

# Modulus Operator | NumberTheory Part 2

 [leetcode.com/discuss/study-guide/3767785/topics-which-you-cant-skip-interview-preparation-part-2](https://leetcode.com/discuss/study-guide/3767785/topics-which-you-cant-skip-interview-preparation-part-2)

▲

The topic included in this article is very important. It can be used in many types of questions. In this article, I will discuss the **modular arithmetic**, including the modulus operator, modular addition, modular subtraction, modular multiplication, and modular division. We will also discuss some of the applications of modular arithmetic.

Now Let's discuss about modulus operator and its operation. Modular arithmetic is a system of arithmetic where **numbers "wrap around" when they reach a certain value, called the modulus.**

For example, if the modulus is 12, then 13 is the same as 1, 14 is the same as 2, and so on.

The modulo operator, also known as the remainder operator, gives the remainder when one number is divided by another. For example,  $10 \% 3 = 1$ , because 10 divided by 3 leaves a remainder of 1.

The following table shows some basic identities about the modulo operator:

Operation	Formula	Explanation
Addition	$(a + b) \% \text{mod} = ((a \% \text{mod}) + (b \% \text{mod})) \% \text{mod}$	Adding two numbers modulo <b>mod</b> is equivalent to adding their remainders modulo <b>mod</b> and then taking the modulo <b>mod</b> again.
Multiplication	$(a * b) \% \text{mod} = ((a \% \text{mod}) * (b \% \text{mod})) \% \text{mod}$	Multiplying two numbers modulo <b>mod</b> is equivalent to multiplying their remainders modulo <b>mod</b> and then taking the modulo <b>mod</b> again.
Subtraction	$(a - b) \% \text{mod} = ((a \% \text{mod}) - (b \% \text{mod}) + \text{mod}) \% \text{mod}$	Subtracting two numbers modulo <b>mod</b> is equivalent to subtracting their remainders modulo <b>mod</b> , adding <b>mod</b> to the result to make it positive, and then taking the modulo <b>mod</b> again.
Division	$(a / b) \% \text{mod} = ((a \% \text{mod}) * (b^{-1} \% \text{mod})) \% \text{mod}$	Dividing <b>a</b> by <b>b</b> modulo <b>mod</b> is equivalent to multiplying the remainder of <b>a</b> modulo <b>mod</b> by the modular multiplicative inverse of <b>b</b> modulo <b>mod</b> , and then taking the modulo <b>mod</b> again. The modular multiplicative inverse is the number $b^{-1}$ such that $(b * b^{-1}) \% \text{mod} = 1$ .

These identities allow us to work with the residues % m instead of the actual values of large numbers. By taking the modulus frequently, we can perform addition, subtraction, and multiplication **without worrying about integer overflow**.

For instance, if we need to calculate the factorial of a number like 23 or even larger, storing the true value becomes challenging. However, **we can still determine the factorial % m for a given modulus**. This approach enables us to handle calculations **even when storing the entire value is not feasible**. Let me provide you with the C++ code:

```
const int MOD = 1e9 + 7; // Modulus value

// Function to calculate factorial modulo m
long long factorialModM(int n) {
    long long factorial = 1;
    for (int i = 2; i <= n; i++) {
        factorial = (factorial * i) % MOD;
    }
    return factorial%MOD;
}
```

### Binary exponentiation:

Imagine you have a number, let's say 25. To calculate  $25^2$ , you would multiply 25 by itself 2 times. But what if you wanted to calculate  $25^{100}$ ? Multiplying 25 by itself 100 times would take a long time. **Basically we are doing in  $O(n)$  time can we do in less time Yes, we can do it in less time using binary exponentiation.** Binary exponentiation is a technique that uses the binary representation of the exponent to calculate the power.

Note : Binary exponentiation is a powerful technique that is used in a variety of applications, including modular arithmetic, cryptography, and optimization algorithms. Binary exponentiation is a technique for computing large powers of a number efficiently. It works by recursively squaring the base and multiplying it by the base at the appropriate places. **The algorithm terminates when all the bits in the binary representation of the exponent have been processed.**

The idea behind binary exponentiation is that **we can split the work of computing  $a^n$  into smaller and smaller pieces by using the binary representation of the exponent**. For example, if  $n = 13$ , then the binary representation of n is 1101<sub>2</sub>. This means that we can compute  $a^n$  by computing  $a^8$ ,  $a^4$ , and  $a^1$ , and then multiplying these three numbers together.

