APS 2023: Session 1

# Modular Combinatorics and Exponentiation in CP

Chahel, Haricharan_B, Aditya_Jain__

# QR for Whatsapp Group

# Topics to be covered

1. Modulo & its properties
2. Fermat's Little Theorem
3. Modular Multiplicative Inverse i.e. Mod_Inv
4. Computing Mod_Inv
5. Fast computation of $a^b$
6. Computing large numbers modulo 1e9+7
7. Power Function
8. Mod_Inv Function
9. Computing n! modulo p
10. nCr Function
11. nPr Function

# Modulo Operation

- In Mathematics, the modulo operation is used to return the remainder of a division i.e. after one number is divided by the other.

# Modulo Operation

- In Mathematics, the modulo operation is used to return the remainder of a division i.e. after one number is divided by the other.
- Given 2 positive numbers *a* and *b*, *a modulo b* (or in programming terms, *a % b*) is the remainder when *a* is divided by *b*.
  Eg.: 14 modulo 5 = 4, 23 modulo 7 = 2, 9 mod 3 = 0.

# Modulo Operation

- In Mathematics, the modulo operation is used to return the remainder of a division i.e. after one number is divided by the other.
- Given 2 positive numbers *a* and *b*, *a modulo b* (or in programming terms, *a % b*) is the remainder when *a* is divided by *b*.
  Eg.: 14 modulo 5 = 4, 23 modulo 7 = 2, 9 mod 3 = 0.

- When exactly one of *a* or *b* is negative, the naive definition breaks down, and different languages implement modulo differently here.
  In C++ (CF and Codeblocks), -5 % 3 = -2.

# Modulo Operation

- In Mathematics, the modulo operation is used to return the remainder of a division i.e. after one number is divided by the other.
- Given 2 positive numbers *a* and *b*, *a modulo b* (or in programming terms, *a % b*) is the remainder when *a* is divided by *b*.
  Eg.: 14 modulo 5 = 4, 23 modulo 7 = 2, 9 mod 3 = 0.
- When exactly one of *a* or *b* is negative, the naive definition breaks down, and different languages implement modulo differently here.
  In C++ (CF & Codeblocks), -5 % 3 = -2.
- For the rest of this session, we will only be dealing with *a % b*, where *a* is non-negative and *b* is positive.

# Properties of Modulo Operator

1. $(a + b) \bmod n = [(a \bmod n) + (b \bmod n)] \bmod n$

# Properties of Modulo Operator

1. $(a + b) \bmod n = [(a \bmod n) + (b \bmod n)] \bmod n$
2. From (1), $(a - b) \bmod n = [(a \bmod n) - (b \bmod n) + n] \bmod n$

# Properties of Modulo Operator

1. $(a + b) \bmod n = [(a \bmod n) + (b \bmod n)] \bmod n$
2. From (1), $(a - b) \bmod n = [(a \bmod n) - (b \bmod n) + n] \bmod n$
3. $ab \bmod n = [(a \bmod n)(b \bmod n)] \bmod n$.

# Properties of Modulo Operator

1. $(a + b) \bmod n = [(a \bmod n) + (b \bmod n)] \bmod n$
2. From (1), $(a - b) \bmod n = [(a \bmod n) - (b \bmod n) + n] \bmod n$
3. $ab \bmod n = [(a \bmod n)(b \bmod n)] \bmod n$.

Why? Can you prove these 3 properties using $a = nq + r$ and $b = ns + t$, where $r = a \bmod n$ & $t = b \bmod n$?

# Properties of Modulo Operator

1. $(a + b) \bmod n = [(a \bmod n) + (b \bmod n)] \bmod n$
2. From (1), $(a - b) \bmod n = [(a \bmod n) - (b \bmod n) + n] \bmod n$
3. $ab \bmod n = [(a \bmod n)(b \bmod n)] \bmod n$.

   Why? Can you prove these 3 properties using $a = nq + r$ and $b = ns + t$, where $r = a \bmod n$ & $t = b \bmod n$?
4. From (3), $a^b \bmod n = (a \bmod n)^b \bmod n$

# Properties of Modulo Operator

1. $(a + b) \bmod n = [(a \bmod n) + (b \bmod n)] \bmod n$
2. From (1), $(a - b) \bmod n = [(a \bmod n) - (b \bmod n) + n] \bmod n$
3. $ab \bmod n = [(a \bmod n)(b \bmod n)] \bmod n$.

