|  |
| --- |
| Close-up image showing the leaf-sides of two oversized books side-by-side on a bookshelf, with additional books in soft focus background |
| Number Theory Handbook  **Part - I** |
| |  |  |  | | --- | --- | --- | | **Fahim Ahmed** | **1/19/18** | **Not For Sale** | |

1. **Greatest Common Divisor (GCD)**

Given two numbers, our task is to find a numbersuch that, and there is no such number for which is also true.

**Solution Approach:** First let. Now if, then obviously m itself is the greatest common divisor among m, n. If hence take n=m, and m=r, and continue this process, until we find a pair of m, n where When we find such pair, in that case *the then* ‘m’ would be considered the greatest common divisor of (a, b), because m divides all the previous reminders from n/m.

**Code:**

**template** <**typename** t1> t1 gcd(t1 a, t1 b)

{

**while**(b != 0 ){

a=a%b;**swap**(a,b);

}

**return** a;

}

1. **Least Common Multiple (LCM)**

Given two numbers our task is to find a number x such thatand there is no such number is true.

**Solution Approach:** let has multiples and has multiples. **Now, if among all these multiples,** and no other common multiples smaller than exists, then is the least common multiple.

To find out the LCM, at first we find the GCD of (a, b). Let.

Hence if,, then we are fairly certain that.

**Hence,** the *Least Common Multiple (smallest common multiple)*,

**Code:**

**template** <**typename** t1> t1 lcm(t1 a, t1 b)

{

**return** a \* (b / gcd(a, b));

}

1. **Remainder of a BigInteger divided by an Integer**

Given 2 numbers, *x* (a BigInteger) and *d* (the integer to divide with). The task is to find(x modulus d) or the remainder of *(x/d)*.

**Solution Approach:** We know that the number **1234 = 4\*100 + 3\*101 + 2\*102 + 1\*103**.

From the above breakdown, we can simplify it by the following code:

**Code:**

#define BigInteger string

**template** <**typename** t1> t1 BigInteger\_remainder(BigInteger x, t1 m){

t1 remainder=0;

**for**(t1 i = 0; i < x.length();i++)

{

**///getting the digit in integer format from character format**

t1 digit = x[i]-'0';

remainder = (remainder\*(t1)10+digit)%m;

}

**return** remainder;

}

1. **GCD of 2 numbers where one of them is a BigInteger**

Given 2 numbers, where ‘a’ is an integer, and b is a BigInteger (represented in string in C++) we need to find the GCD of these 2 numbers. It is obvious that the GCD is less or equal to ‘a’.

**Solution Approach:** We use a very simple technique to solve this problem. As it is guaranteed that *the GCD of an integer and a BigInteger* *will be less or equal to the integer*, we can simply use the basic GCD (01) algorithm.

**long long** GCD(**long long** number1, **long long** number2)

{

**long long** number =**max**(number1,number2);

**long long** remainder=**min**(number1,number2);

**while**(remainder != 0)

{

number = number % remainder;

**swap**(number,remainder);

}

**///this is the GCD**

**return** number;

}

Just before starting the while loop, we will manually calculate the. We would manually calculate this using BigInteger\_remainder algorithm (03).

#define BigInteger string

**template** <**typename** t1> t1 BigInteger\_remainder(BigInteger x, t1 m){

t1 remainder=0;

**for**(t1 i = 0; i < x.length();i++)

{

**///getting the digit in integer format from character format**

t1 digit = x[i]-'0';

remainder = (remainder\*(t1)10+digit)%m;

}

**return** remainder;

}

**template** <**typename** t1> t1 bigGCD(BigInteger number1, t1 number2)

{

t1 number = number2;

t1 remainder=BigInteger\_remainder(number1,number2);

**while**(remainder != 0)

{

number = number % remainder;

**swap**(number,remainder);

}

**///this is the GCD**

**return** number;

}

1. **Sieve of Eratosthenes (Prime Generator from 1 to limit)**

We can use the Sieve of Eratosthenes to find *all the prime numbers that are less than or equal to a given number N* or to find out whether a number is a prime number.

The basic idea behind the Sieve of Eratosthenes is that *at each iteration one prime number is picked up and all its multiples are eliminated*. After the elimination process is complete, all the unmarked numbers that remain are prime.

