Cryptography and Security of Digital Devices

Notes on abstract algebra and modular arithmetic

Multiple Precision Arithmetics and Montgomery Multiplication

A. Barenghi, Gerardo Pelosi

Exam Code 090959 – A.Y. 2011-2012, Semester 2
Politecnico di Milano

1 Multi-precision Arithmetic

This set of notes will describe multiple precision algorithms useful when dealing with *large* non negative integers (as, for instance, the classic representation of the elements of $\mathbb{Z}_p$), expressed in base b, where b can take any integer value larger than 1. We note that, even if the following descriptions will fit for any base, a smart choice of the value of b, which depends on the computing platform, is a crucial point for efficiency. Typically, the base is selected to be of the same size of the underlying architecture word (e.g. $2^8, 2^{32}, 2^{64}$). In particular, b should be taken in such a way that multiplications, divisions and modular reductions mod b^k ($k > 0$) are *fast and simple*. We recall that a single element of a base b representation is called *digit* in base b and we will assume that the underlying architecture computes single digit operations in constant time.

In the following, the modulus will be denoted by N, and expressed in base b as:

$$N = \sum_{i=0}^{n-1} N_i b^i, \quad 0 < N_{n-1} < b \text{ and } 0 \leq N_i < b, \ \ i = 0, 1, \ldots, n-2$$

while let x be a non negative number modulo N

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

1.1 Addition and subtraction algorithms

Additions and subtractions between multiple precision integers in base b are performed according to Algorithm 1.1 and Algorithm 1.2, respectively. Note that the involved operands are assumed to have the *same length* (in terms of number of digits). A subsequent reduction operation modulo N will be performed after the addition/subtraction through checking if the result exceeds N or is negative, and adding or subtracting N to obtain a result in the $[0, N-1]$ range.

Algorithm 1.1: Multiple Precision addition

Input: $A = A_{n-1}b^{n-1} + \ldots + A_1 b + A_0$, $B = B_{n-1}b^{n-1} + \ldots + B_1 b + B_0$
Output: $x = A + B = x_n b^n + \ldots + x_1 b + x_0$

```
1  begin
2      x ← ⟨0, ..., 0⟩
3      k_out ← 0
4      for i ← 0 to n − 1 do
5          k_in ← k_out
6          x_i ← (A_i + B_i) mod b
7          tmp ← (x_i + k_in) mod b
8          k_out ← (x_i < A_i) + (tmp < x_i)
9          x_i ← tmp
10     x_n ← k_out
11     return x
```

Algorithm 1.1 employs the convention of evaluating the expression $(x_i < A_i)$ to one if $x_i < A_i$ is true and 0 otherwise [1]. Algorithm 1.1 is the common schoolbook method to perform additions on multiple digit numbers (the one you're used to perform by hand on paper). Apart from the digit by digit addition (performed at line 6), particular care should be taken in the propagation of the carry.

The loop scanning on the digit of the two operands (lines 4–9) starts through setting the carry in as the one produced by the previous single digit addition (line 5). Subsequently, it adds together the two digits of the two numbers (line 6). After the single digit addition, the algorithm adds in the carry generated from the previous operation (line 7). At line 8, the algorithm merges together the results of two carry checks. The carry generation checks rely on the fact that the single digit result of an addition where a carry has been generated is lower than both addends. Moreover, as it is not possible for the addition of "two single digits and a carry" to trigger another carry, it is safe to simply add together the result of both carry generation checks. Indeed, the maximum value reachable with a single digit addition plus a carry is $(b-1)+(b-1)+1 = 2b-2+1 = 2b-1 < 2b$ – the operation can be rewritten in multi-precision notation as: $(0, b-1)_b + (0, b-1)_b + (0, 1)_b = (1, b-2)_b + (0, 1)_b = (1, b-1)_b$.

Example 1.1. *A running addition example is the following one: we want to add together two operands, three decimal digits wide,i.e., $b = 10$ (Figure 1(a)).*

1	4	5	+	1	4^1	5	+	1^1	4^1	5	+	1^1	4^1	5	+				
	9	7			9	7			9	7			9	7					
						2			4	2		2	4	2					
(a)				(b)				(c)				(d)							

The first iteration of the algorithm adds together 5 and 7, then, as the value of k_{in} is zero, the algorithm checks if a k_{out} is generated. Since the check for

[1] Basically, you copy this down in `C` and it Just Works™ as the unsigned integer arithmetic in `C` is the one of integers modulo $2^{\texttt{word\_length}}$

the generation succeeds, the carry is saved, resulting in the state depicted in Figure 1(b). The second iteration starts by adding 9 and 4 together, obtaining 3, which is saved in tmp . After that, the carry-in, which was set to 1 is added obtaining $x_i = 4$. *After this, the algorithm performs the carry check obtaining* $k_{out} = 1 + 0$ *and thus sets* k_{out} *marking the presence of a carry. The state at the end of the second iteration is depicted in Figure 1(c). In the last iteration, the algorithm adds together the first digit of the first addend* 1, *with the implicit* 0 *in the second addend, and then adds together the carry-in coming from the previous iteration. This results in no further carries being triggered and in the result depicted in Figure 1(d).*

The subtraction algorithm is analogous to the addition one, simply taking care of propagating a borrow instead of a carry. The described subtraction algorithm assumes that the first operand is always lesser than, or equal to, the second one.

