Implementation Of Montgomery Reduction Algorithm For Efficient Modular Arithmetic Operations

Shubh Panchal
21bec080@nirmauni.ac.in

Department of Electronics & Communication, Institute Of Technology, Nirma University, Ahmedabad

Dron Patel
21bec089@nirmauni.ac.in

Department of Electronics & Communication, Institute Of Technology, Nirma University, Ahmedabad

Abstract— If the modulus is an odd integer, the Montgomery reduction algorithm provides an effective way to carry out modular arithmetic operations. This is especially useful when working with big moduli with low Hamming weight. Using Montgomery reduction and the Montgomery multiplication method, this algorithm makes it easier to multiply huge integers. Montgomery multiplication is used in conjunction with conversion to Montgomery representation to facilitate computation.
The benefits of using the Montgomery reduction algorithm over other methods, including Barrett reduction, are emphasised. Notwithstanding its shortcomings, the algorithm is valued for its ease of use, quickness, and adaptability, which make it a top option for many different kinds of applications.

Keywords— Montgomery Reduction Algorithm, Montgomery Multiplication Algorithm, Modular arithmetic, Montgomery Representation, Barrett Reduction Algorithm

INTRODUCTION

In many cryptographic applications, including digital signatures, public key cryptography, RSA cryptography, Elliptic Curve Cryptography (ECC), and key exchange protocols, modular arithmetic is the most often used and significant mathematical operation. Numerous applications include computer algebra systems, error-correcting codes, and number theory. Arithmetic operations modulo a composite integer or a prime number are needed in these applications. The Montgomery Reduction Algorithm was created because traditional modulo procedures can be computationally costly, particularly when dealing with bigger numbers.
Peter Montgomery developed the Montgomery Reduction Algorithm in 1985 to address this issue. A large integer modulo is effectively reduced into a smaller integer using the Montgomery reduction procedure, which makes use of a special representation known as the Montgomery representation. Especially for applications involving cryptography, it provides a It is especially useful for speedier modular arithmetic operations in cryptographic applications. The speed of modular multiplication in particular, as well as other modular arithmetic operations, is critical to the effectiveness of various cryptographic methods. Modular multiplication processes can be accelerated by the Montgomery reduction technique, which accelerates the cryptographic protocol as a whole. Elliptic Curve Cryptography proved to be a particularly beneficial application of the methods. One important function in elliptic curve cryptography is scalar multiplication, which is carried out on elliptic curves using it. Compared to traditional modular reduction procedures, it has several benefits, such as increased speed, simplicity, and flexibility. The Montgomery reduction algorithm functions based on the Montgomery product.
To do modular arithmetic, fewer modular reductions are needed thanks to this particular multiplication operation. The algorithm operates

by first putting both operands into Montgomery form, then on the result, it applies the Montgomery product operation and returns the result to normal form. One of the main benefits of the Montgomery reduction method is its efficiency. Particularly when working with bigger numbers, the Montgomery reduction method outperforms conventional modular reduction algorithms in terms of speed. The overall efficiency is due to the Montgomery product operation, which lowers the number of modular reductions needed to accomplish modular arithmetic.

The approach lowers the total number of operations required and thereby improves performance by substituting a sequence of less expensive modular multiplications for the pricey modular division operation. High modulus and low Hamming weight are desired parameters for the Montgomery reduction technique because they can improve its efficiency and minimal computational cost. The amount of bits in the binary form of the modulus that is nonzero is its hamming weight.

Using the Montgomery reduction procedure, given two integers, a and b, and a modulus m, the value r is calculated so that:

r = (a * b * R^-1) mod m

where R is a power of 2 that is greater than or equal to m, and R^-1 is the inverse of R modulo m.

The Montgomery reduction algorithm can be implemented using the following formula:

t = a * b + u * m

r = t * R^-1 mod m

where u is a value such that u * R is congruent to 1 modulo m. This value can be precomputed using the extended Euclidean algorithm.

Modular exponentiation operations in cryptographic algorithms can be effectively carried out with the aid of the Montgomery reduction technique. For instance, we can compute x^e * R mod m after converting m into its Montgomery form to get x^e mod m. The final result can then be obtained by reducing the result modulo m using the Montgomery reduction procedure. This paper presents an in-depth explanation of the Montgomery reduction algorithm and its application. The processes necessary to convert integers to Montgomery form, perform the reduction operation, and then return to normal form are described in detail in the paper. In the publication, the algorithm's performance is assessed.

An essential tool for creating effective and safe computer algebra and cryptography systems is the Montgomery reduction algorithm. Its extensive use in academics and business is proof of its strength and adaptability in contemporary cryptography systems.

