Big Mod

Given three positive integers a,p and m, find the value of a^P%m.

Simple Code: Divide and Conquer Approach - O(log2(P))

```
#include<bits/stdc++.h>
using namespace std;
long long Big_Mod(long long a,long long p,long long m){
	if(p==0)	return 1%m;
	if(p%2){
		long long x = Big_Mod(a,p-1,m)%m;
		return ((a%m)*(x%m))%m;
	}
	else{
		long long x = Big_Mod(a,p/2,m)%m;
		return ((x%m)*(x%m))%m;
	}
}
int main(){
	ios_base::sync_with_stdio(false);
	cin.tie(NULL);
	long long a,p,m;
	while(cin>>a>>p>>m){
		cout<<Big_Mod(a,p,m)<<"\n";
	}
	return 0;
}
```

Simple Code: Repeated Squaring Method for Modular Exponentiation

```
#include<bits/stdc++.h>
using namespace std;
long long Big_Mod(long long a,long long p,long long m){
	long long ans=1%m,x=a%m;
	while(p){
		if(p&1) ans=(ans*x)%m;
		x=(x*x)%m;
		p=p>>1;
	}
	return ans;
}
int main(){
	long long a,p,m;
	while(cin>>a>>p>>m){
		cout<<Big_Mod(a,p,m)<<'\n';
	}
	return 0;
}
```

Sieve of Eratosthenes - Explanation

The algorithm has runtime complexity of $O(N\log(\log N))$.

```
using namespace std;
bool mark[mx];
vector<int>prime;
void sieve(){
	memset(mark,true,sizeof(mark));
	mark[0]=mark[1]=false;
	for(int i=4;i<=mx;i+=2) mark[i]=false;
	for(int i=3;i<=(int)sqrt(mx);i+=2){	if(mark[i]){
			for(int j=i*i;j<=mx;j+=2*i) mark[j]=false;
		}
	}
	prime.push_back(2);
	for(int i=3;i<=mx;i+=2) if(mark[i]) prime.push_back(i);
}
int main(){
	sieve();
	for(int i=0;i<100;i++){
		cout<<i+1<<" "<<prime[i]<<endl;
	}
	return 0;
}
```

Segmented Sieve of Eratosthenes

```
int Segmented_Sieve(int a,int b){
	int arr2[b-a+1];
	if(a==1) a++;
	int sqrtb=sqrt(b);
	memset(arr2,0,sizeof(arr2));
	for(int i=0;i<prime.size()&&prime[i]<=sqrtb;i++){
		int p=prime[i];
		int j=p*p;
		if(j<a) j=((a+p-1)/p)*p;
		for(;j<=b;j+=p) arr2[j-a]=1;
	}
	int ans=0;
	for(int i=a;i<=b;i++){
		if(arr2[i-a]==0) ans++;
	}
	return ans;
}
```

Prime Factorization of an Integer

The algorithm has runtime complexity of $O(\sqrt{N}/\ln(\sqrt{N})+\log_2(N))$.

```
void prime_factor(int n){
        int sqrtn=sqrt(n);
        for(int i=0;i<prime.size()&&prime[i]<=sqrtn;i++){
                if(mark[n]) break;
                if(n%prime[i]==0){
                        while(n%prime[i]==0){
                                n/=prime[i];
                                factor.push_back(prime[i]);
                        }
                        sqrtn=sqrt(n);
                }
        }
        if(n!=1) factor.push_back(n);
}
```

Number of Divisors of an Integer

```
int NOD(int n){
    int sqrtn=sqrt(n);
    int ans=1;
    for(int i=0;i<prime.size()&&prime[i]<=sqrtn;i++){
         int c=0;
         if(n%prime[i]==0){
              while(n%prime[i]==0){
                   n/=prime[i];
                   c++;
              }
              sqrtn=sqrt(n);
              c++;
              ans*=c;
         }
    }
    if(n!=1) ans*=2;
    return ans;
}
```