Algorithm 1.2: Multiple precision subtraction

Input: $A = A_{n-1}b^{n-1} + \ldots + A_1 b + A_0$, $B = B_{n-1}b^{n-1} + \ldots + B_1 b + B_0$
Output: $x = A - B = x_n b^n + \ldots + x_1 b + x_0$

```
1  begin
2      x ← ⟨0, ..., 0⟩
3      k_out ← 0
4      for j ← 0 to n − 1 do
5          k_in ← k_out
6          x_i ← (A_i − B_i) mod b
7          tmp ← (x_i − k_in) mod b
8          k_out ← (x_i > A_i) + (tmp > x_i)
9          x_i ← tmp
10     x_n ← k
11     return x
```

2 Multiplication Algorithms

There are a number of algorithms to perform efficient multiplications among large, multiple precision, integers. Depending on the bit size of the operands, the choice of the algorithm falls upon a different one, namely:

- Schoolbook multiplication: Complexity $\Theta(n^2)$ fastest below ≈ 320 bits
- Karatsuba: Complexity $\Theta(3n^{\log_2(3)})$ fastest between ≈ 320 and ≈ 10000 bits
- Toom-Cook: Complexity $\Theta(n^{log_d(2d-1)})$, with $d \geq 3$; fastest between ≈ 10000 and ≈ 32000 bits
- Schönage-Strassen: Complexity $\Theta(n \log(n) \log(\log(n)))$ fastest above ≈ 32000 bits

For our purposes the schoolbook multiplication and the Karatsuba method are the most useful techniques. We will now start by presenting two different schoolbook multiplication implementations examining the optimization applicable to a varying detail level.

Algorithm 2.1 reports the pseudo-code of a schoolbook multiplication between two n-digit unsigned integers represented in base b.

Algorithm 2.1: Multiple Precision Multiplication Algorithm

Input: $A = A_{n-1}b^{n-1} + \ldots + A_1 b + A_0$, $B = B_{n-1}b^{n-1} + \ldots + B_1 b + B_0$
Output: $x = A \cdot B = x_{2n-1}b^{2n-1} + \ldots + x_1 b + x_0$

```
1 begin
2     x ← 0
3     for i ← 0 to n − 1 do
4         x ← x + A_i B b^i
5     return x
```

Willing to detail further the single digit operations involved in the previous algorithm, a first optimized version is presented in Algorithm 2.2. We assume that all the single precision operations are working modulo b, as it is the typical case when b is the word length of the machine architecture: i.e.: $b = 2^{\text{word-length}}$, with word-length $\in \{8, 16, 32, 64\}$. In order to add a digit k, to a double precision value $(u, v)_b$, the digit k should be added to the less significant one of the double precision value as $v = v + k$ and the possible occurrence of a carry should be handled adding 1 to the most significant one, if it is needed (i.e. if $v < k$ modulo b).

Algorithm 2.2: Optimized multiple precision multiplication

Input: $A = A_{n-1}b^{n-1} + \ldots + A_1 b + A_0$, $B = B_{n-1}b^{n-1} + \ldots + B_1 b + B_0$
Output: $x = A \cdot B = x_{2n-1}b^{2n-1} + \ldots + x_1 b + x_0$

```
1  begin
2      x ← 0
3      for j ← 0 to n − 1 do
4          k ← 0
5          for i ← 0 to n − 1 do
6              (u, v)_b = A_i B_j
7              v = v + k
8              u = u + (v < k)
9              v = v + x_{i+j}
10             u = u + (v < x_{i+j})
11             x_{i+j} = v
12             k = u
13         x_{j+n} = k
14     return x
```

The Karatsuba technique for fast multiplication relies on the intuition that additions are faster than multiplications and on a divide-et-impera approach to multiple-precision multiplication.

The key idea of the algorithm is that, given two multiple digit numbers A and B represented in base b with $2k$ digits each, it is possible to represent them as $(A_1 b^k + A_0)$ and $(B_1 b^k + B_0)$, respectively, and thus represent the product C as

$$C = AB = c_2 b^{2k} + c_1 b^k + c_0$$

with

$$c_2 = A_1 B_1, \quad c_1 = A_0 B_1 + A_1 B_0, \quad c_0 = A_0 B_0$$

A schoolbook approach would employ 4 k-digit multiplications and 1 k-digit addition to compute the three values c_0, c_1, c_2, but Karatsuba noted that:

$$\begin{aligned} c_1 &= A_0 B_1 + A_1 B_0 \\ &= A_0 B_1 + A_1 B_0 + A_1 B_1 + A_0 B_0 - A_1 B_1 - A_0 B_0 \\ &= (A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0 \end{aligned} \tag{1}$$

which in turn requires only 3 multiplications and 4 additions to be computed. Since the cost of the additions is linear w.r.t. the number of digits while the multiplications are quadratic, the Karatsuba technique is effectively faster than the schoolbook method, provided the numbers are long enough.

In addition to this, it is possible to recursively apply the Karatsuba method for the multiplications needed to compute c_2, c_1, c_0 in order to further exploit its advantages. The limit in the recursion depth is given by either reaching single digit multiplications, or obtaining operands so short that the common schoolbook method performs better, whichever comes first (usually the latter).

The Toom-Cook method is a straightforward extension of the Karatsuba idea, through splitting the two inputs of the multiplication in more than two parts. This, in turn, implies that computing the middle terms requires more

extra-additions, thus making the method suitable for a relatively small number of parts (typically 3).

The Schönage-Strassen method exploits the fact that the convolution of two vectors can be computed efficiently by means of a Fast Fourier Transform. The key intuition is that computing the convolution of the coefficients of a proper large digit representation of the two operands is faster than performing the common multiplication algorithms.

