COME ON CODE ON

A blog about programming and more programming.

Modular Multiplicative Inverse

with 29 comments

The modular multiplicative inverse of an integer a modulo m is an integer x such that $a^{-1} \equiv x \pmod{m}$.

That is, it is the multiplicative inverse in the ring of integers modulo m. This is equivalent to $ax \equiv aa^{-1} \equiv 1 \pmod{m}$.

The multiplicative inverse of a modulo m exists if and only if a and m are coprime (i.e., if gcd(a, m) = 1).

Let’s see various ways to calculate Modular Multiplicative Inverse:

1. Brute Force
We can calculate the inverse using a brute force approach where we multiply *a* with all possible values *x* and find a *x* such that $ax \equiv 1 \pmod{m}$. Here’s a sample C++ code:

```
int modInverse(int a, int m) {
    a %= m;
    for(int x = 1; x < m; x++) {
        if((a*x) % m == 1) return x;
    }
}
```

The time complexity of the above codes is O(m).

2. Using Extended Euclidean Algorithm
We have to find a number x such that a·x = 1 (mod m). This can be written as well as a·x = 1 + m·y, which rearranges into a·x – m·y = 1. Since x and y need not be positive, we can write it as well in the standard form, a·x + m·y = 1.

In number theory, Bézout’s identity for two integers a, b is an expression ax + by = d, where x and y are integers (called Bézout coefficients for (a,b)), such that d is a common divisor of a and b. If d is the greatest common divisor of a and b then Bézout’s identity ax + by = gcd(a,b) can be solved using Extended Euclidean Algorithm.

The Extended Euclidean Algorithm is an extension to the Euclidean algorithm. Besides finding the greatest common divisor of integers a and b, as the Euclidean algorithm does, it also finds integers x and y (one of which is typically negative) that satisfy Bézout’s identity ax + by = gcd(a,b). The Extended Euclidean Algorithm is particularly useful when a and b are coprime, since x is the multiplicative inverse of a modulo b, and y is the multiplicative inverse of b modulo a.

We will look at two ways to find the result of Extended Euclidean Algorithm.

Iterative Method

This method computes expressions of the form $r_i = ax_i + by_i$ for the remainder in each step i of the Euclidean algorithm. Each successive number r_i can be written as the remainder of the division of the previous two such numbers, which remainder can be expressed using the whole quotient q_i of that division as follows:

$r_i = r_{i-2} - q_i r_{i-1}.$

By substitution, this gives:

$r_i = (ax_{i-2} + by_{i-2}) - q_i(ax_{i-1} + by_{i-1}),$ which can be written

$r_i = a(x_{i-2} - q_i x_{i-1}) + b(y_{i-2} - q_i y_{i-1}).$

The first two values are the initial arguments to the algorithm:

$r_1 = a = a \times 1 + b \times 0$

$r_2 = b = a \times 0 + b \times 1.$

So the coefficients start out as $x_1 = 1$, $y_1 = 0$, $x_2 = 0$, and $y_2 = 1$, and the others are given by

$x_i = x_{i-2} - q_i x_{i-1},$

$y_i = y_{i-2} - q_i y_{i-1}.$

The expression for the last non-zero remainder gives the desired results since this method computes every remainder in terms of a and b, as desired.

So the algorithm looks like,

1. Apply Euclidean algorithm, and let qn(n starts from 1) be a finite list of quotients in the division.
2. Initialize x_0, x_1 as 1, 0, and y_0, y_1 as 0,1 respectively.
    1. Then for each i so long as q_i is defined,
    2. Compute $x_{i+1} = x_{i-1} - q_i x_i$
    3. Compute $y_{i+1} = y_{i-1} - q_i y_i$
    4. Repeat the above after incrementing i by 1.
3. The answers are the second-to-last of x_n and y_n.

```
/* This function return the gcd of a and b followed by
    the pair x and y of equation ax + by = gcd(a,b)*/
pair<int, pair<int, int> > extendedEuclid(int a, int b) {
    int x = 1, y = 0;
    int xLast = 0, yLast = 1;
    int q, r, m, n;
    while(a != 0) {
        q = b / a;
        r = b % a;
        m = xLast - q * x;
        n = yLast - q * y;
        xLast = x, yLast = y;
        x = m, y = n;
        b = a, a = r;
    }
    return make_pair(b, make_pair(xLast, yLast));
}

int modInverse(int a, int m) {
    return (extendedEuclid(a,m).second.first + m) % m;
}
```

Recursive Method
This method attempts to solve the original equation directly, by reducing the dividend and divisor gradually, from the first line to the last line, which can then be substituted with trivial value and work backward to obtain the solution.
Notice that the equation remains unchanged after decomposing the original dividend in terms of the divisor plus a remainder, and then regrouping terms. So the algorithm looks like this:

1. If b = 0, the algorithm ends, returning the solution x = 1, y = 0.
2. Otherwise:
   - Determine the quotient q and remainder r of dividing a by b using the integer division algorithm.
   - Then recursively find coefficients s, t such that bs + rt divides both b and r.
   - Finally the algorithm returns the solution x = t, and y = s − qt.

