Divyansh Kumar Singh's Blog

Competitive Programming

COMPETITIVE PROGRAMMING

Number Theory II: Advanced Modular Arithmetic

Date: May 30, 2017 Author: d_skyhawk 0 Comments

Motivation Problem: This month, in Codechef's Long Challenge, there was a problem SANDWICH. In short, All we had to do was calculate nCr modulo M after evaluating proper values of n and r.

The twist was that M is non-prime and constraints of n and r are 10^18. Hence, standard techniques of finding nCr % M would fail for higher values of n and r. This is where CRT comes to the rescue.
Prerequisites: It is expected that you are familiar with basic Modular Math. Read my previous post for the introduction.

Chinese Remainder Theorem

Suppose we wish to solve

$$x \equiv 2 \ (mod\ 5)$$
$$x \equiv 3 \ (mod\ 7)$$

for x. In easier terms, find an x which gives remainder 2 when divided by 5 and which gives remainder 3 when divided by 7. If we have a solution β, then β + 35 is also a solution and more generally β plus any multiple of 35. So, we need to look for solutions modulo 35 only. Brute forcing the solution, we will find the x = 17 (mod 35) solves our system.

$$x \equiv a \ (mod\ p)$$
$$x \equiv b \ (mod\ q)$$

For any system of congruences like this, the ***Chinese Remainder Theorem tells us that, p and q being co-prime, there always exists <u>a unique solution for x modulo pq.</u>***

Now, lets generalise this onto a bigger scale.
Let us store all our divisors in an array, say **div[]**.
Let us store all the remainders from the ith divisor in another array **rem[]**.

It is given that all the divisors are co-prime to each other. Then, one may list the congruences as:

$$x \equiv rem[0] \pmod{div[0]}$$
$$x \equiv rem[1] \pmod{div[1]}$$
$$\ldots$$
$$x \equiv rem[n-1] \pmod{div[n-1]}$$

So, we need to find an x such that

$$x \equiv y \pmod{product}$$

Now, the Chinese Remainder Theorem proves that there will always exist a solution to the above congruences. Hence, using this we may write the brute force solution to our problem.

```
int solveCRT(int div[], int rem[], int k)
{
    int x = 1; // Initialize result

    while (true)
    {
        // Check whether current x satisfies all remainders
        int j;
        for (j=0; j<k; j++ )
            if (x%div[j] != rem[j])
               break;
        // All remainders matched
        if (j == k)
            return x;
        // Else try next number
        x++;
    }
    // Loop Always terminates. Guaranteed by CRT.
    return x;
}
```

Time Complexity: O(M)
Space Complexity: O(1) , (Assuming we already have the divisors and remainders in two arrays).

Now, lets talk of the Efficient Solution.

Gauss's algorithm

It is based on the following formula:

$$x = \left(\sum \left(rem[i] \times prdiv[i] \times inverseModulo(prdiv[i], div[i])\right)\right) \bmod M$$

where
M = Product of all divisors
prdiv[i] = Product of all divisors except div[i]
inverseModulo(a,b) = Multiplicative Modulo Inverse of a with respect to modulo b

The proof to this algorithm is available here. You can think of the above formula to be similar in application as we do when we write (a+M)%M which avoids negative modulo.

So, lets put it to code. I know I am being fast here but theres lot of exciting stuff ahead.

```
int solveCRT(int div[], int rem[], int k)
{
    // Compute product of all numbers
    int M = 1;
    for (int i = 0; i < k; i++)
        M *= div[i];
    int result = 0;
    // Applying above formula
    for (int i = 0; i < k; i++)
    {
        int prdiv = M / div[i];
        result += rem[i] * inverseModulo(prdiv, div[i]) * prdiv;
    }
    return result % M;
}
```

Finding n! mod p (Where p is prime)

Now, this is very important as I see its application in every 1 contest out of 10 and all of its problems are marked "expert" on Hackerrank.

The main idea behind solving this is to represent $n!$ as $a \times P^e$, where a is relatively prime to p . We do this in 2 steps.

1. We group n! = $1 \times 2 \times 3 \times \ldots \times n$, with p elements in each group. Thus, we write, n! = $(1 \times 2 \times \ldots \times p) \times ((p+1) \times (p+2) \times \ldots \times 2p) \times ..$
   Thus, each group of $1 \times 2 \times 3 \times \ldots \times p-1$ is relatively prime to p.
2. Now, use **Wilson's Theorem** which is $(p-1)! \equiv -1 \pmod p$.