Sum of Divisors of an Integer

```
int SOD(int n){
   int sqrtn=sqrt(n);
   int ans=1;
  for(int i=0;i<prime.size()&&prime[i]<=sqrtn;i++){
      int c=1,tempsum=1;
      if(n%prime[i]==0){
         while(n%prime[i]==0){
            n/=prime[i];
            c*=prime[i];
            tempsum+=c;
         }
         sqrtn=sqrt(n);
         ans*=tempsum;
      }
   }
   if(n!=1) ans*=(n+1);
   return ans;
}
```

#Bitwise Sieve:

```
int mark[(mx>>5)+1];
void sieve(){
         memset(mark,0,sizeof(mark));
         for(int i=3;i<=(int)sqrt(mx);i+=2){
         if(!check(mark[i>>5],i&31)) for(int j=i*i;j<=mx;j+=i<<1)
                 mark[j>>5]=biton(mark[j>>5],j&31);
        }
        prime.push_back(2);
        for(int i=3;i<=mx;i+=2) if(!check(mark[i>>5],i&31)) prime.push_back(i);
}
```

Extended Euclidean Algorithm

Forthright48

```
#include<bits/stdc++.h>
using namespace std;
int ext_gcd ( int A, int B, int *X, int *Y ){
    int x2, y2, x1, y1, x, y, r2, r1, q, r;
    x2 = 1; y2 = 0;
    x1 = 0; y1 = 1;
    for (r2 = A, r1 = B; r1 != 0; r2 = r1, r1 = r, x2 = x1, y2 = y1, x1 = x, y1 = y ) {
        q = r2 / r1;
        r = r2 % r1;
        x = x2 - (q * x1);
        y = y2 - (q * y1);
    }
    *X = x2; *Y = y2;
    return r2;
}
int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int a,b,x,y;
    cin>>a>>b;
    int ans=ext_gcd(a,b,&x,&y);
    printf("gcd(%d, %d) = %d, x = %d, y= %d",a,b,ans,x,y);
    return 0;
}
```

gfg

```
using namespace std;
int ext_gcd(int a,int b,int *x,int *y){
    if(a==0){
        *x=0;
        *y=1;
        return b;
    }
    int x1,y1;
    int gcd = ext_gcd(b%a,a,&x1,&y1);
    *x=y1-(b/a)*x1;
    *y=x1;
    return gcd;
}
int main(){
    int a,b,x,y;
    cin>>a>>b;
    int ans=ext_gcd(a,b,&x,&y);
    printf("gcd(%d, %d) = %d, x = %d, y= %d",a,b,ans,x,y);
    return 0;
}
```

Linear Diophantine Equation

```
///Given three integers a, b, c representing a linear equation of the form : ax + by = c.
#include<bits/stdc++.h>
using namespace std;
int gcd(int a,int b){
///    return b==0?a:gcd(b,a%b);
    return (a%b==0)?abs(b):gcd(b,a%b);
}
bool isPossible(int a,int b,int c){
    return (c%gcd(a,b)==0);
}
int main(){
    int a,b,c;
    cin>>a>>b>>c;
    isPossible(a,b,c)?cout<<"Possible\n":cout<<"Not Possible\n";
    return 0;
}
```

#Print Also the x and y values:

```
bool linearDiophantine ( int A, int B, int C, int *x, int *y ) {
    int g = gcd ( A, B );
    if ( C % g != 0 ) return false; //No Solution
    int a = A / g, b = B / g, c = C / g;
    ext_gcd( a, b, x, y ); //Solve ax + by = 1
    if ( g < 0 ) { //Make Sure gcd(a,b) = 1
        a *= -1; b *= -1; c *= -1;
    }
    *x *= c; *y *= c; //ax + by = c
    return true; //Solution Exists
}
int main () {
    int x, y, A = 2, B = 3, C = 5;
    bool res = linearDiophantine ( A, B, C, &x, &y );

    if ( res == false ) printf ( "No Solution\n" );
    else {
        printf ( "One Possible Solution (%d %d) \n", x, y );
        int g = gcd ( A, B );
        int k = 1; //Use different value of k to get different solutions
        printf ( "Another Possible Solution (%d %d)\n", x + k * ( B / g ), y - k * ( A / g ) );
    }

    return 0;
}
```