When considering the implementation of "modular" multiplications, the most straightforward strategy is to perform a sequence of interleaved multiplications and reductions in order to keep the size of the result within reasonable limits. However, such a straightforward strategy needs an efficient way to perform reductions modulo N, a typically challenging task, if the modulus is not fixed [2]. For fixed moduli, it is possible to efficiently perform the modular reduction through a chain of subtractions of fixed values depending on the value of the digits of the result which exceed the ones of the modulus. In case the value of the modulus N is not fixed, the Montgomery multiplication strategy, presented in the following section, is the most efficient method.

[2] For instance, on the ARM architecture, there's no hardware support for division, which in turn implies that a schoolbook division algorithm is performed by the processor every time!

3 Montgomery Representation and Single Precision Multiplication

Computing a modular multiplication employing the multiply-and-reduce strategy mentioned before assumes that the modular reduction is extremely efficient. Since we wish to avoid the usage of single (or even worse, multiple) precision divisions, we need a way to overcome this empasse. Paul Montgomery described a technique able to perform a modular multiplication at a computational cost almost equal to the one of a simple multiplication between multiple precision integers. More in detail, if we define as n the number of digits of the factors, the computational complexity of a Montgomery multiplication is of $2n(n+1) = 2n^2 + 2n$ single precision multiplications, while a regular multiple precision multiplication needs n^2 of them.
Let N be a positive integer and let us consider the *ring* $\mathbb{Z}_N$=$\{0, 1, 2, \ldots, N-1\}$. We define as Montgomery transformation a specific permutation of the elements of $\mathbb{Z}_N$ defined as follows:

$$\mu : \quad (\mathbb{Z}_N, +, \cdot) \quad \mapsto \quad (\widetilde{\mathbb{Z}_N}, +, *)$$

Basically, we can see the Montgomery map as an *homomorphism* between two rings: $\mathbb{Z}_N$ with its usual (modular) multiplication and addition operations and a set of numbers $\widetilde{\mathbb{Z}_N}$ (henceforth known as Montgomery domain) with the same addition operation, but a different (more computationally efficient, modular) multiplication operation.
Given $a, b \in \mathbb{Z}_N$, it must be true that:

$$\begin{aligned} \mu(a + b \bmod N) &= \mu(a) + \mu(b) \bmod N \\ \mu(a \cdot b \bmod N) &= \mu(a) * \mu(b) \bmod N \end{aligned}$$

The modular multiplication in $\widetilde{\mathbb{Z}_N}$ is faster than in $\mathbb{Z}_N$ so, if an algorithm requires a long sequence of modular multiplications on operands in $\mathbb{Z}_N$, a good strategy is to map them in the Montgomery domain, do the multiplications in $\widetilde{\mathbb{Z}_N}$ and map them back to $\mathbb{Z}_N$ only when the computation is finished.
Note that, since the addition operation is the same in the two domains, we can also perform additions between the multiplications in the Montgomery domain without the need to map the elements back.

Given our desiderata, we will need to define operatively the μ transformation. Given x, an element of $\mathbb{Z}_N$, and a positive integer R>N coprime with N. The parameters R' and N' are defined as follows:

$$\gcd(R, N) = 1 = R \cdot R' - N \cdot N'$$

where R' and N' satisfy the following relations:

$$\begin{aligned} R' &\equiv R^{-1} \bmod N \text{ with } 0 \leq R' < N \\ N' &\equiv -N^{-1} \bmod R \text{ with } 0 \leq N' < R \end{aligned}$$

The Montgomery transformation μ is defined as follows:

$$\begin{aligned} \widetilde{x} &= \mu(x) = x \cdot R \bmod N \\ x &= \mu^{-1}(\widetilde{x}) = \widetilde{x} \cdot R' \bmod N \end{aligned}$$

Note that the functions μ, μ^{-1} are bijective due to the coprimality of R and N.

Since the conditions binding a valid choice for the value of R, which is also known as *Montgomery Radix*, only require it to be greater than N and coprime with it, the most common choice is to pick $R{=}2^k$ for a sufficiently large positive integer k. The coprimality condition with N is satisfied easily in practice, as N is either a large prime number or the product of two large prime numbers, thus $\gcd(2^k, N){=}1$.

The reason for choosing the Montgomery Radix equal to a power of 2 is the fact that an integer division by 2^k is simply a k-bit right shift: an extremely fast and simple operation in both hardware and software implementations.

Now that we have an operative definition of the Montgomery transformation, it is easy to check that μ is an homomorphism w.r.t. the usual addition and subtraction operations but not w.r.t. the usual modular multiplication:

$$\begin{aligned}
\mu(a \pm b \bmod N) &= (a \pm b) \cdot R \bmod N = \\
&= (a \cdot R \bmod N) \pm (b \cdot R \bmod N) \bmod N = \\
&= \mu(a) \pm \mu(b) \bmod N \\
\mu(a \cdot b \bmod N) &= (a \cdot b) \cdot R \bmod N \neq \mu(a) \cdot \mu(b) \bmod N
\end{aligned}$$

We want to have a modular multiplication operation (denoted as $*$) in $\widetilde{\mathbb{Z}_N}$ such that $\mu(a \cdot b \bmod N) = \mu(a) * \mu(b) \bmod N$. Therefore, the definition of the operation $*$ must be substantially different from the common product $(\mathbb{Z}_N, \cdot)$.

