

Arithmetic in \mathbb{Z}_N (or \mathbb{F}_p)

- ▶ Imagine you want to represent and perform various operations on **integers modulo N** , i.e., members of the set

$$\mathbb{Z}_N = \{0, 1, 2, \dots, N - 1\}$$

where for convenience, we'll say if N is written in base- b , it has n digits.

- ▶ When N is composite \mathbb{Z}_N is a **ring**; when N is prime \mathbb{Z}_N is a **finite field** so we write \mathbb{F}_p , where $p = N$, instead.
- ▶ One could implement \mathbb{Z}_N simply using integer operations, i.e., “on top of” \mathbb{Z} , but this has some disadvantages ...
 1. Elements of \mathbb{Z}_N are **unsigned**; since they are always positive, any checks for sign in the implementation of \mathbb{Z} are needless overhead.
 2. Elements of \mathbb{Z}_N have a **fixed size** bounded by N , so we can avoid keeping track of the size, plus specialise (e.g., unroll loops) for a particular N .
- ▶ ... so instead we'll investigate efficient techniques to implement \mathbb{Z}_N from scratch, using similar techniques to \mathbb{Z} .

Arithmetic in \mathbb{Z}_N (or \mathbb{F}_p)

- ▶ Imagine we want to compute $x \pm y \pmod{N}$:

- ▶ If $0 \leq x, y < N$ note that

$$0 \leq x + y \leq 2 \cdot (N - 1)$$

and

$$-(N - 1) \leq x - y \leq (N - 1)$$

so, for $t = x \pm y$, we need to compute we can simply subtract (resp. add) N to (resp. from) t to perform a reduction that we term “cheap” or “easy”.

- ▶ Note that this implies $-x \pmod{N}$ can be computed as $N - x$.
- ▶ Imagine we want to compute $x \cdot y \pmod{N}$:

- ▶ If $0 \leq x, y < N$ note that

$$0 \leq x \cdot y \leq (N - 1)^2$$

so, for $t = x \cdot y$, we need to compute

$$t - (N \cdot \left\lfloor \frac{t}{N} \right\rfloor)$$

to perform a reduction we term “expensive” or “full”.

Data Structure (1)

- ▶ The **data structure** we'll use is based exactly on a simplification of the approach we used for \mathbb{Z} :
 1. Each digit will be represented by an unsigned integer data type that matches the processor word size; for $w = 32$

```
typedef unsigned int      WORD;  
typedef unsigned long long  DWORD;
```

2. A given $x \in \mathbb{Z}_N$ will be represented by `mpz_n`

```
typedef struct __mpz_n  
{  
    WORD data[ MPZ_N_SIZE_MAX ];  
}  
mpz_n;
```

where if `x` is an instance of said structure, then

- ▶ `x.data` is the sequence of digits representing the magnitude of `x`.
- ▶ We assume **all** the digits within `x.data` are used; this is sort of like fixing `x.size` to `n`.
- ▶ **Alternatively**, we could just use `mpz` and **ignore** `x.size` and `x.sign` where appropriate (i.e., when we are doing arithmetic modulo N).

Algorithms (1): Addition and Subtraction

- ▶ Our basic approach for addition and subtraction modulo N will be:
 1. Compute the integer sum (or difference) of x and y , i.e., $t = x + y$ or $t = x - y$.
 2. Perform a cheap modular reduction of t .

Algorithm (\mathbb{Z}_N -ADD)

Input: Two multi-precision integers,
 $0 \leq x, y < N$, represented in base- b .

Output: A multi-precision integer $r = x + y$
(mod N), represented in base- b .

```
 $t \leftarrow x + y$   
if  $t \geq N$  then  
     $t \leftarrow t - N$   
end  
return  $t$ 
```

Algorithm (\mathbb{Z}_N -SUB)

Input: Two multi-precision integers,
 $0 \leq x, y < N$, represented in base- b .