For example, the binary representation of 100 is 1100100. This means that  $25^{100}$  can be calculated by computing  $25^8$ ,  $25^4$ , and  $25^1$ , and then multiplying these three numbers together.

The binary exponentiation algorithm works as follows:

1. Start with the base, a.

2. Square the base.
3. If the most significant bit of the exponent is 1, then multiply the result by the base.
4. Shift the exponent one bit to the right.
5. Repeat steps 2-4 until the exponent is 0.

**The final result is the power of the base.**

$$x^n = \begin{cases} x(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even} \end{cases}$$

Lets code it

//Recursive

```
int MOD+1e9+7;
int BinaryExp(int a ,int b )
{
    if(b==0) return 1;
    int res= BinaryExp(a,b/2);
    res= ( res*1LL*res )%MOD;
    if(b%2)
        return ( a* res)%MOD;
    else return res;
}
```

// Iterative code

```
int MOD+1e9+7;
int BinaryExp(int a ,int b )
{
    int ans=1;
    while(b!=0)
    {
        if(b%2) ans = (ans*1LL*a)%MOD;
        a= (a*1LL*a)%MOD;
        b/=2;
    }
    return ans;
}
```

Lets understand throug example

suppose you have  $a=3$   $b=13$  and you wanna to cal value of  $(a)^{\text{power}(b)}$

↳ general approach come in your mind of  $O(n)$

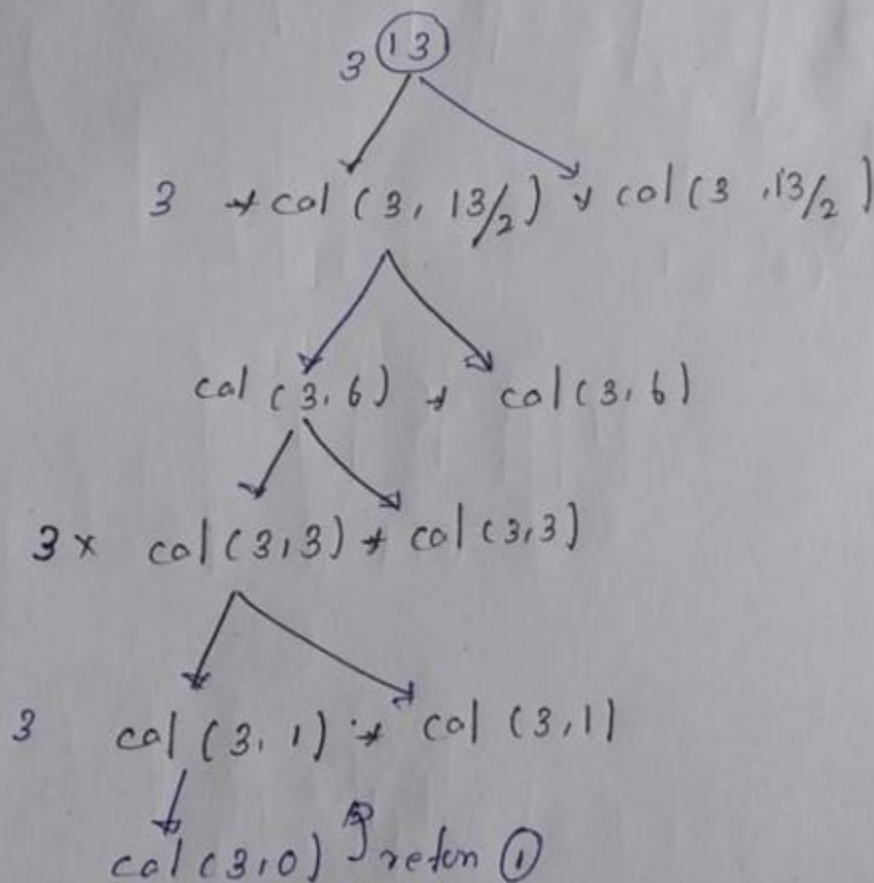
```
int ans=1
for(int i=1; i<=b; i++) {
    ans = ans * a
}
```