With the final goal of providing an operative description of how the multiplication operation, $*$, in the Montgomery domain can be performed, we now tackle the issue of performing the following computation efficiently:

$$x \in \mathbb{Z}_N, \quad x \leftarrow \frac{x}{R} \bmod N, \text{ with } 0 < x < R \cdot N$$

To the end of efficiently divide by R, it should be true that the Radix-R representation of x has a zero least significant digit (i.e., $x{=}(x_{d-1}, x_{d-2}, \ldots, x_1, 0)_R$): in this case it is possible to divide x by R through simply shifting to the right the digits of x by one position.
The clever idea in the Montgomery reduction method is to add to x a positive integer $t{\in}\mathbb{N}$, which causes the last digit of the Radix-R representation of the sum $x{+}t$ to be zero, and subsequently compute both the right-shift and the modulo operations:

$$x \leftarrow \frac{x + t}{R} \bmod N$$

This procedure will yield the same result as $(x/R) \bmod N$, provided that:

$$\begin{cases} x + t \equiv_N x \text{ (adding } t \text{ does not change the value of } x \bmod N) \\ x + t \equiv_R 0 \text{ (condition for fast division by } R) \end{cases}$$

Through manipulation of the previous conditions, we obtain the value of t which we should employ as:

$$\begin{cases} t \equiv_N 0 \Rightarrow t = t'N,\ t' \in \mathbb{N}^+ \\ x + t'N \equiv_R 0 \end{cases} \Rightarrow \begin{cases} t = t'N,\ t' \in \mathbb{N}^+ \\ t' \equiv_R (-N^{-1})x = N'x \bmod R \end{cases}$$

Algorithm 3.1: MRED(·) - Montgomery Reduction

Input: $0 < x \leq RN$
N, R con $R > N$, $\gcd(R, N) = 1 = RR' - NN'$
Output: $xR^{-1} \mod N$

```
1 begin
2     t' ← xN' mod R // Cost:  1 modular multiplication mod R
3     x ← x + Nt' // Cost:  1 addition mod R
4     x ← x/R // This is just a shift
5     if x ≥ N then
6         x ← x − N
7     return x
```

We note that the value of N' can easily be precomputed offline if the modulus N is fixed. Employing the value of t which we derived, we can thus define the Montgomery Reduction Algorithm 3.1 as follows:
The algorithm is thus able to compute $(xR^{-1} \mod N)$ at the cost of 1 multiplication, 1 addition and two reductions modulo R. In order to justify the need for the check and the eventual single multiprecision subtraction to be performed at the end of the algorithm, we observe that, since $t'<R$ and $x<RN$, consequentially $x+Nt'<RN+RN$, thus we obtain that $\frac{x+Nt'}{R}<\frac{2RN}{R}=2N$, hence the need to compare the result with N and subtract N from the result at most once.

Exploiting the Montgomery reduction operation MRED(.), we can define the modular multiplication operation between two elements of the Montgomery domain $\widetilde{x}, \widetilde{y} \in \widetilde{\mathbb{Z}_N}$ to compute the value: $\widetilde{x} * \widetilde{y} \mod N$, as described in Algorithm 3.2.

Algorithm 3.2: MMUL(·,·) - Montgomery Multiplication

Input: $\widetilde{x} = xR, \widetilde{y} = yR \in \mathbb{Z}_N$
Output: $\widetilde{z} = xyR \mod N$

```
1 begin
2     return z̃ ← MRed(x̃ · ỹ)
```

We note that the MMUL(·,·) operation yields a result which is still in the Montgomery domain, since it basically acts as follows:

$$\widetilde{x} * \widetilde{y} \bmod N = (xR \cdot yR) \cdot R^{-1} \mod N = (xyR^2) \cdot R^{-1} \mod N = \widetilde{xy} \mod N$$

The net effect of this is thatMMUL(·,·) takes 1 multiplication (the one to perform $\widetilde{x} \cdot \widetilde{y}$), plus the cost of the Montgomery reduction (1 mul + 1add + 1shift), which is significantly faster than performing the regular reduction modulo N.
Considering the previous definitions, we will now express the transformation functions $\mu(\cdot)$ and $\mu^{-1}(\cdot)$ employing the MMUL(·,·). This is particularly useful in practice as we will only need to implement the MMUL(·,·) to have a fully functioning Montgomery multiplier.
We thus recall the definition of $\mu(\cdot)$ and $\mu^{-1}(\cdot)$:

$$\begin{aligned} \mu: &\quad (\mathbb{Z}_N, +, \cdot) \to (\widetilde{\mathbb{Z}_N}, +, *) \ : \ \widetilde{x} = \mu(x) = xR \mod N \\ \mu^{-1}: &\quad (\widetilde{\mathbb{Z}_N}, +, *) \to (\mathbb{Z}_N, +, \cdot) \ : \ x = \mu^{-1}(\widetilde{x}) = \widetilde{x}R' \mod N \end{aligned}$$

And we note that, assuming the constant value ($R^2 \bmod N$) is known, we can define them as:

$$\widetilde{x} = \mu(x) = x\,R \bmod N = \text{MMul}(x, R^2 \bmod N) \in (\widetilde{\mathbb{Z}_N}, +, *)$$

$$x = \mu^{-1}(\widetilde{x}) = \widetilde{x}\,R' \bmod N = \text{MMul}(\widetilde{x}, 1) \in (\mathbb{Z}_N, +, \cdot)$$

4 Multiple Precision Montgomery

The previous section presented the Montgomery multiplication technique, assuming that all the operations (multiplications and reductions) were done on a single digit number. We will now show the actual benefits of employing this technique when dealing with multiple precision multiplications such as the ones described in previous sections. From now on, we use the following notation to represent the modulus N and the involved number x as d digits numbers in base b:

$$N = (N_{d-1}N_{d-2}\dots N_1N_0); \quad x = (x_{d-1}x_{d-2}\dots x_1x_0); \quad x < N$$

and we assume the Montgomery radix R to be $R = b^d$.

