**NUMBER THEORY**

Primality test: to check number is prime or not

Naïve approach is to run a loop n times and check its divisibility upto n;

Better approach:

Divisors of number occur in pair 12: (1,12),(2, 6),(3,4)

So we have to run upto 3 and divide it by number to find other number………..but problem is we don’t know the limit he loop till which we have to run t.

So…….. for that we should know for a pair of divisor one is less then sqrt(n) and other should be greater than sqrt(n).

* If the number is perfect square one factor lies on sqrt.

So the sqrt of n is the upper bound.

Time complexity O(sqrt(n)).

Sieve of Eratosthenes:

If suppose u have given million queries ………then if we run it million times it will give tle

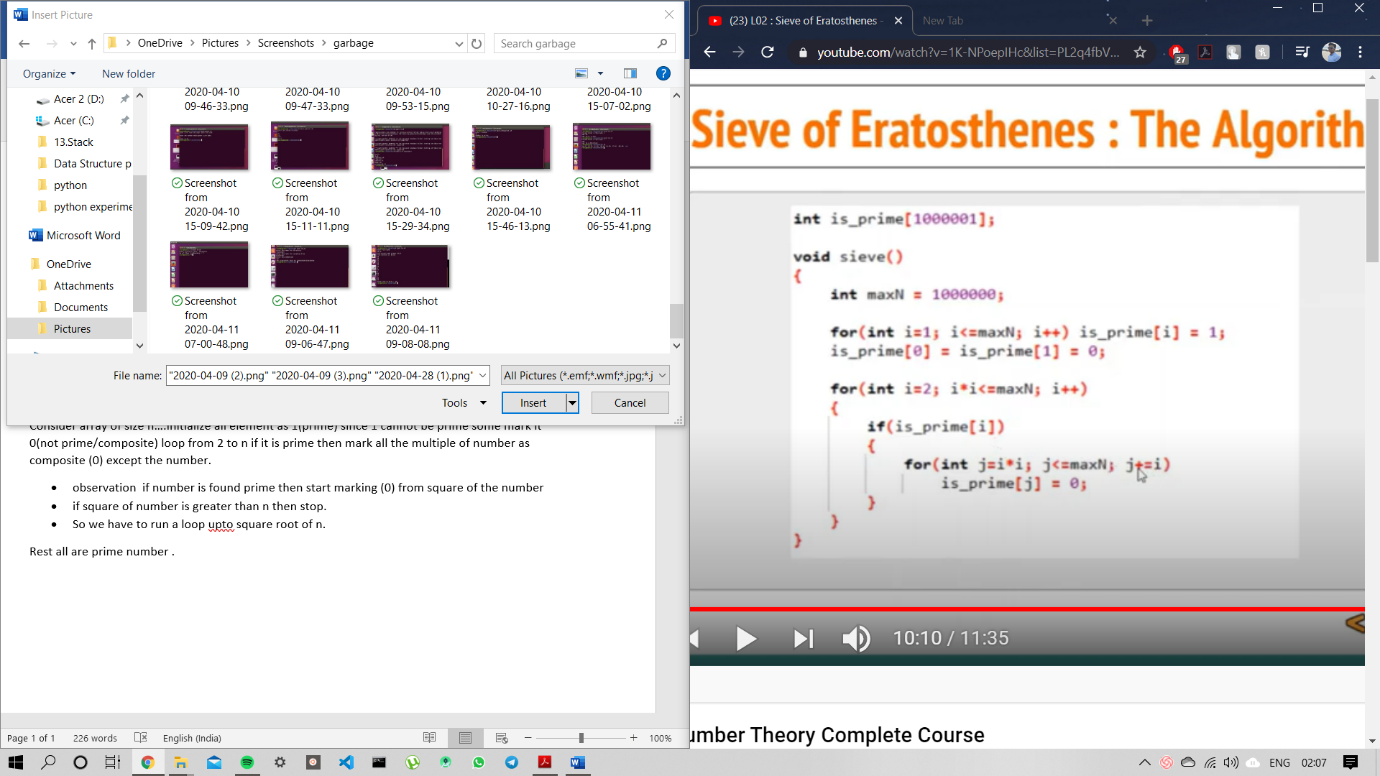
For this Sieve is uses

n=50

Consider array of size n….initialize all element as 1(prime) since 1 cannot be prime so mark it 0(not prime/composite) loop from 2 to n if it is prime then mark all the multiple of number as composite (0) except the number.

* observation if number is found prime then start marking (0) from square of the number
* if square of number is greater than n then stop.
* So we have to run a loop upto square root of n.

Rest all are prime number .



