Understanding the Montgomery reduction algorithm

The Montgomery reduction algorithm finds the remainder of a division. Many cryptographic schemes work with numbers modulo a prime. When you have to multiply two numbers with e.g. 128 bits each, first you multiply them the usual way (there are many techniques for this) to obtain a 256-bit ("double precision") number. Then you need to reduce this result modulo the prime you're working with, that is, compute the remainder of the division of this number over the prime.

You can compute the remainder of a division with the "schoolbook" technique everyone learns in school, but that is expensive and requires divisions, with are costly in many platforms (some microcontrollers don't even have a division instruction). Montgomery reduction only needs a division by a powers of the integer size, which are cheap for computers.

Here I'll try to explain how it works, in an informal approach. For detailed proofs of its correctness, check e.g. the chapter 14 of the Handbook of Applied Cryptography or the original paper.

First approach

When working modulo a prime (or modulo anything), you can add or subtract the prime (call it p) from the number you're working with (call it x) and you'll always get the same number. For example, 3 = 3 + 7 = 10 = 3 (mod 7), or 3 = 3 – 7 = -4 = 3 (mod 7). This gives the basic approach for finding remainders: just subtract the p from x until you reach a number smaller than p.

For example, you want to compute 6 * 5 modulo 7. You have 6 * 5 = 30, now you just need the remainder of the division of 30 over 7, that is, 30 modulo 7. Now it's just subtractions:

```
30 - 7 = 23 (mod 7)
23 - 7 = 16 (mod 7)
16 - 7 = 9 (mod 7)
9 - 7 = 2 (mod 7)
```

And you're done: 6 * 5 = 2 (mod 7). But if the number you're working with is big, the number of subtractions you'll need will be too large. You can speed up this process by dividing the number by the modulus and computing r = x – (x/p * p). But as I said, division is expensive.

Let's add, not subtract

The first insight of the Montgomery reduction is this: what if instead of subtracting, you add the prime modulus many times? And more, what if you add the modulus in a way that the number being reduced is filled with zeros to the right? For example, let's work modulo 97. You want to compute 43 * 56 (mod 97). You multiply the operands obtaining 2408, and you'll need to reduce it modulo 97. You can add the 97 to this number any times you want, that is, you can add any multiple of 97. Which multiple of 97 that, when added to 2408, leads to a number with last digit 0?

```
2408 + 6 * 97 =
2408 + 582 =
2990
```

It's 6 times 97. It's easy to find this "magic number", but I won't go into this right now. OK, you have a rightmost zero. Let's try to add again in order to have two zeros:

```
2990 + 30 * 97 =
2990 + 2910 =
5900
```

Great! But… so what? Well, suppose these numbers live in a world where they must be divided by 100 after you multiply them and use the above procedure. Then you divide 5900 by 100, which is cheap, and you have 59.

Sadly, these numbers don't live in such a world, and 59 is not the right answer. If the division is exact, you can divide by any multiple of 97, but you can't

simply divide by 100.

The Montgomery World

The second insight of the Montgomery reduction is: if you need such a world were you must divide by 100, then just create it!

The Montgomery reduction requires transforming the numbers into the "Montgomery domain", which is exactly the world we need. First let's look at our modulus: 97. It has two digits in base 10. Then let R be smaller power of the base that is greater than the modulus. For 97, we have R = 10^2 = 100.

To transform a number x into the Montgomery domain, compute x * R (mod p). In our example, suppose x = 35. Then 35 * 100 (mod 97) = 8 (mod 97).

(You may notice that this transformation simply begs the question – you need to reduce modulo p in order to convert the numbers to a form where it's easier to reduce modulo p! I'll talk about this later on.)

Now, the beauty of the Montgomery domain: suppose that you want to compute x * y (mod p). Converting the operands, you'll actually compute (x * R) * (y * R) (mod p). But you want the result to still live in the Montgomery domain. That's because you may need to do many other multiplications. Since converting the number back to the real world is somewhat expensive, it's best to keep the computations in the Montgomery domain. Therefore, you want to compute x * y * R (mod p), i.e., you want x * y, but in the Montgomery domain.