As a first observation we want to note that we can represent the operation $\frac{x}{R} \mod N$ in base b through explicitly expanding all the the divisions by b and putting into evidence every actual digit:

$$x \leftarrow \frac{x}{R} \mod N = \frac{x}{b^d} \mod N = (\cdots(\underbrace{\underbrace{(\underbrace{\frac{x}{b} \mod N}_{i=0})\frac{1}{b} \mod N}_{i=1})\cdots\frac{1}{b} \mod N}_{i=d-1})$$

This representation also operatively defines the division by R is performed by means of d iterated divisions by b. We will now exploit the aforementioned equality to build a multiple precision MRED(.) algorithm step by step.

We will start by analyzing the value of the first (i.e., the least significant) digit of the result, i.e., when $i = 0$.

$$x \leftarrow \frac{x}{b} \mod N \Leftrightarrow x \leftarrow \frac{x + t'}{b} \mod N$$

Applying the proper conditions, we obtain the value of t' as follows:

$$\begin{cases} x + t' \equiv_N x \\ x + t' \equiv_b 0 \end{cases} \Rightarrow \begin{cases} t' \equiv_N 0 \Rightarrow t' = tN,\ t \in \mathbb{N}^+ \\ x + tN \equiv_b 0 \Rightarrow x_0 + tN_0 \equiv_b 0 \end{cases} \Rightarrow \begin{cases} t' = tN,\ t' \in \mathbb{N}^+ \\ t \equiv_b (-N_0^{-1})x_0 \equiv_b N_0'x_0 \end{cases}$$

Consequentially we can state the following equality :

$$x = \frac{x + t'}{b} \mod N \Leftrightarrow x = \frac{x + (N_0'x_0 \bmod b)N}{b} \mod N$$

Note that the value of t' correctly depends only on a single digit of x and the first digit of N'. If we assume to delay both the division by b and the modular reduction mod$N$ we obtain that:

$$x = x + (N_0'x_0 \mod b)Nb^0 \Rightarrow x = (x_dx_{d-1}\dots x_2x_10)$$

that is, the multiple precision number x can be represented on $d + 1$ digits, of which the least significant one is equal to zero.
We now move on to the next iteration of the division loop ($i = 1$): for the sake of simplicity, we will still note the digits in the accumulator as x_i, but it is clear

that now they are containing the values obtained from the previous operation. As in the first iteration we proceed to compute the value of t' for this digit, recalling that the value we are dealing with is the result of the (integer) division $\frac{x}{b}$ done in the previous iteration :

$$\begin{cases} \frac{x}{b} + t' \equiv_N x \\ \frac{x}{b} + t' \equiv_b 0 \end{cases} \Rightarrow \begin{cases} t' \equiv_N 0 \Rightarrow t' = tN,\ t \in \mathbb{N}^+ \\ \frac{x}{b} + tN \equiv_b 0 \Rightarrow x_1 + tN_0 \equiv_b 0 \end{cases} \Rightarrow \begin{cases} t' = tN,\ t' \in \mathbb{N}^+ \\ t \equiv_b (-N_0^{-1})x_1 \equiv_b N_0' x_1 \end{cases}$$

Thus, similarly to the previous iteration, we obtain the value of t' as:

$$x \leftarrow \frac{\frac{x}{b} + t'}{b} \bmod N \Leftrightarrow x \leftarrow \frac{\frac{x}{b} + (N_0' x_1 \mod b)N}{b} \bmod N$$

Delaying again the divisions by b and the modular reduction by N we can state that:

$$x = x + (N_0' x_1 \mod b)Nb^1 \Rightarrow x = (x_{d+1}x_d \dots x_3 x_2 00)$$

that is, now the value of x, represented on $d+2$ digits, has the two least significant ones equal to 0.
Let's consider now the last iteration of the division loop ($i = d-1$), the value of the last value t can be computed as:

$$\begin{cases} \frac{x}{b^{d-1}} + t' \equiv_N x \\ \frac{x}{b^{d-1}} + t' \equiv_b 0 \end{cases} \Rightarrow \begin{cases} t' \equiv_N 0 \Rightarrow t' = tN,\ t \in \mathbb{N}^+ \\ \frac{x}{b^{d-1}} + tN \equiv_b 0 \Rightarrow x_{d-1} + tN_0 \equiv_b 0 \end{cases} \Rightarrow \begin{cases} t' = tN,\ t' \in \mathbb{N}^+ \\ t \equiv_b (-N_0^{-1})x_{d-1} \equiv_b N_0' x_{d-1} \end{cases}$$

Thus obtaining a final value for x of:

$$x = \frac{\frac{x}{b^{d-1}} + t'}{b} \bmod N \Leftrightarrow x = \frac{\frac{x}{b^{d-1}} + (N_0' x_{d-1} \mod b)N}{b} \bmod N$$

Assuming to have delayed all the divisions by b and reductions by N up to now, we have that the value of x is now:

$$x \leftarrow x + (N_0' x_{d-1} \mod b)N \Rightarrow x = (x_{2d}x_{2d-1} \dots x_d \underbrace{00 \dots 0}_{d \text{ digits}})$$

Note that, if no reductions are performed, the number x will end up in being represented with $2d$ digits, but the d least significant ones will all be equal to zero. It is thus possible to perform the division by b^d which has accumulated up to now in a single step, by simply performing a right shift of d digits.
Similarly to the single precision case, we need to check if the result is bigger than the modulus, and eventually subtract N once.