Run time :nLnLn(n)

PRIME FACTORIZATION:

Numbers of prime factors of number

Brute force

Run a loop from i=2 and check if the number is giving mod 0 if so then run while it is divisible and count++;

Prime(n)

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

{

If(n%i==0)

` {

Int count=0;

While(n%i==0)

{

Count++; n/=i;

Cout<<i<<”^”<<count<<endl;

}

}

}

}

Complexity :o(log(n))

Worst case(if n is prime): complexity is o(n)

Optimized:

* If n is composite number then there is atleast 1 prime divisor of N below sqrt(N)

Prime(n)

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

{

If(n%i==0)

{

Int count=0;

While(n%i==0)

{

Count++; n/=I;

Cout<<i<<”^”<<count<<endl;

}

}

If(n>1)

{

Cout<<n<<”^”<<1<<endl;

}

}

Time complexity o(sqrt(n))

IMP

Suppose a number is given in form of power of its prime factors,

Then number of divisor I given by product of power + 1 of prime number

N=p^a\*q^b\*r^c

Then total factors are (a+1)(b+1)(c+1)

BINARY EXPONENTIATIOIN:

Used to calculate power of any number in log(n) time

Ans=1, base=2, power=13

If power is odd then multiple ans bus base

ie ans =2

if even then divide by 2 and multiple base with itself.

ans base power

1 2 13

2 2 12

2 4 6

2 16 3

32 16 2

32 256 1

8192 256 0

Void power(int a , int n)

{  
 int ans=1;

While(n)

{

If(n%2)

Ans\*=a;

n--;

else

a\*=a;

n/=2;

}

return ans;

}

in modular exponentiation we have to calculate (A^n)%p;

so there is slight modification in the code replace

ans=ans\*a by ans=(ans\*a)%p

and a=a\*a by a=(a\*a)%p;

PRIME FACTORIZATION USING SEIVE

MATRIX EXPONENTIATION

Finding power of a matrix

Normal Time complexity :O(M^3\*N)

M is dimension

On optimization we get O(M^3^log(N))

Naïve approach

\_\_power(matrix ,int n)

{

Result;

For(int I =1;i<=n; i++)

{  
 result \*= base

}

Return 0;

}

Result should be a identity matrix of size n , bcoz initially while multiplying it wont affect the matrix

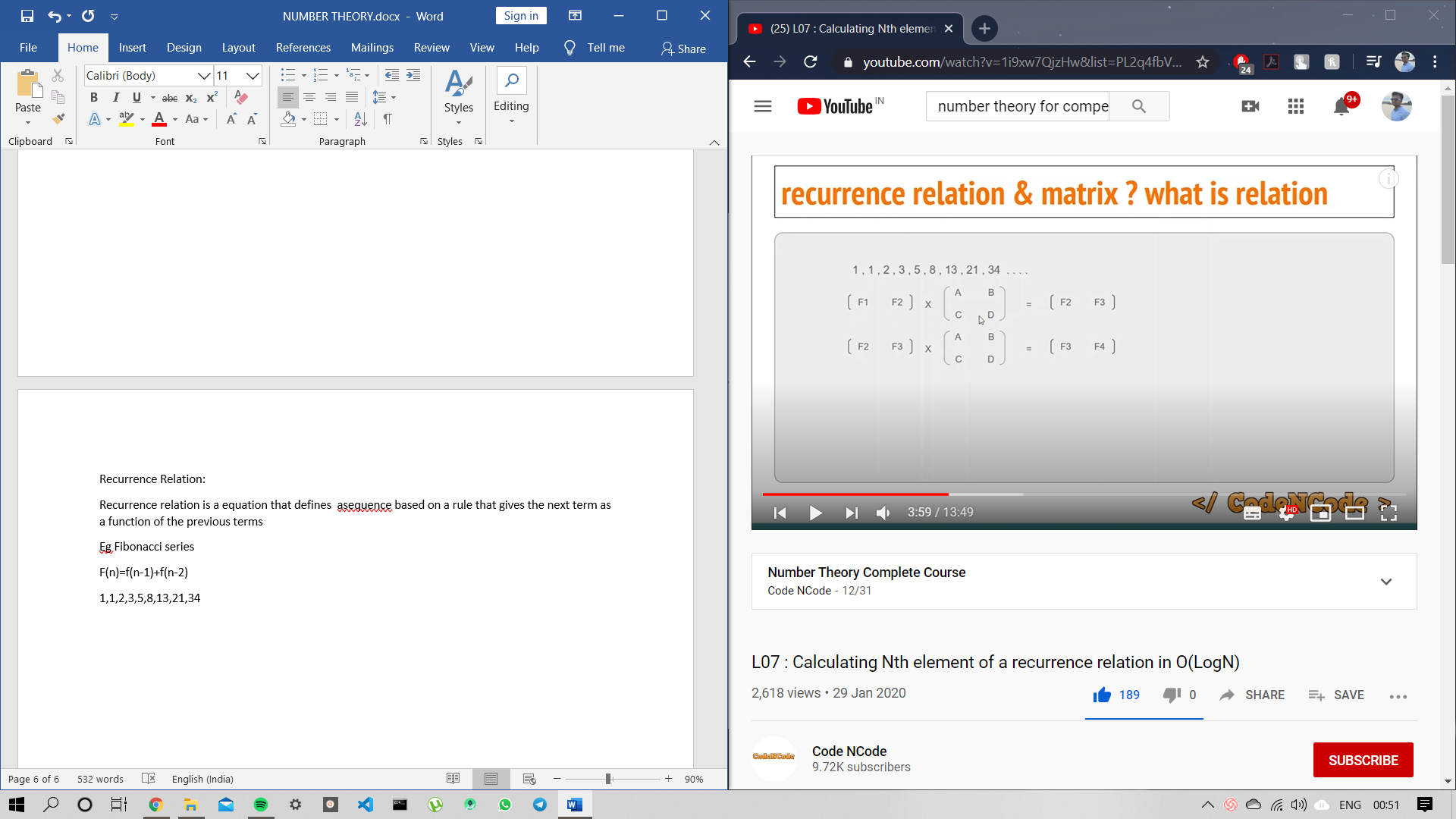
Recurrence Relation:

Recurrence relation is a equation that defines a sequence based on a rule that gives the next term as a function of the previous terms

Eg Fibonacci series

F(n)=f(n-1)+f(n-2)

1,1,2,3,5,8,13,21,34



(A,B,C,D) is the magic matrix/ transition matrix

#include<bits/stdc++.h>

#define N 101

#define REP(i,n) for(int i=1;i<=n;i++)

int ar[N][N] , I[N][N];

void mul(int Identity[][N], int A[][N], int dim)

{

int res[dim+1][dim+1];

REP(i,dim)

{

REP(j,dim)

{

res[i][j]=0;

REP(k,dim) res[i][j]+=Identity[i][k]\*A[k][j];

}

}

REP(i,dim) REP(j,dim) Identity[i][j]=res[i][j];

}

void power(int A[][N],int dim ,int n)

{

REP(i,dim) REP(j,dim)

{

if(i==j) I[i][j]=1;

else I[i][j]=0;

}

REP(i,n)

{

mul(I,A,dim);

}

REP(i,dim) REP(j,dim) A[i][j]=I[i][j];

}

void printMAt(int a[][N] , int dim)

{

REP(i,dim)

{

REP(j,dim) std::cout<<a[i][j]<<" ";

std::cout<<"\n";

}

}

int main(void)

{

int dim,n;

std::cin>>dim>>n;

REP(i,dim) REP(j,dim) std::cin>>ar[i][j];

power(ar,dim,n);

printMAt(ar,dim);

return 0;

}

Optimal approach

Replace

REP(i,n)

{

mul(I,A,dim);

}

With

while(n)

{

if(n%2)

{

mul(I, A, dim) , n--;

}

else

{

mul(A,A,dim),n/=2;