**Pseudo code**

* Mark all the numbers as prime numbers except 1
* Traverse over each prime numbers smaller than sqrt(N)
* For each prime number, mark its multiples as composite numbers
* Numbers, which are not the multiples of any number, will remain marked as prime number and others will change to composite numbers.

**Code:**

**///Sieve Prime Generator to generate primes between 1 to 10^7**

**///Prime Number Theorem: upper bound of primes in between n = N/ln(N)**

**int** primes[10000000];

**int** pr\_ctr = 0;

**bitset**<10000010> st(0);

**void** siv(**int** n=10000000)

{

st.**set**(0,1); **///marking 0 as nonprime**

st.**set**(1,1); **///marking 1 as nonprime**

primes[pr\_ctr++] = 2; **///listing 2 as a prime**

**///listing all the even numbers as nonprime**

**for**(**int** i = 4; i <= n; i+=2)

{

st.**set**(i,1);

}

**///Running O(sqrt(n)) iteration**

**int** sqn = sqrt(n);

**for**(**int** i = 3; i <= sqn ; i+=2)

{

**if**(st.test(i) == 0)

{

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

{

st.**set**(j,1);

}

}

}

**///listing all the determined primes**

**for**(**int** i = 3; i <= n; i+=2)

{

**if**(st.test(i) == 0)primes[pr\_ctr++]=i;

}

}

**/// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///**

1. **Procedure to store all *divisors*, *Number of divisors*, *Sum of divisors* of range [1, N]**

So, now I am going to discuss an idea about how we can count all the *divisors, Number of divisors, Sum of divisors* of all the numbers in between [1, N] within time complexity. Awesome right? Fun fact is I don’t think too many people know this trick. People know the complexity way. But theis not probably known to everybody. I myself figured this trick out in my ICPC regional contest back in 2015. Anyway, the procedure is described below.

We try to use a moderated version of Sieve’s technique to accomplish our task. For the O(NlogN) approach what we do is:

* Iterate from 1 to N.
* For each number,
  + Put into divisor list of .
  + At the time when putting divisor list, remember to increment the **number of divisors (NOD)** for that ‘j’. And also add ‘i’ to **sum of divisors (SOD)** for ‘j’.

Time complexity:

This outer loop takes O(N) instructions.

The inner loop in total takes

Hence, total complexity O(N) + O(N\*log(N)) ~ **O(NlogN).**

So, how to speed it up totime complexity? Well, you will figure it out yourself once you see the implementation I think.

**Code:**

**///ALL DIVISORS, NOD of a range, SOD of a range:**

**vector**<**int**> divisors[20000002];

**int** Number\_of\_divisors[20000002] = {0}; **///NOD**

**int** Sum\_of\_divisors[20000002] = {0}; **///SOD**

**void** all\_divisors(**int** n)

{

**int** i,j,k;

**int** sq = sqrt(n);

Iterating up to rather than N

**for**(i = 1; i <= sq; i++)

{

**for**(j = i; j <= n; j+=i)

{

divisors[j].**push\_back**(i);

Number\_of\_divisors[j]++;

Sum\_of\_divisors[j]+=i;

k = j/i;

**if**(k > sq)

{

Check out this part. This is how we reduced complexity from   
.

divisors[j].**push\_back**(k);

Number\_of\_divisors[j]++;

Sum\_of\_divisors[j]+=k;

}

}

}

}

**/// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///**

1. **Sum of SOD for range [1, N] incomplexity.**

Using the above procedure described in **(06)**, we can calculate individual Sum of Divisors **(SOD)** for all numbers in range [1, N] using time complexity.

To find *Sum of SOD* for a range [1, N], simply run a **O(N) iteration** to keep **cumulative sum** of SOD’s for every number .

**///SUM of SOD from i to j; O(sqrt(n) \* log2(n))**

**///SUM OF MAX\_LCM = SUM\_OF\_SOD**

**///This is suitable if 104 testcase 10^5 and timelimit >= 3 seconds**

li Sum\_SOD[20000001];

**void** SUM\_of\_SOD(**int** start, **int** fin)

{

Refers to method of (06)

all\_divisors(fin);

Sum\_SOD[start] = Sum\_of\_divisors[start];

**for**(**int** i = start+1; i <= fin; i++)

{

{Sum\_SOD[i] = Sum\_SOD[i-1] + Sum\_of\_divisors[i];}

}

}

**/// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///**

1. **Sum of Sum of Proper Divisors of a range [1, N]**

Proper divisor: For any number ‘x’, proper divisor would mean all of ‘x’s divisors except 1 and x.

