Number Theory in Competitive Programming  
  
GCD, LCM, Euclidean Algorithm  
  
The definitions of GCD and LCM are well-known, (and taught in middle school I think) I will skip the definitions.  
Also, since $\text{lcm} (a,b) \cdot \text{gcd} (a,b) = ab$, calculating GCD is equivalent to calculating LCM.  
Now, how do we calculate the GCD of two numbers?  
  
A naive solution would be iterating over all positive integers no more than $\text{min} (a,b)$.  
This will get the GCD in $O(\text{min}(a,b))$, very very slow.  
  
We can calculate the GCD of $a,b$in $O(\log ab)$using Euclidean Algorithm.  
This algorithm uses the easy-to-prove fact $\text{gcd}(a,b) = \text{gcd} (b, r)$, where $r$is the remainder when $a$is divided by $b$, or just a%b.  
  
We can now use the following code.

#include <iostream>

using namespace std;

int gcd(int u, int v)

{

return u%v==0?v:gcd(v,u%v);

}

int main(void)

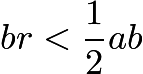
{

int x, y;

cin>>x>>y;

cout<<gcd(x,y);

}

How do we prove that this algorithm is $O(\log ab)$? Well, let's suppose that we started with $(a,b)$.  
Then, we go to $(b,r)$, where $r$is defined similarly as above. It can be proved that .  
Therefore, the product of two numbers in the function decreases by half every time. Done!  
  
Here's a challenge. Can we find the numbers $x, y$such that $ux+vy=\text{gcd} (u,v)$?  
There exists infinitely many pairs - this is Bezout's Lemma. The algorithm to generate such pairs is called Extended Euclidean Algorithm.  
  
Generating Primes, Prime Test, Prime Factorization  
  
Generating primes fast is very important in some problems.  
Let's cut to the chase and introduce Eratosthenes's Sieve.  
  
The main idea is the following. Suppose we want to find all primes between 2 and 50.  
Iterate from 2 to 50. We start with 2. Since it is not checked, it is a prime number. Now check all numbers that are multiple of $2$**except** 2. Now we move on, to number 3. It's not checked, so it is a prime number.  
Now check all numbers that are multiple of $3$, **except** 3. Now move on to 4. We see that this is checked - this is a multiple of 2! So 4 is not a prime. We continue doing this.  
  
Here's the implementation.

#include <stdio.h>

int primechk[21000];

void preprocess(void)

{

int i, j;

for(i=2 ; i<=20000 ; i++)

{

primechk[i]=1;

}

for(i=2 ; i<=20000 ; i++)

{

if(primechk[i]==1)

{

for(j=2 ; i\*j<=20000 ; j++)

{

primechk[i\*j]=0;

}

}

}

}

int main(void)

{

preprocess();

int i, cnt=0;

for(i=2 ; i<=20000 ; i++)

{

if(primechk[i]==1)

{

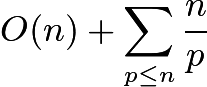
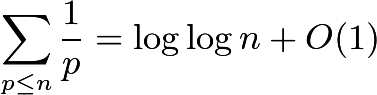
cnt++;

printf("%d is the %dth prime!**\n**",i,cnt);

}

}

}

Okay, so what is the time complexity? To get all primes in the interval $[1,n]$, the TC of this algorithm is $O(n \log \log n)$  
Very very fast. To prove this, notice that the number of iterations are something like where $p$is a prime.  
  
Well, **Merten's Second Theorem** states that (natural logarithm, by the way) so this prove the TC.  
  
Now we know how to generate prime numbers fast. How about primality testing?  
  
Naive solutions first. Given an integer $n$, we can check numbers up to $\sqrt{n}$to find if a number divides $n$. If there is such a number, $n$is composite. If not, $n$is a prime. This gives the solution in $O(\sqrt{n})$.  
  
Here's a "solution" in $O(c\ln n)$using the fast exponentiation we will talk about in the next section. It's called Miller-Rabin Primality Test.  
I'll introduce the deterministic version. We probably (hopefully) won't see stuff like $n> 4 \cdot 10^9$in contests.  
  
Here's the sketch of the algorithm. Choose some set of $a$. We will run the algorithm with different $a$s, and the more $a$s we run this algorithm with, the more accurate this primality test is going to be.  
  