}

}

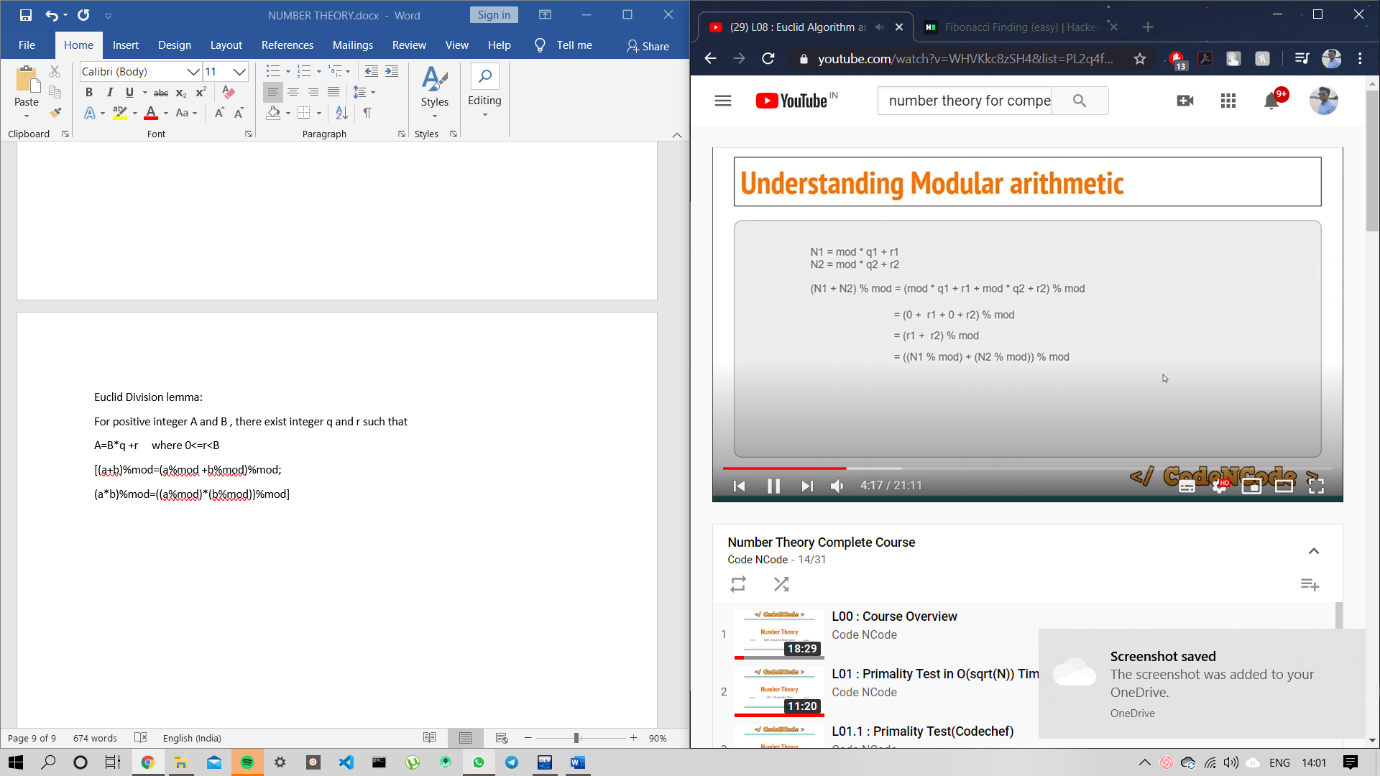
Euclid Division lemma:

For positive integer A and B , there exist integer q and r such that

A=B\*q +r where 0<=r<B

[(a+b)%mod=(a%mod +b%mod)%mod;

(a\*b)%mod=((a%mod)\*(b%mod))%mod]



R1 and r2 are remainder.

Euclid Division Algorithm

To find GCD (HCF)

Int gcd( int a,int b)

{

If(b==0) return a;

Return gcd(b,a%b)

}

Complexity (log(max(a,b))

Let GCD of A and B be g

Obs : GCD (A, B) = GCD(A-B, B) = GCD(A,B-A);

A= g\*X

B = g\*Y

A-B =g(X-Y)