MONTGOMERY FORM

To expedite the computation of big integers in modular arithmetic, the Montgomery form is employed. Exponentiation, inversion, and modular multiplication are the specialized applications for it. Converting numbers into a unique "reduced" form that enables the more effective use of modular operations is the aim of Montgomery reduction. The operands are multiplied by a carefully chosen factor to bring the result identical to the modulus after the modulus is changed to a higher value which is a power of two in the Montgomery form. There are several advantages when comparing the Montgomery form to traditional modular arithmetic. Its elimination of the costly division operation required in traditional modular arithmetic is a key benefit. All it uses are simple addition and shift operations, which are substantially faster. A further advantage is that it allows modular powers to be computed rapidly through the use of "exponentiation by squaring."

MONTGOMERY MULTIPLICATION ALGORITHM

Montgomery multiplication is a fast modular multiplication method that was created by Peter L. Montgomery in 1985. It is effective for multiplying two big numbers modulo a third number. The primary advantage of the Montgomery multiplication algorithm over other modular multiplication algorithms is its lack of division operation, which is a laborious task on most computer architectures. Three phases make up the Montgomery multiplication algorithm: initialization, multiplication, and reduction. In the initialization step, precomputation is carried out to modify the operands' form and facilitate quicker computation. In the multiplication phase, a

sequence of modular operations is used to multiply the operands.
The result is changed back into the desired form during the reduction stage. In cryptographic applications involving large numbers, like public-key cryptography algorithms like RSA and elliptic curve cryptography, the Montgomery multiplication algorithm is frequently used.

The Montgomery multiplication formula is used to get the product (A*B) mod N given three integers: A, B, and N, where N is a positive odd integer. The first step is to pre calculate a value R such that gcd (R, N) = 1 and that R is a power of two greater than N. A measurement of the parameter R is called the "Montgomery modulus". We can first put A and B into their "Montgomery form" and then utilize the Montgomery multiplication formula by multiplying A and B by R and getting the result modulo N. This provides us with two integers, A' and B', between 0 and R1. After calculating the product, a sequence of modular additions and multiplications is utilized to restore C to its initial form modulo N.
C = A'*B'.

APPLICATION OF MONTGOMERY MULTIPLICATION ALGORITHM USING MONTGOMERY REDUCTION ALGORITHM IN CRYPTOGRAPHY

There are numerous uses for the Montgomery multiplication algorithm in cryptography, which employs Montgomery reduction. One of its primary applications is in modular exponentiation, a technique found in public key cryptography algorithms such as RSA and Diffie-Hellman. An exponent e and a modulus n make up the public key for RSA encryption. Encrypting a message involves raising its plaintext value m to the power of e modulo n. Modular exponentiation can be efficiently performed with the Montgomery multiplication algorithm by first transforming the plaintext and exponent into Montgomery form using Montgomery reduction, and then using the standard square-and-multiply procedure. Similar to this, Alice and Bob decide on a prime modulus of p and a base of g, and then they each select a secret key in the Diffie-Hellman key exchange procedure.
After that, they trade public keys, which are produced by raising the g-power modulo p of each secret key. The shared secret key is obtained by raising the received public key to the power of their own secret key modulo p. Once more, the Montgomery multiplication technique can be used to accomplish modular exponentiation efficiently. Elliptic curve cryptography is another popular public key cryptography algorithm that uses the Montgomery multiplication algorithm. Calar multiplication of points on an elliptic curve in this case is carried out by the Montgomery ladder algorithm. All things considered, a key tool in many cryptographic applications, the Montgomery multiplication technique utilizing Montgomery reduction provides a rapid and efficient means to perform modular arithmetic operations in encryption.

BARRET REDUCTION ALGORITHM

Barrett reduction is an additional modular reduction algorithm, akin to Montgomery reduction. It functions according to the same idea of calculating a value ahead of time and using it to speed up modular reduction. Using a different precalculated value, the Barrett reduction process can be thought of as a variation of the Montgomery reduction technique. In particular, the Barrett reduction uses a number mu = floor((2^2k)/m), where m is the modulus, but the Montgomery reduction uses R = 2^k, where k denotes the bit length of the modulus. The following is the primary distinction between the Montgomery reduction algorithm and the Barrett reduction algorithm:

1. Managing Larger Numbers: Montgomery Reduction is effective for handling huge numbers since it optimises arithmetic operations by converting numbers to Montgomery form. Barrett Reduction computes modulo directly; for very large moduli, this method may become less effective.
2. Supercharged Exponentiations: Montgomery Reduction is perfect for modular exponentiation because of its unique multiplication technique, which minimises the number of operations. The regular multiplication of Barrett Reduction is effective, but it is not enhanced by Montgomery Reduction's optimisations.
3. Memory Usage: Barrett Reduction simplifies memory management by requiring less memory for precomputed values than Montgomery

Reduction does.