Decompose $n-1$as $2^s \cdot d$. Then check if the following holds.  
$$a^d \not\equiv 1 \pmod{n} \text{   and   }a^{2^rd} \not\equiv -1 \pmod{n} \text{    for all    } r \in [0,s-1]$$If there is an $a$that satisfies this, $n$is composite. If not, $n$is a prime.  
  
For $n<4.7 \cdot 10^9$, we can just check for $a=2,7,61$and be sure about it.  
For $n<2^{64}$, we can check for $a=2,3,5,7,11,13,17,19,23,29,31,37$and be confident.  
  
Now let us look at prime factorization of numbers.  
  
Assume that we generated prime numbers using the Eratosthenes's Sieve.  
If $n$is in the "prime-generated" range, we can actually do prime factorization in $O(\log n)$.  
Make another array. While we are doing the Sieve, for composite numbers, put "the first prime that verified that this number is composite" and for prime numbers, put itself. This is easy to implement.  
  
Then we can start with $n$, and continue to divide the prime numbers in the array.

#include <stdio.h>

int primechk[21000];

int fprime[21000];

void preprocess(void)

{

int i, j;

for(i=2 ; i<=20000 ; i++)

{

primechk[i]=1;

}

for(i=2 ; i<=20000 ; i++)

{

if(primechk[i]==1)

{

fprime[i]=i;

for(j=2 ; i\*j<=20000 ; j++)

{

primechk[i\*j]=0;

if(fprime[i\*j]==0)

{

fprime[i\*j]=i;

}

}

}

}

}

int main(void)

{

preprocess();

int n;

scanf("%d",&n);

while(n!=1)

{

printf("%d**\n**",fprime[n]);

n=n/fprime[n];

}

}

If not, we can just check through all primes less than $\sqrt{n}$and divide by those primes until we can't.  
If all the primes multiply up to $n$, we are done. If not, there is exactly one prime more than $\sqrt{n}$that divides $n$.  
  
Another algorithm involving prime factorization is Pollard's rho algorithm - since the pseudocode is simple, I'll leave you the wikipedia link. <https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm#Algorithm>  
  
Playing with Modulars, and Euler Phi Function  
  
Quick stuff first, fast exponentiation in logarithm time.  
  
Let us calculate $a^b$in modular $m$in $O(\log b)$.  
It uses binary expansion of $b$, and is very very straightforward.

ll exp(ll x, ll n)

{

if(n==0) return 1;

if(n==1) return x;

if(n%2==0) return exp((x\*x)%mod,n/2);

if(n%2==1) return (x\*exp((x\*x)%mod,n/2))%mod;

}

Now, let us talk about modular inverses.  
  
By using Extended Euclidean Algorithm, we can get the inverse of $a$modulo $m$.

#include <iostream>

int inv(int a, int m)

{

int temp=m, q, t, u=0, v=1;

if(m==1) return 0;

while(a>1)

{

q=a/m;

t=m;

m=a%m;

a=t;

t=u;

u=v-q\*u;

v=t;

}

if(v<0) v+=temp;

return v;

}

int main(void)

{

int a, m;

std::cin>>a>>m;

std::cout<<inv(a,m);

}

Of course, logarithm time. If $m$is prime, we can do a lot of different things.  
  
Fermat's Little Theorem gives $a^{p-1} \equiv 1 \pmod{p}$if $(a,p)=1$, where $p$is a prime.  
Therefore, we can calculate the modular inverse of $a$as $a^{p-2}$, by fast exponentiation.  
Time Complexity is $O(\log p)$.  
  
Also, you can get the modular inverse of all numbers in $[1,n]$in $O(n)$.  
The code for this is shown below. The proof for the correctness is left to the reader (not difficult)

#include <iostream>

typedef long long ll;

using namespace std;

int inv[111111], n;

ll mod=1e9+7;

int main(void)

{

cin>>n;

int i;

inv[1]=1;

for(i=2 ; i<=n ; i++)

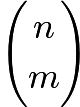
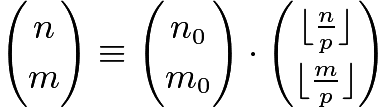
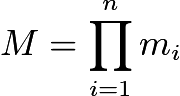
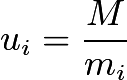
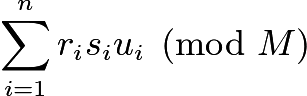
{

inv[i]=((mod-mod/i)\*inv[mod%i])%mod;

cout<<inv[i]<<endl;

}

}

We can also calculate in modulo $p$($p$ is a prime) very fast using Lucas' Theorem.  
  