Putting together these observation we are now able to write down Algorithm 4.1, which effectively performs Montgomery on multiple precision numbers.

Algorithm 4.1: Multiple precision Montgomery Reduction

Input: $x = x_{d-1}b^{d-1} + \ldots + x_1 b + x_0$, $N = N_{d-1}b^{d-1} + \ldots + N_1 b + N_0$,
$R = b^d$, $\gcd(R, N) = 1 = RR' - NN'$
Output: $xR^{-1} \mod N$

```
1  begin
2      for i ← 0 to d − 1 do
3          t ← N'_0 x_i mod b
4          x ← x + tNb^i
5      x ← x/b^n
6      if x ≥ N then
7          x ← x − N
8      return x
```

Putting together the aforementioned algorithm with the common multiple precision multiplication algorithm (i.e. Algorithm 2.1) we obtain the Montgomery multiplication algorithm for two multiple precision integers (Algorithm 4.2).

Algorithm 4.2: Multiple precision Montgomery multiplication

Input: $A = A_{d-1}b^{d-1} + \ldots + A_1 b + A_0$, $B = B_{d-1}b^{d-1} + \ldots + B_1 b + B_0$,
$N = N_{d-1}b^{d-1} + \ldots + N_1 b + N_0$,
$R = b^d$, $\gcd(R, N) = 1 = RR' - NN'$,
$N'_0 = (-N^{-1} \mod R) \mod b$
Output: $x = ABR^{-1} \mod N$

```
1  begin
2      x ← 0
3      for i ← 0 to d − 1 do
4          x ← x + A_i B b^i
5          t ← N'_0 x_i mod b
6          x ← x + tNb^i
7      x ← x/b^d
8      if x ≥ N then
9          x ← x − N
10     return x
```

Remark 4.1. *At each iteration of the* `for` *loop (lines 3–6) the i-th least significant digit of the accumulator x becomes zero.*

Algorithm 4.2 can be modified slightly to reduce the number of single digit operations, as shown in Algorithm 4.3.

Algorithm 4.3: Optimized Multiple Precision Montgomery Multiplication

Input: $A = A_{d-1}b^{d-1} + \ldots + A_1 b + A_0$, $B = B_{d-1}b^{d-1} + \ldots + B_1 b + B_0$,
$N = N_{d-1}b^{d-1} + \ldots + N_1 b + N_0$,
$R = b^d$, $\gcd(R, N) = 1 = RR' - NN'$,
$N'_0 = (-N^{-1} \mod R) \mod b$
Output: $x = ABR^{-1} \mod N$

```
1 begin
2     x ← 0
3     for i ← 0 to d − 1 do
4         x ← x + A_i B
5         t ← N'_0 x_0 mod b
6         x ← (x + tN)/b
7     if x ≥ N then
8         x ← x − N
9     return x
```

Remark 4.2. *In both Algorithm 4.2,and Algorithm 4.3 a smart choice of the value of the Montgomery radix R can further speed up their execution. In particular, when computing a modular exponentiation the value of R can be chosen in such a way to skip the final subtraction at each intermediate step, and doing it only once at the end of the computation. Picking R as $R > 4N$ ($R \geq 4b^d$) instead of $R > N$ ($R \geq b^d$), multiplying two operands $A, B < 2N$ all the partial results will be lesser than $2N$. This can be easily proven considering the following inequality chain:*

$$\frac{(A \cdot B + N(R-1))}{R} \leq \frac{(2N-1)^2 + N(R-1)}{R} < \frac{4N^2 + NR}{R} = \frac{4N^2}{R} + \frac{NR}{R} < 2N.$$

The first link in the inequality chain holds because both A and B are bounded by $N-1$; the second is obtained by algebraic manipulation and the last exploits the fact that $R > 4N$.

Remark 4.3. *Through the exam of Algorithm 4.3 we note that its computational complexity is $\Theta((2d+1)d)$, with the computational unit being a multiplication of a two digits in base b. If we compare this complexity with the one of the ordinary multiple precision multiplication ($\Theta(d^2)$), and recall the fact that the Montgomery multiplication also deals with the modular reduction at the end, the advantages of the representation are evident.*

For the sake of completeness in the following we show the pseudo-code of the Mongomery multiplication algorithm as reported by Menezes et al. in the *Hand-Book of Applied Cryptography* (Algorithm 14.36).
Note that this is only a different way to write the operations in the loop-body of Algorithm 4.3, while the number of digits of the multiprecision integers is denoted as n instead of denoting it as d.

Algorithm 4.4: Montgomery Multiplication as described by Menezes et al., HandBook of Applied Cryptography, Algorithm 14.36

Input: $A = A_{n-1}b^{n-1} + \ldots + A_1 b + A_0$, $B = B_{n-1}b^{n-1} + \ldots + B_1 b + B_0$,
$N = N_{n-1}b^{n-1} + \ldots + N_1 b + N_0$,
$R = b^n$, $\gcd(R, N) = 1 = RR' - NN'$,
$N'_0 = (-N^{-1} \mod R) \mod b$
Output: $x = ABR^{-1} \mod N$

```
1 begin
2     x ← 0
3     for i ← 0 to n − 1 do
4         t ← N'_0(x_0 + A_i B_0) mod b
5         x ← (x + A_i B + tN)/b
6     if x ≥ N then
7         x ← x − N
8     return x
```

5 Examples

5.1 Example 1

Given the following parameters:

$$b = 2,\ n = 4,\ N = 11,\ R = 2^n = 16$$
$$\mathbb{Z}_N = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$
$$R\,\mathbb{Z}_N = \{0, 5, 10, 4, 9, 3, 8, 2, 7, 1, 6\}$$
$$\gcd(R, N) = RR' - NN' \Leftrightarrow \gcd(16, 11) = 1$$