   Why? Can you prove these 3 properties using $a = nq + r$ and $b = ns + t$, where $r = a \bmod n$ & $t = b \bmod n$?
4. From (3), $a^b \bmod n = (a \bmod n)^b \bmod n$

- What about $a/b \bmod n$? There is a conditional property to this, but it's not as simple as the previous ones. Before we can understand a/b mod n, there is a prerequisite theorem that should be understood.

# Fermat's Little Theorem

- If $p$ is a prime number, then for any positive integer $a$,

$$a^p \equiv a \bmod p$$

(Means $a^p$ % p = a % p)

# Fermat's Little Theorem

- **If $p$ is a prime number, then for any positive integer $a$,**

$$a^p \equiv a \bmod p$$

(Means $a^p$ % p = a % p)

- If $a$ is not divisible by p, then:

  From FLT, we know $a^p - a = qp$

# Fermat's Little Theorem

- **If $p$ is a prime number, then for any positive integer $a$,**

$$a^p \equiv a \bmod p$$

(Means $a^p \% p = a \% p$)

- If $a$ is not divisible by p, then:

  From FLT, we know $a^p - a = qp$

  As LHS is divisible by a, RHS must also be divisible by a. As p is not divisible by a, q must be divisible by a. Let q = ka.

  => $a^p - a = kap$

# Fermat's Little Theorem

- **If $p$ is a prime number, then for any positive integer $a$,**

$$a^p \equiv a \bmod p$$

(Means $a^p \% p = a \% p$)

- If $a$ is not divisible by p, then:

  From FLT, we know $a^p - a = qp$

  As LHS is divisible by a, RHS must also be divisible by a. As p is not divisible by a, q must be divisible by a. Let q = ka.

$$\Rightarrow \quad a^p - a = kap$$

$$\Rightarrow \quad a^{p-1} - 1 = kp$$

This becomes: $a^{p-1} \equiv 1 \bmod p$ *(a is not divisible by p & p is prime)*

We will be using this form of the theorem in the upcoming discussions.

# Modular Multiplicative Inverse - Mod_Inv

- In modulo language, the additive inverse of a is -a. Why?
  Because for any $n$, $(a + (-a)) \equiv 0 \mod n$.

# Modular Multiplicative Inverse - Mod_Inv

- In modulo language, the additive inverse of a is -a. Why?
  Because for any $n$, $( a + (-a) ) \equiv 0 \mod n$.

- Similarly, is there a multiplicative inverse of a wrt modulo?
  Yes, under certain terms and conditions!

# Modular Multiplicative Inverse - Mod_Inv

- In modulo language, the additive inverse of a is -a. Why?
  Because for any $n$, $( a + (-a) ) \equiv 0 \mod n$.

- Similarly, is there a multiplicative inverse of a wrt modulo?
  Yes, under certain terms and conditions!

- **The modular multiplicative inverse of a number *a* (wrt modulo *n*) is an integer *x* such that:**

$$ax \equiv 1 \pmod{n}.$$

# Modular Multiplicative Inverse - Mod_Inv

- In modulo language, the additive inverse of a is -a. Why?
  Because for any *n*, $( a + (-a) ) \equiv 0 \mod n$.

- Similarly, is there a multiplicative inverse of a wrt modulo?
  Yes, under certain terms and conditions!

- **The modular multiplicative inverse of a number *a* (wrt modulo *n*) is an integer *x* such that:**

$$ax \equiv 1 \pmod{n}.$$

- Now that we have seen the definition, for what kind of pairs (a,n) is Mod_Inv(a,n) defined? If and only if *a* and *n* are coprime i.e. *GCD(a,n) = 1. Why so?*

# Bezout's Identity

**Let $a$ and $b$ be integers with greatest common divisor $d$. Then there exist integers $x$ and $y$ such that $ax + by = d$. More generally, the integers of the form $ax + by$ are exactly the multiples of $d$.**

How is this useful?

