Why "OUTPUT THE ANSWER MODULO 10^9 + 7"?

178 Competitive Programming Algorithm

You might have noticed that many programming problems ask you to output the answer "modulo 1000000007 (10^9 + 7)". In this note I'm going to discuss what this means and the right way to deal with this type of questions. So, here it is...

First of all, I'd like to go through some prerequisites.

The **modulo** operation is the same as ' the remainder of the division '. If I say *a* modulo *b* is *c*, it means that the remainder when a is divided by b is c. The modulo operation is represented by the '%' operator in most programming languages (including C/C++/Java/Python). So, 5 % 2 = 1, 17 % 5 = 2, 7 % 9 = 7 and so on.

WHY IS MODULO NEEDED..

The largest integer data type in C/C++ is the *long long int*; its size is 64 bits and can store integers from (–2^63) to (+2^63 -1) . Integers as large as *9 X 10^18* can be stored in a *long long int*.

But in certain problems, for instance when calculating the number of permutations of a size n array, even this large range may prove insufficient. We know that the number of permutations of a size n array is **n!**. Even for a small value of n, the answer can be very large. Eg, for n=21, the answer is 21! which is about 5 x 10^19 and too large for a long long int variable to store. This makes calculating values of large factorials difficult.

So, instead of asking the exact value of the answer, the problem setters ask the answer modulo some number M; so that the answer still remain in the range that can be stored easily in a variable.

Some languages such as Java and Python offer data types that are capable of storing infinitely large numbers. But data type size is not the only problem. As the size of the number increases the time required to perform mathematical operations on them also increases.

There are certain requirements on the choice of M:
1. It should just be large enough to fit in an *int* data type.
2. It should be a prime number.
10^9 + 7 fits both criteria; which is why you nearly always find 10^9 + 7 in modulo type

questions.
I've explained the logic behind the 2nd point at the end of the note.

HOW TO HANDLE QUESTIONS INVOLVING MODULO:

Some basic knowledge of modulo arithmetic is required to understand this part.

A few distributive properties of modulo are as follows:
1. (a + b) % c = ((a % c) + (b % c)) % c
2. (a * b) % c = ((a % c) * (b % c)) % c
3. (a – b) % c = ((a % c) - (b % c)) % c (see notes at bottom)
4. (a / b) % c **NOT EQUAL TO** ((a % c) / (b % c)) % c
So, modulo is distributive over +, * and - but not / .

One observation that I'd like to make here is that **the result of (a % b) will always be less than b**.

If I were to write the code to find factorial of a number n, it would look something like this:

```
long long factorial(int n,int M)
{
     long long ans=1;
     while(n>=1)
     {
          ans=(ans*n)%M;
          n--;
     }
     return ans;
}
```

Notice that on line 6, I performed the modulo operation at EACH intermediate stage. It doesn't make any difference if we first multiply all numbers and then modulo it by M, or we modulo at each stage of multiplication.
(a * b * c) % M is the same as (((a * b) % M) * c) % M

But in case of computer programs, due to size of variable limitations we avoid the first approach and **perform modulo M at each intermediate stage** so that range overflow never occurs.
So the following approach is wrong:

```
long long factorial(int n,int M)//WRONG APPROACH!!!
{
     long long ans=1;
     while(n>=1)
```

```
    {
        ans=ans*n;
        n--;
    }
        ans=ans%M;
    return ans;
}
```

The same procedure can be followed for addition too.
(a + b + c) % M is the same as (((a + b) % M) + c) % M
Again we prefer the second way while writing programs. Perform % M every time a number is added so as to avoid overflow.

The rules are a little different for division. This is the main part of this note;
As I mentioned earlier,
(a / b) % c NOT EQUAL TO ((a % c) / (b % c)) % c which means that modulo operation is not distributive over division.
The following concept is most important to find nCr (ways of selecting r objects from n objects) modulo M. (As an example, I've included the code to find nCr modulo M at the end of this note)
To perform division in modulo arithmetic we need to first understand the concept of **modulo multiplicative inverse.**

Lets go over some basics first.

The **multiplicative inverse** of a number y is z iff (z * y) == 1.

Dividing a number x by another number y is same as multiplying x with the multiplicative inverse of y.

x / y == x * y^(-1) == x * z (where z is multiplicative inverse of y)

In normal arithmetic, the multiplicative inverse of y is y^(-1); which will correspond to some float value. Eg. Multiplicative inverse of 5 is 0.2, of 3 is 0.333... etc.
But in modulo arithmetic the definition of multiplicative inverse of a number y is a little different. **The modulo multiplicative inverse (MMI) of a number y is z iff (z * y) % M == 1.**

Eg. if M= 7 the MMI of 4 is 2 as (4 * 2) %7 ==1,
if M=11, the MMI of 7 is 8 as (7 * 8)%11 ==1,
if M=13, the MMI of 7 is 2 as (7 * 2) % 13==1.
Observe that **the MMI of a number may be different for different M**.
So, if we are performing modulo arithmetic in our program and we need the result of the operation x / y, we should NOT perform

```
z=(x/y)%M;
```

instead we should perform

```
y2=findMMI(y,M);
z=(x*y2)%M;
```

Now one question remains.. How to find the MMI of a number n.
The brute force approach would look something like this

```
int findMMI_bruteforce(int n,int M)
{
    int i=1;
    while(i<M)// we need to go only uptil M-1
    {
        if(( (long long)i * n ) % M ==1)
            return i;
        i++;
    }
    return -1;//MMI doesn't exist
}
```

The complexity of this approach is O(M) and as M is commonly equal to 10^9 + 7, this method is not efficient enough.