Let's see… if you just multiply the operands the usual way, you get (x * R) * (y * R) (mod p) = x * y * R * R (mod p). But you want x * y * R (mod p). Therefore, every time you multiply two numbers, you'll need to reduce the result modulo p, but you'll also need to divide by R. Recall our example, where R = 100, and we wanted to be required to divide by 100. That's exactly what we got: the Montgomery domain requires the division by 100!

An example

Let x = 43, y = 56, p = 97, R = 100. You want to compute x * y (mod p). First you convert x and y to the Montgomery domain. For x, compute x' = x * R (mod p) =

43 * 100 (mod 97) = 32, and for y, compute y' = y * R (mod p) = 56 * 100 (mod 97) = 71.

Compute a := x' * y' = 32 * 71 = 2272.

In order to zero the first digit, compute a := a + (4p) = 2272 + 388 = 2660.

In order to zero the second digit, compute a := a + (20p) = 2660 + 1940 = 4600.

Compute a := a / R = 4600 / 100 = 46.

We have that 46 is the Montgomery representation of x * y (mod p), that is, x * y * R (mod p). In order to convert it back, compute a * (1/R) (mod p) = 46 * 65 (mod 97) = 80. You can check that 43 * 56 (mod 97) is indeed 80.

Converting the numbers

Let Montgomery(x',y') = (x'y')/R (mod p) be the Montgomery reduction of x' and y'.

Like I mentioned, in order to compute the Montgomery representation of a number or in order to convert it back, you need to multiply by R modulo p or divide by R modulo p. You can compute this remainders using a classic, expensive algorithm, since for practical applications you'll rarely need to convert the numbers (for example, in a cryptographic application: since no one will need to actually see the real form of the numbers, there is no need for conversions and it's possible to work entirely in the Montgomery domain). But there is also another approach. You can precompute R' = R^2 (mod p) and then convert a number x to xR (mod p) by simple computing the Montgomery reduction of x and R', since Montgomery(x, R') = Montgomery(x, R^2) = (xRR)/R (mod p) = xR (mod p). To convert back a number x' = xR (mod p), simply compute the Montgomery reduction of x' and 1, since Montgomery(x', 1) = Montgomery(xR, 1) = (xR)/R (mod p) = x (mod p).

The magic numbers

Let B be the base you're working with. The main step of the Montgomery reduction is "add a multiple k of p that, when added to a, makes its i-th digit

zero". It's not hard to find k. First, notice that the only relevant digits are the first digit of p (call it p[0]) and the i-th digit of a (call it a[i]). Then:

```
a[i] + k * p[0] = 0 (mod B)
k = a[i] * -(1/p[0]) (mod B)
```

You can precompute -(1/p[0]) (mod B) and then multiply it by a[i] in order to find k. In our example, with p = 97 and base 10, this value is -(1/7) (mod 10) = -3 (mod 10) = 7.

In the real world

For example, in elliptic curve cryptography in the 128-bit level of security, the operands have 256 bits. Suppose the target platform is a desktop PC with 32 bits. Then the operands are represented by eight 32-bit digits (i.e. B=2^32 as opposed to B=10 in the examples); R is 2^256; and the Montgomery reduction requires eight steps (since the numbers have eight integers).

If you look carefully, you can notice that the Montgomery reduction is nothing more than a multiplication of p with an operand which is computed on the fly (the "magic numbers"). This allows the use of many optimizations from multiplication algorithms.

December 24, 2009 Conrado Cryptography, Math algorithm, montgomery, understanding

29 thoughts on "Understanding the Montgomery reduction algorithm"

Daniel

January 2, 2012 at 20:48

Can you go more in depth on "the magic number" section? I'm a little confused on how to calculate -(1/p[0]) mod B. I've got a little toy implementation that uses B=256.

Conrado

January 2, 2012 at 22:19

Sure! I am assuming that you are storing your numbers in byte vectors, therefore p[0] is the lower byte of the prime modulus. Then, 1/p[0] mod B is the inverse, modulo B, of p[0]; i.e. the number x such that x * p[0] mod B = 1. In order to compute it, you can use the extended Euclidean algorithm; but in your case you can simply compute x * p[0] mod B for all bytes x until the result is one.