# Bezout's Identity

**Let *a* and *b* be integers with greatest common divisor *d*. Then there exist integers *x* and *y* such that *ax* + *by* = *d*. More generally, the integers of the form *ax* + *by* are exactly the multiples of *d*.**

How is this useful?

- If *GCD(a,n) = g != 1*, then all numbers of the form *ax - ny* will be divisible by *g*.
  For Mod_Inv(a,n) to exist, we need to have ax - 1 = ny i.e. *ax - ny = 1*.
  But this is not possible since *ax - ny* is divisible by *g* for all integers *x* and *y*.

# Bezout's Identity

**Let $a$ and $b$ be integers with greatest common divisor $d$. Then there exist integers $x$ and $y$ such that $ax + by = d$. More generally, the integers of the form $ax + by$ are exactly the multiples of $d$.**

How is this useful?

- If $GCD(a,n) = g$ != 1, then all numbers of the form $ax - ny$ will be divisible by $g$.
  For Mod_Inv(a,n) to exist, we need to have ax - 1 = ny i.e. $ax - ny = 1$.
  But this is not possible since $ax - ny$ is divisible by $g$ for all integers $x$ and $y$.

- If $GCD(a,n) = 1$, then there exist integers x and y such that $ax - ny = 1$
  i.e. $ax \equiv 1 \pmod{n}$. Hence, Mod_Inv(a,n) exists.

# Bezout's Identity

**Let $a$ and $b$ be integers with greatest common divisor $d$. Then there exist integers $x$ and $y$ such that $ax + by = d$. More generally, the integers of the form $ax + by$ are exactly the multiples of $d$.**

How is this useful?

- If $GCD(a,n) = g$ != 1, then all numbers of the form $ax - ny$ will be divisible by $g$.
  For Mod_Inv(a,n) to exist, we need to have ax - 1 = ny i.e. $ax - ny = 1$.
  But this is not possible since $ax - ny$ is divisible by $g$ for all integers $x$ and $y$.

- If $GCD(a,n) = 1$, then there exist integers x and y such that $ax - ny = 1$
  i.e. $ax \equiv 1 \pmod{n}$. Hence, Mod_Inv(a,n) exists.

Can the Bezout's Identity be proved by me?
Yes! Can you think of an intuitive proof? I leave it as an exercise to the reader. :)

# Is Mod_Inv(a,n) unique?

- Question: Is Mod_Inv(a,n) unique?

# Is Mod_Inv(a,n) unique?

- Question: Is Mod_Inv(a,n) unique?
  No! If $ax \equiv 1 \bmod n$, then $a(x+n) \equiv 1 \bmod n$ i.e. if $x$ works, then $x + n$ also works.

# Is Mod_Inv(a,n) unique?

- Question: Is Mod_Inv(a,n) unique?
  No! If $ax \equiv 1 \bmod n$, then $a(x+n) \equiv 1 \bmod n$ i.e. if $x$ works, then $x + n$ also works.

- But in range [1,n), there is only 1 Mod_Inv(a,n). Why?

# Is Mod_Inv(a,n) unique?

- Question: Is Mod_Inv(a,n) unique?
  No! If $ax \equiv 1 \bmod n$, then $a(x+n) \equiv 1 \bmod n$ i.e. if $x$ works, then $x + n$ also works.

- But in range [1,n), there is only 1 Mod_Inv(a,n). Why?
  You can think about it for some time. :)

# Application of Mod_Inv

- What exactly is the application of Mod_Inv for us now?
- Our initial aim was to find $(a/b)$ mod $n$ in terms of $a$ and $b$.

  Prerequisite: b and n must be coprime.

# Application of Mod_Inv

- What exactly is the application of Mod_Inv for us now?
- Our initial aim was to find *(a/b)* mod *n* in terms of *a* and *b*.
  Prerequisite: b and n must be coprime.

Let a/b = c.
As b and n are coprime, there exists an integer x such that $bx \equiv 1$ mod $n$.
Let us assume we can find *x* i.e. Mod_Inv(b,n).

# Application of Mod_Inv

- What exactly is the application of Mod_Inv for us now?
- Our initial aim was to find *(a/b)* mod *n* in terms of *a* and *b*.
  Prerequisite: b and n must be coprime.