That is why *1 and all the prime numbers don’t have any proper divisors.*

Here the given problem is:

You will be given a number N. The task is to count theor the SUM of (sum of proper divisor) for all number in range [1, N]. This task can be solved in complexity by the way.

As we will be checking for proper divisors only, we would run the iteration from 2 to . The reason behind running it upto is that if any number has at least 1 proper divisor it would be less or equal to .

The steps are:

* Sum\_of\_SOPD = 0
* for

This is how we can calculate sum of (Sum of Proper Divisors) of numbers in range from 1 to N. ***I do not have a clear confident understanding of how this works at this time, hence no explanation is to be written.***

#define li long long int

**///SUM\_of\_sum\_of\_proper\_divisors of a range 1 to n (PROPER DIVISORS)**

**///Caution: This doesn't calculate the SSOPD of sub-problems**

**/// O(sqrt(n))**

li sum\_of\_SOPD(li n)

{

li ans=0;

**for**(li i=2;i\*i<=n;i++)

{

li j=n/i;

ans+=(i+j)\*(j-i+1)/2;

ans+=i\*(j-i);

}

**return** ans;

}

**/// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///**

1. **Sum of Sum of Divisors of a range [1, N] in time complexity**

Although **(06)** and shows us a way to calculate, and in complexity, in this section we would see another way to do the same in complexity.

In the previous section (08), we saw a method to find the sum of **SUM OF PROPER DIVISORS** for a range [1, N] within O() time. ***Notice the fact that,*** to find the sum of divisors , we can simply ***add 1*** and ***add ‘i’*** with each ‘i’s sum of proper divisor.

*Because, when we calculate proper divisor for any number ‘x’, we simply exclude 1 and x from its complete divisor list.*

So the steps would be:

* Calculate the Sum of (Sum of Proper Divisors) of [1, N] using algorithm from (08) in complexity.
* Then
  + **(Because 1 is a divisor for each element in [1, N] )**
  + ; **(In case of 1, it has only 1 improper divisor.)**
* But the thing to notice here is that, if we ran a loop to add (1 and ‘i’) for each it would turn into a complexity. So we try a bit smarter move. **We are eventually doing this after calculating Sum\_SOPD:**
* So, we simply calculate Sum of (Sum of Proper Divisors) of [1, N] using algorithm from (08) in complexity. Then add and this summation would be the Sum\_of\_SOD for range [1, N].

#define li long long int

**///SUM\_OF\_sum\_of\_all\_divisors of a range of 1 to n**

**///Suitable when O(testcase\*sqrt(n)) <= 10^7**

li sum\_of\_SOD(li n)

{

//sum of all proper divisors between [1,n]

li SSOPD = sum\_of\_SOPD(n);

//adding 1 for all [1,n] numbers.

li SSOD = SSOPD + (n);

//adding all i 's for (2<=i<=n);

//as i was not a proper divisor of the number i itself

**///which means adding (2 + 3 + .... +n)**

**/// = (1 + 2 + 3+ .... + n) - 1**

**/// = [(n\*(n+1))/2)] - 1**

SSOD+=((n\*(n+1))/2)-1;

**return** SSOD;

}

**/// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///**

Hence, we learn two way to calculate Sum\_SOD from ***(07) and (09)*.** Now we need to know when to use which one.

* The (09) though calculates Sum\_SOD in O(complexity, it doesn’t at the same time calculate the solutions for its sub-problems (for range [1, M] where M < N). Hence for each distinct value of N, it would have to be recalculated. **Hence use this only when .** Considering time limit is 1s and instruction per second = 107.
* The (07) though takes much longer time complexity, **BUT** *if you calculate the* ***Sum\_SOD*** *for the maximum value of N for once,* as it stores in cumulative sum format, the solutions for all other values less than N can be found from the cumulative array as well. So no matter how many testcases they give you, you only need 1 pre-calculation for the maximum possible value of N, and from there, just answer in O(1) complexity for any test-case. Follow this procedure if and only if . (Considering time limit is 1s and instruction per second = 107.

