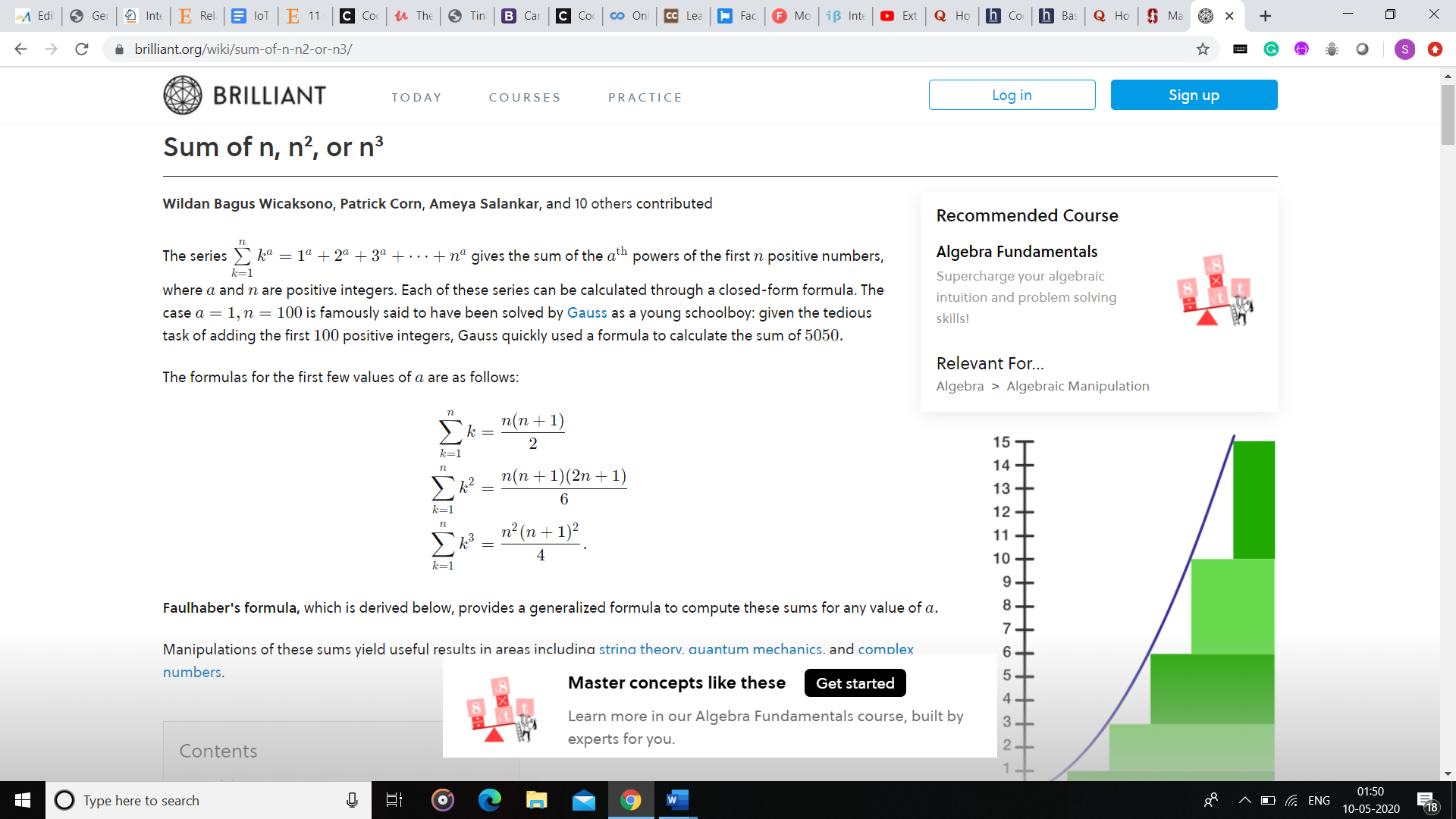
Number Theory-1



<https://www.hackerearth.com/practice/math/number-theory/basic-number-theory-1/tutorial/>