Lets see this by an example. Say we need to find out *79! mod 7.*

So, first lets group the first 79 numbers.
$(1 \times 2 \times \ldots \times 7) \times (8 \times 9 \times \ldots \times 14)$ and so on.
Thus there are total 11 groups of 7 plus 1 group of 2 (=79%7).
Now, we have to represent 79! as $a \times 7^e$. Hence, for each group we have one multiple of 7 (the last number) that adds to the total power e. This means this multiple provided one of the powers of 7 in 79! .
Also, for each group we have a (7-1)! mod 7 = -1 from Wilson's Theorem.

Hence, we have 79/7 = 11 multiples of 7 and because for each of those 11 groups we have one -1 as remainder. Hence, for odd number of groups we have negative remainder and for even number of groups we have positive remainder.

The remainders get multiplied to the final remainder ***a*** and the powers get added to final power ***e***. We also need to account for the last group, which serves only to the remainder and not to the power.

Hence we have

$$\begin{aligned} &e = (n/p) \\ &a = -1 \times (n \quad mod \quad p)! \qquad for \quad odd \\ &a = +1 \times (n \quad mod \quad p)! \qquad for \quad even \end{aligned}$$

But, that was only for multiples of 7. The multiples of powers of 7 also contribute to 79! We need to add them to the final power and remainder of our required expression also.

So we do the same with 7^2 = 49 and evaluate the same results. Then again for 7^3 and so on. If its still not clear, take a look at the recursive code below and you will understand why we need to do this.

```
//facts is a vector of integers that stores i! mod p for each i
pair<int,int> fact_mod(int n, int p, vector<int> facts) {
    if (n == 0)
          return make_pair(1, 0);
    pair<int,int> temp = fact_mod(n / p, p, facts);

    int a = temp.first;
    int e = temp.second;
    e += n / p;
    if (n / p % 2 != 0) //Wilson's Theorem Application.
         return make_pair(a * (p - facts[n % p]) %p, e);
    else
         return make_pair(a * facts[n % p] % p, e);
}
```

Hopefully the above explanation and code was clear.

Finding n! mod p^e

This is the last topic we need to know in order to solve our motivation problem.

Let us suppose $f(n, p^e) = n! \quad mod \quad p^e$. The steps are similar to finding n! mod p.

1. **We separate n! as $n! = H \times p^b$.**
   That is, we find the highest power *b of p* such that $n! \equiv 0 \quad (mod \quad p^b)$
   To find $f(n, p^e)$, the trick is to take all the numbers divisible by ***p*** out of ***n!***. *By all numbers, we mean all multiples of p.*
   Thus H will not be divisible by p.
   Example: 6! mod 2^4
   We take out all the terms divisible by 2 out of 6! that is 2, 4 and 6.
   Hence, we have taken out *2 x 4 x 6 = (1 x 2 x 3) x (2^3).*
   Hence, we precisely take out $\lfloor n/p \rfloor ! \times p^{\lfloor n/p \rfloor}$.
   This procedure is recursive as the above step only takes out multiples of p and not multiples of powers of p (Similar to the reasoning when we were finding n! mod p). *For instance in our current example while taking out all 2s, we got a 2^3 as above, but there are*

more 2s in the part ***1 x 2 x 3.*** *In order to take those 2s out too, we need to recurse for n=n/p = 6/2= 3.*

2. *Having found H and b, we evaluate H mod p^e and p^b mod p^e separately and merge the two answers modulo p^e.*

Hopefully the algorithm must be clear by now. Even if it isnt take a look at the code below to get a taste of what exactly we are doing. I have included a few points below to the code also to clear some common doubts.

```
//To find highest power b of p such that n!=0 mod p^b
//fmodp is an array containing i! modulo p
long long factmaxpower(long long n,long long p){
    if(n==0)
        return 0;
    long long a=1,b=0;
    while(n!=0){
        b+=n/p;
        if((n/p)%2)
            a=(a*(p-fmodp[n%p]))%p;
        else
            a=(a*fmodp[n%p])%p;
        n/=p;    //ITERATE FOR EACH POWER MULTIPLE
    }
    return b;
}

//To find H mod p^e
//fpmode is an array containing i! mod p^e
long long factmod(long long n,long long p, long long m){
    //m = p^e
    if(n<=1)
        return 1;
    else if(n<m)
        return (fpmode[n]*factmod(n/p,p,m))%m;
    else{
        long long a=fpmode[m-1];
        long long b=fpmode[n%m];
        long long c=factmod(n/p,p,m);
        return (powmod(a,n/m,m)*((b*c)%m))%m;
//powmod is fast power modulo m function
    }
}
```

Note:

1. While finding highest power b, we multpily ***a*** with *p-fmodp[n%p] for odd number of groups because* of Wilson's Theorem (See Wilson's Theorem above for details).
2. While calculating H, we use modulo M but when we are recursively call the same function we divide it by *p. Reason being that we intend to remove all terms of p from H and at the same time get modulo of H*. The example given above will help in understanding the division by *p*.

Motivation Problem