4. Adaptability to Changing Moduli: Barrett Reduction is more appropriate for fixed moduli, whereas Montgomery Reduction adjusts well to fluctuating moduli because of precomputed values.

5. Implementation Complexity: Handling Montgomery form and precomputed values makes Montgomery Reduction more difficult to accomplish. Barrett Reduction requires only elementary modular arithmetic, making it easier to apply.

6. Theoretical Foundations: Fermat's Little Theorem and other solid theoretical foundations provide the basis of Montgomery Reduction. Barrett Reduction is less dependent on theoretical ideas and more on real-world optimisations.

ADVANTAGES AND LIMITATIONS

- **Advantages:**

  1. Effectiveness with Big Numbers: Montgomery Reduction is particularly good at handling large integer values, which is important for cryptographic operations involving a lot of numerical input.

  2. Optimised Arithmetic Operations: This method reduces computing overhead and boosts operational efficiency by streamlining arithmetic operations by converting integers into Montgomery form.

  3. Better for Modular Exponentiation: Montgomery Reduction, which uses a specific multiplication method, works incredibly well for modular exponentiation jobs. It makes it easier to compute exponential operations quickly and accurately in cryptographic contexts.

  4. Streamlined Resource Utilisation: Montgomery Reduction's careful memory management guarantees optimal resource allocation, promoting efficient utilisation in memory-constrained contexts, even though it requires additional memory for precomputed values.

  5. Dynamic Moduli Handling: Montgomery Reduction demonstrates adaptability by skilfully handling situations involving dynamic modulus values. This allows for stable performance levels even in the face of fluctuations, which is very useful in flexible cryptography systems.

  6. Mathematical Rigour and Reliability: Montgomery Reduction's foundational mathematical ideas provide assurance for its accuracy and dependability, which supports the credibility of cryptographic solutions created with this method.

- **Limitations:**

  1. Initial Setup Overhead: Before efficient computation can start, Montgomery Reduction necessitates precomputing values such as Montgomery's R-squared value. This results in an initial setup overhead. This initial investment might not be negligible, particularly for small or seldom utilised moduli.

  2. Complexity of Implementation: When compared to less complex modular reduction approaches, the Montgomery Reduction might be more intricate to implement. Managing precomputed values and Montgomery form transformations increases implementation complexity, which could result in longer development times and a higher risk of errors.

  3. Limited Applicability to Fixed Moduli: Although Montgomery Reduction performs exceptionally well in situations involving huge numbers or dynamic moduli, its benefits are reduced when working with fixed moduli, particularly when the modulus is tiny. In some situations, the precomputed value overhead can offset the computing advantages.

  5. Possible Trade-off with Memory Usage: The Montgomery Reduction's memory requirements, especially for storing precomputed values, could result in a trade-off between memory usage and computational performance. This trade-off becomes important to take into account in situations where memory resources are limited or need optimisation.

  6. Optimisation for Particular

Processes: Montgomery Reduction works quite well for modular exponentiation, but not all cryptographic processes will profit equally from it. Montgomery Reduction's general applicability may be limited by the lack of optimization experienced by other operations, such as modular multiplication or inversion.

FLOWCHART

Start

- Step 1: Initialization
  - Compute R = b^n, where b is the base and n is the number of bits in the modulus
  - Compute R^-1 mod m, where m is the modulus
  - Set A = a * R mod m and B = b * R mod m
- Step 2: Multiplication
  - Set C = 0
  - Repeat n times:
  - Compute q_i = (C + (A mod b) * (B mod b) * R^-1) mod b
  - Compute C = (C + A * (B mod b) * R^-1 + q_i * m) / b
  - Compute A = floor(A / b) and B = floor(B / b)
- Step 3: Reduction
  - If C >= m, then C = C - m
  - Set C = C * R^-1 mod m
  - Output the result

End

EXPLANATION

- Step 1: Initialization

In this step, we will define x1, x2 & m and then compute the value of R, which is r1 and r2. The value of R is computed as R = b^n, where n is the number of bits in the modulus. Then, we compute r1 as mont.reduce(t1) and r2 as mont.reduce(t2). Finally, then we compute R as 1<< n. Hence now we can say that R is reduced to the form of T. Here mont.n is the number of digits in binary form of n.