Lucas' Theorem basically states that , where $n_0$is $n$modulo $p$and $m_0$is $m$modulo $p$.  
  
This is very efficient when $p$is small and $n, m$is huge. We can precalculate the factorials and inverse of factorials modulo $p$by using the above code, and solve each queries in $O(\log_p \text{max} (n,m))$.  
  
Also, we can use Chinese Remainder Theorem to solve a system of modular equations.  
  
Let us solve $x \equiv r_i \pmod {m_i}$, where $m_i$are pairwise coprime.  
(If they are not coprime, break them into prime powers, and if some are contradictory, there are no solutions.)  
  
The CRT itself gives an algorithm to get our answer.  
Set , and . Also, set $s_i$as the modular inverse of $u_i$in modulo $m_i$. Then our answer is   
  
We learned how to calculate modular inverse in logarithm time above. So the time complexity is $O(n \log MAX)$.

long long int r[111111]; // remainders

long long int m[111111]; // modulars

long long int M=1; // product

int n; // number of equation

int res(void)

{

int i;

for(i=1 ; i<=n ; i++)

{

M=M\*m[i];

}

long long int ret=0;

for(i=1 ; i<=n ; i++)

{

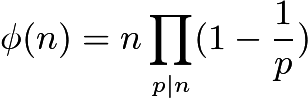
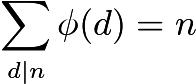
ret+=r[i]\*inv(M/m[i],m[i])\*(M/m[i]);

ret=ret%M;

}

return ret;

}

$\phi (n)$is the number of positive integers no more than $n$which is coprime with $n$.  
Formula is . Proof is Inclusion-Exclusion.  
Also, we have the formula .  
  
Of course, for the calculation of Euler Phi numbers, we can tweak the Eratosthenes's Sieve algorithm a little bit.

void preprocess(void)

{

int i, j;

eulerphi[1]=1;

for(i=2 ; i<=122000 ; i++)

{

eulerphi[i]=i;

primechk[i]=1;

}

for(i=2 ; i<=122000 ; i++)

{

if(primechk[i]==1)

{

eulerphi[i]-=eulerphi[i]/i;

for(j=2 ; i\*j<=122000 ; j++)

{

primechk[i\*j]=0;

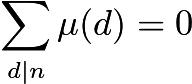
eulerphi[i\*j]-=eulerphi[i\*j]/i;

}

}

}

}

You could also calculate $\phi (n)$by using prime factorization of $n$.  
  
Now let's get to the fun stuff. The mobius function.  
  
Mobius Function Part 1. The Introduction  
  
What is mobius function? This function is notated as $\mu (n)$. This function has lots of definitions.  
However, the main definition is the following.  
  
$\mu (n)$is $1$if $n=1$or $n$is square-free and has even number of prime divisors.  
$\mu (n)$is $-1$if $n$is square-free and has odd number of prime divisors.  
$\mu (n)$is $0$if $n$is not square-free.  
  
$\mu (n)$also has a lot of interesting properties that make $\mu (n)$so important.  
  
for all $n>1$, and $\mu (n)$is multiplicative.  
(A function $f$is multiplicative if $f(mn)=f(m)f(n)$for all $(m,n)=1$.)  
  
How do we calculate $\mu (n)$fast? Again, we can tweak the Eratosthenes's Sieve a little bit.

void preprocess(void)

{

int i, j;

for(i=1 ; i<=111100 ; i++)

{

mu[i]=1;

primechk[i]=1;

}

primechk[1]=0;

for(i=2 ; i<=111100 ; i++)

{

if(primechk[i]==1)

{

mu[i]=-mu[i];

for(j=2 ; i\*j<=111100 ; j++)

{

primechk[i\*j]=0;

if(j%i==0)

{

mu[i\*j]=0;

}

else

{

mu[i\*j]=-mu[i\*j];