Through employing the Euclidean algorithm for the gcd computation, we find the values of R' and N':

$u = (16, 1, 0)$	$v = (11, 0, 1)$	$q = [16/11] = 1$
$w = (5, 1, -1)$		
$u = (11, 0, 1)$	$v = (5, 1, -1)$	$q = [11/5] = 2$
$w = (1, -2, 3)$		
$u = (5, 1, -1)$	$v = (1, -2, 3)$	$q = [5/1] = 5$
$w = (0, 11, -16)$		
$u = (1, -2, 3)$	$v = (0, 11, -16)$	

$$R' = -2, \quad N' = -3, \quad N'_0 = 1 \bmod 2$$

Perform the modular multiplication between $A = 4_{\text{dec}}$, and $B = 8_{\text{dec}}$:

$AB \bmod N = 32 \bmod 11 = 10$,

$\widetilde{(AB)} = \text{MMul}((A \cdot B), R^2 \bmod N) = (AB)R \bmod N = 6$
$\widetilde{A} = \text{MMul}(A, R^2 \bmod N) = AR \bmod N = 9$,
$\widetilde{B} = \text{MMul}(B, R^2 \bmod N) = BR \bmod N = 7$
$\widetilde{(AB)} = \text{MMul}(\widetilde{A} \cdot \widetilde{B}) = \widetilde{A}\widetilde{B}R^{-1} \bmod N = 63(-2) \bmod 11 = 6$

Referring to Algorithm 4.3:

$\widetilde{A} = 9_{\text{dec}} = 1001_2, \widetilde{B} = 7_{\text{dec}} = 0111_2, N = 1011_2, N'_0 = 1$

0000	+
0111	$A_0B = 0111$
0111	+
1011	$t\,N = (N'_0X_0)N = 1011$
10010	$right\ shift$
1001	+
0000	$A_1B = 0000$
1001	+
1011	$t\,N = (N'_0X_0)N = 1011$
10100	$right\ shift$
1010	+
0000	$A_2B = 0000$
1010	+
0000	$t\,N = (N'_0X_0)N = 0000$
01010	$right\ shift$
0101	+
0111	$A_3B = 0111$
1100	+
0000	$t\,N = (N'_0X_0)N = 0000$
01100	$right\ shift$

The result is 0110, which should be checked to be smaller than $\widetilde{(AB)} = 0110_2 = 6$

5.2 Exercise 1

We want to compute multiplications in $\mathbb{Z}_{29}$ employing Montgomery's Multiplication technique

1. Compute the *smallest* useful Montgomery root R.
2. Compute the values of R', N' and N'_0 needed to perform the Montgomery direct and inverse transforms.
3. Compute the following quantities, given $a = 13, b = 17$:
   - $\alpha = \mu(a)$
   - $\beta = \mu(b)$
   - $\gamma = \texttt{MMul}(\alpha, \beta)$
   - $c = \mu^{-1}(\gamma)$
   - $d = a \cdot b \bmod N$
4. Consider a multiple precision implementation of the $\texttt{MMul}(\alpha, \beta)$ function with base $b = 2$. Write down all the steps needed to perform the multiplication performed in the previous steps.

Solution

1. We recall that a useful value for the Montgomery root should be $R = 2^n$, and it must hold that $R > N$ and $\gcd(R, N) = 1$. Thus , in this case , choosing $R = 2^5 = 32$ fits all the requirements

2. Through employing the Euclidean algorithm for the gcd computation, and recalling that $\gcd(R, N) = 1 = RR' - NN'$, we find the values of R' and N' to be: $R' = 10, \quad N' = 11, \quad N'_0 = 1 \bmod 2$

3. The required values can be computed as follows :

   - $\alpha = \mu(a) = aR \bmod N = 13 \cdot 32 \bmod 29 = 10 \bmod 29$
   - $\beta = \mu(b) = aR \bmod N = 17 \cdot 32 \bmod 29 = 22 \bmod 29$
   - $\gamma = \texttt{MMul}(\alpha, \beta) = \alpha \cdot \beta \cdot R' \bmod N = (10 \cdot 22 \cdot 10) \bmod 29 = 25 \bmod 29$
   - $c = \mu^{-1}(\gamma) = \gamma \cdot R' \bmod N = 28 \bmod 29$
   - $d = a \cdot b \bmod N = 13 \cdot 17 \bmod 29$

4. Referring to Algorithm 4.3, and recalling that $\alpha = 10_{\text{dec}} = 01010_2, \beta = 22_{\text{dec}} = 10110_2, N = 11101_2, N'_0 = 1$

00000	+
00000	$A_0B = 0 \cdot 10110$
00000	+
00000	$t\,N = (N'_0\gamma_0)N = 1 \cdot 0 \cdot 11101$
00000	$right\ shift$
00000	+
10110	$A_1B = 1 \cdot 10110$
10110	+
00000	$t\,N = (N'_0\gamma_0)N = 1 \cdot 0 \cdot 11101$
01011	$right\ shift$
01011	+
00000	$A_2B = 0 \cdot 10110$
01011	+
11101	$t\,N = (N'_0\gamma_0)N = 1 \cdot 1 \cdot 11101$
101000	$right\ shift$
10100	+
10110	$A_3B = 1 \cdot 10110$
101010	+
00000	$t\,N = (N'_0\gamma_0)N = 1 \cdot 0 \cdot 11101$
101010	$right\ shift$
10101	+
00000	$A_4B = 0 \cdot 10110$
10101	+
11101	$t\,N = (N'_0\gamma_0)N = 1 \cdot 0 \cdot 11101$
110010	$right\ shift$