- Step 2: Method Declaration

In this step, we define the reduce method(.rrm) and Montgomery class.

- For computing rrm we will define m as self.m and the bit length of m as n. Then rrm(m) as calculated as 1 times left shift of (2*n) and then modulo to m.
- Now we will be defining reduce method Here in a for loop we will perform bitwise and operation of t which is stored in a; and update the value of a by a+self.m . Here we will also have one if statement to keep a in the range of self.m
- Finally, we update the values of prod and base as mont.reduce(mont.rrm) and mont.reduce(x1 * mont.rrm) respectively. We will store x2 in x variable

- Step 3: Exponential and Result verification.

In this step,we check if the value of exp.bit_length() is greater than 0. And run a while loop also perform a bitwise and operation exp and 1 and keep updating the value of product and exp and base.

The key idea behind the Montgomery reduction algorithm is that by choosing the constant R carefully, the multiplication by R can be used to effectively "shift" the number into a higher bit position, making the reduction by m easier and faster to compute. In Montgomery arithmetic, r represents the base used in calculations and is often referred to as the "Montgomery modulus." It is crucial for Montgomery multiplication and exponentiation operations. So we need to carefully calculate R.

MONTOMERY ALGORITHM

```
#User Inputs

class Montgomery:
   BASE = 2

# Calculating rrm
   def __init__(self, m):
      self.m = m
      self.n = m.bit_length()
      self.rrm = (1 << (self.n * 2)) % m
# .rrm is = 1 is left shifted by 2n then modulo m
# hence .rrm = (1* 2 to power 2n)  % m

# Reduce method
   def reduce(self, t):
      a = t
      for i in range(self.n):
         if (a & 1) == 1:
            a = a + self.m
         a = a >> 1
      if a >= self.m:
         a = a - self.m
      return a

# These are the modulus m and two operands x1 and
x2 for the modular exponentiation operation
((x1^x2) % m)
m =  750791094644726559640638407699
x1 = 540019781128412936473322405310
x2 = 515692107665463680305819378593

# Instantiating Montgomery and Performing
Reduction:
mont = Montgomery(m)
t1 = x1 * mont.rrm
t2 = x2 * mont.rrm


# r1 & r2 are reduced form of t1 & t2.
r1 = mont.reduce(t1)
r2 = mont.reduce(t2)
r = 1 << mont.n
#mont.n is the number of bits in binary form of m

#Printing Information:
print ("b : ", Montgomery.BASE)
print ("n : ", mont.n)
print ("r : ", r)
print ("m : ", mont.m)
print("mont.rrm:",mont.rrm);
print ("t1: ", t1)
print ("t2: ", t2)
print ("r1: ", r1)
print ("r2: ", r2)
print("")
print ("Original x1      :", x1)
print ("Recovered from r1 :", mont.reduce(r1))
print ("Original x2      :", x2)
print ("Recovered from r2 :", mont.reduce(r2))

print ("\nMontgomery computation of x1 ^ x2 mod
m:")
print("")
prod = mont.reduce(mont.rrm)
base = mont.reduce(x1 * mont.rrm)
exp = x2
# Checking initialized values
print("precomputed-prod",prod)
print("precomputed-base",base)
print("precomputed-exp",exp)

# Exponentiation and Result Verification:
while exp.bit_length() > 0:
   if (exp & 1) == 1:
      prod = mont.reduce(prod * base)
   exp = exp >> 1
   base = mont.reduce(base * base)
print                              ("\nMontgomery
exponentiation:",mont.reduce(prod))

# Alternate method for computation of x1 ^ x2 mod
m :
print ("\nAlternate computation of x1 ^ x2 mod m
:",pow(x1, x2, m))
```