Here's a C++ implementation:

```cpp
/* This function return the gcd of a and b followed by
    the pair x and y of equation ax + by = gcd(a,b)*/
pair<int, pair<int, int> > extendedEuclid(int a, int b) {
    if(a == 0) return make_pair(b, make_pair(0, 1));
    pair<int, pair<int, int> > p;
    p = extendedEuclid(b % a, a);
    return make_pair(p.first, make_pair(p.second.second - p.secc
}

int modInverse(int a, int m) {
    return (extendedEuclid(a,m).second.first + m) % m;
}
```

The time complexity of the above codes is *O(log(m)*^{2}*)*.

3. Using Fermat's Little Theorem

Fermat's little theorem states that if m is a prime and a is an integer co-prime to m, then $a^p - 1$ will be evenly divisible by m. That is $a^{m-1} \equiv 1 \pmod{m}$. or $a^{m-2} \equiv a^{-1} \pmod{m}$. Here's a sample C++ code:

```cpp
/* This function calculates (a^b)%MOD */
int pow(int a, int b, int MOD) {
int x = 1, y = a;
    while(b > 0) {
        if(b%2 == 1) {
            x=(x*y);
            if(x>MOD) x%=MOD;
        }
        y = (y*y);
        if(y>MOD) y%=MOD;
        b /= 2;
    }
    return x;
}

int modInverse(int a, int m) {
    return pow(a,m-2,m);
}
```

The time complexity of the above codes is O(log(m)).

4. Using Euler's Theorem
Fermat's Little theorem can only be used if m is a prime. If m is not a prime we can use Euler's Theorem, which is a generalization of Fermat's Little theorem. According to Euler's theorem, if a is coprime to m, that is, gcd(a, m) = 1, then $a^{\varphi(m)} \equiv 1 \pmod{m}$, where where φ(m) is Euler Totient Function. Therefore the modular multiplicative inverse can be found directly:
$a^{\varphi(m)-1} \equiv a^{-1} \pmod{m}$. The problem here is finding φ(m). If we know φ(m), then it is very similar to above method.

Now lets take a little different question. Now suppose you have to calculate the inverse of first n numbers. From above the best we can do is O(n log(m)). Can we do any better? Yes.

We can use sieve to find a factor of composite numbers less than n. So for composite numbers inverse(i) = (inverse(i/factor(i)) * inverse(factor(i))) % m, and we can use either Extended Euclidean Algorithm or Fermat's Theorem to find inverse for prime numbers. But we can still do better.

a * (m / a) + m % a = m
(a * (m / a) + m % a) mod m = m mod m, or
(a * (m / a) + m % a) mod m = 0, or
(- (m % a)) mod m = (a * (m / a)) mod m.
Dividing both sides by (a * (m % a)), we get
– inverse(a) mod m = ((m/a) * inverse(m % a)) mod m
inverse(a) mod m = (- (m/a) * inverse(m % a)) mod m

Here's a sample C++ code:

```
vector<int> inverseArray(int n, int m) {
    vector<int> modInverse(n + 1,0);
    modInverse[1] = 1;
    for(int i = 2; i <= n; i++) {
        modInverse[i] = (-(m/i) * modInverse[m % i]) % m + m;
    }
    return modInverse;
}
```

The time complexity of the above code is O(n).

-fR0DDY

About these ads

Written by fR0DDY

October 9, 2011 at 12:29 AM

Posted in Programming

Tagged with algorithm, C, code, euclidean, Euler, fermat, inverse, little, modular, multiplicative, theorem

29 Responses

Subscribe to comments with RSS.

hi! nice blog. could u help me in computing multiplicative inverse in java?

siti

December 4, 2011 at 10:20 PM

Reply

The codes are given in C++. Java should be very similar.

fR0DDY

December 5, 2011 at 12:17 AM

Reply

Excellent Post

Mitesh

June 27, 2012 at 6:06 PM

Reply

How would we calculate modInverse(a, m)
Where ‘a’ can be as large as 500! and m=1000000007

rn

February 8, 2013 at 5:49 PM

Reply

first calculate (a!) mod m :
f=1; for(i=1;i<=a;i++) f=(f*i)%m;
then use above algorithm for its inverse.

shahid

March 8, 2013 at 2:15 AM

Reply

Reblogged this on Saurabh Vats.

bit_cracker007

May 26, 2013 at 3:53 AM

Reply

how to calculate 512 bit (156 decimal digits) numbers using c coding.

navya

July 5, 2013 at 8:59 PM

Reply

Running your brute force code in Java, I've gotten number pairs that have more than 1 multiplicative mod inverse. Is there any significance to that?