$\gamma = 11001_2 = 25 < N \Rightarrow \gamma = 25 \bmod 29$

6 Optimizing Montgomery Multiplication

The Montgomery multiplication Algorithm (see either Algorithm 4.3 or Algorithm 4.4) can be further optimized through reserving extra storage for the factors and the intermediate variables. The key observation lies on assuming a zero digit as most significant digit of the first factor.
Indeed, if

$$R = b^d, A = \langle 0, A_{d-1}, A_{d-2}, \ldots, A_1, A_0 \rangle, B = \langle B_{d-1}, B_{d-2}, \ldots, B_1, B_0 \rangle$$

then $$\text{MMUL(A, B)} = A\,B\,R \bmod N = A\,(bB)\,(bR) \bmod N$$

The above relation allow to modify the `for`-loop in Algorithm 4.4 with an extra iteration (that is equivalent to think the Montgomery Radix equal to $R_1 = R \cdot b = b^{d+1}$ instead of R) and to simplify the computation of the parameter t because the least significant digit of $(b\,B)$ is zero. Thus, we can re-write Algorithm 4.4 as shown in Algorithm 6.1 and re-arrange the pseudo-code in a simpler manner as shown in Algorithm 6.2.

Algorithm 6.1: Optimized Montgomery multiplication Ver-1

Input: $A = \langle 0, A_{d-1}, \ldots, A_1, A_0 \rangle$, $B = \langle B_{d-1}, \ldots, B_1, B_0 \rangle$,
$N = \langle N_{d-1}, \ldots, N_1, N_0 \rangle$, $R = b^d$, $\gcd(R, N) = 1 = RR' - NN'$,
$N'_0 = (-N^{-1} \bmod R) \bmod b$
Output: $x = ABR^{-1} \bmod N$

```
1 begin
2     x ← 0
3     for i ← 0 to d do
4         t ← N'_0 x_0 mod b
5         x ← ((x + A_i(bB)) + tN)/b
6     if x ≥ N then
7         x ← x − N
8     return x
```

Algorithm 6.2: Optimized Montgomery multiplication Ver-2

Input: $A = \langle 0, A_{d-1}, \ldots, A_1, A_0 \rangle$,$B = \langle B_{d-1}, \ldots, B_1, B_0 \rangle$,
$N = \langle N_{d-1}, \ldots, N_1, N_0 \rangle$, $R = b^d$, $\gcd(R, N) = 1 = RR' - NN'$,
$N'_0 = (-N^{-1} \bmod R) \bmod b$
Output: $x = ABR^{-1} \bmod N$

```
1 begin
2     x ← 0
3     for i ← 0 to d do
4         if (N'_0 x_0 mod b ≠ 0) then
5             x ← x + tN
6         x ← x/b + A_iB // A right-shift plus ...
7     if x ≥ N then
8         x ← x − N
9     return x
```

The Montgomery multiplication algorithm can be effectively implemented through employing only addition and right-shift operations (i.e. without a Hardware multiplier), when it is applied to factors represented with the common binary encoding (Radix-2), thus assuming $R = 2^{\lceil \log_2 N \rceil}$.

7 Modular Exponentiations

Considering a modular exponentiation, where the modulus N is an odd integer (typical cases are the RSA algorithm, where it is obtained as the product of two prime numbers, and discrete log based schemes where it is a large prime) thus, we know in advance that the Montgomery multiplication primitive will have $N'_0 = -N_0^{-1} \bmod 2 = 1$.
Moreover, through

- reserving an extra digit (bit) for the storage of the employed variables (see Algorithm 6.2)
- and reserving a further couple of bits as stated in Remark 4.2 in order to postpone the final test (`if (`$x \geq N$`)`) as late as possible

we obtain the so called **EXP – Radix-2 Montgomery Multiplication**, EXP-R2-MMul$(\cdot,\cdot)$, as shown in Algorithm 7.1.

Algorithm 7.1: EXP – Radix-2 Montgomery Multiplication

Input: $A = \langle 0, 0, A_d, A_{d-1}, \ldots, A_1, A_0 \rangle$, $A_i \in \{0,1\}$,
$B = \langle 0, 0, B_d, B_{d-1}, \ldots, B_1, B_0 \rangle$, $B \in \{0,1\}$
$N = \langle N_{d-1}, \ldots, N_1, N_0 \rangle$, N odd, $R = 2^{d+2}$
Output: $x = ABR^{-1} \bmod N$ or $x = ABR^{-1} \bmod 2N$

```
1 begin
2     x ← 0
3     for i ← 0 to d + 2 do
4         if (x_0 == 1) then
5             x ← x + N
6         x ← x/2 + A_i B
7     return x
```

A cheap implementation (in terms of HW components) of a modular exponentiation may be realized as follows. Given $m^e \bmod N$, with $m \in \mathbb{Z}_N$ and $e = \langle e_{\lceil \log_2 N \rceil - 1}, \ldots, e_1, e_0 \rangle$ the computation is performed through:

- mapping the integer m in the Montgomery domain with $R = 2^{\lceil \log_2 N \rceil + 2}$,
- performing a repeated S&M strategy through employing the EXP-R2-MMul$(\cdot,\cdot)$ primitive,
- checking if the result x is greater than N and in such a case compute $x \leftarrow x - N$,
- finally, mapping the result back in the domain $\mathbb{Z}_n$