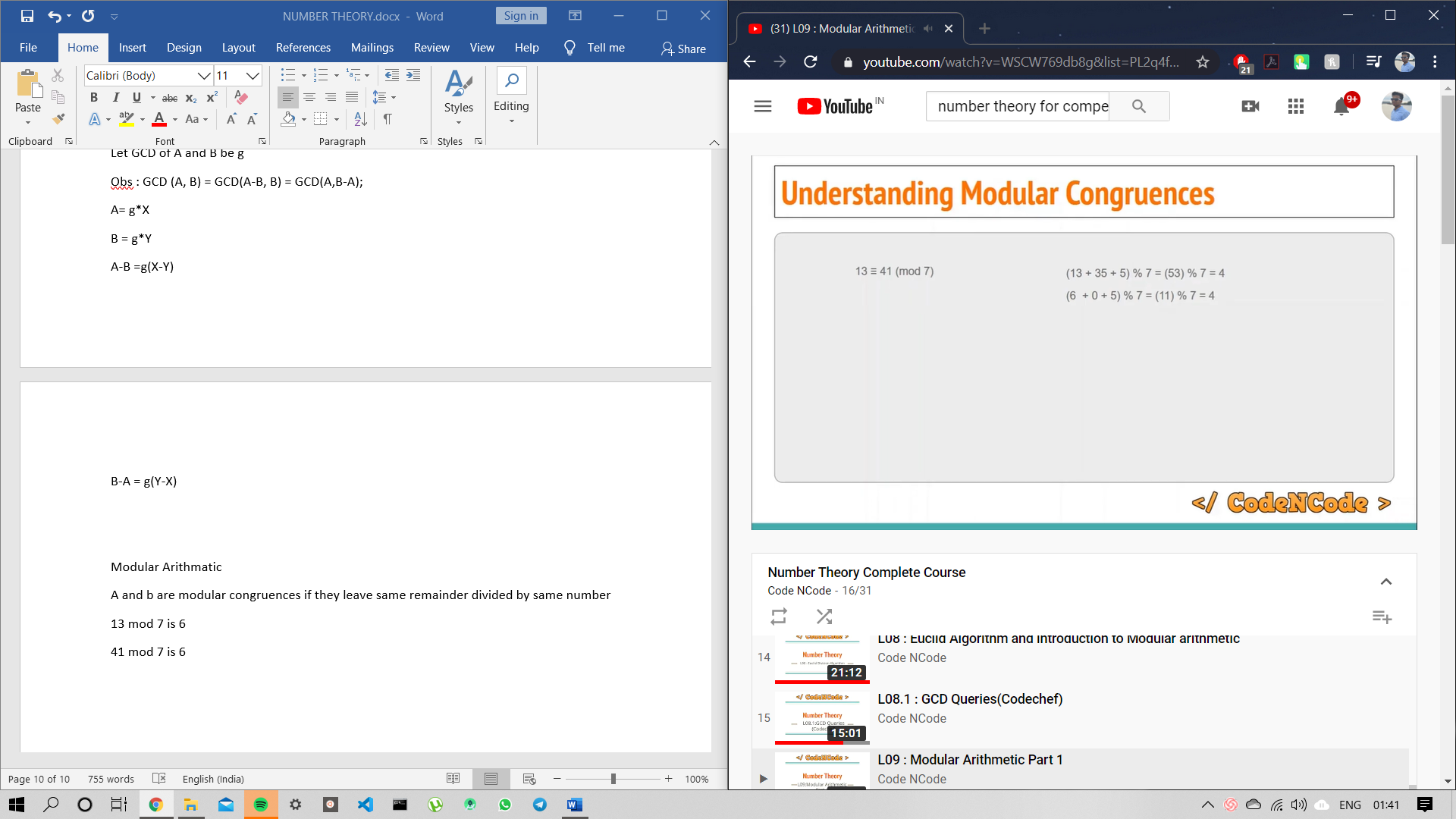
B-A = g(Y-X)

Modular Arithmatic

A and b are modular congruences if they leave same remainder divided by same number

13 mod 7 is 6

41 mod 7 is 6



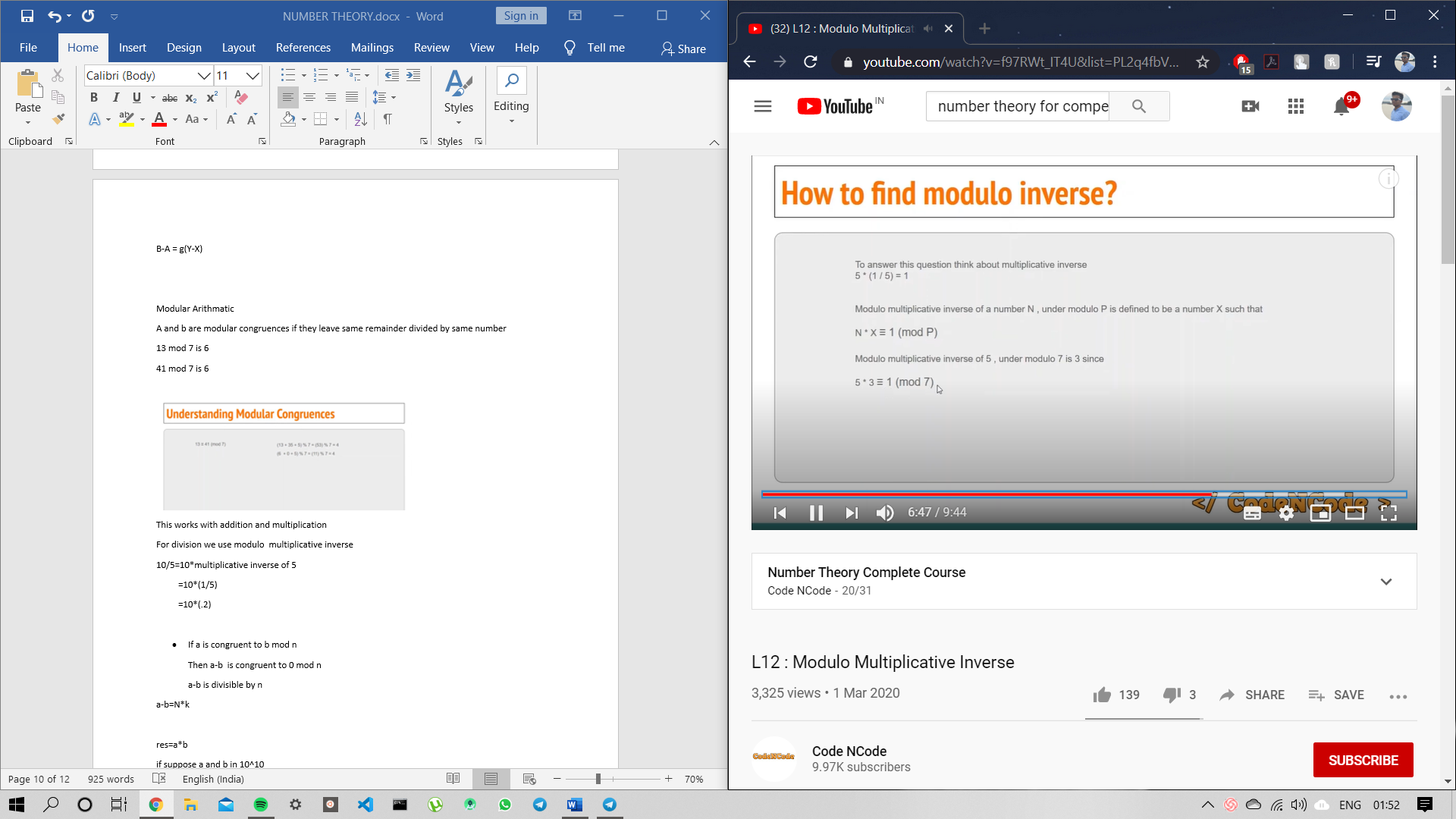
This works with addition and multiplication