1. **Prime factorization of a single integer**

**In order to prime factorize an integer x, at first we need to generate primes that are less or equal to .** The smart approach is to generate primes up to (*where, L is the constraint for x :* ) for once, and use that prime list over and over again.

The prime generation would use Sieve prime generation mentioned in (05) which would take complexity. Here, N denotes the range’s limit for prime generation. Hence the practical complexity would be .

After the prime generation is complete, we simply proceed according to the following lines.

**///Prime factorization of a single integer, O((sqrt(n)/(logn))+(logn))**

**vector**<**int**> factors;

**void** factorize( **int** n )

{

**int** sqrtn = sqrt ( n ) ;

**for** ( **int** i = 0; i < pr\_ctr && primes[i] <= sqrtn; i++ )

{

**if** ( n % primes[i] == 0 )

{

**while** ( n % primes[i] == 0 )

{

n /= primes[i] ;

factors.**push\_back**(primes[i]) ;

}

**sqrtn = sqrt ( n ) ;**

}

}

**if** ( n != 1 ) **///that means current value of n itself is a prime > sqrtn**

{

factors. **push\_back**(n) ;

}

}

//factors would contain the prime factorization of the integer n

//in practical case the complexity can be counted as O(sqrt(n)+logn)

/\*

Additional Info on this:

1. to factorize n, you need integers only upto floor(sqrt(n))

\*/

**/// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///**

Total Complexity: + .

*\*To prime factorize you would only need to* ***generate primes upto .***

1. **Number of Divisors for a single integer (NOD)**

To find the number of divisors of a single integer, we would at first need to prime factorize it **(10)**. Now let a number (Where P1, P2, P3 are all primes); then, the number of divisors of X would be (a1+1)\*(a2+1)\*(a3+1).

Why number of divisors = (a1+1)\*(a2+1)\*(a3+1)? Let me explain with an example.

Suppose .

Let, Set with all of 2’s powers that divide 60, A = {20, 21, 22}

Let, Set with all of 3’s powers that divide 60, B = {30, 31}

Let, Set with all of 5’s powers that divide 60, C = {50, 51}

*Now, notice that the cross multiplication of these 3 sets would produce all the divisors of 60 including 60 and 1 itself.*

To verify it, we cross multiply the three sets.

S=A x B x C

* S = {(20, 30), (20, 31), (21, 30), (21, 31), (22, 30), (22, 31)} x {50, 51}
* S = {(20, 30, 50), (20, 31, 50), (21, 30, 50), (21, 31, 50), (22, 30, 50), (22, 31, 50), (20, 30, 50), (20, 31, 51), (21, 30, 51), (21, 31, 51), (22, 30, 51), (22, 31, 51)}

Now, we check *each element tuple* from the *cross multiplied* set, and find the ***product of the tuple’s values.***

These above 12 in-fact are the only divisors of 60. Hence, now we can clearly see why the cross multiplication of all the primes possible powers provide all the divisors of the number.

The code is given below. This one also has the same complexity as the prime factorization one.

**+ .**

**///NUMBER OF DIVISOR OF A SINGLE INTEGER:/// (NOD)**

**///UPPER BOUND FOR NOD(n) = [2 x n^(1/3)]**

**int** NOD ( **int** n )

{

**int** sqrtn = sqrt ( n );

**int** res = 1;

**for** ( **int** i = 0; i < pr\_ctr && primes[i] <= sqrtn; i++ ) {

**if** ( n % primes[i] == 0 ) {

**int** p = 0; /\*Counter for power of prime\*/

**while** ( n % primes[i] == 0 ) {

n /= primes[i];

p++;

}

sqrtn = sqrt ( n );

p++;/\*Increase it by one at end for prime^0\*/

res \*= p; /\*Multiply with answer\*/

}

}

**if** ( n != 1 ) {

res \*= 2; /\*Remaining prime has power p^1. So {p^0, p^1} 2 elements\*/

}

**return** res;

}

**/// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///**

1. **Sum of Divisors for a single integer (SOD)**

To find the number of divisors of a single integer, we would at first need to prime factorize it **(10)**. Now let a number . If you understood the last section (11) then now you know that *all the divisors can be found from the cross multiplication of all the primes possible powers in the number X.*

So let 6 = .

Hence, available primes and their possible powers in 6 = .