***Properties***

1. (a+b)%c=(a%c+b%c)%c
2. (a∗b)%c=((a%c)∗(b%c))%c
3. (a−b)%c=((a%c)−(b%c)+c)%c
4. (a/b)%c=((a%c)∗(b−1%c))%c

**Note**: In the last property, b−1 is the multiplicative modulo inverse of b and c.

***Examples***

If a=5, b=3, and c=2, then:

* (5+3)%2=8%2=0  
  Similarly, (5%2+3%2)%2=(1+1)%2=0
* (5∗3)%2=15%2=1  
  Similarly, ((5%2)∗(3%2))%2=(1∗1)%2=1

If a=12,b=15, and c=4, then the answer in some languages is (12−15)%4=(12%4−15%4)%4=(0−3)%4=−3. However, **the answer of the % operator cannot be negative.**

**Therefore, to make the answer positive, add c to the formula and compute it as follows:**(12−15)%4=(12%4−15%4+4)%4=(0−3+4)%4=1

***When are these properties used?***

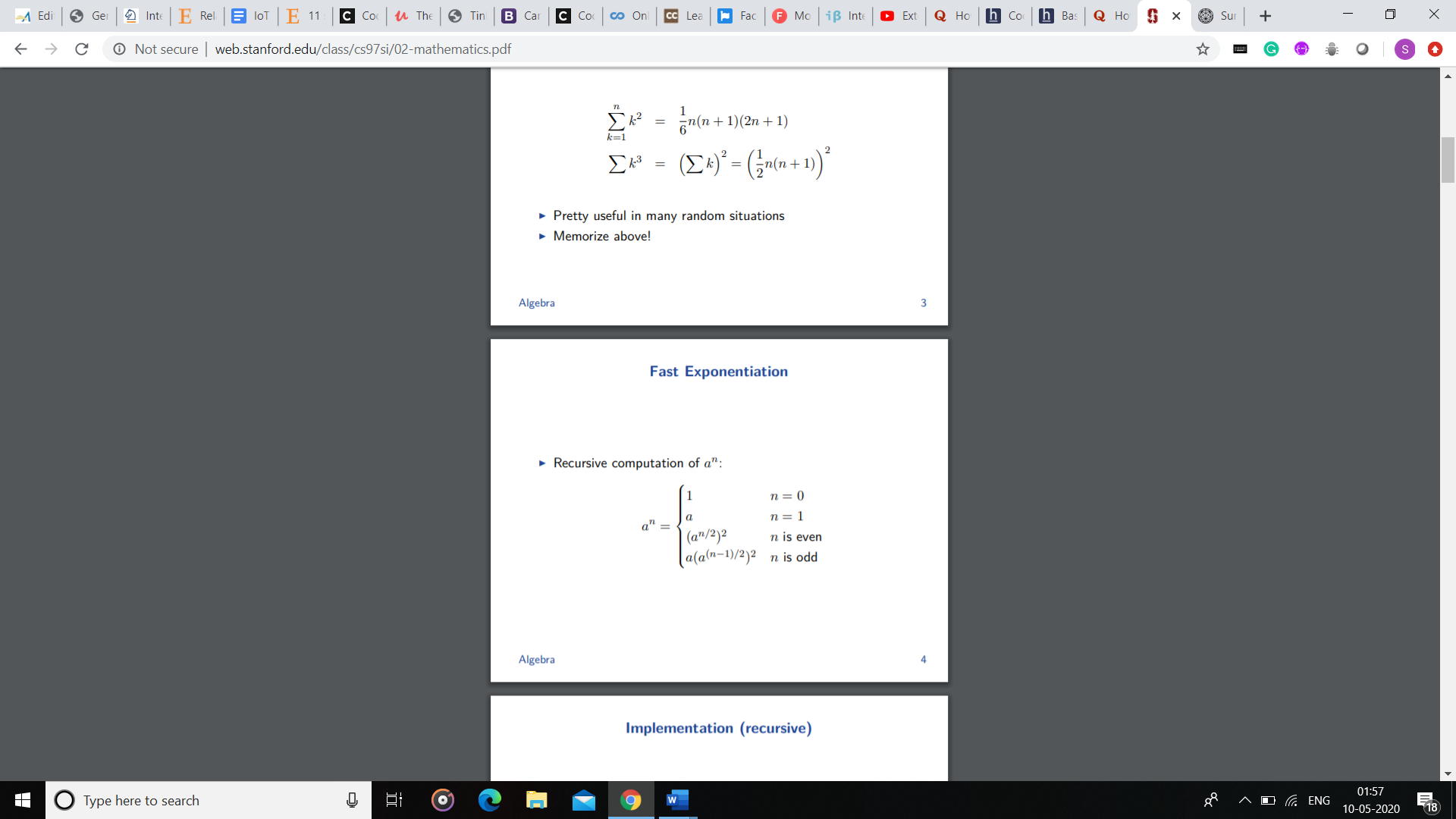
Assume that a = 1018, b = 1018, and c = 109+7. You have to find (a∗b)%c.