For division we use modulo multiplicative inverse

10/5=10\*multiplicative inverse of 5

=10\*(1/5)

=10\*(.2)



(6/2)%5=( (6%5)\* (3%5) )%5

=(1\*3)%5

=3

Fermit little theorem

For prime number m and any coprime number a

Coprime means gcd of m and a is 1

then

a^(m-1) is congruent to 1 mod m

a^m-1 = 1 mod m

then

a^m-2 = a^-1 mod m

* If a is congruent to b mod n

Then a-b is congruent to 0 mod n

a-b is divisible by n

a-b=N\*k

res=a\*b

if suppose a and b in 10^10

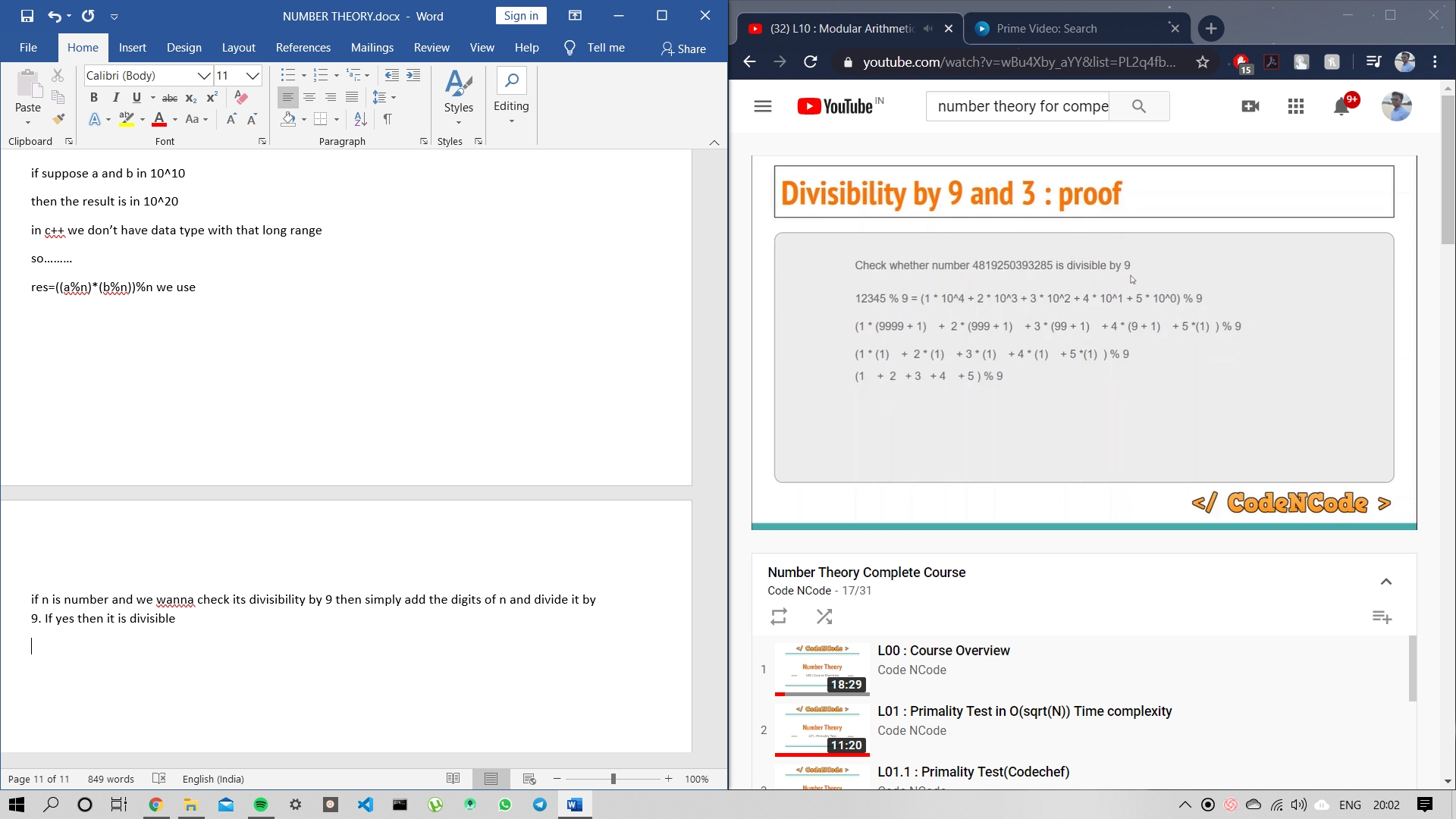
then the result is in 10^20