Let a/b = c.
As b and n are coprime, there exists an integer x such that $bx \equiv 1$ mod *n*.
Let us assume we can find *x* i.e. Mod_Inv(b,n).

If we know x, we know bx.
Multiply c on both sides of the modulo congruence. (Why can we do this?)
If a % n = b % n, then ac % n = bc % n. (From Property (3))
We now have $cbx \equiv c$ mod *n*. But cb = a => $ax \equiv c$ mod *n*.

# Application of Mod_Inv

- What exactly is the application of Mod_Inv for us now?
- Our initial aim was to find *(a/b)* mod *n* in terms of *a* and *b*.
  Prerequisite: b and n must be coprime.

Let a/b = c.
As b and n are coprime, there exists an integer x such that $bx \equiv 1 \bmod n$.
Let us assume we can find *x* i.e. Mod_Inv(b,n).

If we know x, we know bx.
Multiply c on both sides of the modulo congruence. (Why can we do this?)
If a % n = b % n, then ac % n = bc % n. (From Property (3))
We now have $cbx \equiv c \bmod n$. But cb = a => $ax \equiv c \bmod n$.

Thus, **(a/b) % n = [ a*Mod_Inv(b,n) ] % n.**
It looks like 1/b got replaced with Mod_Inv(b,n).

# Finding Mod_Inv(b,p) where p is prime

Why exactly did we study Fermat's Little Theorem for?

# Finding Mod_Inv(b,p) where p is prime

Why exactly did we study Fermat's Little Theorem for?

It was to find Mod_Inv(b,p) where *p* is prime.

# Finding Mod_Inv(b,p) where p is prime

Why exactly did we study Fermat's Little Theorem for?
It was to find Mod_Inv(b,p) where *p* is prime.

Fermat's Little Theorem: If b is not divisible by a prime p, then $b^{p-1} \equiv 1 \bmod p$.
Mod_Inv(b,n) is a number x such that $bx \equiv 1 \bmod p$.

# Finding Mod_Inv(b,p) where p is prime

Why exactly did we study Fermat's Little Theorem for?
It was to find Mod_Inv(b,p) where *p* is prime.

Fermat's Little Theorem: If b is not divisible by a prime p, then $b^{p-1} \equiv 1 \bmod p$.
Mod_Inv(b,n) is a number x such that $bx \equiv 1 \bmod p$.

From the above 2 equations, one solution to x is $b^{p-2}$.
In fact, $b^{p-2}$ % p also works!

# Finding Mod_Inv(b,p) where p is prime

Why exactly did we study Fermat's Little Theorem for?

It was to find Mod_Inv(b,p) where *p* is prime.

Fermat's Little Theorem: If b is not divisible by a prime p, then $b^{p-1} \equiv 1 \bmod p$.

Mod_Inv(b,n) is a number x such that $bx \equiv 1 \bmod p$.

From the above 2 equations, one solution to x is $b^{p-2}$.

In fact, $b^{p-2}$ % p also works!

Hence, **Mod_Inv(b,p) = $b^{p-2}$ % p.** **[One of the solutions]**

Summarizing, **(a/b) % p = [ a*Mod_Inv(b,n) ] % p = [ a * ($b^{p-2}$ % p) ] % p**

**(a/b) % p = [ (a % p) * ($b^{p-2}$ % p) ] % p**

# How fast can you compute pow(a,b) i.e. $a^b$?

Let us hypothetically say the computer can deal with numbers upto 10^(10^10).

Also, in this set up, assume addition, subtraction, multiplication & division each take 1 second to do.

Around how many seconds will you take to find 3^(10^9)?

Note: Error of upto 1000% is allowed :)

# How fast can you compute pow(a,b) i.e. $a^b$?

Let us hypothetically say the computer can deal with numbers upto 10^(10^10).

Also, in this set up, assume addition, subtraction, multiplication & division each take 1 second to do.

Around how many seconds will you take to find 3^(10^9)?

Note: Error of upto 1000% is allowed :)

I give you guarantee that I can do it within 1 minute!

But how can I do it in such short time?

# How am I so fast at multiplication? :)

I will compute 3^(2^i) for i from 1 to j, where j is the least number such that 2^j > 1e9.

i.e. from 1 to log_2 (1e9) = 30

# How am I so fast at multiplication? :)

I will compute 3^(2^i) for i from 1 to j, where j is the least number such that 2^j > 1e9.

i.e. from 1 to log_2 (1e9) = 30

Now I write the binary representation of 1e9, which is 111011100110101100101000000000.