Simple Hyperbolic Diophantine Equation

Given the integers A,B,C,D, find a pair of (x,y) such that it satisfies the equation Axy+Bx+Cy=D.

```
using namespace std;
vector<pair<int,int> >sol;
bool isValidSolution(int a,int b,int c,int p,int divisor){
    if(((divisor-c)%a)!=0) return false;//x = (div - c) / a
    if(((p-b*divisor)%(a*divisor))!=0) return false;// y = (p-b*div)
    sol.push_back(make_pair((divisor-c)/a,(p-b*divisor)/(a*divisor)));
    return true;
}
int hyperbolicDiophantine(int a,int b,int c,int d){
    int p=a*d+b*c;
    if(p==0){
        if(-c%a==0) return -1;///Infinite Solution (-c/a,k)
        else if(-b%a==0) return -1;///Infinite Solution(k,-b/a)
        else return 0;///NO Solution
    }
    else{
        int sqrtn=sqrt(p),ans=0;
        for(int i=1;i<=sqrtn;i++){
            if(p%i==0){          //Check if divisors i,-i,p/i,-p/i produces valid solutions
                if(isValidSolution(a,b,c,p,i)) ans++;
                if(isValidSolution(a,b,c,p,-i)) ans++;
                if(p/i!=i){
                    if(isValidSolution(a,b,c,p,p/i)) ans++;
                    if(isValidSolution(a,b,c,p,-p/i)) ans++;
                }
            }
        }
        return ans;
    }
}
int main(){
    int a,b,c,d;
    scanf("%d %d %d %d",&a,&b,&c,&d);
    int ans=hyperbolicDiophantine(a,b,c,d);
    printf("%d\n",ans);
    for(int i=0;i<ans;i++)   cout<<sol[i].first<<" "<<sol[i].second<<'\n';
    return 0;
}
```

Introduction to Number Systems

Here is a code that can convert a number in base B into decimal.

```
int baseToDecimal(string s,int base){
    int ans=0,c=1;
    for(int i=s.size()-1;i>=0;i--){
        if(isalpha(s[i])) ans+=(s[i]-'0'-7)*c;
        else ans+=(s[i]-'0')*c;
        c*=base;
    }
    return ans;
}
```

Here is a code that can convert a decimal number into base B:

```
string symbol={'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};
string decimalToBase(int x,int base){
    string ans="";
    while(x){
        int r=x%base;
        ans=ans+symbol[r];
        x/=base;
    }
    if(ans=="") ans=symbol[0];
    reverse(ans.begin(),ans.end());
    return ans;
}
```

Number of Digits of Factorial

```
#include<bits/stdc++.h>
#define M_E 2.718281828
#define M_PI 22/7
using namespace std;
//int num_digit(int n){
//    double x=0;
//    for(int i=1;i<=n;i++){
//        x+=log10(i);
//    }
//    int ans=((int)x)+1;
//    return ans;
//}
```

```
///However that solution would not be able to handle cases where n >10^6
///So, can we improve our solution ?
///Yes ! we can.
///We can use Kamenetsky's formula to find our answer !
long long num_digit(int n){
    if(n<0) return 0;
    if(n<=1) return 1;
    double x=((n*log10(n/M_E)+log10(2*M_PI*n)/2.0));
    return floor(x)+1;
}
int main(){
    int n;
    cin>>n;
    cout<<num_digit(n)<<'\n';
    return 0;
}
```

Digits of $N!$ in Different Base

Can we use logarithms to solve this problem too? Yes.

$$\text{number of digits of } x \text{ in base } B = \log_B(x)$$

All we need to do is change the base of our $\log$ and it will find number of digits in that base.
But, how do we change base in our code? We can only use log with base 2 and 10 in C++. Fear not, we can use the following law to change base of logartihm from B to C.