When you multiply a with b, the answer is 1036, which does not conform with the standard integer data types. Therefore, to avoid this we used the properties.

(a∗b)%c=((a%c)∗(b%c))%c=(49∗49)%(109+7)=2401

**2. Modular exponentiation**

Exponentiation is a mathematical operation that is expressed as xn and computed as xn=x⋅x⋅...⋅x (n times).



If n is even, then xn can be broken down to (x2)n/2. Programmatically, finding x2 is a one-step process. However, the problem is to find (x2)n/2.

Notice how the computation steps were reduced from n to n/2 in just **one** step? You can continue to divide the power by 2 as long as it is even.

When n is odd, try and convert it into an even value. xn can be written as x⋅xn−1. This ensures that n−1 is even.

* If n is even, replace xn by (x2)n/2.
* If n is odd, replace xn by x⋅xn−1. n−1 becomes even and you can apply the relevant formula.

***Example***

You are required to compute 310. The steps are as follows:

1. The power of 3 is 10, which is even. Break it down as follows:

310⇒(32)5⇒95

1. Find 95. The power of 9 is 5, which is odd. Convert it into an even power and then apply the following formula:  
   95⇒9⋅94⇒9⋅(92)2⇒9⋅(812)
2. 812 is a one-step computation process

The result is 9⋅81⋅81=59049.

This is an efficient method and the *ten-step process* of determining 310 is reduced to a *three-step process*. At every step, n is divided by 2. Therefore, the time complexity is *O(log N)*.

Code for modular exponentiation, here M can be for eg: 109+7.

int modularExponentiation(int x,int n,int M)

{

int result=1;

while(n>0)

{

if(power % 2 ==1)

result=(result \* x)%M;

x=(x\*x)%M;

n=n/2;

}

return result;

}

**Extended Euclidean algorithm**

This algorithm is an extended form of Euclid’s algorithm. GCD(A,B) has a special property so that it can always be represented in the form of an equation i.e. Ax+By=GCD(A,B).

The coefficients (x and y) of this equation will be used to find the modular multiplicative inverse. The coefficients can be zero, positive or negative in value.

This algorithm takes two inputs as A and B and returns GCD(A,B) and coefficients of the above equation as output.

***Example***

If A=30 and B=20, then 30∗(1)+20∗(−1)=10 where 10 is the GCD of 20 and 30.