Output: A multi-precision integer $r = x - y$
(mod N), represented in base- b .

```
 $t \leftarrow x - y$   
if  $t < 0$  then  
     $t \leftarrow t + N$   
end  
return  $t$ 
```

Algorithms (2): Addition and Subtraction

```
void mpz_n_add( mpz_n* r, const mpz_n* x, const mpz_n* y )
{
    int n = N->size;

    mpn_add( r->data, x->data, n+0, y->data, n+0 );

    if( !mpn_islth( r->data, n+1, N->data, n+0 ) ) {
        mpn_sub( r->data, r->data, n+1, N->data, n+0 );
    }
}

void mpz_n_sub( mpz_n* r, const mpz_n* x, const mpz_n* y )
{
    int n = N->size;

    WORD T[ n + 1 ];

    if( mpn_islth( x->data, n+0, y->data, n+0 ) ) {
        mpn_add( T,          x->data, n+0, N->data, n+0 );
        mpn_sub( r->data, T,          n+1, y->data, n+0 );
    }
    else {
        mpn_sub( r->data, x->data, n+0, y->data, n+0 );
    }
}
```

Algorithms (3): Division

- ▶ Given x and y , the **Extended Euclidean Algorithm (EEA)**, or **XGCD**, computes $\text{xgcd}(x, y) = (f, g, h)$ where

$$f = \text{xgcd}(x, y) = g \cdot x + h \cdot y.$$

- ▶ We can use this to compute the **inverse** of some x modulo N :

1. Set $y = N$, and compute

$$\text{xgcd}(x, N) = (f, g, h).$$

2. For a suitable x and N , we should have $f = 1$ (i.e., x and N are coprime) so

$$g \cdot x + h \cdot N = 1$$

3. If you look at this modulo N , it means that

$$g \cdot x \equiv 1 \pmod{N},$$

i.e., g is the inverse of x modulo N so

$$g \equiv x^{-1} \pmod{N}.$$

- ▶ A **division** operation is then just an inversion followed by a multiplication, i.e.,

$$x/y \equiv x \cdot y^{-1} \pmod{N}.$$

Algorithms (4): Multiplication

- ▶ Given we already have modular addition, we **could** implement modular multiplication using a bit-serial approach such as ...

```
void mpz_n_mul( mpz_n* r, const mpz_n* x, const mpz_n* y )
{
    int n = N->size;

    mpz_n t;

    memset( t.data, 0,          n * sizeof( WORD ) );

    for( int i = n - 1; i >= 0; i-- ) {
        WORD w = y->data[ i ];

        for( int j = BITSOF( WORD ) - 1; j >= 0; j-- ) {
            mpz_n_add( &t, &t, &t );

            if( ( w >> j ) & 1 ) {
                mpz_n_add( &t, &t, x );
            }
        }
    }

    memcpy( r->data, t.data, n * sizeof( WORD ) );
}
```

Algorithms (5): Multiplication

- ▶ ... but this is an extraordinarily **bad** approach; we aren't using the $(w \times w)$ -bit integer multiplier in the processor at all !
- ▶ Our basic approach for multiplication modulo N will be:
 1. Compute the integer product of x and y , i.e., $t = x \cdot y$.
 2. Perform a full modular reduction of t ; remember that if N has n base- b digits, t has at most $2n$ digits.
- ▶ There are two main categories of interest ...
 1. N is “**random**”, i.e., has no special form; an example is an RSA modulus $N = p \cdot q$ for large prime p and q .
 - 1.1 N is known in advance of use, so any pre-computation can be done offline, or
 - 1.2 N changes per-use, so any pre-computation has to be done online.
 2. N is “**special**”, i.e., has a special form; an example is the Mersenne prime $2^{31} - 1$.

Algorithms (6): Multiplication, Random Modulus

- ▶ For a random modulus N , we've got several options:

1. Compute a full **division**, i.e.,

$$t \pmod{N} = t - \left(N \cdot \left\lfloor \frac{t}{N} \right\rfloor\right)$$

- ▶ Uses a standard integer representation.
 - ▶ Requires no pre-computation.
 - ▶ Requires an integer division ... this is expensive !
2. Use **Barrett reduction**.
 - ▶ Uses a standard integer representation.
 - ▶ Requires some pre-computation.
 - ▶ Requires $2 \cdot (n + 1) \cdot (n + 1)$ digit multiplications for an n -digit N .
 3. Use **Montgomery reduction**.
 - ▶ Uses a non-standard representation.
 - ▶ Requires some pre-computation.
 - ▶ Requires $2 \cdot n \cdot n$ digit-multiplications for an n -digit N .

