Montgomery Multiplication

An efficient algorithm for computing $R = a \cdot b \bmod n$ where a, b, and n are k-bit binary numbers, was introduced by P. L. Montgomery [5]. The algorithm is particularly suitable for implementation on general-purpose computers (signal processors or microprocessors) which are capable of performing fast arithmetic modulo a power of 2. The Montgomery reduction algorithm computes the resulting k-bit number R without performing a division by the modulus n. Via an ingenious representation of the residue class modulo n, this algorithm replaces division by n operation with division by a power of 2. This operation is easily accomplished on a computer since the numbers are represented in binary form. Assuming the modulus n is a k-bit number, i.e., $2^{k-1} \le n < 2^k$, let r be 2^k. The Montgomery reduction algorithm requires that r and n be relatively prime, i.e., $\gcd(r, n) = \gcd(2^k, n) = 1$. This requirement is satisfied if n is odd. In the following we summarize the basic idea behind the Montgomery reduction algorithm.

Given an integer $a < n$, we define its n-residue with respect to r as

$$\bar{a} = a \cdot r \bmod n \ .$$

It is straightforward to show that the set

$$\{ \ i \cdot r \bmod n \mid 0 \le i \le n-1 \ \}$$

is a complete residue system, i.e., it contains all numbers between 0 and $n-1$. Also there is a one-to-one correspondence between the numbers in the range 0 and $n-1$ and the numbers in the above set. The Montgomery reduction algorithm exploits this property by introducing a much faster multiplication routine which computes the n-residue of the product of the two integers whose n-residues are given. Given two n-residues $\bar{a}$ and $\bar{b}$, the *Montgomery product* is defined as the n-residue

$$\bar{R} = \bar{a} \cdot \bar{b} \cdot r^{-1} \bmod n$$

where r^{-1} is the inverse of r modulo n, i.e., it is the number with the property

$$r^{-1} \cdot r = 1 \bmod n \ .$$

The resulting number $\bar{R}$ is indeed the n-residue of the product

$$R = a \cdot b \bmod n$$

since

$$\begin{aligned} \bar{R} &= \bar{a} \cdot \bar{b} \cdot r^{-1} \bmod n \\ &= a \cdot r \cdot b \cdot r \cdot r^{-1} \bmod n \\ &= a \cdot b \cdot r \bmod n \ . \end{aligned}$$

In order to describe the Montgomery reduction algorithm, we need an additional quantity, n', which is the integer with the property

$$r \cdot r^{-1} - n \cdot n' = 1 \ .$$

The integers r^{-1} and n' can both be computed by the extended Euclid algorithm. The Montgomery product computation is given below:

```
**function** MonPro($\bar{a}, \bar{b}$)
Step 1. $t := \bar{a} \cdot \bar{b} \bmod r$
Step 2. $m := t \cdot n' \bmod r$
Step 3. $u := (\bar{a} \cdot \bar{b} + m \cdot n)/r$
Step 4. **if** $u \geq n$ **then return** $u - n$ **else return** $u$
```

The most important feature of the Montgomery product algorithm is that the operations involved are multiplications modulo r and divisions by r, both of which are intrinsically fast operations since r is a power 2. The MonPro algorithm can be used to compute the product of a and b modulo n provided n is odd. The algorithm given below achieves this purpose:

```
**function** ModMul($a, b, n$) { $n$ is an odd number }
Step 1. Compute $n'$ using the extended Euclid algorithm.
Step 2. $\bar{a} := a \cdot r \bmod n$
Step 3. $\bar{b} := b \cdot r \bmod n$
Step 4. $\bar{x} :=$ MonPro($\bar{a}, \bar{b}$)
Step 5. $x :=$ MonPro($\bar{x}, 1$)
Step 6. **return** $x$
```

Since the preprocessing operations (conversion from ordinary residue to n-residue, computation of n', and converting the result back to ordinary residue) are rather time-consuming, it is not a good idea to use the Montgomery product computation algorithm when a single modular multiplication is to be performed.

0.1 Montgomery Exponentiation

The Montgomery product algorithm is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute *modular exponentiation*, i.e., the computation of $a^e \bmod n$. Using one of the addition chain algorithms given in Chapter 2, we replace the exponentiation operation by a series of square and multiplication operations modulo n. This is where the Montgomery product operation finds its best use. In the following we summarize the modular exponentiation operation which makes use of the Montgomery product function MonPro. The exponentiation algorithm uses the binary method.

```
function ModExp($a, e, n$) { $n$ is an odd number }
Step 1. Compute $n'$ using the extended Euclid algorithm.
Step 2. $\bar{a} := a \cdot r \bmod n$
Step 3. $\bar{x} := 1 \cdot r \bmod n$
Step 4. for $i = k - 1$ down to 0 do
Step 5.         $\bar{x} :=$ MonPro($\bar{x}, \bar{x}$)
Step 6.         if $e_i = 1$ then $\bar{x} :=$ MonPro($\bar{a}, \bar{x}$)
Step 7. $x :=$ MonPro($\bar{x}, 1$)
Step 8. return $x$
```