***Key idea***

A.x+B.y=GCD(A,B). ---(1)

You know that GCD(A,B)=GCD(B,A%B). Therefore, you can write the equation as follows: B.x1+ (A % B).y1=GCD(A,B). ---(2)

You can write A%B=A−B∗⌊A/B⌋ where ⌊⌋ means floor value .B and substitute it in equation 2. Your equation will be as follows: B.x1+ (A - ⌊A/B⌋.B).y1=GCD(A,B)

When you solve it further, your equation is as follows: B.(x1 - ⌊A/B⌋.y1)+A.y1=GCD(A,B). ---(3)

Comparing coefficients in equations 1 and 3, you get the following:

* x=y1
* y=x1 - ⌊A/B⌋.y1

These equations are key in understanding the extended Euclidean algorithm.

In this algorithm, recursive calls are made to GCD(B,A%B). The values that are returned from recursive calls are x1 and y1, which are used to get x and y.

***Implementation***

#include < iostream >

int d, x, y;

void extendedEuclid(int A, int B) {

if(B == 0) {

d = A;

x = 1;

y = 0;

}

else {

extendedEuclid(B, A%B);

int temp = x;

x = y;

y = temp - (A/B)\*y;

}

}

int main( ) {

extendedEuclid(16, 10);

cout << ”The GCD of 16 and 10 is ” << d << endl;

cout << ”Coefficients x and y are ”<< x << “and “ << y << endl;

return 0;

}

**Output**

The GCD of 16 and 10 is 2.

Coefficients x and y are 2 and -3.

Initially, the extended Euclidean algorithm will run as Euclid's algorithm until you determine GCD(A,B) or until B = 0. It will then assign x = 1 and y = 0.

In the current scenario, since B = 0 and GCD(A,B) is A, the equation Ax+By=GCD(A,B) will be changed to A∗1+0∗0=A.

The values of d, x, and y in the process of the extendedEuclid( ) function are as follows:

* d=2,x=1,y=0
* d=2,x=0,y=1−(4/2)∗0=1
* d=2,x=1,y=0−(6/4)∗1=−1
* d=2,x=−1,y=1−(10/6)∗−1=2
* d=2,x=2,y=−1−(16/10)∗2=−3

***Time complexity***

The time complexity of the extended Euclidean algorithm is O(log(max(A,B))).

***When is this algorithm used?***

This algorithm is used when A and B are co-prime. In such cases, x becomes the multiplicative modulo inverse of A under modulo B, and y becomes the multiplicative modulo inverse of B under modulo A. This has been explained in detail in the ***Modular multiplicative inverse*** section.

**Modular multiplicative inverse**

What is a multiplicative inverse? If A.B=1, you are required to find B such that it satisfies the equation. The solution is simple. The value of B is 1/A or A−1. Here, B is the multiplicative inverse of A.

What is modular multiplicative inverse? If you have two numbers A and M, you are required to find B such it that satisfies the following equation:

(A.B)%M=1

Here B is the modular multiplicative inverse of A under modulo M.

Formally, if you have two integers A and M, B is said to be modular multiplicative inverse of A under modulo M if it satisfies the following equation:

A.B≡1(modM). where B is in the range [1,M-1]

This equation is a formal representation of the equation discussed earlier.

**Why is B in the range [1,M-1]?**

(A∗B)%M=(A%M∗B%M)%M

Since we have B%M, the inverse must be in the range [0,M-1]. However, since 0 is invalid, the inverse must be in the range [1,M-1].