Algorithms (7): Multiplication, Random Modulus (Barrett)

Description

Given N can be written as an n -digit base- b expansion, pre-compute

$$\mu = \lfloor b^{2n}/N \rfloor.$$

Algorithm (\mathbb{Z}_N -BARRETT-RED)

Input: A multi-precision integer, $0 \leq t \leq (N - 1)^2$, represented in base- b .

Output: A multi-precision integer $r = t \pmod{N}$, represented in base- b .

```
q1 ← ⌊t/bn-1⌋
q2 ← q1 · μ
q3 ← ⌊q2/bn+1⌋
r1 ← t (mod bn+1)
r2 ← q3 · N (mod bn+1)
r ← r1 - r2
if r < 0 then
    r ← r + bn+1
end
while r ≥ N do
    r ← r - N
end
return r
```

- ▶ Some things to note:
 - ▶ All variables in the algorithm (e.g., q_i) are members of \mathbb{Z} ; we've abused the subscripts to stress that q_i relates to the **quotient** and r_i relates to the **remainder**.
 - ▶ Several divisions and reductions are by powers of b ; these map to inexpensive shifting and masking.

Algorithms (8): Multiplication, Random Modulus (Barrett)

Implementation removed !

Algorithms (9): Multiplication, Random Modulus (Barrett)

Example

$$\begin{aligned} N &= 667 \\ n &= 3 \\ \mu &= \lfloor 1000000/667 \rfloor \\ &= 1499 \\ \\ t &= 123 \cdot 456 \\ &= 56088 \\ \\ q_1 &= \lfloor 56088/100 \rfloor &= 560 \\ q_2 &= 560 \cdot 1499 &= 839440 \\ q_3 &= \lfloor 839440/10000 \rfloor &= 83 \\ r_1 &= 56088 \pmod{10000} &= 6088 \\ r_2 &= 83 \cdot 667 \pmod{10000} &= 5361 \\ \\ r &= 6088 - 5361 &= 727 \\ \\ r - N &= 727 - 667 &= 60 \end{aligned}$$

Algorithms (12): Multiplication, Random Modulus (Montgomery)

- Challenge #3: compute $123 \cdot 456/1000 \pmod{667}$; this is a bit harder ...

Example

	1	2	3	
	4	5	6	×
<hr/>				
	7	3	8	
	7	3	8	
4	7	4	0	
<hr/>				
	4	7	4	
<hr/>				
	6	1	5	
1	0	8	9	
3	0	9	0	
<hr/>				
	3	0	9	
<hr/>				
	4	9	2	
	8	0	1	
5	4	7	0	
<hr/>				
	5	4	7	
<hr/>				

$p_0 = 6 \cdot 123$
 $t = t + p_0$
 $t = t + 6 \cdot 667$
 $t = t/10$
 $p_1 = 5 \cdot 123$
 $t = t + p_1$
 $t = t + 3 \cdot 667$
 $t = t/10$
 $p_2 = 4 \cdot 123$
 $t = t + p_2$
 $t = t + 7 \cdot 667$
 $t = t/10$

Example

	2	7	2	
	4	3	9	×
<hr/>				
2	4	4	8	
2	4	4	8	
6	4	5	0	
<hr/>				
	6	4	5	
<hr/>				
	8	1	6	
1	4	6	1	
6	1	3	0	
<hr/>				
	6	1	3	
<hr/>				
	1	0	8	8
1	7	0	1	
6	3	7	0	
<hr/>				
	6	3	7	
<hr/>				
	6	3	7	
<hr/>				

$p_0 = 9 \cdot 272$
 $t = t + p_0$
 $t = t + 6 \cdot 667$
 $t = t/10$
 $p_1 = 3 \cdot 272$
 $t = t + p_1$
 $t = t + 7 \cdot 667$
 $t = t/10$
 $p_2 = 4 \cdot 272$
 $t = t + p_2$
 $t = t + 7 \cdot 667$
 $t = t/10$

- ... the difference is basically:
 - We **can't** have a fractional part; when we want to divide by 10 but can't, we add some multiple of 667 (which is always 0 (mod 667)) ...
 - ... the trick is to choose the right multiple so afterwards, we **can** divide by 10.