Thus, we start with the ordinary residue a and obtain its n-residue $\bar{a}$ using a division-like operation, which can be achieved, for example, by a series of shift and subtract operations. Additionally, Steps 2 and 3 require divisions. However, once the preprocessing has been completed, the inner-loop of the binary exponentiation method uses the Montgomery product operations which performs only multiplications modulo 2^k and divisions by 2^k. When the binary method finishes, we obtain the n-residue $\bar{x}$ of the quantity $x = a^e \bmod n$. The ordinary residue number is obtained from the n-residue by executing the MonPro function with arguments $\bar{x}$ and 1. This easily shown to be correct since

$$\bar{x} = x \cdot r \bmod n$$

immediately implies that

$$x = \bar{x} \cdot r^{-1} \bmod n \; = \; \bar{x} \cdot 1 \cdot r^{-1} \bmod n \; := \; \text{MonPro}(\bar{x}, 1) \; .$$

The resulting algorithm is quite fast as was demonstrated by many researchers and engineers who have implemented it. However, the above algorithm can be refined and made more efficient, particularly when the numbers involved are multi-precision integers.

0.2 An Example of Exponentiation

Here we show how to compute $x = 7^{10} \bmod 13$ using the Montgomery exponentiation algorithm.

- Since $n = 13$, we take $r = 2^4 = 16 > n$.
- Computation of n':

  Using the extended Euclid algorithm, we determine that $16 \cdot 9 - 13 \cdot 11 = 1$, thus, $r^{-1} = 9$ and $n' = 11$.
- Computation of $\bar{M}$:

  Since $M = 7$, we have $\bar{M} := M \cdot r \pmod{n} = 7 \cdot 16 \pmod{13} = 8$.

- Computation of $\bar{x}$ for $x = 1$:

  We have $\bar{x} := x \cdot r \pmod{n} = 1 \cdot 16 \pmod{13} = 3$.

- Steps 5 and 6 of the ModExp routine:

e_i	Step 5	Step 6
1	$\text{MonPro}(3,3) = 3$	$\text{MonPro}(8,3) = 8$
0	$\text{MonPro}(8,8) = 4$	
1	$\text{MonPro}(4,4) = 1$	$\text{MonPro}(8,1) = 7$
0	$\text{MonPro}(7,7) = 12$	

  ◦ Computation of $\text{MonPro}(3,3) = 3$:
  $t := 3 \cdot 3 = 9$
  $m := 9 \cdot 11 \pmod{16} = 3$
  $u := (9 + 3 \cdot 13)/16 = 48/16 = 3$

  ◦ Computation of $\text{MonPro}(8,3) = 8$:
  $t := 8 \cdot 3 = 24$
  $m := 24 \cdot 11 \pmod{16} = 8$
  $u := (24 + 8 \cdot 13)/16 = 128/16 = 8$

  ◦ Computation of $\text{MonPro}(8,8) = 4$:
  $t := 8 \cdot 8 = 64$
  $m := 64 \cdot 11 \pmod{16} = 0$
  $u := (64 + 0 \cdot 13)/16 = 64/16 = 4$

  ◦ Computation of $\text{MonPro}(4,4) = 1$:
  $t := 4 \cdot 4 = 16$
  $m := 16 \cdot 11 \pmod{16} = 0$
  $u := (16 + 0 \cdot 13)/16 = 16/16 = 1$

  ◦ Computation of $\text{MonPro}(8,1) = 7$:
  $t := 8 \cdot 1 = 8$
  $m := 8 \cdot 11 \pmod{16} = 8$
  $u := (8 + 8 \cdot 13)/16 = 112/16 = 7$

  ◦ Computation of $\text{MonPro}(7,7) = 12$:
  $t := 7 \cdot 7 = 49$
  $m := 49 \cdot 11 \pmod{16} = 11$
  $u := (49 + 11 \cdot 13)/16 = 192/16 = 12$

- Step 7 of the ModExp routine: $x = \text{MonPro}(12,1) = 4$
  $t := 12 \cdot 1 = 12$
  $m := 12 \cdot 11 \pmod{16} = 4$
  $u := (12 + 4 \cdot 13)/16 = 64/16 = 4$

Thus, we obtain $x = 4$ as the result of the operation $7^{10} \bmod 13$.