$$\log_B(x) = \log_C(x) / \log_C(B)$$

```
#include<bits/stdc++.h>
using namespace std;
int num_digit_fac(int n,int base){
    double x=0;
    for(int i=1;i<=n;i++){
        x+=log10(i)/log10(base);
    }
    int ans=((int)x)+1;
    return ans;
}
int main(){
    int n,base;
    cin>>n>>base;
    cout<<num_digit_fac(n,base)<<'\n';
    return 0;
}
```

Prime Factorization of Factorial

Given a positive integer N, find the prime factorization of N!.
For example, 5!=5×4×3×2×1=120=23×3×5.

First - Linear Loop From 1 to N

We know that N!=N×(N−1)×(N−2)×...×2×1. So we could simply factorize every number from 1 to Nand add to a global array that tracks the frequency of primes. Using the code for factorize() from here, we could write a solution like below.

```
void prime_factor(int n){
   int sqrtn=sqrt(n);
   for(int i=0;i<prime.size()&&prime[i]<=sqrtn;i++){
      if(n%prime[i]==0){
         while(n%prime[i]==0){
            n/=prime[i];
            factor[prime[i]]++;
         }
         sqrtn=sqrt(n);
      }
   }
   if(n!=1) factor[n]++;
}
void fact_factor(int n){
   for(int i=2;i<=n;i++) prime_factor(i);
   printf("%d! =",n);
   for(int i=0;prime[i]<=n;i++){
      if(i==0) printf(" %d^%d",prime[i],factor[prime[i]]);
      else printf(" x %d^%d",prime[i],factor[prime[i]]);
   }
   printf("\n");
}
```

It is simple and straight forward, but takes $O(N\times factorize())$ amount of time. We can do better.

Second - Summation of Each Prime Frequency

So, using this idea our code will look as the following.

```
void prime_factor(int n){
    for(int i=0;i<prime.size()&&prime[i]<=n;i++){
        int p=prime[i];
        int c=0;
        if(n/p){
            while(n/p){
                c+=n/p;
                p*=prime[i];
            }
            factor[prime[i]]=c;
//            printf("%d^%d\n",prime[i],c);
        }
    }
```

```
}
void fact_factor(int n){
//    for(int i=2;i<=n;i++)
    prime_factor(n);
    printf("%d! =",n);
    for(int i=0;i<factor.size();i++){
        if(i==0) printf(" %d^%d",prime[i],factor[prime[i]]);
        else printf(" x %d^%d",prime[i],factor[prime[i]]);
    }
    printf("\n");
}
```

This code factorizes N! as long as we can generate all primes less than or equal to N!. The loop in line 6 runs until npbecomes 0.

This code has 3 advantages over the "First" code.

1. We don't have to write factorize() code.
2. Using this code, we can find how many times a specific prime p occurs in N! in O(logp(N)) time. In the "First" code, we will need to run O(N) loop and add occurrences of p in each number.
3. It has a better complexity for Factorization. Assuming the loop in line 6 runs log2(N) times, this code has a complexity of O(Nlog2(N)). The code runs faster than this since we only loop over primes less than N and at each prime the loop runs only O(logp(N)) times. The "First" code ran with O(N×factorize()) complexity, where factorize() has complexity of O(N√/ln(N√)+log2(N)).

This idea still has a small flaw. So the next one is better than this one.

<u>Three - Better Code than Two</u>

Suppose, we want to find out how many times 1009 occurs in 9×1018!. Let us modify the "Second" code to write another function that will count the result for us.

Simple Code:

```
long long factorialPrimePower(long long n,long long p){
    long long freq=0;
    long long x=n;
    while(x/p){
        freq += x/p;
        x=x/p;
    }
    return freq;
}
```

There might still be inputs for which this code will overflow, but chances for that is now lower.. Now if we send in n=9×1018 and p=1009, then this time we get 8928571428571425 as our result.