Martin

January 6, 2012 at 05:48

x * p[0] mod B = 1
not x * p[0] mod B = 0.

Conrado

January 6, 2012 at 07:22

Oops. Thanks!

David

June 8, 2012 at 00:44

hi, you said “In order to convert it back, compute a * (1/R) (mod p) = 46 * 65 (mod 97) = 80.” I wonder how you get 65? Thanks.

Conrado

June 8, 2012 at 10:39

65 is (1/R) (mod p), that is, the multiplicative inverse of R modulo p. You can compute it using the extended Euclidean algorithm.

heba

December 5, 2012 at 05:38

Thank you very much , I relay understood Montgomery reduction algorithm, can you please explain the Montgomery Multiplication algorithm and the five types(SOS,CIOS,FIOS…)with a number example please,Thanks

khalid

October 11, 2013 at 08:57

Indeed it is a great explanation but i still need your help on implementation side, i am implementing 256 * 256 bit montgomery multiplier for pairing computation.
my implementation is block wise of 64 bit size, two approaches are available; one is to get 512 bit product first and than apply montgormery reduction, and second is to reduce intermediate products block wise.(which is suitable and fast).
second question is how to compute (P mod R, in my case R= 2^256) and M' (M'= 1/M, where M is a modulus of 254 bits).

Albert

November 12, 2013 at 08:47

Where could I find test vectors to test my own implementation of Montgomery multiplication in C language ? I need a golden to verify if my answer is right ? I need to test different configuration , such as numbers of 255 bit for 2^24 base …

Conrado

November 12, 2013 at 08:51

I don't think you'll find test vectors, you'll have to generate them on your own with a parallel implementation. I usually just test the multiplication itself (not any intermediate steps), and for that it's simple to generate test vectors (just multiply numbers).

Nic

February 22, 2014 at 21:16

Hi

This a confusion about Montgomery reduction for me: is it still work in in the RSA group(Z_N) where N=p*q ?

Conrado

February 22, 2014 at 22:45

Yes, it works with any odd number number (actually with any modulus which is coprime to R, but since R is usually a power of 2, any odd number will do).

sanaullah

March 2, 2014 at 12:23

Hi! would you like to explain montgomery modular multiplication step by step please as i am doing my final year project under the title “FPGA implementation of montgomery modular multiplier”…. there is no problem in hardware implementation for me but i can’t understand the algorithm … i have searched and read so many articles but none has explained the algorithm by example.
will be waiting for your kind response.
thanks in advance.

Conrado

March 2, 2014 at 12:33

Montgomery multiplication is multiplication followed by Montgomery reduction, which I explained in the post… if you have a specific question, please ask.

sanaullah

March 4, 2014 at 17:41

Bro following is the algorithm which i am going to implement on FPGA,
radix r=2, A,B<2M, M<R= 2^m+2, gcd(2,M)=1
1) So =0
2) for i=0 to m+2 DO
3) qi = Si mod 2

4) Si+1= (Si+qi.M)/2+ai.B
5) END FOR

actually i am not getting step 4, can u please please explain ???
Thanks in advance…

Shah Khalid

April 14, 2014 at 13:25

Sana here is an example .

Example: Let N = 79, and instead of using a power of 2 for R, we'll use R = 100 for readability. We find that 64 · 100 − 81 · 79 = 1, so we have R = 100, R = 64, N = 79, N = 81.
Now let's say that we multiply a = 17 times b = 26 to get 442. The number 17 is really a · 100 modulo 79 for some a. Multiplying 17 · 64 ≡ 61 (mod 79), we find that a = 61. Similarly, 26 · 64 ≡ 5 (mod 79). So when we multiply 17 and 26 in this representation, we're really trying to multiply 61 · 5 = 305 ≡ 68 (mod 79).
Knowing that we can in fact work modulo 79, we know that what we have is
17 · 26 = 442 ≡ (61 · 100) · (5 · 100)
≡ 305 · 100 · 100
≡ 68 · 100 · 100 (mod 79)
and if we multiply by 64 and reduce modulo 79 we should get the right answer:
442 · 64 ≡ 28288 ≡ 6 ≡ 68 · 100 (mod 79).
The function we want is the function that will take as input the 442 and return 6. And the function described above does exactly that:
m = (442 (mod 100)) · 81 (mod 100)
= 42 · 81 (mod 100)
= 3402 (mod 100)
≡ 2 (mod 100)
t = (442 + 2 · 79)/100
= (442 + 158)/100
= 600/100
= 6
and we return t = 6 as the result.