in c++ we don’t have data type with that long range

so………

res=((a%n)\*(b%n))%n we use

if n is number and we wanna check its divisibility by 9 then simply add the digits of n and divide it by 9. If yes then it is divisible



If a is congruent to bmod(n)

Then a^k is congruent to b^k mod(n).

Eq

29^10 mod 3 is congruent to 2 mod(3)

29=2 mod(3)

(2^10 )%3 =(1024)%3

(7)%3=1

Modular GCD:

If we wanna find gcd of two number then one very lar ge and other small

Then we find potential gcd candidates and apply modulo arithmatic to find gcd

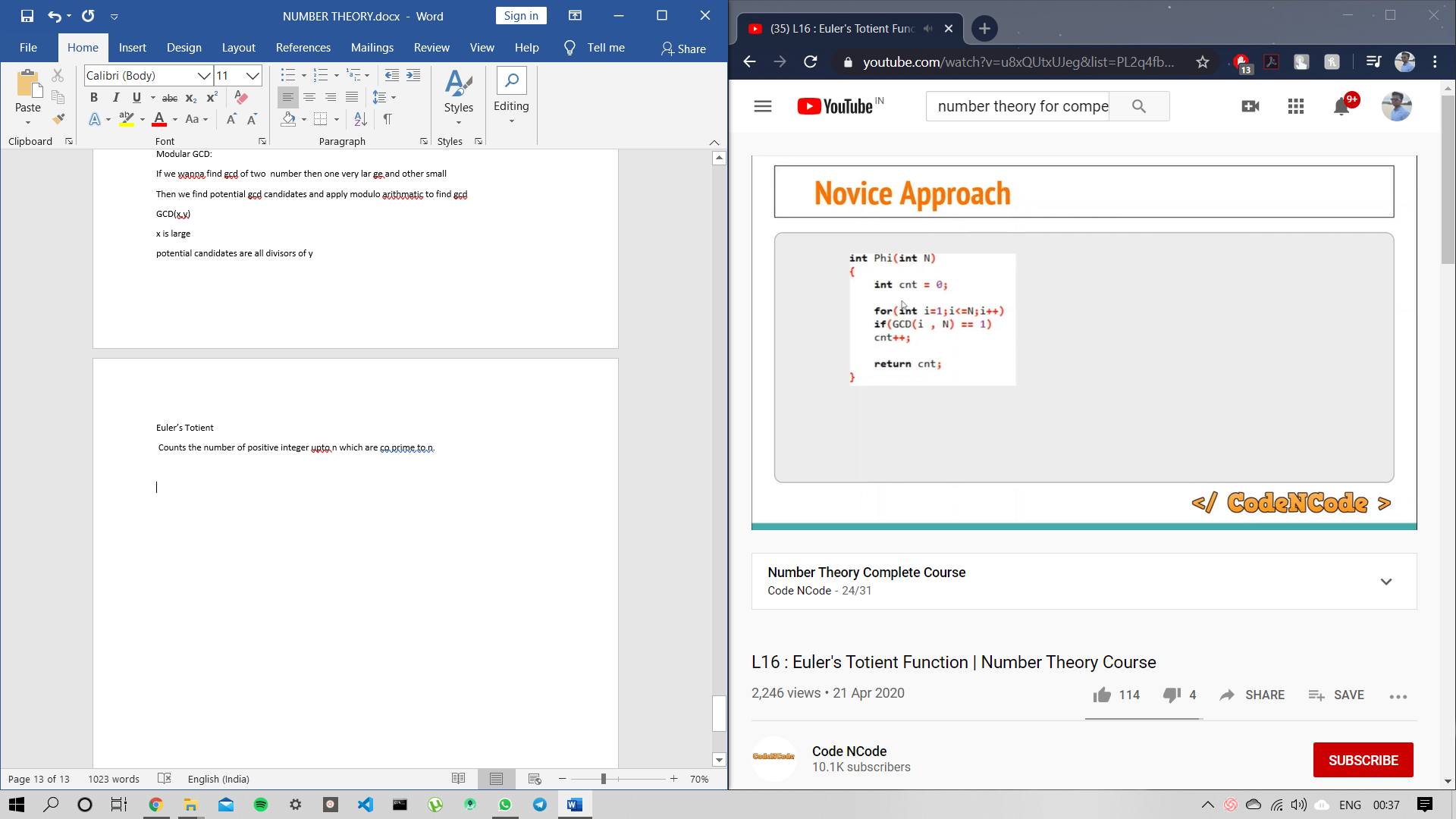
GCD(x,y)