**Existence of modular multiplicative inverse**

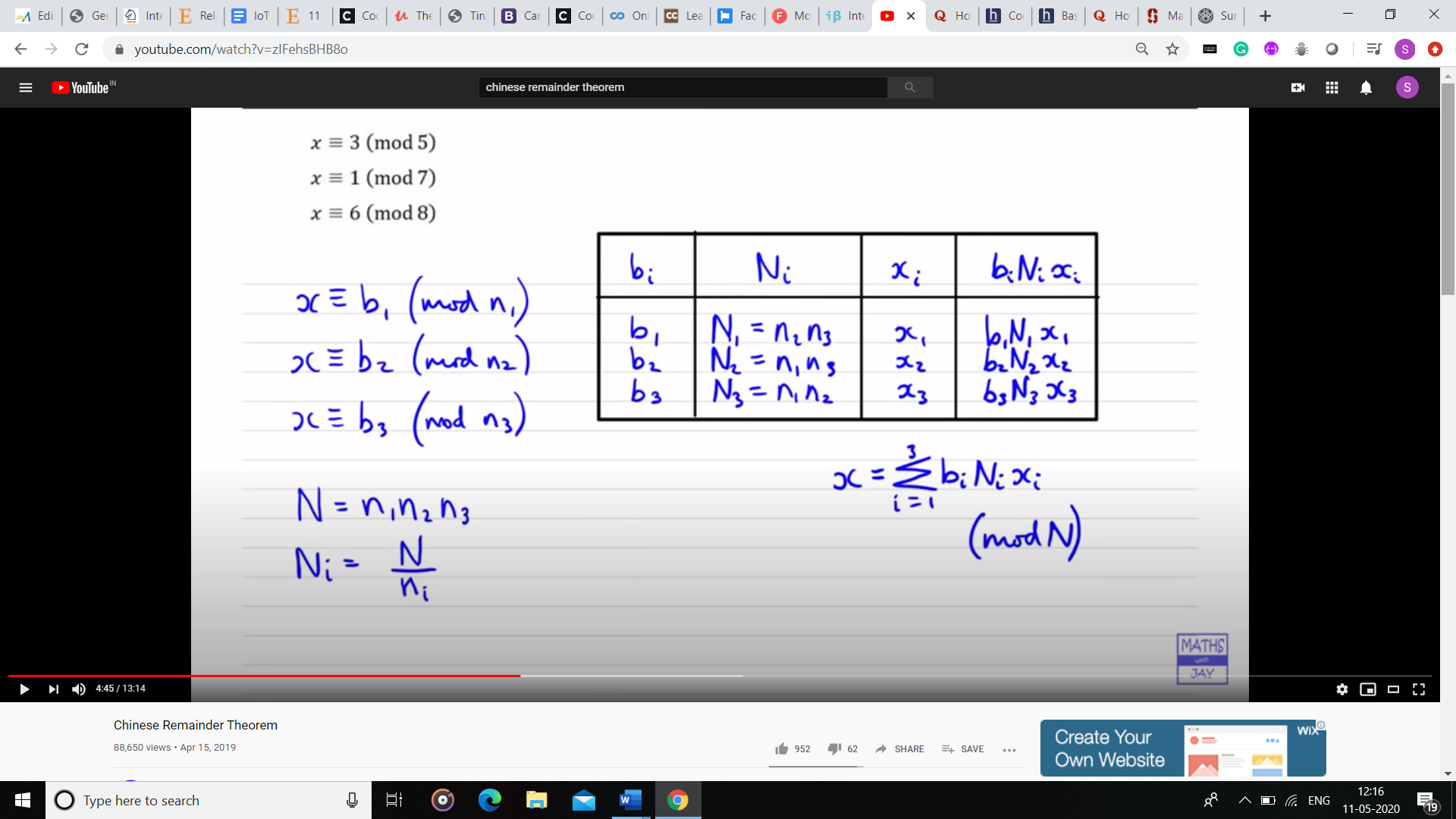
An inverse exists only when A and M are coprime i.e. GCD(A,M)=1.