Algorithms (13): Multiplication, Random Modulus (Montgomery)

► Some things to note:

1. Division by b , in this case $b = 10$, is **free** (i.e., there is no computation): we simply right-shift the digits once we force the accumulator to be divisible by b .
2. Since

$$\begin{aligned}(123 \cdot 1000) \pmod{667} &= 272 \\ (456 \cdot 1000) \pmod{667} &= 439,\end{aligned}$$

in the last example, we've sort of computed

$$\begin{aligned}(123 \cdot 1000) \cdot (456 \cdot 1000) / 1000 &= 637 \pmod{667} \\ &= 60 \cdot 1000 \pmod{667} \\ &= (123 \cdot 456) \cdot 1000 \pmod{667}\end{aligned}$$

i.e., in this example, the value 1000 is “magic”.

3. Is isn't luck that we arrived at the right multiple of 667 to add each time

$$\begin{aligned}(738 \cdot 7) \pmod{10} &= 6 \rightsquigarrow 6 \cdot 667 \\ (1089 \cdot 7) \pmod{10} &= 3 \rightsquigarrow 3 \cdot 667 \\ (801 \cdot 7) \pmod{10} &= 7 \rightsquigarrow 7 \cdot 667\end{aligned}$$

i.e., in this example, the value 7 is “magic”.

Algorithms (14): Multiplication, Random Modulus (Montgomery)

Description

Given N can be written as an n -digit base- b expansion, pre-compute

1. $\rho = b^k$, for the smallest k st. $b^k > N$, and
2. $\omega = -N^{-1} \pmod{\rho}$.

and let the **Montgomery representation** of some integer $0 \leq x \leq N - 1$ be

$$\hat{x} = x \cdot \rho \pmod{N}.$$

Given some integer $0 \leq t \leq N \cdot \rho - 1$, **Montgomery reduction** computes

$$t \cdot \rho^{-1} \pmod{N}.$$

So if we have

$$\begin{aligned}\hat{x} &= x \cdot \rho \pmod{N} \\ \hat{y} &= y \cdot \rho \pmod{N}\end{aligned}$$

then their integer product is $t = x \cdot y \cdot \rho^2$ and, finally, Montgomery reduction yields

$$\hat{t} = x \cdot y \cdot \rho \pmod{N}.$$

Algorithm (\mathbb{Z}_N -MONTGOMERY-RED)

Input: A multi-precision integer, $0 \leq t \leq N \cdot \rho - 1$, represented in base- b .

Output: A multi-precision integer $r = t \cdot \rho^{-1} \pmod{N}$, represented in base- b .

```
r ← t
for i = 0 upto n - 1 step 1 do
    u ← ri · ω (mod b)
    r ← r + (u · N · bi)
end
r ← r / bn
if r ≥ N then
    r ← r - N
end
return r
```

Algorithms (15): Multiplication, Random Modulus (Montgomery)

► Some things to note:

1. In the original example we had $b = 10$, $N = 667$ and $k = 3$ which gives $\rho = 1000$, $\omega = 7$; we can now show algorithmically what is going on:

Example

Assuming the inputs are in their Montgomery representation

$$\begin{aligned}x &= 123 & \mapsto & \hat{x} = 123 \cdot 1000 \pmod{667} = 272 \\y &= 456 & \mapsto & \hat{y} = 456 \cdot 1000 \pmod{667} = 439,\end{aligned}$$

first compute their integer product

$$272 \cdot 439 = 119408$$

then compute the Montgomery reduction:

i	r	u	$u \cdot N \cdot b^i$	r
0	$\langle 8, 0, 4, 9, 1, 1 \rangle$	6	4002	$\langle 0, 1, 4, 3, 2, 1 \rangle$
1	$\langle 0, 1, 4, 3, 2, 1 \rangle$	1	46690	$\langle 0, 0, 1, 0, 7, 1 \rangle$
2	$\langle 0, 0, 1, 0, 7, 1 \rangle$	1	466900	$\langle 0, 0, 0, 7, 3, 6 \rangle$
	$\langle 0, 0, 0, 7, 3, 6 \rangle$			$\langle 7, 3, 6 \rangle$

2. Several divisions and reductions are by powers of b ; these map to inexpensive shifting and masking.
3. Although we can separate the integer multiplication from the modular reduction, it is also possible to **interleave** the two ...