Hence, all the divisors from cross multiplication:

So Sum of Divisors, SOD = (1+3+2+6) =

Hence, for any number. The **SOD =**.

**///SUM OF DIVISOR OF A SINGLE INTEGER: ///SOD**

**///if N = p1^a1 \* p2^a2;**

**///then SOD(N) = (p1^0 + p1^1 + p1^2 + ... + p1^a1) x (p2^0 + p2^1 + p2^2 + .... + p2^a2)**

**int** SOD( **int** n )

{

**int** sum\_of\_divisors = 0;

**int** sqrtn = sqrt (n) ;

**for** ( **int** i = 0; i < pr\_ctr && primes[i] <= sqrtn; i++ )

{

**if** ( n % primes[i] == 0 )

{

**int** primesum = 1;

**int** p\_mul=1; //serves as pi^0, pi^1,...,pi^ai

**while** ( n % primes[i] == 0 )

{

n /= prime[i] ;

p\_mul\*=prime[i];

primesum+=p\_mul;

}

sqrtn = sqrt ( n ) ;

sum\_of\_divisors\*= primesum;

}

}

**if** ( n != 1 )

{

**int** primesum = 1 + n;

sum\_of\_divisors\*=primesum;

}

**return** sum\_of\_divisors;

}

**/// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///**

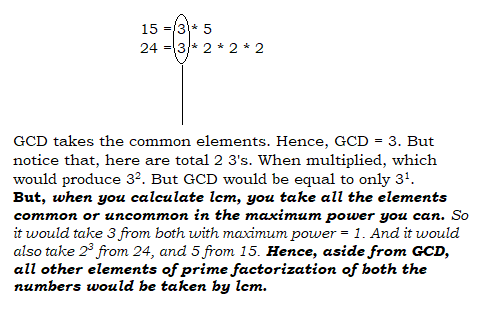
1. **Given two number’s GCD G, and LCM L; how to find the product of the two numbers?**

Let are two numbers. And . Then the product of a,b which is .

Explanation:

Let, **a = 15, b=24**. Then .

Now, product of a,b . If we look at the following figure:

Hence, among 2 numbers, if GCD takes a common part from both, all the other uncommon parts would be taken into LCM. **Hence, multiplying GCD\*LCM would result into multiplication of prime factorization of both numbers a and b.**

1. **Given two number’s GCD G, and LCM L; how to find the two numbers?**

Let .

Now we know from **(13);**

**G\*L = a\*b.**

Now, can be possible if and only if in other words if G is divisible by L. If ; then there cannot exist any pair of (a,b) such that their gcd is G and lcm is L. Else the pair is (G, L) themselve.

**///Pair of integers whose gcd is G and LCM is L**

**pair**<**int**,**int**> pair\_gcd\_lcm(**int** G, **int** L)

{

**int** a, b;

a = G;

//gcd(a,b) \* lcm(a,b) = a\*b

//if lcm % gcd != 0, that means no a,b exists such that (a\*b) = G \* L

//if L % G == 0 ; that means G\*L = a\*b

//we can simply put a = G, b = L (we could swap a,b, but given condition is a<=b)

**if** ( L % G != 0 ) {

**return** make\_pair(-1,-1);

}

b = L;

**return** make\_pair(a,b);

}

**/// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///**

1. **The largest value of such that**

Find . *Now L is the minimum number which is divisible by both a and b.* We need to check *exactly which multiple of L is strictly less or equal to N*. and that would be the answer.

Hence, Hence

1. **Prime Factorization of factorial(N)**

Given N, we have to *prime factorize factorial(N) or (N!)*.

At first, we have to generate primes <= N using Sieve.

Then

Check how many appearances of that prime can be in N!.

How to do that? Let me demonstrate.

Let, we want to prime factorize factorial(15).

Now let’s list the primes within 15. They are 2, 3, 5, 7, 11, 13.

* **Now,** we will check how many occurrence of 2 are in .
* To check how many are there from 1 to 15 (), we simply take the = 7. ***Hence,*** *in prime factorization of 15! There would be at least present.*
* But, because of numbers like 4, 8 which have more than inside it, there would be some more 2’s in the prime factorization of 15!.
* So, each present in 1 to 15 would present one more 2 each.
* Each present in 1 to 15 would present one more 2 each.
* We would keep checking higher powers of a prime as long .
* In this case . So we would check for upto .
* So there are of 22 present within 1 to 15. Hence power of 2 increase by 3 for each 22. (*as we already counted this numbers for 21 once,* ***we are only adding the increased power from last time, hence for each, only 1 power is being added****).* Hence, currently at least (7+3) = 10 powers of 2 are present in 15!.
* Same way, , hence 1 more power of 2 would be added.
* In total 7+3+1 = 10 powers of 2 would be in 15!.

