
    of  $p$  integers.

    Set  $T[r + p + 1] = 0$ 
    for  $0 \leq i < r$  do
        (Make T divisible by  $B^{i+1}$ )

         $c \leftarrow 0$ 
         $m \leftarrow T[i] \cdot N' \bmod B$ 
        for  $0 \leq j < p$  do
            (Add the low word of  $m \cdot N[j]$  and the carry from earlier, and find the
            new carry)

             $x \leftarrow T[i + j] + m \cdot N[j] + c$ 
             $T[i + j] \leftarrow x \bmod B$ 
             $c \leftarrow \lfloor x / B \rfloor$ 
        end for
        for  $p \leq j \leq r + p - i + 1$  do
            (Continue carrying)

             $x \leftarrow T[i + j] + c$ 
             $T[i + j] \leftarrow x \bmod B$ 
             $c \leftarrow \lfloor x / B \rfloor$ 
        end for
    end for

    for  $0 \leq i < p + 1$  do
         $S[i] \leftarrow T[i + r]$ 
    end for

    if  $S > N$  then
        return  $S - N$ 
    else
        return  $S$ 
    end if
end function

```

The final comparison and subtraction is done by the standard algorithms.

The above algorithm is correct for essentially the same reasons that REDC is correct. Each time through the  $i$  loop,  $m$  is chosen so that  $T[i] + mN[0]$  is divisible by  $B$ . Then  $mNB^i$  is added to  $T$ . Because this quantity is zero mod  $N$ , adding it does not affect the value of  $T \bmod N$ . If  $m_i$  denotes the value of  $m$  computed in the  $i$ th iteration of the loop, then the algorithm sets  $S$  to  $T + (\sum m_i B^i)N$ . Because MultiPrecisionREDC and REDC produce the same output, this sum is the same as the choice of  $m$  that the REDC algorithm would make.

The last word of  $T$ ,  $T[r + p + 1]$ , is used only to hold a carry, and so consequently it is either zero or one. Depending upon the processor, it may be possible to store this word as a carry flag instead of a full-sized word.

It is possible to combine multiprecision multiplication and REDC into a single algorithm. This combined algorithm is usually called **Montgomery multiplication**. Several different implementations are described by Koç, Acar, and Kalinski.<sup>[5]</sup> The algorithm may use as little as  $p + 2$  words of storage (plus a carry bit).

As an example, let  $B = 10$ ,  $N = 997$ , and  $R = 1000$ . Suppose that  $a = 314$  and  $b = 271$ . The Montgomery representations of  $a$  and  $b$  are  $314000 \bmod 997 = 942$  and  $271000 \bmod 997 = 813$ . Compute  $942 \cdot 813 = 765846$ . The initial input  $T$  to MultiPrecisionREDC will be  $[6, 4, 8, 5, 6, 7]$ . The number  $N$  will be represented as  $[7, 9, 9]$ . The extended Euclidean algorithm says that  $-299 \cdot 10 + 3 \cdot 997 = 1$ , so  $N'$  will be 7.

```

i ← 0
m ← 6 · 7 mod 10 = 2

j T      c
- - - - -
0 0485670 1      (After first iteration of first loop)
1 0485670 1
2 0485670 1
3 0487670 0      (After first iteration of second loop)
4 0487670 0
5 0487670 0
6 0487670 0

i ← 1
m ← 4 · 7 mod 10 = 8

j T      c
- - - - -
0 0087670 1      (After first iteration of first loop)
1 0067670 1
2 0067670 1
3 0067470 1      (After first iteration of second loop)
4 0067480 0
5 0067480 0

i ← 2
m ← 6 · 7 mod 10 = 2

j T      c
- - - - -
0 0007480 1      (After first iteration of first loop)
1 0007480 1
2 0007480 1
3 0007400 1      (After first iteration of second loop)
4 0007401 0

```

Therefore, before the final comparison and subtraction,  $S = 1047$ . The final subtraction yields the number 50. Since the Montgomery representation of  $314 \cdot 271 \bmod 997 = 349$  is  $349000 \bmod 997 = 50$ , this is the expected result.




When working in base 2, determining the correct  $m$  at each stage is particularly easy: If the current working bit is even, then  $m$  is zero and if it's odd, then  $m$  is one. Furthermore, because each step of MultiPrecisionREDC requires knowing only the lowest bit, Montgomery multiplication can be easily combined with a [carry-save adder](#).

## Side channel attacks [\[edit\]](#)

When using it as a part of a cryptographically secure algorithm, unmodified Montgomery reduction is vulnerable to [side channel attacks](#), where the attacker can learn about the inner workings of the algorithm by studying the differences in time, power-consumption or any other parameter affected by the fact that the algorithm performs very different actions depending on the input. However it is simple to modify the algorithm or the hardware to make it resistant to such attacks.<sup>[4][6]</sup>

## References [\[edit\]](#)

- ↑ Peter Montgomery. *Modular Multiplication Without Trial Division*, *Math. Computation*, vol. 44, pp. 519–521, 1985.
- ↑ Martin Kochanski, [Montgomery Multiplication](#) <sup>[a]</sup> A colloquial explanation.
- ↑ Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography* <sup>[a]</sup>. CRC Press, 1996. ISBN 0-8493-8523-7, chapter 14.

4. <sup>a</sup> <sup>b</sup> Zhe Liu, Johann Großschädl, and Ilya Kizhvatov. "Efficient and Side-Channel Resistant RSA Implementation for 8-bit AVR Microcontrollers" .
5. <sup>a</sup> Çetin K. Koç, Tolga Acar, Burton S. Kalinski, Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms" **16** (3). 1996. pp. 26–33. [CiteSeerX 10.1.1.26.3120](#) .
6. <sup>a</sup> Marc Joye and Sung-Ming Yen. "The Montgomery Powering Ladder" . 2002.

Categories: [Computer arithmetic](#) | [Cryptographic algorithms](#) | [Modular arithmetic](#)

This page was last modified on 29 July 2015, at 10:23.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