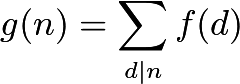
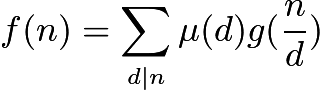
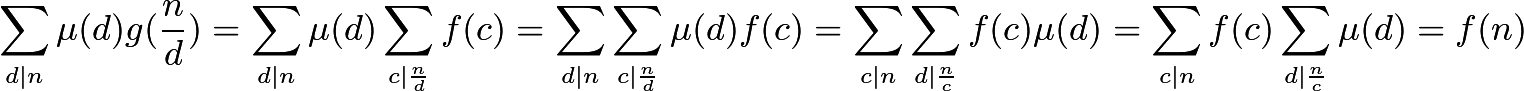
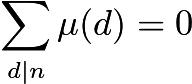
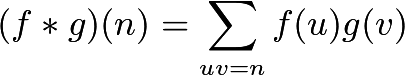
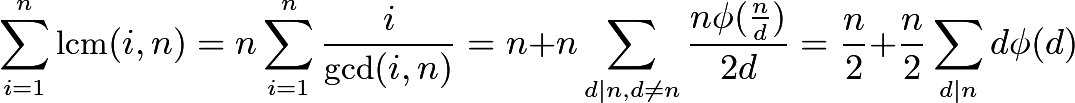
}

}

}

}

}

Okay, $O(n \log \log n)$is good. Now, how do we use $\mu (n)$?  
  
Mobius Function Part 2. Mobius Inversion, Dirichlet Convolution  
  
**Theorem.** Let $f(n), g(n)$be arithmetic function such that . Then, .  
  
**Proof.** using for all $n>1$and $\mu (1)=1$.  
  
The Dirichlet Convolution of two function $f, g$is the following.  
  
Denote $1$as the constant function, $f(n)=1$.  
$\epsilon (n)$is a function which satisfies $\epsilon (1)=1$and $\epsilon (n) = 0$for $n \not= 1$.  
$Id(n)$is the identity function, $Id(n)=n$  
  
The mobius inversion formula and the basic property of mobius functions give $1 * \mu = \epsilon$, and $g = f * 1 \iff f=g * \mu$  
  
Okay. So what? How can we use this formula to ~~improve our lives~~ solve problems?  
  
We can, change our "sum" or our answer using mobius inversion and mobius functions to calculate them fast.  
  
I will give 3 examples which changes the desired sum by using number theory (like mobius function) to calculate them fast.  
  
Example Problems  
  
Problem 1. <http://www.spoj.com/problems/LCMSUM/>  
  
  
Let $res[x]$be the answer when $n=x$.  
Precalculate $\phi (d)$, and add $d \phi (d)$to every $res[i \cdot d]$. Then multiply $\frac{x}{2}$to $res[x]+1$.  
This gives our solution in $O(n \log n + T)$. yay

#include <stdio.h>

#include <iostream>

using namespace std;

typedef long long int ll;

ll res[1000010];

ll phi[1000010];

void preprocess(void)

{

for(int i=1 ; i<=1000000 ; i++)

{

phi[i]=i;

}

for(int i=2 ; i<=1000000 ; i++)

{

if(phi[i]==i)

{

for(int j=1; i\*j<=n; j++)

{

phi[i\*j]-=phi[i\*j]/i;

}

}

}

for(int i=1 ; i<=1000000 ; i++)

{

for(int j=1; i\*j<=1000000 ; j++)

{

res[i\*j]+=i\*phi[i];

}

}

}

int main(void)

{

preprocess();

int tc;

cin>>tc;

while (tc--)

{

ll n;

scanf("%lld",&n);

ll ans=res[n]+1;

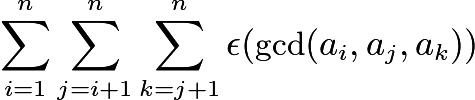
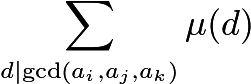
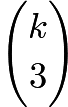
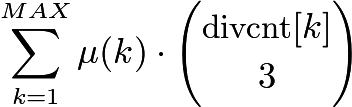
ans=(ans\*n)/2;

printf("%lld**\n**",ans);

}

return 0;

}

2. <https://www.codechef.com/LTIME13/problems/COPRIME3>  
  
Here's the key sum transformation.  
We want .  
Using $1 * \mu = \epsilon$, we can transform this sum. I'll explain this step-by-step.  
  