If we apply this improvement in our factFactorize() function, then it will become:

```
#include<bits/stdc++.h>
#define mx 1000001
using namespace std;
bool mark[mx];
vector<int>prime;
//vector<int>factor;
map<int,int>factor;
/// Here Sieve Function:
void prime_factor(long long n){
    for(int i=0;i<prime.size()&&prime[i]<=n;i++){
        long long x=n;
        int c=0;
        if(x/prime[i]){
            while(x/prime[i]){
                c+=n/prime[i];
                x=x/prime[i];
            }
            factor[prime[i]]=c;
//            printf("%d^%d\n",prime[i],c);
        }
    }
}
void fact_factor(long long n){
//    for(int i=2;i<=n;i++)
    prime_factor(n);
    printf("%lld! =",n);
    for(int i=0;i<factor.size();i++){
        if(i==0) printf(" %d^%d",prime[i],factor[prime[i]]);
        else printf(" x %d^%d",prime[i],factor[prime[i]]);
    }
    printf("\n");
}
int main(){
    sieve();
    long long n;
    scanf("%d",&n);
    fact_factor(n);
    return 0;
}
```

Big Integer Remainder

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    string s;
    long long int d,reminder=0;
    printf("Enter Dividend And Divisor\n");
    cin>>s>>d;
    for(int i=0;i<s.length();i++){
        reminder=((reminder*10)+(s[i]-'0'))%d;
    }
    printf("Reminder is = %lld\n",reminder);
    return 0;
}
```

Three Intersting Problem

The three leading digits (most significant) and three trailing digits (least significant). You can assume that the input is given such that $\mathbf{n}^{\mathbf{k}}$ contains at least six digits.

Code:

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const double eps = 0.000000001;
int Leading_Digits(int n,int k){
    double x = k*log10(n);
    x-=floor(x+eps);
    x=pow(10.0,x)*100;
    return floor(x+eps);
}
ll Trailing_Digits(ll a,ll p,ll m){
    ll ans=1%m,x=a%m;
    while(p){
        if(p&1) ans=(ans*x)%m;
        x=(x*x)%m;
        p=p>>1;
    }
    return ans;
}
```

```
int main(){
    int t;
    scanf("%d",&t);
    for(int i=1;i<=t;i++){
        int n,k;
        scanf("%d %d",&n,&k);
        int most=Leading_Digits(n,k);
        int lest=Trailing_Digits(n,k,1000);
        printf("Case %d: %3d %03d\n",i,most,lest);
//        printf("Case %d: %3d %3d\n",i,most,lest);
    }
    return 0;
}
```

$^{n}C_{r} * p^{q}$

The number of trailing zeroes:

```
#include<bits/stdc++.h>
using namespace std;
int zero_factorial(int N,int x){
    int counter=0;
    while(N>=1){
        counter+=N/x;
        N=N/x;
    }
    return counter;
}
int zero_pow(int N,int x){
    int counter = 0;
    while(N%x==0){
        counter++;
        N=N/x;
    }
    return counter;
}
```

```
int main(){
    int T,h=0;
    cin>>T;
    while(T--){
    int n,r,p,q;
    cin>>n>>r>>p>>q;
    int i = zero_factorial(n, 2);
    int j = zero_factorial(n, 5);
    int k = zero_factorial(r, 2);
    int l = zero_factorial(r, 5);
    int m = zero_factorial(n-r, 2);
    int mm = zero_factorial(n-r, 5);
    int px = q*zero_pow(p, 2);
    int py = q*zero_pow(p, 5);
    int ans = min(i-k-m+px,j-l-mm+py);
    cout<<"Case "<<++h<<": "<<ans<<"\n";
    }
    return 0;
}
```