BARRET ALGORITHM

```
class Barrett:
   BASE = 2

   def __init__(self, m):
      self.m = m
      self.n = m.bit_length()
      # Calculate Barrett constant
      self.k = (1 << (2 * self.n)) // m

   # Reduction method
   def reduce(self, t):
      q = (t * self.k) >> (2 * self.n)
      a = t - q * self.m
      if a < 0:
         a += self.m
      return a

# Modulus m and two operands x1 and x2 for
```

```
modular exponentiation operation ((x1^x2) % m)
m = 75079109464472655964063840769 9
x1 = 54001978112841293647332240531 0
x2 = 51569210766546368030581937859 3

# Instantiating Barrett Reduction and Performing Reduction:
barrett = Barrett(m)
t1 = x1 * Barrett.BASE
t2 = x2 * Barrett.BASE

r1 = barrett.reduce(t1)
r2 = barrett.reduce(t2)
r = 1 << barrett.n

# Printing Information:
print("b:", Barrett.BASE)
print("n:", barrett.n)
print("m:", barrett.m)
print("t1:", t1)
print("t2:", t2)
print("r1:", r1)
print("r2:", r2)
print("")
print("Original x1      :", x1)
print("Recovered from r1 :", barrett.reduce(r1))
print("Original x2      :", x2)
print("Recovered from r2 :", barrett.reduce(r2))

# Checking Barrett constant
print("k:", barrett.k)

# Exponentiation and Result Verification:
prod = barrett.reduce(Barrett.BASE)
base = barrett.reduce(x1 * Barrett.BASE)
exp = x2

while exp.bit_length() > 0:
    if (exp & 1) == 1:
        prod = barrett.reduce(prod * base)
    exp >>= 1
    base = barrett.reduce(base * base)

print("\nBarrett exponentiation:", prod)
```

PROGRAMMING PROCEDURE

Montgomery reduction algorithm is implemented using python programming language. The code will define a Montgomery reducer class and a test case to check it's correctness. The Montgomery Reducer class takes a modulus as input and provides methods to convert numbers to and from the Montgomery domain, multiply numbers in the Montgomery domain, and compute modular exponentiation using the Montgomery reduction.
Montgomery Reducer class by generating random inputs and comparing the output of the Montgomery reduction with the output of the standard modular arithmetic. The test generates random bit lengths for the modulus between 2 and 100, generates 100 random pairs of numbers in the range [0, modulus] and multiplies them using the Montgomery Reducer. Multiply method, and checks that the result in the Montgomery domain is equivalent to the standard modular multiplication of the input numbers. Then, the test generates 10 random pairs of numbers in the range [0, modulus) and computes the modular exponentiation using the Montgomery Reducer. Pow method, and checks that the result in the Montgomery domain is equivalent to the standard modular exponentiation of the input numbers.

The Montgomery Reducer class constructor takes a modulus as input and initializes some variables that are used in the Montgomery reduction. The reducer bits variable is the number of bits needed to represent the Montgomery reduction factor, which is a power of 256 that is greater than the modulus. The reducer variable is the Montgomery reduction factor itself, which is a power of 256 that is greater than the modulus. The mask variable is the bitmask used to extract the lower bits of the product of two numbers in the Montgomery domain. The reciprocal variable is the modular inverse of the Montgomery reduction factor with respect to the modulus. The factor variable is the value (reducer * reciprocal - 1), modulus, which is used to reduce the product of two numbers in the Montgomery domain to a number in the Montgomery domain.

The convert in method takes a number x in the standard domain and converts it to the Montgomery domain by multiplying it by 2^reducerbits and reducing the result modulo the modulus. The convert out method takes a number

x in the Montgomery domain and converts it to the standard domain by multiplying it by the modular inverse of 2^reducerbits modulo the modulus. The multiply method takes two numbers x and y in the Montgomery domain, multiplies them using the Montgomery reduction algorithm, and returns the result in the Montgomery domain. The pow method takes a number x in the Montgomery domain and an exponent y in the standard domain, computes the modular exponentiation x^y using the Montgomery reduction algorithm, and returns the result in the Montgomery domain.

Conclusion

In summary, while Montgomery Reduction offers advantages such as optimized arithmetic operations and adaptability to changing moduli, it comes with additional memory overhead and implementation complexity. On the other hand, Barrett Reduction is simpler to implement and doesn't require precomputed values, making it suitable for fixed moduli and memory-constrained environments. Both methods have their strengths and weaknesses and the choice between them depends on the specific requirements and constraints.

We have also written the JAVA and C++ code for the Montomery method ,but we chose writing our final code in Python, because in python the biggest integer that one can use virtually unbounded and in C++, JAVA that limits upto (2^64 -1).

In our python program we have optimized it to calculate the computational cost and proved that Montogomery reduces time complexity in the program itself.

References

[1] "Handbook of Applied Cryptography" by Alfred Menezes, Paul van Oorschot, and Scott Vanstone: This book is a comprehensive reference on cryptography and includes a chapter on modular arithmetic and the Montgomery reduction algorithm.

[2] "Cryptography Engineering: Design Principles and Practical Applications" by Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno: This book is a practical guide to cryptography and includes a chapter on modular arithmetic and the Montgomery reduction algorithm.

[3] www.navuki.io

[4] "A Fast Montgomery Multiplication Algorithm for Cryptographic Applications" by A. Menezes and M. Vanstone.