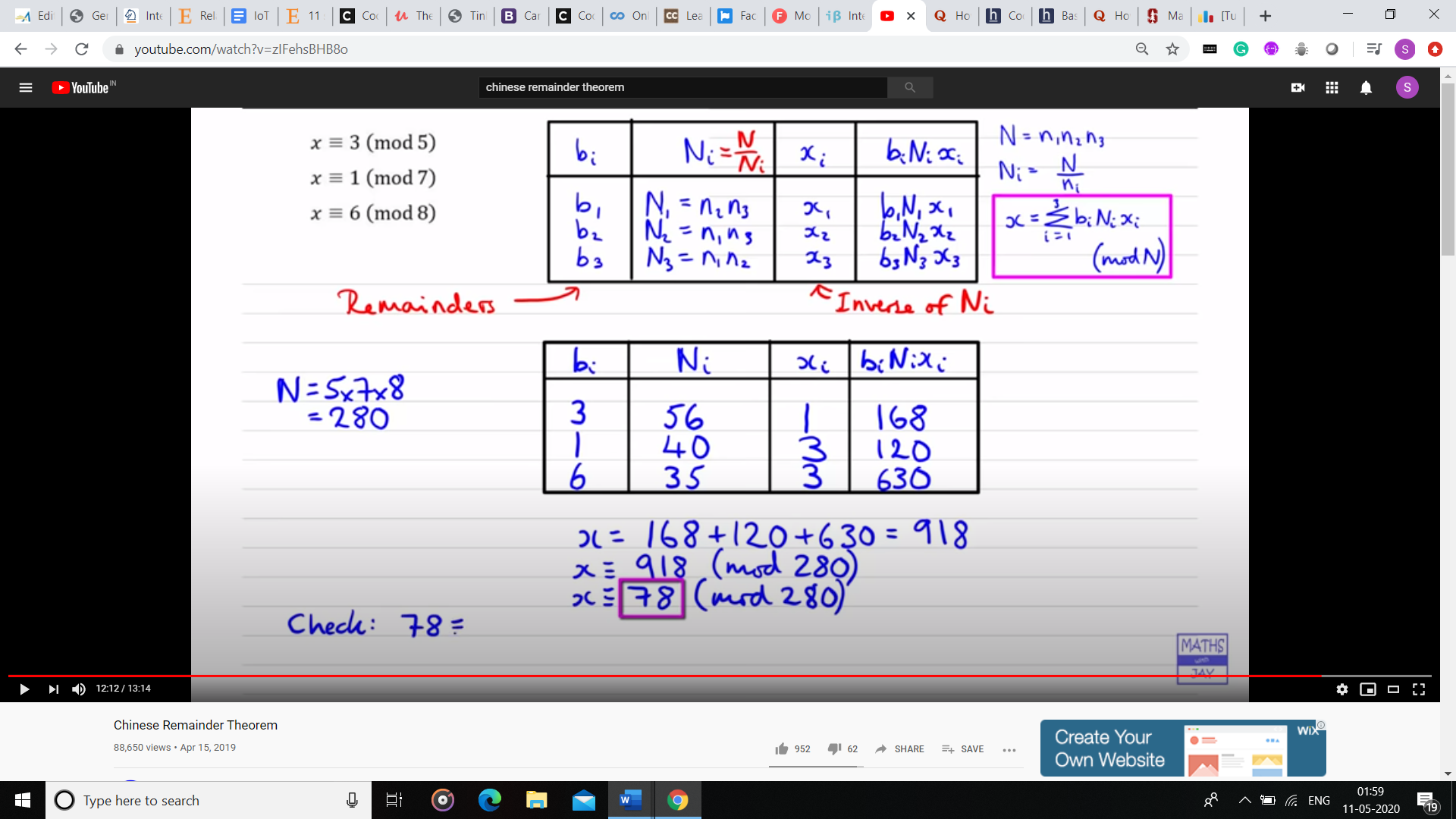
For example, if A=5 and M=12, then (A∗5)%M=(5∗5)%12=1. Here, 5 is the modular multiplicative inverse of 5 under modulo 12.  
Though (5∗17)%12=1, but since 17 > 12, it isn't considered.

Therefore, the answer is 5.

**Chinese Remainder Theorem:**

This theorem is used to solve problem such as find the number which when divided by 5 gives a remainder 3, remainder is 1 when divided by 7 and remainder is 6 when divided by 8.





<https://www.geeksforgeeks.org/chinese-remainder-theorem-set-2-implementation/>

<https://codeforces.com/blog/entry/61290>