Eric

July 10, 2013 at 4:55 AM

Reply

Very good blog. Thanks!!

Just a very tiny bit of addition in the last method, the one to compute inverse of all numbers till n. Here, m should be relatively prime (i.e. co-prime) to all the numbers from 2 to n. If this is not the case, then (m%a) = 0, and we cannot divide by (a * (m % a)) and so the equation will break.

Prashant

January 19, 2014 at 7:39 PM

Reply

Can you tell me how to solve, say, (2n C n) % 1000000006? Since n! in denominator will not be co-prime to 1000000006, we can not apply any of the above method. I heard somewhere that CRT can be used. Can you tell me how?
Thanks!

Dhruv

April 14, 2014 at 8:26 PM

Reply

This question is a part of http://www.spoj.com/problems/POWPOW/

Dhruv

April 14, 2014 at 8:31 PM

Reply

Check this blog post:
https://comeoncodeon.wordpress.com/2011/07/31/combination/

fR0DDY

April 15, 2014 at 1:54 PM

"If we have to find nCr mod m(where m is not prime), we can factorize m into primes and then use Chinese Remainder Theorem(CRT) to find nCr mod m"
But my problem is how to use CRT here? I mean, from wikipedia, I got to know that it is used to solve a set of congruence. But, here, we don't have any congruence in the first place!.
Thanks!

Dhruv

April 15, 2014 at 7:16 PM

nice post understood the logic but what does this make_pair function look like ?

hemanth

May 21, 2014 at 12:55 PM

Reply

It is a standard C++ function.
http://www.cplusplus.com/reference/utility/make_pair/

fR0DDY

May 21, 2014 at 12:57 PM

Reply

thank you very much. very useful post

anamzahid

August 21, 2014 at 12:52 AM

Reply

Reblogged this on Anam Zahid and commented:
A very useful post … helps me very much writing key splitter in c++ .. reference code is taking from http://en.wikipedia.org/wiki/Shamir's_Secret_Sharing

anamzahid

August 21, 2014 at 12:54 AM

Reply
[…] with Problem & Implementation […]

Data Structures and Algorithms Tutorials

January 20, 2015 at 11:10 AM

Reply
[…] একটুখানি Modular Multiplicative Inverse ৩. Chinese Remainder Theorem ৪. Modular Multiplicative Inverse ৫. Primality Testing : Non-deterministic […]

মডুলার এরিথমেটিক, বিগ মড, মডুলার ইনভার্স, এক্সটেন্ডেড ইউক্লিড, চাইনিজ রিমেইন্ডার থিওরেম কোড, প্র�

May 27, 2015 at 4:51 PM

Reply
[…] একটুখানি Modular Multiplicative Inverse ৩. Chinese Remainder Theorem ৪. Modular Multiplicative Inverse ৫. Primality Testing : Non-deterministic […]

মডুলার এরিথমেটিক, বিগ মড, মডুলার ইনভার্স, এক্সটেন্ডেড ইউক্লিড, চাইনিজ রিমেইন্ডার থিওরেম, প্রাইম��

May 27, 2015 at 4:57 PM

Reply

```
int modInverse(int a, int m) {
return (extendedEuclid(a,m).second.first + m) % m;
}
```

Why do i need to add m before performing mod? if a*x mod m =1 i have to calculate x as it is inverse modulo of a.why do i need to add m with value of x?

Farsan

June 1, 2015 at 10:07 PM

Reply

So that even if extendedEuclid returns a negative value, we return a positive modular inverse.

fR0DDY

June 3, 2015 at 1:42 AM

Reply

Hi! Thanks for this! I'm looking for some material about a method based on Euler Sieve Method to calculate multiplicative inverse but can't find one, could you provide me with some? (Sorry for my poor Engilish…)

Likecer

June 16, 2015 at 7:32 PM

Reply

please i need a java implementation of the extended euclidean algorithm. i dont know how to implement this in java…please help

ebitunyan

June 22, 2015 at 2:00 AM

Reply

its urgent pleaseeeeeeee

ebitunyan

June 22, 2015 at 2:05 AM

Reply

please help its urgent

ebitunyan

June 22, 2015 at 2:01 AM

Reply

[…] Modular Multiplicative Inverse […]

Getting started with competitive coding | Sameer Chaudhari

July 27, 2015 at 7:21 PM

Reply

[…] tutorial, implementation, problem, reading the chapter from clrs is highly recommended. Modular Multiplicative Inverse nCr % […]

A List Of Some Algorithms with a lot of Resources |

September 10, 2015 at 4:47 AM

Reply

I have been looking for the proof for Modular Inverse from 1 to N. Thanks. That's a beautiful proof

forthright48

September 29, 2015 at 2:19 PM

Reply