Start with $\epsilon ( \text{gcd} (a_i,a_j,a_k) )$as .  
Now let's decide how many $\mu (d)$appears in this sum. Clearly, it is the number of 3-tuples $(i, j, k)$such that $d|a_i, d|a_j, d|a_k$. Therefore, if $k$is the number of $a_i$s which is a multiple of $d$, $\mu (d)$appears times.  
  
Therefore, our result is , where $\text{divcnt}[k]$is the number of $a_i$s which are a multiple of $k$.  
  
We can precalculate binomial numbers, $\mu (k)$s, and $\text{divcnt}[k]$easily in $O(MAX \log MAX)$, where $MAX$is the maximum of $a_i$s.

#include <stdio.h>

#include <algorithm>

#include <vector>

#include <string.h>

#include <string>

#include <stack>

#include <queue>

#include <iostream>

#include <assert.h>

#include <math.h>

using namespace std;

typedef long long int ll;

ll primechk[111111];

ll mu[111111];

ll divcnt[111111];

ll cnt[111111];

ll com[333][333];

ll ans, mod=1e7+3;

int n;

void input(void)

{

int i, j, x;

cin>>n;

for(i=1 ; i<=n ; i++)

{

scanf("%d",&x);

cnt[x]++;

}

}

void divpproc(void)

{

int i, j, x;

for(i=1 ; i<=110000 ; i++)

{

for(j=1 ; i\*j<=110000 ; j++)

{

divcnt[i]+=cnt[i\*j];

}

}

}

void compproc(void)

{

com[0][0]=1;

int i, j;

for(i=1 ; i<=320 ; i++)

{

com[i][0]=1;

com[i][i]=1;

}

for(i=2 ; i<=320 ; i++)

{

for(j=1 ; j<=i-1 ; j++)

{

com[i][j]=(com[i-1][j-1]+com[i-1][j]);

}

}

}

void preprocess(void)

{

int i, j;

for(i=1 ; i<=111100 ; i++)

{

mu[i]=1;

primechk[i]=1;

}

primechk[1]=0;

for(i=2 ; i<=111100 ; i++)

{

if(primechk[i]==1)

{

mu[i]=-mu[i];

for(j=2 ; i\*j<=111100 ; j++)

{

primechk[i\*j]=0;

if(j%i==0)

{

mu[i\*j]=0;

}

else

{

mu[i\*j]=-mu[i\*j];

}

}

}

}

}

void calc(void)

{

int i, j;

for(i=1 ; i<=105000 ; i++)

{

ans=ans+mu[i]\*com[divcnt[i]][3];

}

}

int main(void)

{

preprocess();

input();

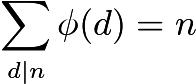
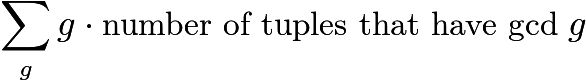
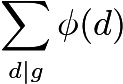
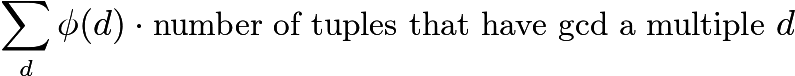
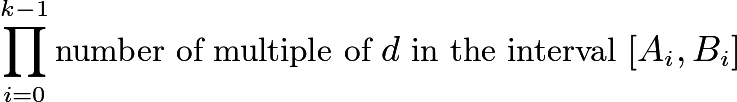
divpproc();

compproc();

calc();

cout<<ans;

}

3. <https://www.codechef.com/COOK29/problems/EXGCD>  
  
Here's a sketch. Calculating denominator + calculating inverse of it is just modular inverse calculation.  
  
The main problem is calculating the sum of all gcds. Using , we can change the sum.  
  
The sum is pretty much .  
  
Now change $g$to . Therefore, we can change the sum to .  
  
But this is way easier to calculate! For the $gcd$to be a multiple of $d$, we can just use . Done!  
  
In the more harder problems, the result of mobius inversion gets more complex, and in some problems we also have to keep track of some prefix sums and use the fact that $\lfloor \frac{n}{i} \rfloor$takes $O(\sqrt{n})$values. But that is for later ~~when I can actually solve those problems~~ .