1 The Case of Even Modulus

Since the existence of r^{-1} and n' requires that n and r be relatively prime, we cannot use the Montgomery product algorithm when this rule is not satisfied. We take $r = 2^k$ since arithmetic operations are based on binary arithmetic modulo 2^w where w is the word-size of the computer. In case of single-precision integers, we take $k = w$. However, when the numbers are large, we choose k to be an integer multiple of w. Since $r = 2^k$, the Montgomery modular exponentiation algorithm requires that

$$\gcd(r, n) = \gcd(2^k, n) = 1$$

which is satisfied if and only if n is odd. We now describe a simple technique [2] which can be used whenever one needs to compute modular exponentiation with respect to an even modulus. Let n be factored such that

$$n = q \cdot 2^j$$

where q is an odd integer. This can easily be accomplished by shifting the even number n to the right until its least-significant bit becomes one. Then, by the application of the Chinese remainder theorem, the computation of

$$x = a^e \bmod n$$

is broken into two independent parts such that

$$\begin{aligned} x_1 &= a^e \bmod q \ , \\ x_2 &= a^e \bmod 2^j \ . \end{aligned}$$

The final result x has the property

$$\begin{aligned} x &= x_1 \bmod q \ , \\ x &= x_2 \bmod 2^j \ , \end{aligned}$$

and can be found using one of the Chinese remainder algorithms: The single-radix conversion algorithm or the mixed-radix conversion algorithm [6, 1, 4]. The computation of x_1 can be performed using the ModExp algorithm since q is odd. Meanwhile the computation of x_2 can be performed even more easily since it involves arithmetic modulo 2^j. There is however some overhead involved due to the introduction of the Chinese remainder theorem. According to the mixed-radix conversion algorithm, the number whose residues are x_1 and x_2 modulo q and 2^j, respectively, is equal to

$$x = x_1 + q \cdot y$$

where

$$y = (x_2 - x_1) \cdot q^{-1} \bmod 2^j \ .$$

The inverse $q^{-1} \bmod 2^j$ exists since q is odd. It can be computed using a simplified Euclidean algorithm. We thus have the following algorithm:

```
function EvenModExp($a, e, n$) { $n$ is an even number }
1. Shift $n$ to the right obtain the factorization $n = q \cdot 2^j$.
2. Compute $x_1 := a^e \bmod q$ using ModExp routine above.
3. Compute $x_2 := a^e \bmod 2^j$ using the binary method and modulo $2^j$ arithmetic.
4. Compute $q^{-1} \bmod 2^j$ and $y := (x_2 - x_1) \cdot q^{-1} \bmod 2^j$.
5. Compute $x := x_1 + q \cdot y$ and return $x$.
```

1.1 An Example of Even Modulus Case

The computation of $a^e \bmod n$ for $a = 375$, $e = 249$, and $n = 388$ is illustrated below.

Step 1. $n = 388 = (110000100)_2 = (11000001)_2 \times 2^2 = 97 \times 2^2$. Thus, $q = 97$ and $j = 2$.

Step 2. Compute $x_1 = a^e \bmod q$ by calling ModExp with parameters $a = 375$, $e = 249$, and $q = 97$. We must remark, however, that we can reduce a and e modulo q and $\phi(q)$, respectively. The latter is possible if we know the factorization of q. Such knowledge is not necessary but would further decrease the computation time of the ModExp routine. Assuming we do not know the factorization of q, we only reduce a to obtain

$$a \bmod q \quad = 375 \bmod 97 \quad = 84$$

and call the ModExp routine with parameters $(84, 249, 97)$. Since q is odd, the ModExp routine successfully computes the result as $x_1 = 78$.

Step 3. Compute $x_2 = a^e \bmod 2^j$ by calling an exponentiation routine based on the binary method and modulo 2^j arithmetic. Before calling such routine we should reduce the parameters as

$$\begin{array}{lll} a \bmod 2^j & = 375 \bmod 4 & = 3 \\ e \bmod \phi(2^j) & = 249 \bmod 2 & = 1 \end{array}$$

In this case, we are able to reduce the exponent since we know that $\phi(2^j) = 2^{j-1}$. Thus, we call the exponentiation routine with the parameters $(3, 1, 4)$. The routine computes the result as $x_2 = 3$.

Step 4. Using the extended Euclidean algorithm, compute

$$q^{-1} \bmod 2^j = 97^{-1} \bmod 4 = 1 \ .$$

Now compute

$$\begin{array}{rcl} y & = & (x_2 - x_1) \cdot q^{-1} \bmod 2^j \\ & = & (3 - 78) \cdot 1 \bmod 4 \\ & = & 1 \ . \end{array}$$

Step 5. Compute and return the final result

$$x = x_1 + q \cdot y = 78 + 97 \cdot 1 = 175 \ .$$

References

[1] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Second edition, 1981.

[2] Ç. K. Koç. Montgomery reduction with even modulus. *IEE Proceedings - Computers and Digital Techniques*, 141(5):314–316, September 1994.

[3] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[4] J. D. Lipson. *Elements of Algebra and Algebraic Computing.* Addison-Wesley, 1981.

[5] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[6] N. S. Szabo and R. I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology.* McGraw-Hill, 1967.