Algorithms (16): Multiplication, Random Modulus (Montgomery)

- ▶ ... there are numerous approaches; this is so-called **Coarsely Integrated Operand Scanning (CIOS)**:

Algorithm (\mathbb{Z}_N -MONTGOMERY-MUL)

Input: Two multi-precision integers, $0 \leq x, y < N$, represented in base- b .

Output: A multi-precision integer $r = x \cdot y \cdot \rho^{-1} \pmod{N}$, represented in base- b .

$r \leftarrow 0$

for $i = 0$ **upto** $n - 1$ **step** 1 **do**

$u \leftarrow (r_0 + x_i \cdot y_0) \cdot \omega \pmod{b}$

$r \leftarrow (r + x_i \cdot y + u \cdot N) / b$

end

if $r \geq N$ **then**

$r \leftarrow r - N$

end

return r

- ▶ The interleaving provides a total solution for modular multiplication **if** we keep x and y in Montgomery representation; note that
 1. \mathbb{Z}_N -MONTGOMERY-MUL($x, \rho^2 \pmod{N}$) = $x \cdot \rho = \hat{x}$, and
 2. \mathbb{Z}_N -MONTGOMERY-MUL($\hat{x}, 1$) = \mathbb{Z}_N -MONTGOMERY-MUL($x \cdot \rho, 1$) = x .

Algorithms (17): Multiplication, Random Modulus (Montgomery)

Implementation removed !

Algorithms (18): Multiplication, Random Modulus (Montgomery)

Implementation removed !

Algorithms (19): Multiplication, Random Modulus (Montgomery)

- ▶ **Problem #1:** Need to pre-compute $\omega = -N^{-1} \pmod{\rho}$; noting that ...
 1. Since ω is **only** used modulo b , only need $\omega = -N^{-1} \pmod{b}$.
 2. Although N could be composite, usually N and ρ (and hence N and b) are **coprime**.
- ▶ ... we have a few options:

1. We know that $\text{xcgcd}(x, y) = (f, g, h)$ where

$$f = \text{xcgcd}(x, y) = g \cdot x + h \cdot y$$

so if N and b are coprime, then setting $x = N$ and $y = b$ computes

$$1 = \text{xcgcd}(N, b) = g \cdot N + h \cdot b$$

i.e., $g \cdot N \equiv 1 \pmod{b}$ so $g \equiv N^{-1} \pmod{b}$.

2. The theorem of Lagrange + Euler tells us that if N and b are coprime then

$$N^{\phi(b)} \equiv 1 \pmod{b}$$

so

$$N^{\phi(b)-1} \equiv N^{-1} \pmod{b}.$$

Note that since $b = 2^w$, we know that $\phi(b) = 2^{w-1}$.

- ▶ **Problem #2:** Need to pre-compute $\rho^2 \pmod{N}$; note that we know $\rho = 2^k$ for some k , so $\rho^2 = 2^{2k}$ which we compute via $2k$ modular “doublings”.

Algorithms (20): Multiplication, Random Modulus (Montgomery)

Algorithm (\mathbb{Z}_N -MONTGOMERY- ω)

Input: The modulus N , represented in base- b .

Output: The Montgomery parameter $\omega = -N^{-1} \pmod{\rho}$.

$t \leftarrow 1 \pmod{b}$

for $i = 1$ **upto** $w - 1$ **step 1 do**

$t \leftarrow t \cdot t \cdot N \pmod{b}$

end

$t = -t \pmod{b}$

return t

Algorithm (\mathbb{Z}_N -MONTGOMERY- ρ^2)

Input: The modulus N , represented in base- b .

Output: The Montgomery parameter $\rho^2 \pmod{N}$.

$t \leftarrow 1 \pmod{N}$

for $i = 0$ **upto** $2 \cdot n \cdot w - 1$ **step 1 do**

$t \leftarrow t + t \pmod{N}$

end

return t

Algorithms (21): Multiplication, Random Modulus (Montgomery)

Implementation removed !