→ But not eff  
one

So we will use Binary exponentiation

$$a^b = \begin{cases} a \times f(a^2, b/2) \times f(a, b/2) & \text{if } b \text{ is odd} \\ f(a^2, b/2) \times f(a, b/2) & \text{if } b \text{ is even} \end{cases}$$

$$\text{eg- } 3^{13} = \underline{3^8 \times 3^4 \times 3^1}$$



## Now lets talk about Binary Exponentiation for large numbers

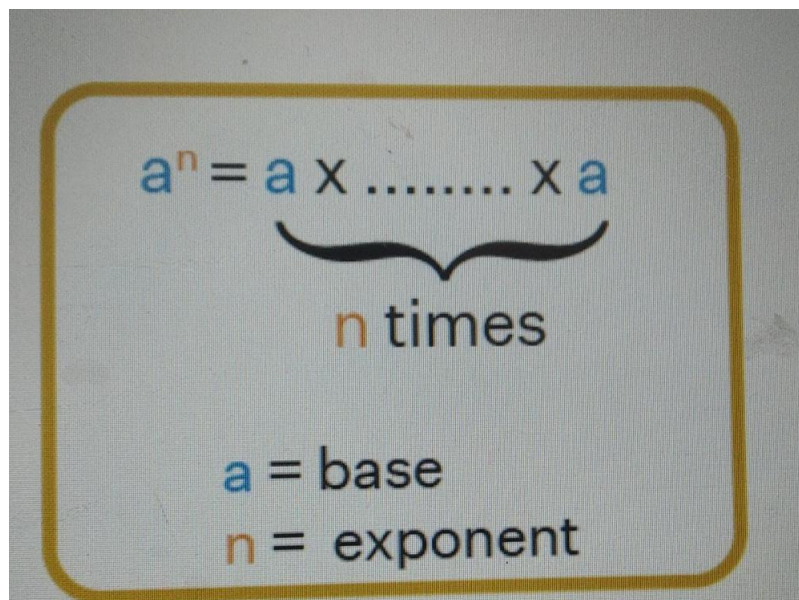
NOTE: If  $a \leq 10^9$ ,  $b \leq 10^9$ , and  $\text{mod} \leq 10^9 + 7$ , then we can apply binary exponentiation without overflow. However, what if  $a$  is greater than  $10^{10}$  or even larger?

**In C++, the maximum value that a long long can store is less than or equal to  $10^{18}$ . So, what if we have  $a \leq 10^{18}$ ,  $b \leq 10^{18}$ , and  $\text{mod} \leq 10^{18}$ ? Will our binary exponentiation still work?**

---

**Let's discuss the first case:  $a \leq 10^{18}$  and  $b \leq 10^9$  and  $\text{mod} \leq 10^9$ .**

This case is relatively easy to deal with. We can simply calculate the binary exponentiation of  $a$  first. Then, we can use the binary exponentiation of  $a$  to calculate  $a^b$ . The mod operator, %, returns the remainder of a division. So, if we take the mod of  $a^b$ , we are essentially taking the remainder of the calculation of  $a^b$  divided by a large number. the mod operator only cares about the remainder of the calculation. The actual value of  $a^b$  is irrelevant.


$$a^n = a \times \dots \times a$$

n times

$a = \text{base}$   
 $n = \text{exponent}$

For example, let's say we want to calculate `binaryexp(5, 2)`, where  $\text{mod} = 2$ . We can do this by first calculating `binaryexp(5, 1000)`. Then, we can use the binary exponentiation of 5 to calculate  $5^2$ , which is 25. Finally, we can take the remainder of 25 divided by 2, which is 1.  $10^9$  order and now we can calculate  $a$  to the power of  $b$

```
int res = BinaryExp(a,b)
```

**In the second case, we have  $a \leq 10^{18}$ ,  $b \leq 10^9$ , and  $\text{mod} \leq 10^{18}$ .** This means that  $a$  is a very large number, and  $\text{mod}$  is also a very large number.