**You task is to find minimal natural number N, so that N! contains exactly Q zeroes on the trail in decimal notation.*

Code:

```
long long int findnum(long long int n){
    if(n==1) return 5;
    long long int low = 0,high =
5*n,ans=0,temp;
    while(low<=high){
        long long int mid = (low+high)/2;
        temp = check(mid);
        if(temp<n) low=mid+1;
        else if(temp>n) high=mid-1;
        else{
            ans = mid;
            high = mid-1;
        }
    }
    return ans;
}
```

```
int check(long long int mid){
    long long int c=0,i=5;
    while(i<=mid){
        c+=mid/i;
        i*=5;
    }
    return c;
}
int main(){
    long long int t,n,k;
    cin>>t;
    for(int i=1;i<=t;i++){
        cin>>n;
        k = findnum(n);
     if(k==0) cout<<"Case "<<i<<":impossible"<<"\n";
        else cout<<"Case "<<i<<": "<<k<<"\n";
    }
    return 0;
}
```

Number of Trailing Zeroes of Factorial

Code:

Base 10

```
int trailing(long long int n){
    long long int i=5,c=0;
    ///for(long long int i=5;n>=i;i*=5){
    while(i<=n){
        c+=n/i;
        i*=5;
    }
    return c;
}
```

Base 16

```
int cal(int N,int x){
    int counter=0;
    while(x<=N){
        counter+=N/x;
        x*=2;
    }
    return counter;
}
int main(){
    int t;
    cin>>t;
    for(int h=1;h<=t;h++){
        int n; cin>>n;
        int ans = cal(n, 2);
        cout<<"Case "<<h<<": "<<ans/4<<"\n";
    }
    return 0;
}
```

Leading Digits of Factorial

Problem

Given an integer N, find the first K leading digits of $N!$.

Solution:

```
#include<bits/stdc++.h>
using namespace std;
const double eps=.000000001;
int leadingDigitFact(int n,int k){
    double fact=0;
    for(int i=1;i<=n;i++){
        fact+=log10(i);
    }
    double q=fact-floor(fact+eps);
    double b=pow(10,q);
    for(int i=0;i<k-1;i++){
        b*=10;
    }
    return floor(b+eps);
}
```

```
int main(){
    int n,k;
    cin>>n>>k;
    int ans=leadingDigitFact(n,k);
    cout<<ans<<'\n';
    return 0;
}
```

Euler Totient or Phi Function

Given an integer N, how many numbers less than or equal N are there such that they are coprime to N? A number Xis coprime to N if gcd(X,N)=1.

Before we go into its proof, let us first see the end result. Here is the formula using which we can find the value of the ϕ() function. If we are finding Euler Phi of $N=p1^{a_1}p2^{a_2}...pk^{a_k}$, then:

$$\phi(n)=n\times(p_1-1/p_1)\times(p_2-1/p_2)...\times(p_k-1/p_k).$$

Theorem 1: If m and n are coprime, then $\phi(m\times n)=\phi(m)\times\phi(n)$.

Theorem 2: In an arithmetic progression with difference of m, if we take n terms and find their modulo by n, and if n and m are coprimes, then we will get the numbers from 0 to $n-1$ in some order.

Theorem 3: If a number x is coprime to y, then $(x\%y)$ will also be coprime to y.

Solution:

```
int Eular_Phi(int n){
	int ans=n;
	int sqrtn=sqrt(n);
	for(int i=0;i<prime.size()&&prime[i]<=sqrtn;i++){
		if(n%prime[i]==0){
			while(n%prime[i]==0){
				n/=prime[i];
			}
			sqrtn=sqrt(n);
			ans/=prime[i];
			ans*=(prime[i]-1);
		}
	}
	if(n!=1){
		ans/=n;
		ans*=(n-1);
		}
	return ans;
}
```

Euler's Theorem

Theorem - Euler's Theorem states that, if a and n are coprime, then $a^{\phi(n)}\equiv 1 \pmod n$

Here $\phi(n)$ is Euler Phi Function.

Proof

Let us consider a set $A=\{b_1,b_2,b_3...,b_{\phi(n)}\} \pmod n$, where b_i is coprime to n and distinct. Since there are $\phi(n)$elements which are coprime to n, A contains $\phi(n)$ integers.

Now, consider the set $B=\{ab_1,ab_2,ab_3....ab_{\phi(n)}\} \pmod n$. That is, B is simply set A where we multiplied a with each element. Let a be coprime to n.