This way, we continue checking for all the other primes 15.

**Code:**

**///keeps the (prime, power) pair of each factor**

**vector**<**pair**<**int**,**int**> > factors;

**void** factorial\_factorize(**int** n){

factors.**clear**();

**int** prime\_size=primes.**size**();

**for**(**int** i=0; i < prime\_size **and** primes[i] <= n; i++)

{

**if**((n/primes[i]) > 0)

{

**int** power=0;

**///first checking prime^1;**

**///later power would gradually increase**

**int** prime\_multiple = primes[i];

**while**((n/prime\_multiple) > 0)

{

power+=(n/prime\_multiple);

**///raising the power of prime^power**

prime\_multiple\*=primes[i];

}

**pair**<**int**,**int**> P = make\_pair(primes[i], power);

factors.pb(P);

}

}

}

1. **Co-prime Calculation of a single Integer**

Two numbers are co-primes iff . Function phi(n) is defined as count of how many number from 1 to n are coprime(having gcd value 1) to n. ***For example*** phi(6) is 2 as 1,5 are coprime to 6.

Few properties of phi function :

* phi(p) = p - 1. Where p is prime. All the numbers from 1 to p - 1 are coprime to p.
* phi(a \* b) = phi(a) \* phi(b) where a and b are coprime.
* phi(p^k) = p^k - p^(k - 1). Note that here ^ denotes power. Here all the numbers from 1 to p^k are coprime to p^k except all the multiples of p, which are exactly p^(k -1).

**The principle behind calculation of phi(n)** is, first consider that n has n co-primes. Then find out how many primes n can divide it. And when you find some prime that divides it, simply decrease the amount of co-primes that should be decreased from n due to that prime. **This number is: . Means,** If we considered a number 9 to have 9 co-primes, and we find an integer 3 dividing it, then due to ‘3’, the number of probable co-prime for 9 decreases by (9/3) numbers, or 3 numbers. Hence left would be 6 numbers. We can reach number 6 using formula

**We do this same thing for all possible primes < n.**

**///EULER PHI/SINGLE INPUT COPRIME COUNTER**

**///complexity O(sqrt(n))**

**int** EulerPhi(**int** n){

**if**(n==1){

**return** 0;

}

**int** sqn=sqrt(n);

**int** phiCounter=n;

**for**(**int** i=0; i < primes.**size**() && primes[i] <= sqn; i++){

**if**(n%primes[i] == 0){

**while**(n%primes[i] == 0){

n/=primes[i];

}

phiCounter = (phiCounter / primes[i]) \* (primes[i]-1);

**///update the current value of sqrt(N) according to current N;**

sqn=sqrt(n);

}

}

**if**(n != 1)**///that means n currently itself is a prime number**

{

phiCounter = (phiCounter / n) \* (n-1);

n/=n;

}

**return** phiCounter;

}

**int** main(){

sieve(31623); **///because we need primes upto sqrt(max) = sqrt(10e9) = 31622**

**int** n;

**while**(scanf("%d", &n) && n != 0){

printf("%d\n", EulerPhi(n));

}

}

1. **Calculating Number of Co-prime for a range [1, N] using sieve (Euler Sieve Phi)**

Below we describe the procedure to calculate phi(n) for a range of number [1, N] using Sieve technique.

If you don’t understand, just simulate yourself with a small sample size, & you will get it.