---

safiul islam

July 6, 2015 at 08:22

Can you solve this question?
Q:The Montgomery reduction of 25 modulo 109 w.r.t. 128 is

Shah Khalid

April 14, 2014 at 13:39

In the above algorithm u are running 2 extra loop . 1 loop due to ai.B in step 4 and one loop for converting back from m residue, means for getting exact answer .

---

boob

May 29, 2014 at 04:03

Sir plz explain Algo step by step with example. what is the difference between divide by r and mod r.

Kyle Orlando

December 13, 2014 at 21:23

The magic numbers section doesn't make any sense to me. Say that a = 2660, p = 97, and B = 10 (like the second part of one of your examples above). Say that we want to zero the 2nd element of a, or a[1]. Then, to determine k, we would compute
10 ^ 1 * a[1] * -(1 / p[0]) (mod 10) = 0 , because it's obviously a multiple of 10. This would happen for any i >= 1. But according to your example (before you went into the actual details of how to compute magic numbers), k is 20. Is there something that I'm misunderstanding here?

Conrado ▲

December 13, 2014 at 21:51

Yep, I've written it wrong. You don't need the B^i:

a[i] + k * p[0] = 0 (mod B)

k = a[i] * -(1/p[0]) (mod B)

it's just that you add (k * p * B^i) to a in order to zero the i-th digit of a.

Nick

February 8, 2015 at 14:37

Thank you for the incredibly clear explanation. There was one point where I must be misunderstanding (and a commenter above).

"We have that 46 is the Montgomery representation of x * y (mod p), that is, x * y * R (mod p). In order to convert it back, compute a * (1/R) (mod p) = 46 * 65 (mod 97) = 80. You can check that 43 * 56 (mod 97) is indeed 80."

From the example:
compute: a * [(1/R) (mod p)]
a=46
R=100
p=97

46 * (1/100) mod 97
46 * (0.01 mod 97)
46 * 0.01
0.46 (not 80)

I have a reference calculation in my code that calculates (43*56) mod p = 80, without any Montgomery steps. There must be a property of modular arithmetic that I am missing. Idea?

Nick

February 8, 2015 at 15:06

Oh, maybe I see: you *must* use the extended Euclidean algorithm to calculate the modular inversion?

Conrado ▲

February 8, 2015 at 15:20

Exactly. Division is actually the multiplication by the inverse, and the inverse of x is the number that, when multiplyed by x, gives 1. When you're working with integers modulo n, the inverse is another integer that when multiplyied by x (mod n), gives 1, and you can compute it using EGCD.

Nick

February 8, 2015 at 17:04

I see, thank you. This is such a great resource and it's nice that you answer questions.

Nick

February 8, 2015 at 17:49

Just to verify, I realized I got a little lost:

(1/R) mod p = gcd(R,p) mod p ?

Conrado

February 8, 2015 at 20:03

No, the EGCD can help you compute it the inverse, but the inverse is not the result of the operation itself. Check the other post where I explain it:

http://alicebob.cryptoland.net/understanding-the-extended-euclidian-algorithm/

sara

May 28, 2015 at 05:07

Thank you very much for your compete comments and useful subjects, I'm a master student ,and my thesis about implementation of RSA cryptography with residue number ,
but i don't know Montgomery multiplication or redundant well.i confused about why we multiple our real numbers with R,why adding zeros help us, pleas tell me more,you explain very well, you will help me more,until i finish my study,I'm waiting for your answer,

best regard

Debapriyay Mukhopadhyay

July 2, 2015 at 19:46

The article published in GanitCharcha on Montgomery Multiplication has described the Mathematics in a very easy to understand and lucid way.

http://www.ganitcharcha.com/view-article-Montgomery-Modular-Multiplication.html