# How am I so fast at multiplication? :)

I will compute 3^(2^i) for i from 1 to j, where j is the least number such that 2^j > 1e9.

i.e. from 1 to log_2 (1e9) = 30

Now I write the binary representation of 1e9, which is 111011100110101100101000000000.

I initialize result to 1.
Now I travel through the bits of 1e9. If the $i^{th}$ bit of 1e9 is 1, I multiply 3^(2^i) to result.
Else I don't.

# How am I so fast at multiplication? :)

I will compute 3^(2^i) for i from 1 to j, where j is the least number such that 2^j > 1e9.
i.e. from 1 to log_2 (1e9) = 30

Now I write the binary representation of 1e9, which is 111011100110101100101000000000.

I initialize result to 1.
Now I travel through the bits of 1e9. If the $i^{th}$ bit of 1e9 is 1, I multiply 3^(2^i) to result.
Else I don't.

Now the final result I will have will be 3^(summation of 2^i such that ith bit of 1e9 is 1)
The exponent I have is 1e9 itself.

# How am I so fast at multiplication? :)

I will compute 3^(2^i) for i from 1 to j, where j is the least number such that 2^j > 1e9.
i.e. from 1 to log_2 (1e9) = 30

Now I write the binary representation of 1e9, which is 111011100110101100101000000000.

I initialize result to 1.
Now I travel through the bits of 1e9. If the $i^{th}$ bit of 1e9 is 1, I multiply 3^(2^i) to result.
Else I don't.

Now the final result I will have will be 3^(summation of 2^i such that ith bit of 1e9 is 1)
The exponent I have is 1e9 itself.

Number of multiplications I did < 2*log_2 (1e9) < 60.

# Code for the “fast” multiplier

```
typedef long long ll;
ll FastMultiplier(ll a,ll b) // Evaluates a^b
{
    ll result=1, a_pwr=a;
    while(b!=0)
    {
        if(b%2==1)
        result*=a_pwr;
        a_pwr=a_pwr*a_pwr;
        b/=2;
    }
    return result;
}
```

# Computing large values modulo 1e9+7

What is the time complexity of the FastMultiplier(a,b) function?

# Computing large values modulo 1e9+7

What is the time complexity of the FastMultiplier(a,b) function?

O(log_2 b).

# Computing large values modulo 1e9+7

What is the time complexity of the FastMultiplier(a,b) function?
$O(\log_2 b)$.

However, our computer is capable of handling integers only till $10^{19}$.
Hence, in CP Questions where large numbers come as the answer, we are asked to output them modulo 1000000007 or 998244353.

(What's so special about these numbers?)

# Computing large values modulo 1e9+7

What is the time complexity of the FastMultiplier(a,b) function?
O(log_2 b).

However, our computer is capable of handling integers only till $10^{19}$.
Hence, in CP Questions where large numbers come as the answer, we are asked to output them modulo 1000000007 or 998244353.

(What's so special about these numbers?)
They are large, and they are prime. Note that they are around $10^9$ as well.

# Computing large values modulo 1e9+7

What is the time complexity of the FastMultiplier(a,b) function?
O(log_2 b).

However, our computer is capable of handling integers only till $10^{19}$.
Hence, in CP Questions where large numbers come as the answer, we are asked to output them modulo 1000000007 or 998244353.

(What's so special about these numbers?)
They are large, and they are prime. Note that they are around $10^9$ as well.

So now, can we modify our FastMultiplier to output 2^(10^9) modulo 1000000007?

# Computing large values modulo 1e9+7

What is the time complexity of the FastMultiplier(a,b) function?
O(log_2 b).

However, our computer is capable of handling integers only till $10^{19}$.
Hence, in CP Questions where large numbers come as the answer, we are asked to output them modulo 1000000007 or 998244353.

(What's so special about these numbers?)
They are large, and they are prime. Note that they are around $10^9$ as well.

So now, can we modify our FastMultiplier to output 2^(10^9) modulo 1000000007?
Yes, we can.
We exploit property (3) of modulo: $ab \bmod n = [(a \bmod n)(b \bmod n)] \bmod n$.