Lemma - Set A and set B contains the same integers.

We can prove the above lemma in three steps.

1. **A and B has the same number of elements**
Since B is simply every element of A multiplied with a, it contains the same number of elements as A. This is obvious.
2. **Every integer in B is coprime to n**
An integer in B is of form $a \times b_i$. We know that both b_i and a are coprime to n, so ab_i is also coprime to n.
3. **B contains distinct integers only**
Suppose B does not contain distinct integers, then it would mean that there is such a b_i and b_j such that:
$ab_i \equiv ab_j \pmod n$
$b_i \equiv b_j \pmod n$
But this is not possible since all elements of A are distinct, that is, b_i is never equal to b_j.
Hence, B contains distinct elements.

With these three steps, we claim that, since B has the same number of elements as A which are distinct and coprime to n, it has same elements as A.

Now, we can easily prove Euler's Theorem.

$ab_1 \times ab_2 \times ab_3 \ldots \times ab_{\phi(n)} \equiv b_1 \times b_2 \times b_3 \ldots \times b_{\phi(n)} \pmod n$

$a^{\phi(n)} \times b_1 \times b_2 \times b_3 \ldots \times b_{\phi(n)} \equiv b_1 \times b_2 \times b_3 \ldots \times b_{\phi(n)} \pmod n$

$$\therefore a^{\phi(n)} \equiv 1 \pmod n$$

Fermat's Little Theorem

Fermat's Little Theorem is just a special case of Euler's Theorem.

Theorem - Fermat's Little Theorem states that, if a and p are coprime and p is a prime, then $a^{p-1} \equiv 1 \pmod p$

As you can see, Fermat's Little Theorem is just a special case of Euler's Theorem. In Euler's Theorem, we worked with any pair of value for a and n where they are coprime, here n just needs to be prime.
We can use Euler's Theorem to prove Fermat's Little Theorem.
Let a and p be coprime and p be prime, then using Euler's Theorem we can say that:
$a^{\phi(p)} \equiv 1 \pmod p$ (But we know that for any prime p, $\phi(p) = p-1$)
$a^{p-1} \equiv 1 \pmod p$

Conclusion

Both theorems have various applications. Finding Modular Inverse is a popular application of Euler's Theorem. It can also be used to reduce the cost of modular exponentiation. Fermat's Little Theorem is used in Fermat's Primality Test.

Modular Multiplicative Inverse

Problem

Given value of A and M, find the value of X such that $AX \equiv 1 \pmod{M}$.

For example, if $A=2$ and $M=3$, then $X=2$, since $2 \times 2 = 4 \equiv 1 \pmod{3}$.

We can rewrite the above equation to this:

$$AX \equiv 1 \pmod{M}$$

$$X \equiv \frac{1}{A} \pmod{M}$$

$$X \equiv A^{-1} \pmod{M}$$

Modular Inverse of A with respect to M, that is, $X = A^{-1} \pmod{M}$ exists, if and only if A and M are coprime.

Hence, the value X is known as Modular Multiplicative Inverse of A with respect to M.

How to Find Modular Inverse?

First we have to determine whether Modular Inverse even exists for given A and M before we jump to finding the solution. Modular Inverse doesn't exist for every pair of given value.

Existence of Modular Inverse

Modular Inverse of A with respect to M, that is, $X = A^{-1} \pmod{M}$ exists, if and only if A and M are coprime.

Why is that?

$$AX \equiv 1 \pmod{M}$$

$$AX - 1 \equiv 0 \pmod{M}$$

Therefore, M divides $AX-1$. Since M divides $AX-1$, then a divisor of M will also divide$AX-1$. Now suppose, A and M are not coprime. Let D be a number greater than 1 which divides both A and M. So, D will divide $AX-1$. Since D already divides A, D must divide 1. But this is not possible. Therefore, the equation is unsolvable when A and M are not coprime.

From here on, we will assume that A and M are coprime unless state otherwise.