There exist two other algorithms to find MMI of a number. First is the **Extended Eucledian algorithm** and the second using **Fermat's Little Theorem**.

If you are new to modulo arithmetic, you'll probably not find these topics easy to understand. These algorithms require prerequisites. If you want to read them they are very well explained on many online sources and some of you will study them in depth in your college's algorithm course too.
I'll keep this note simple for now and only give the code using Fermat Little Theorem.

```
int fast_pow(long long base, long long n,long long M)
{
    if(n==0)
       return 1;
    if(n==1)
    return base;
    long long halfn=fast_pow(base,n/2,M);
    if(n%2==0)
        return ( halfn * halfn ) % M;
    else
```

```
        return ( ( ( halfn * halfn ) % M ) * base ) % M;
}
int findMMI_fermat(int n,int M)
{
    return fast_pow(n,M-2,M);
}
```

This code uses a function fast_pow() that is used to calculate the value of base^p. Its complexity is O(log p). It is a very efficient method of computing power of a number.
It is based on the fact that to find a^n, we just need to find a^(n/2) and the required answer will be *a^(n/2) * a^(n/2)* if n is even; and *a^((n-1)/2) * a^((n-1)/2) * a* if n is odd (if n is odd n/2 == (n-1)/2 in most programming languages).

NOTES:

1. For M to be a prime number is really important. Because if it is not a prime number then it is possible that the result of a modulo operation may become 0. Eg. if M=12 and we perform (8 * 3) % 12, we'll get 0. But if M is prime then ((a % M) * (b % M)) % M can never be 0 (unless a or b == 0)
2. If M is prime then we can find MMI for any number n such that 1<=n<M
3. (a – b) % c = ((a % c) - (b % c)) %c is fine mathematically. But, while programming, don't use

   a=(a%c);
   b=(b%c);
   ans=(a - b)%c;

   instead use

   a=a%c;
   b=b%c;
   ans = (a - b + c) % c;

% works differently with -ve numbers

IMPORTANT:
1. If n1,n2 are int type variables and M=10^9+7, then the result of (n1 * n2) % M will surely be < M (and capable of fitting in a simple int variable). BUT the value of (n1 * n2) can be greater than the capacity of an int variable. Internally, first (n1 * n2) is computed. So, to avoid overflow either declare n or m as long long int OR use explicit type casting ((long long) n * m) % M.

As an example, here is the code to find nCr modulo 1000000007, (0<=r<=n<=100000)

?

```
int fast_pow(long long base, long long n,long long M)
{
    if(n==0)
       return 1;
    if(n==1)
    return base;
    long long halfn=fast_pow(base,n/2,M);
    if(n%2==0)
        return ( halfn * halfn ) % M;
    else
        return ( ( ( halfn * halfn ) % M ) * base ) % M;
}
int findMMI_fermat(int n,int M)
{
    return fast_pow(n,M-2,M);
}
int main()
{
    long long fact[100001];
    fact[0]=1;
    int i=1;
    int MOD=1000000007;
    while(i<=100000)
    {
        fact[i]=(fact[i-1]*i)%MOD;
        i++;
    }
    while(1)
    {
        int n,r;
        printf("Enter n: ");
        scanf(" %d",&n);
        printf("Enter r: ");
        scanf(" %d",&r);
        long long numerator,denominator,mmi_denominator,ans;
        //I declared these variable as long long so that there is no need to
use explicit typecasting
        numerator=fact[n];
        denominator=(fact[r]*fact[n-r])%MOD;
        mmi_denominator=findMMI_fermat(denominator,MOD);
        ans=(numerator*mmi_denominator)%MOD;
        printf("%lld\n",ans);
```

```
    }
    return 0;
}
```


AUTHOR

Abhinav Sharma
Graduate student M.E. (CS...
Bangalore, Karnataka, India
2 notes

TRENDING NOTES

Python Diaries Chapter 3 Map | Filter | For-else | List Comprehension
written by Divyanshu Bansal

Bokeh | Interactive Visualization Library | Use Graph with Django Template
written by Prateek Kumar

Bokeh | Interactive Visualization Library | Graph Plotting
written by Prateek Kumar

Python Diaries chapter 2
written by Divyanshu Bansal

Python Diaries chapter 1
written by Divyanshu Bansal

more ...

Developer hiring guide

Engineering Blog

Developers Blog

Developers Wiki

Competitive Programming

Start a Programming Club

Practice Machine Learning

Assess University Talent

Organize Hackathons

?