# Power(a,b) - Computes (a^b) modulo 1e9+7

Time Complexity: O(log b).

```
typedef long long ll;
#define mod 1000000007
// Take a and b as input and returns : The power (a,b) , (a^b) % mod.
// mod need not be a prime OR coprime to b.
ll Power(ll a, ll b)
{
    ll result=1;
    ll a_pwr=a%mod;
    while(b)
    {
        if(b%2==1) result*=a_pwr;
        a_pwr*=a_pwr;
        a_pwr%=mod; // Take modulo everywhere
        result%=mod;
        b/=2;
    }
    return result;
}
```

Note that mod need not be coprime to b or be a prime number.

# Function to compute Mod_Inv(b,p) where p is prime

Now that we have developed an efficient function to calculate (a^b) % p, we can use this to find Mod_Inv(b,p).

$$\text{Mod\_Inv}(b,p) = b^{p-2} \% p$$

$$= \text{Power}(b,p-2) \text{ where mod=p.}$$

```
typedef long long ll;
#define p 998244353
// It gives the modulo inverse of a, (1/a)%p.
ll Mod_Inv(ll a)
{
    a%=p;
    ll x= Power(a,p-2);
    return x;
}
```

Time Complexity: O(log p)

# Computing n! % p efficiently

How many digits do you think 300C100 has?

# Computing n! % p efficiently

How many digits do you think 300C100 has?

Definitely greater than 48 digits!

# Computing n! % p efficiently

How many digits do you think 300C100 has?
Definitely greater than 48 digits!

When faced with combinatorial computations with large n,r, we are again asked to output the final value modulo 1e9 + 7 or some big prime number.

What do we do in this case?

# Computing n! % p efficiently

How many digits do you think 300C100 has?
Definitely greater than 48 digits!

When faced with combinatorial computations with large n,r, we are again asked to output the final value modulo 1e9 + 7 or some big prime number.

What do we do in this case?

First, we pre-compute n! % p i.e. compute n! % p (generally for n from 1 to $10^5$) before running test cases.

# Computing n! % p efficiently

How many digits do you think 300C100 has?
Definitely greater than 48 digits!

When faced with combinatorial computations with large n,r, we are again asked to output the final value modulo 1e9 + 7 or some big prime number.

What do we do in this case?

First, we pre-compute n! % p i.e. compute n! % p (generally for n from 1 to $10^5$) before running test cases.
We use the idea of factorial(n) = factorial(n-1) * n & memoization to make our computation process efficient.

# Computing n! % p efficiently

Time Complexity: O(Fact_Length)

```
typedef long long ll;
#define mod 1000000007
#define Fact_length 200001

ll Factorial[Fact_length];

ll Make_Factorial()
{
    Factorial[0]=1;
    for(ll i=1;i<Fact_length;i++)
    {
        Factorial[i]=i*Factorial[i-1];
        Factorial[i]%=mod;
    }
    return 0;
}
```

# Computing nCr % p

Given n and r, how would you compute nCr % p? (Given p is a prime)

# Computing nCr % p

Given n and r, how would you compute nCr % p? (Given p is a prime)

nCr = n! / (n-r)!r!

Hence nCr modulo p = [ (n!,p)*(Mod_Inv((n-r)!,p)*(Mod_Inv(r!,p) ] % p

# Computing nCr % p

Given n and r, how would you compute nCr % p? (Given p is a prime)

nCr = n! / (n-r)!r!

Hence nCr `modulo p = [ (n!,p)*(Mod_Inv((n-r)!,p)*(Mod_Inv(r!,p) ] % p`

But we can't find r! for large r...What do we do now?

# Computing nCr % p

Given n and r, how would you compute nCr % p? (Given p is a prime)

nCr = n! / (n-r)!r!

Hence nCr `modulo p = [ (n!,p)*(Mod_Inv((n-r)!,p)*(Mod_Inv(r!,p) ] % p`

But we can't find r! for large r...What do we do now?

Lemma: If $a \equiv b \bmod p$ ($a\%p=b\%p=r$) & $p$ is prime, then Mod_Inv(a,p) = Mod_Inv(b,p)

# Computing nCr % p

Given n and r, how would you compute nCr % p? (Given p is a prime)

nCr = n! / (n-r)!r!