If we try to calculate  $a^b$  using the binary exponentiation method, we will eventually reach a point where the value of  $a$  is greater than  $\text{mod}$ . This is because the binary exponentiation method works by repeatedly squaring  $a$ . So, if  $a$  is already greater than  $\text{mod}$ , then squaring  $a$  will only make it larger.

This means that we cannot use the binary exponentiation method to calculate  $a^b$  directly. but why lets see

To understand this better, let's say you can only store numbers up to 9 (means our data limit is 9 that is maximum digit it can store upto 9 here ) If you have  $a = 4$ ,  $b = 4$ , and  $\text{mod} = 5$ , then you want to calculate  $4^4$ . However,  $4 * 4 = 16$ , which is greater than 9. This means that you cannot store the result of  $4 * 4$ .

If you try to calculate  $4^4$  using the binary exponentiation method, you will eventually reach a point where the value of  $a$  is greater than 9. This is because the binary exponentiation method works by repeatedly squaring  $a$ . So, if  $a$  is already greater than 9, then squaring  $a$  will only make it larger.

This means that the result of the exponentiation will overflow. In other words, the result will be a number that is greater than 9, which is not allowed.

So how we can calculate binary exponetion here we can use binary Multiplication  
eg ->  $4*4=16$  we can not able to do in above case but we can do  $4+4+4+4=>$  as  $(4+4)\%5$  is allowed we will do like this AS AA means adding A (A times) so we can multiply number by binary multiplication so we will use binary multiplication here to overcome form overflow

```
long long mod=10e18+7;
long long BinaryMul( long long a ,long long b)
{
    long long ans=0;
    while(b>0)
    {
        if(b&1) ans = ( ans+a )%mod;
        a= ( a + a )%mod;
        b>=1;
    }
    return ans;
}
```

```
long long BinaryExp( long long a ,long long b)
{
    long long ans=0;
    while(b>0)
    {
        if(b&1)
            ans= BinaryMul(ans,a);
        a=BinaryMul(a,a);
        b>=1;
    }
    return ans;
}
```

The idea is to use binary multiplication to avoid the overflow that would occur if we tried to multiply two large numbers directly.



### case 3:-> when $b \leq 10^{18}$ or more

this part is actually hard so i would say read it care fully or be with me with your all paitence lets discuss

#### we have coded like this

```
int MOD=1e9+7;
long long BinaryExp(long long a ,long long b )
{
    long long ans=1;
    while(b!=0)
    {
        if(b%2) ans = (ans*a)%MOD;
        a= (a*a)%MOD;
        b/=2;
    }
    return ans;
}
```

The above code runs only  $\log(b)$  times for any value of b it wil run eg  $b=10^{20}$   $\log(10^{20})$  is actually small number but the problem arries when its taking b as input you can not able to take b as input as maximum number you can store in c++ is in long long at that is  $10^{18}$  range so how b is given to you

eg-> question is like this  $b = b^c$  and you have to calulate  $a^b$  where ( $b = b^c$ ) so  $b^c$  can be large You are thingking you will take  $\text{mod } b\% \text{mod}$  but this is not allowed as its leed wrong result b is power not base as in case first when ("A") is large

#### So Here we will use Euler's theorem

**Euler's Theorem states** that if n is a positive integer and a is an integer such that  $\text{gcd}(a,n)=1$ , then  $a^{\phi(n)} \equiv 1 \pmod{n}$ . Here,  $\phi(n)$  is Euler's totient function, which counts the number of positive integers less than or equal to n that are relatively prime to n.

**Fermat's Little Theorem** is a special case of Euler's Theorem, where n is a prime number. Fermat's Little Theorem states that if p is a prime number and a is an integer such that  $\text{gcd}(a,p)=1$ , then  $a^{p-1} \equiv 1 \pmod{p}$ .

#### The main properties of Euler's Theorem are as follows:

1. If n is a prime number, then  $\phi(n)=n-1$ .
2.  $\phi(n)$  is multiplicative, i.e. if  $\text{gcd}(m,n)=1$ , then  $\phi(mn)=\phi(m)\phi(n)$ .
3. If a is an integer such that  $1 \leq a \leq n$ , then  $a^{\phi(n)} \equiv 1 \pmod{n}$ .