Using Fermat's Little Theorem

Recall Fermat's Little Theorem from a previous post, "Euler's Theorem and Fermat's Little Theorem". It stated that, if $A$and M are coprime and M is a prime, then, $A^{M-1} \equiv 1 \pmod{M}$. We can use this equation to find the modular inverse.

$A^{M-1} \equiv 1 \pmod{M}$ (Divide both side by A)

$A^{M-2} \equiv \frac{1}{A} \pmod{M}$

$A^{M-2} \equiv A^{-1} \pmod{M}$

Therefore, when M is prime, we can find modular inverse by calculating the value of A^{M-2}. How do we calculate this? Using Modular Exponentiation.

This is the easiest method, but it doesn't work for non-prime M. But no worries since we have other ways to find the inverse.

Code: When $\mathbf{M}$ is Prime…!

using namespace std;
int gcd(int a,int m){
   return a==0?abs(m):gcd(m%a,a);
}
int Big_Mod(int a,int p,int m){
   int ans=1%m,x=a%m;
   while(p){
      if(p&1) ans=(ans*x)%m;
      x=(x*x)%m;
      p>>=1;
   }
   return ans;
}
	void Modular_Inverse(int a,int m){
	   int g=gcd(a,m);
	   if(g!=1) cout<<"Inverse doesn't exist"<<'\n';
	   else{
	      cout<<"Modular multiplicative inverse: "<<Big_Mod(a,m-2,m)<<'\n';
	   }
	}
	int main(){
	   int a,m;
	   while(cin>>a>>m){
	      Modular_Inverse(a,m);
	   }
	   return 0;
	}

Using Euler's Theorem

It is possible to use Euler's Theorem to find the modular inverse. We know that:

$A^{\phi(M)} \equiv 1 \pmod{M}$

$\therefore A^{\phi(M)-1} \equiv A^{-1} \pmod{M}$

This process works for any M as long as it's coprime to A, but it is rarely used since we have to calculate Euler Phi value of M which requires more processing. There is an easier way.

Using Extended Euclidean Algorithm

We are trying to solve the congruence, $AX \equiv 1 \pmod{M}$. We can convert this to an equation.

$AX \equiv 1 \pmod{M}$

$AX + MY = 1$

Here, both X and Y are unknown. This is a linear equation and we want to find integer solution for it. Which means, this is a Linear Diophantine Equation.

Linear Diophantine Equation can be solved using Extended Euclidean Algorithm. Just pass ext_gcd() the value of $A$and M and it will provide you with values of X and Y. We don't need Y so we can discard it. Then we simply take the mod value of X as the inverse value of A.

Code: When **M** is not Prime….!

using namespace std;
int Ext_GCD(int a,int m,int *x,int *y){
    if(a==0){///Base case
        *x=0;
        *y=1;
        return m;
    }
    int x1,y1;
    int gcd=Ext_GCD(m%a,a,&x1,&y1);
    *x=y1-(m/a)*x1;
    *y=x1;
    return gcd;
}
	void Modular_Inverse(int a,int m){
	    int x,y;
	    int g=Ext_GCD(a,m,&x,&y);
	
	    if(g!=1) cout<<"Inverse doesn't exist"<<'\n';
	    else{
	        int ans=(x%m+m)%m;
	        cout<<"Modular Multiplicative Inverse is: "<<ans<<'\n';
	    }
	}

Modular Inverse from 1 to N

Problem

Given N and M (N<M and M is prime), find modular inverse of all numbers between 1 to N with respect to M.

Code:

```
#include<bits/stdc++.h>
#define mx 10000001
using namespace std;
int m_inverse[mx];
int main(){
        int N,M;
        cin>>N>>M;
        m_inverse[1]=1;
        for(int i=2;i<=N;i++){
                m_inverse[i]=(-(M/i)*m_inverse[M%i])%M;
                m_inverse[i]=m_inverse[i]+M;
        }
        for(int i=1;i<=N;i++){
                cout<<m_inverse[i]<<'\n';
        }
        return 0;
}
```