Algorithms (22): Multiplication, Special Modulus

- ▶ A **pseudo Mersenne prime** is of the form

$$N = 2^k \pm c.$$

- ▶ This affords a method for inexpensive reduction: since

$$2^k \pm c = 0 \pmod{N},$$

by writing t in base- 2^k , i.e.,

$$t = \langle t_0, t_1 \rangle_{(2^k)}$$

it turns out that

$$t = t_0 \pm c \cdot t_1 \pmod{N}.$$

- ▶ In selecting N , the aim is to have a small c so the multiplication by c is inexpensive, i.e., a few additions ...
- ▶ ... the ideal case is $c = 1$ where we have a real **Mersenne prime** and multiplication by c is free !

Algorithms (23): Multiplication, Special Modulus

- ▶ A **generalised Mersenne prime** can be written as the sum (or difference) of a number of powers-of-two ...
- ▶ ... the less of them the better, and if those powers are multiples of the word size w then even better still; e.g.,

$$N = 2^{192} - 2^{64} - 1.$$

- ▶ This allows us to pull more or less the same trick as before by writing t in base- 2^w , e.g. if $w = 64$

$$t = \langle t_0, t_1, t_2, t_3, t_4, t_5 \rangle_{(2^{64})},$$

and it turns out that for this N

$$t \equiv a + b + c + d \pmod{N}$$

where

$$a = \langle t_0, t_1, t_2 \rangle_{(2^{64})}$$

$$b = \langle t_3, t_3, 0 \rangle_{(2^{64})}$$

$$c = \langle 0, t_4, t_4 \rangle_{(2^{64})}$$

$$d = \langle t_5, t_5, t_5 \rangle_{(2^{64})}$$

Algorithms (24): Multiplication, Special Modulus

Example

$$\begin{aligned}N &= 2^{13} - 1 \\t &= 123 \cdot 456 \\&= 56088 \\&= \langle 6936, 6 \rangle_{(2^{13})} \\&= 6936 + 1 \cdot 6 \quad (\text{mod } N) \\&= 6942 \quad (\text{mod } N)\end{aligned}$$

Example

$$\begin{aligned}N &= 2^{12} - 2^4 - 1 \\t &= 123 \cdot 456 \\&= 56088 \\&= \langle 8, 1, 11, 13, 0, 0 \rangle_{(2^4)} \\a &= \langle 8, 1, 11 \rangle_{(2^4)} = 2840_{(10)} \\b &= \langle 13, 13, 0 \rangle_{(2^4)} = 221_{(10)} \\c &= \langle 0, 0, 0 \rangle_{(2^4)} = 0_{(10)} \\d &= \langle 0, 0, 0 \rangle_{(2^4)} = 0_{(10)} \\t &= a + b + c + d \quad (\text{mod } N) \\&= 2840 + 221 + 0 + 0 \quad (\text{mod } N) \\&= 3061 \quad (\text{mod } N)\end{aligned}$$

Conclusions

- ▶ Arithmetic in \mathbb{Z}_N is **fundamentally** important to many, asymmetric in particular, cryptographic primitives ...
 1. RSA uses a random modulus $N = p \cdot q$ for large prime p and q .
 2. ECC might use special or random moduli to construct a field \mathbb{F}_p (where $p = N$) and hence a curve $E(\mathbb{F}_p)$.
 3. NTRU operates on elements of some polynomial ring over \mathbb{Z}_N .
- ▶ ... and efficient implementation is **fundamental** to said primitives being useful.

Further Reading

- ▶ V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2008. ISBN: 0-521-85154-8.
 - ▶ Chapter 3 – Computing with large integers
- ▶ A.J. Menezes, P.C. van Oorschot and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN: 0-8493-8523-7.
 - ▶ Chapter 14 – Efficient implementation
- ▶ GNU Multiple Precision Arithmetic Library (GMP). <http://gmplib.org/>
- ▶ P.D. Barrett. Implementing the Rivest, Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology (CRYPTO)*, Springer-Verlag LNCS 263, 311–323, 1986.
- ▶ P.L. Montgomery. Modular Multiplication Without Trial Division. In *Mathematics of Computation*, **44**, 519–521, 1985.
- ▶ Ç.K. Koç, T. Acar and B.S. Kaliski. Analyzing and Comparing Montgomery Multiplication Algorithms. In *IEEE Micro* **16** (3), 26–33, 1996.