#### The main properties of Fermat's Little Theorem are as follows:

1. If a is an integer such that  $1 \leq a \leq p-1$ , then  $a^{p-1} \equiv 1 \pmod{p}$ .
2. If a is an integer such that  $\text{gcd}(a,p)=1$ , then  $a^{p-1} \equiv 1 \pmod{p}$ .

Exponentiation reduction using multiplicative inverse by Euler's theorem and Fermat's theorem is a technique for reducing the exponent of a number modulo another number. The technique uses the fact that  $a^b \equiv a^{b \phi(n) \pmod n}$  for any integers  $a$ ,  $b$ , and  $n$  such that  $\gcd(a, n) = 1$ .



Basically I am saying you have given input  
is like this

$$(50^{60^{90}}) \% 10^9 + 7 \quad \text{something like this}$$

So  $b = 60^{90}$  but you cannot take  
mod

$$(50^{(60^{90} \% \text{mod})}) \% \text{mod} \rightarrow x \quad \text{Not allowed}$$

↪ As  $b$  is power not base so it  
Leads to wrong result

⇒ So Here we have to learn about Euler's  
theorem which says

$$(ab) \% \text{mod} \Rightarrow [a^b \equiv (a^{b \% \phi(\text{mod})}) \% \text{mod}]$$

$$[ (a^{b \% \text{mod}}) \equiv (a^{b \% \phi(\text{mod})}) \% \text{mod} ]$$

↪ Basically E T F of  
mod

$$\phi(n) = n \times \prod_{\substack{p \mid n \\ \text{all prime factor of } n}} (1 - 1/p)$$

eg  $\phi(6) \rightarrow$  find prime factor of 6  $\rightarrow 2, 3$

$$\begin{aligned} \phi(6) &= 6 \times (1 - 1/2) (1 - 1/3) \\ &= 6 \times 1/2 \times 2/3 = 2 \end{aligned}$$

$\Rightarrow$  In most cases we have given mod = some prime number  
eg -  $(10^9 + 7)$  It is for our use

$\phi(n) = n(1 - 1/n)$  if  $n$  is prime only one possible case

$\phi(n) = (n-1)$  if  $n$  is prime

[So Euler's value of prime number = number - 1]

Then it's easy for us to calculate ~~from~~ Euler's value

$$\begin{aligned} \text{eg } \left( \begin{matrix} 60 & 90 \\ 50 & \end{matrix} \right) \cdot 10^9 + 7 &\rightarrow \text{As Euler's value so we can do this} \\ &= \left( \begin{matrix} 60 & 90 \\ 50 & \end{matrix} \right) \cdot (10^9 + 6) \cdot 10^9 + 7 \end{aligned}$$

Euler's Theorem which states that:  $a^{\phi(n)} \equiv 1 \pmod{n}$ , where  $\phi(n)$  is Euler's Totient Function, provided that  $a$  and  $n$  are relatively prime (meaning they have no common factors). In our case,  $n = 1337$ , which has prime factorization  $1337 = 7 \times 191$ . We know that  $\phi(p) = p - 1$  when  $p$  is prime and also that  $\phi(p \times q) = \phi(p) \times \phi(q)$ . see how 1140 is calculated out:  $\phi(1337) = \phi(7) \times \phi(191) = 6 \times 190 = 1140$

```
class Solution {
public:
    int BinaryExp(int a ,int b ,int mod )
    {
        int ans=1;
        while(b!=0)
        {
            if(b%2) ans = (ans*a)%mod;
            a= (a*1LL*a)%mod;
            b/=2;
        }
        return ans;
    }
    int superPow(int a, vector<int>& b) {
        int p = 0;
        for (int i : b) p = (p * 10 + i) % 1140;
        if (p == 0) p += 1140;
        return BinaryExp(a, p, 1337);
    }
};
```

*just leaving this topic here, I will not go into detail, this is enough for binaryexp for our purpose*

### Now lets talk about modular multiplicative inverse

$(a / b) \% \text{mod} = ((a \% \text{mod}) * (b^{(-1)} \% \text{mod})) \% \text{mod}$

if num is prime, then we can use Fermat's theorem to calculate the modular multiplicative inverse of a modulo num. This is because Fermat's theorem states that if  $a$  is an integer and  $p$  is a prime number, then  $a^{p-1} \equiv 1 \pmod{p}$ .