Hence nCr modulo p = [ (n!,p)*(Mod_Inv((n-r)!,p)*(Mod_Inv(r!,p) ] % p

But we can't find r! for large r...What do we do now?

Lemma: If *a* ≡ *b* mod *p* *(a%p=b%p=r)* & *p* is prime, then Mod_Inv(a,p) = Mod_Inv(b,p)

Proof: Mod_Inv(a,p) = $a^{p-2}$ % p = $((a\%p)^{p-2})$ % p = $(r^{p-2})$ % p

Mod_Inv(b,p) = $b^{p-2}$ % p = $((b\%p)^{p-2})$ % p = $(r^{p-2})$ % p

Hence, both are equal.

# Computing nCr % p

Given n and r, how would you compute nCr % p? (Given p is a prime)

nCr = n! / (n-r)!r!

Hence nCr `modulo p = [ (n!,p)*(Mod_Inv((n-r)!,p)*(Mod_Inv(r!,p) ] % p`

But we can't find r! for large r...What do we do now?

Lemma: If $a \equiv b \bmod p$ $(a\%p=b\%p=r)$ & $p$ is prime, then `Mod_Inv(a,p) = Mod_Inv(b,p)`

Proof: `Mod_Inv(a,p)` $= a^{p-2} \% p = ((a\%p)^{p-2}) \% p = (r^{p-2}) \% p$

`Mod_Inv(b,p)` $= b^{p-2} \% p = ((b\%p)^{p-2}) \% p = (r^{p-2}) \% p$

Hence, both are equal.

Corollary: `Mod_Inv(a,p) = Mod_Inv(a%p,p)`

# Computing nCr % p

Given n and r, how would you compute nCr % p? (Given p is a prime)

nCr = n! / (n-r)!r!

Hence nCr modulo p = [ (n!,p)*(Mod_Inv((n-r)!,p)*(Mod_Inv(r!,p) ] % p

But we can't find r! for large r...What do we do now?

Lemma: If *p* is prime, then Mod_Inv(a,p) = Mod_Inv(a%p,p)

Proof: Mod_Inv(a,p) = $a^{p-2}$ % p = $((a\%p)^{p-2})$ % p = Mod_Inv(a%p,p

Hence, both are equal.

For this scenario, Mod_Inv(r!,p) = Mod_Inv(r!%p,p)=Mod_Inv(Factorial[r]);

This makes it convenient because we can compute r! % p easily. In fact, we have precomputed it.

```
Hence nCr modulo p  = [ (n!%p)*(Mod_Inv((n-r)!,p)*(Mod_Inv(r!,p) ] % p
                    = [ (n!% p) * (Mod_Inv((n-r)!%p, p) * (Mod_Inv(r!%p, p) ] % p
                    = [ (Factorial[n],p) * (Mod_Inv(Factorial[n-r],p)) * (Mod_Inv(Factorial[r],p)) ] %
p]
```

# Computing nCr % p

```
typedef long long ll;
#define mod 1000000007
ll nCr(ll n, ll r)
{
    if(n<r) return 0;
    ll ans=Factorial[n];
    ans%=mod;
    ans*=Mod_Inv(Factorial[r]);
    ans%=mod;
    ans*=Mod_Inv(Factorial[n-r]);
    ans%=mod;
    return ans;
}
```

Time Complexity: O(log mod)

# Computing nPr % p

nPr = n!/(n-r)!

nPr % p = [ (n!%p)*(Mod_Inv((n-r)!,p)] % p

= [ (n!%p) * (Mod_Inv((n-r)!%p,p) ] % p

= [ (Factorial[n], p) * (Mod_Inv(Factorial[n-r], p)) ] % p

```
typedef long long ll;
#define mod 1000000007
ll nPr(ll n, ll r)
{
    ll ans=Factorial[n];
    ans%=mod;
    ans*=Mod_Inv(Factorial[n-r]);
    ans%=mod;
    return ans;
}
```

Time Complexity: O(log mod)

# Feedback Form:

# Illustrative Problems

1. Distributing Apples
2. Christmas Party
3. Check out this CF Blog!