**///EULER SIEVE PHI/ RANGED COPRIME COUNTER:**

**///complexity O(nlogn)**

//MUST HAVE SIV WRITTEN ABOVE THIS

ui phi[5000001];

**bitset**<5000001> mark(0);

**void** sievephi(**int** n)

{

**int** i,j;

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

{

phi[i]=i;

}

mark.**set**(1,1);//marking as non-prime

phi[1] = 1;

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

{

**if**( i != 2)mark[i]=1;//marking as non-prime

phi[i]/=2;

}

**for**(i = 3; i <= n; i+=2)

{

**if**(mark[i]==0)

{

phi[i]--;

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

{

mark.**set**(j,1);

phi[j] = phi[j]/i \* (i-1);

}

}

}

}

**/// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*///**

1. **BigMod or Modulus of Big Power (A^P) % M**

**Explanation :** [**http://www.shafaetsplanet.com/?p=936**](http://www.shafaetsplanet.com/?p=936)

#ifndef li

#define li long long int

#endif // li

#ifndef MOD

#define MOD 1000000007

#else

#define MOD 1000000007

#endif // MOD

li BigMod(li a, li p){

**if**(p==0){

**return** 1LL % MOD;

}

li x = BigMod(a,p/2LL);

x = (x\*x) % MOD;

**if**(p%2){

x=(x % MOD \* a % MOD) % MOD;

}

**return** x;

}

1. **Modular Inverse ( calculation)**

**Explanation:** [**https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/**](https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/)

**///Inverse Modulus**

**///(X^-1)%MOD = mod\_inverse( (X % MOD), MOD)% MOD**

**///(When X and MOD are co-prime) -> (X^-1)%MOD = (X ^ (phi(MOD)-1))%MOD [use bigmod]**

**///(When MOD is prime) -> (X^-1)%MOD = (X^(MOD-2))%MOD**

**template** <**typename** t1> mod\_inverse(t1 a, t1 m){

t1 m0 = m, temp;

t1 x0=0, x=1;

**if**(m==1)**return** 1;

**while**(a > 1){

t1 quotient = a/m;

a=a%m;**swap**(a,m);

temp=x0; x0=x-x0\*quotient; x=temp;

}

**if**(x < 0){

x+=m0;

}

**return** x;

}

1. **Number of Digits of in any base B.**

We know that for a number **x** in base-10would result in the number of digits of that number.

Let, N = factorial(n) = n \* (n-1) \* (n-2) \* … \* 3 \* 2 \* 1.

According to previous statement,

Hence, to find out **Number of digits of factorial(n)**, we simply need to:

**int** Num\_of\_digits\_Factorial(**int** n){

**double** sum=0;

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

sum+=log10((**double**)i);

}

**return** floor(sum)+1;

}

Now, if you only need to calculate factorial(n) for only once, this O(N) algorithm is fine.

But what if *you need to calculate the factorial for any value in between [1, N] at most of N times??*

In that case, for worst case, required time complexity of algorithm using the above method is equal to O(N\*N) = O(N2).

So what we can do is apply Dynamic Programming (Specifically Memoization) to memoize all the digit of factorials for all. Hence, whenever the digit of factorial(i) would be needed for any, we simply return floor(sum[i])+1;

**Memoized Code:**

#ifndef N

#define N 1000006

#endif // N

**double** memoized\_sum\_of\_log10[N+10];

**int** preProcess\_digits\_of\_fact()

{

**double** sum=0;

**for**(**int** i=1;i<=N;i++){

sum+=log10((**double**)i);

memoized\_sum\_of\_log10[i]=sum;

}

}

**int** Num\_of\_digits\_Factorial(**int** n){

**if**(n>N)**return** -1;

**return** floor(memoized\_sum\_of\_log10[n])+1;

}

Using this above code, the preprocess takes O(N) time, and all the later queries would be processed in O(1) time. Hence total complexity has been decreased from **O(N2) to O(N)**.

***So, now we know how to calculate the digits of factorial(N) in base 10.***

*But, what if we need to calculate the number o digits in factorial(N) when fact(N) is in base B??*

Now, there is a formula for logarithm base transformation:

Using this formula, as we have number of digits of factorial(N) in decimal from,

We can have **digits of fact(N) in base B = .**

**Final Code:**

#ifndef N

#define N 1000006

#endif // N

**double** memoized\_sum\_of\_log[N+10];

**void** preProcess\_digits\_of\_fact()

{

**double** sum=0;

**int** indexx=1;

**for**(**double** i=1.0;i<=(**double**)N;i+=1.0,indexx++){

sum+=log(i);

memoized\_sum\_of\_log[indexx]=sum;

}

}

**int** Num\_of\_digits\_Factorial(**int** n, **int** Base){

**if**(n>N)**return** -1;

**return** floor(memoized\_sum\_of\_log[n]/log((**double**)Base))+1;

}