x is large

potential candidates are all divisors of y

Euler’s Totient

Counts the number of positive integer upto n which are co prime to n.



Co primes of p^x= (p^(x-1))\*(p-1)

Segmented sieve: Used to generate prime number between given range

Find all prime number upto sqrt(upperlimt)

Then in the range check divisibility of every prime number generated

And mark it 1/0 acc.

Now……we need to find index of number …….simply add lower limit to index ull get the number.

Non Deterministic Primality Test // bcoz sqrrt(N) is very slow.

Fermat Primality Test

A^(p-1) = 1 mod p

P is prime;

Extended Euclidean algorithm

If ax+by =gcd(a,b) then there always exist x and y .

It is reverse of Euclid algo

Eg

81 = 1(57) + 24

57 =2(24)+9

24 = 2(9)+6

9 = 1(6) +3

6 = 2(3) +0

Consider second last step

3 = 9 – 1(6)

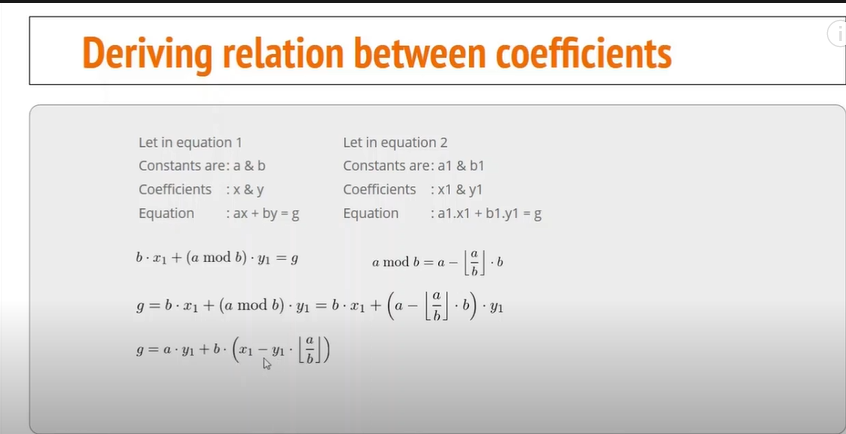
3 = 9 – 1(24 -2(9))

3 = 3(57 -2(24)) -24

3 = 3(57 )-7(24)

3 = 3(57) – 7(81 – 1(57))

3 = 10(57) – 7(81)



By comparison x= y1

Y = x1 – y1\*(a/b)

Linear Diophantine equation

Equation of the form ax + by = c where a , b are known and c is unknown where c is not gcd

Special case a=b=0

2 sub cases

C=0 c!=0

Total solution infinite total solution is 0

Ax+by =c

Ax1+ by1 = gcd

Then x = x1\*(c/gcd)

Y = y1 \*(c/gcd);

\*solution exist only if c is divisible by gcd