In other words, if we raise  $a$  to the power of  $p-1$  modulo  $p$ , then the remainder will be 1. This means that  $a^{p-1}$  is the modular multiplicative inverse of  $a$  modulo  $p$ .

So, if num is prime, then we can calculate the modular multiplicative inverse of a modulo num by simply raising  $a$  to the power of  $p-1$  modulo  $p$ .  $a^{(m-1)} \equiv 1 \pmod{m}$  If we multiply both sides with  $a^{(-1)}$ , we get  $a^{(-1)} \equiv a^{(m-2)} \pmod{m}$  so we just need to calculate  $\text{modPower}(a, m-2)$

However, if num is not prime, then we cannot use Fermat's theorem to calculate the modular multiplicative inverse of a modulo num. In this case, we need to use the extended Euclidean algorithm.

The extended Euclidean algorithm is a mathematical algorithm that can be used to find the greatest common divisor of two numbers and the modular multiplicative inverse of one number modulo another number.

```
code :
int MOD=1e9+7;
int BinaryExp(int a ,int b )
{
    int ans=1;
    while(b!=0)
    {
        if(b%2) ans = (ans*1LL*a)%MOD;
        a= (a*1LL*a)%MOD;
        b/=2;
    }
    return ans;
}

int main()
{
    int a=2;
    int aInverse= BinaryExp(a,MOD-2); // this is how we will calculate
    // inverse of a number
}
```

**question:**

**Find the number of permutations and combinations, if  $n = 100$  and  $r = 3$ .**

Ans-> Combination,  $c = nCr = n!/(n-r)!$

as you know you can not simply take mod so here inverse came in role

```

code :
const int MOD = 1e9 + 7; // Modulus value

// Function to calculate factorial modulo m
long long factorialModM(int n) {
    long long factorial = 1;
    for (int i = 2; i <= n; i++) {
        factorial = (factorial * i) % MOD;
    }
    return factorial%MOD;
}

long long BinaryExp( long long a ,long long b)
{
    long long ans=0;
    while(b>0)
    {
        if(b&1)
            ans= BinaryMul(ans,a);
        a=BinaryMul(a,a);
        b>=1;
    }
    return ans;
}

int solve()
{
    int n=100,r=3;
    int facN= factorialModM(100);
    int facNmiusR=factorialModM(100-3);
    int facR=factorialModM(3);
    int denominator= (facNmiusR*facR)%mod;
    int denominatorInv= BinaryExp(denominator,mod-2);
    int res = ( facN*denominatorInv)%mod << endl;
}

```

question : Fancy-sequence

```

code: int mod97 = 1000000007;
/**
Calculates multiplicative inverse
*/
unsigned long modPow(unsigned long x, int y) {
    unsigned long tot = 1, p = x;
    for (; y; y >>= 1) {
        if (y & 1)
            tot = (tot * p) % mod97;
        p = (p * p) % mod97;
    }
    return tot;
}

class Fancy {
public:
    unsigned long seq[100001];
    unsigned int length = 0;
    unsigned long increment = 0;
    unsigned long mult = 1;
    Fancy() {
        ios_base::sync_with_stdio(false);
        cin.tie(NULL);
    }

    void append(int val) {
        seq[length++] = (((mod97 + val - increment)%mod97) * modPow(mult, mod97-
2))%mod97;
    }
    void addAll(int inc) {
        increment = (increment+ inc%mod97)%mod97;
    }

    void multAll(int m) {
        mult = (mult* m%mod97)%mod97;
        increment = (increment* m%mod97)%mod97;
    }

    int getIndex(int idx) {
        if (idx >= length){
            return -1;
        }else{
            return ((seq[idx] * mult)%mod97+increment)%mod97;
        }
    }
};

```

you can read this for better understanding of this question

link : [fancy-sequence/solutions](#)

**Question:**Count Anagrams

logic:

1. The basic idea is to multiply the number of ways to write each word.

2. The number of ways to write a word of size  $n$  is  $n$  factorial, i.e.,  $n!$ . For example, for the word "abc," there are 6 ways: "abc," "acb," "bac," "bca," "cab," "cba," which is equal to  $3!$  (3 factorial).
3. Since we want the total unique count, words with repeating characters like "aa" can be written in only 1 way. So we need to divide our ways by the factorial of the frequencies of repeating characters.
4. We create a frequency array of the word, `freq[]`.
5. Our formula becomes:  $\text{ways} = n! / (\text{freq}[i]! * \text{freq}[i+1]! * \dots * \text{freq}[n-1]!)$ .
6. The overall answer is obtained by multiplying `ways[i] * ways[i+1] * ... * ways[n]`.
7. However, the answer can be a large number, so we need to return it modulo  $1e9+7$ .
8. To handle this, we use modular arithmetic for every computation since numbers can be very large. In our formula:  $a = n!, b = (\text{freq}[i]! * \text{freq}[i+1]! * \dots * \text{freq}[n-1]!)$ .
9. But  $(a / b) \% \text{mod} \neq (a \% \text{mod}) / (b \% \text{mod})$ . To solve this, we use Modular Multiplicative Inverse.
10. Modular Multiplicative Inverse states that  $(A / B) \% \text{mod} = A * (B^{-1}) \% \text{mod}$ .
11. We can find  $(B^{-1}) \% \text{mod}$  using Fermat's little theorem, which states that  $(b^{-1}) \% \text{mod} = b^{(\text{mod} - 2)}$  if  $\text{mod}$  is prime.
12. In our case,  $\text{mod}$  is  $1e9+7$ . Instead of calculating  $b^{(\text{mod} - 2)}$  using a loop, we can use Binary Exponentiation, which allows us to calculate  $a^n$  in  $O(\log(n))$  time.



```

class Solution {
public:
    int mod=1e9+7;
    int fact[100002];

    int modmul(int a,int b){
        return((long long)(a%mod)*(b%mod))%mod;
    }

    int binExpo(int a,int b){
        if(!b)return 1;
        int res=binExpo(a,b/2);
        if(b&1){
            return modmul(a,modmul(res,res));
        }else{
            return modmul(res,res);
        }
    }

    int modmulinv(int a){
        return binExpo(a,mod-2);
    }

    void getfact() {
        fact[0]=1;
        for(int i=1;i<=100001;i++){
            fact[i]=modmul(fact[i-1],i);
        }
    }

    int ways(string str) {
        int freq[26] = {0};
        for(int i = 0; i<str.size(); i++) {
            freq[str[i] - 'a']++;
        }
        int totalWays = fact[str.size()];
        int factR=1;
        for(int i = 0; i<26; i++) {
            factR=modmul(factR,fact[freq[i]]);
        }
        return modmul(totalWays,modmulinv(factR));
    }

    int countAnagrams(string s) {
        getfact();
        istringstream ss(s);
        string word;
        int ans=1;
        while (ss >> word)
        {
            ans=modmul(ans,ways(word));
        }
        return ans;
    }
}

```

```
};
```

The above code and logic is of devanshu171 for question Count Anagrams

### Questions From above discuss topic

Please let me know if you have any specific questions related to these problems!

This is Part 2 of a series on Number theory. If you haven't read Part 1, you can **find it here: [topics-which-you-cant-skip-interview-preparation-part-1](#)**

**My Other post:->**

**[All-types-of-patterns-for-bits-manipulations-and-how-to-use-it](#)**  
**[Binary-search-a-comprehensive-guide](#)**

**If you found it helpful, please upvote. If you have any questions or comments, please feel free to leave them below. I'd love to hear your thoughts.**

modulosbinaryexponention

Read More

You can check out my **[YouTube channel.](#)**, where I upload videos covering all the articles I've written,

along with new topics. **[Click Here To Subscribe.](#)**

**Do You want Article on graph in detail**

eg-> 1. When to use graph

2 .when to use dfs and bfs

3. How bfs and dijistra relatated to each other

4. when to use union find

5. when Topological sort is needed

6.How just finding cycle in graph fill help you to find atleast 30 questions and much much more...

?

▲

60

▼

🗨 Show 9 replies

↩ Reply

🔗 Share

⚠ Report



Read More

This is easily the best material that I have found on internet recently for free. Such clear explanations for not so easy to understand topics. Great stuff!

▲

3



-  Reply
-  Share
-  Report