Baby steps Titan steps

110 • 7 Jan 2025

introduction

During my recent studies we were introduced to the baby-step giant step algorithm (BSGS). The baby-step giant step algorithm is a method that solves the discrete logarithm fairly efficiently. Let’s describe it quickly here.

The discrete logarithm problem can be described like that: We want to find

$$\alpha \in \mathbb{N}$$

such that

$$g^{\alpha} \equiv r \mod N \Leftrightarrow g^{\alpha} = r + kN$$

with N usually a prime, g a generator of the multiplicative group, $\alpha < N$

The BSGS algorithm offers to solve the discrete log problem by rewriting

$$\alpha = am + j$$

with $j < \sqrt{N}$

Doing so, you can rewrite the problem as

$$g^{am} g^{j} \equiv r \mod N \Leftrightarrow g^{j} \equiv g^{-am} r \mod N$$

So the BSGS algorithm proposes to repeatedly apply a modular multiplication by $g^{-a} \equiv g^{N-1-a} \mod N$ to r until one value equals g^{j}. To check whether g^{j} equals $g^{-am} r$, the values of all the $g^{j} \mod N, \forall j \in [\![0, \lceil \sqrt{N} \rceil]\!]$ along with the exponents (indexes) j can be stored in a hash table. The index can then be recovered immediately.

I somehow forgot about the details of the BSGS for unknown reasons, thinking the algorithm was parsing all the potential j. At a point in 2024, I told myself the discrete logarithm did not look as a problem which is that difficult when you look at it...which is a bit stupid. I thus decided to write an algorithm which is, in the end, exactly like the BSGS one, by searching for "the smallest k such that $g^{\alpha} = r + kN$". Among the long list of silly rediscoveries, I understood that you can actually compute a multiplication by the inverse of g^{a} by dividing $r + kN$ for some convenient k. That method is more or less what is called the Montgomery reduction method, silly me, it already exists.

Having a hard time to learn correctly what I read, I often end up doing some mental loops reconstructing things that already exist or that are inspired by something I read a while ago that I do not properly regurgitate. An attitude that does not get the reward it deserves in the modern world...

That being said, we can still do some interesting things to slightly modify the BSGS algorithm. By tinkering, I realised that you can actually have smaller hash tables, and still get the algorithm to converge. Conversely, it means that you can have hashtable of the same size, and make "bigger" steps.

Let's prove it.

Let's say we are searching for α, i.e.

$$g^{\alpha} \equiv r \mod N$$

and we take a generator g and we make steps of size a

then

$$\alpha = aq + r, 0 \leqslant r < a$$

Now let's take the generator g and we make steps of size am

then

$$\alpha = (am)q' + r', 0 \leqslant r' < am$$

Now the result, if

$$N - 1 = mQ + 1$$

Then you can store only all the residues r from g^a in a hashtable, while making steps of size g^{am} in the Baby steps, giant steps algorithm.

First

$$(am)q' + r' = aq + r \Leftrightarrow a(mq' - q) = r - r'$$

which means $a|r' - r$. So the difference between r and r' is a multiple of a.

Now if

$$N - 1 = mQ + 1$$

then

$$a(N - 1) = amQ + a \Leftrightarrow amQ \equiv -a \mod N - 1$$

Thus

$$\begin{aligned}
\alpha + (am)Q &\equiv \alpha - a \mod N-1 \\
\alpha + (am)Q &\equiv (am)q' + r' - a \mod N-1 \\
\alpha + 2(am)Q &\equiv (am)q' + r' - 2a \mod N-1 \\
\alpha + 3(am)Q &\equiv (am)q' + r' - 3a \mod N-1 \\
&\dots
\end{aligned}$$

And so there is always a $k \in \mathbb{Z}$ such that

$$\alpha + k(am)Q \equiv (am)q' + r' - ka \equiv (am)q' + r \mod N-1$$

And we are guaranteed to land on a value stored in the hashtable. And the algorithm converges.

$-1 < k < m$ because

$$0 \leqslant r' < ma$$

$$0 \leqslant r < a$$

so

$$-a < r' - r < ma$$

$$-a < ka < ma$$

$$-1 < k < m$$

So k can only take m values.

Neat, but I just wondered what would happen if the algo did not land on any stored value. Would it run forever? I then tried several values, and it works, it's mostly serendipidity, feelings, good baths, and some Ritalin.

Is it useful? Likely, you need to search for the divisor of $N-2$, which might be complicated but you can hope for a small one. Then, the discrete logarithm is likely to be tackled more easily especially if you have plenty of computer running in parallel.

On a single processor, making steps of size am with remainders of the division by a, the solutions are found in $[\![0, mN)$, so with m processors, you can make an acceleration to the BSGS by a factor of m. That is, if you can store $O(f(N)$ values on your compute and the classical BSGS algorithm runs in $O(\frac{N}{f(N)})$ time, then you can still store $O(f(N)$ values on m computers and run the program on each computer in $O(\frac{N}{mf(N)})$ time.

That's a slightly different use of parallelism for the BSGS algorithm than the usual although, in the end, it simply provides a sort of map between the remainders of two generators. You could say it's a reversed use of parallelism. The classical use for m processor would be to divide the number of possible values taken by q by m. Here instead we multiply a by m. Tomato tomato. Maybe the only advantage would be to store only low values.

Let me show why. Let's say we have a generator g such that

$$g^{aq+r} \equiv g^{amq'+r'} \mod N \Leftrightarrow g^{aq+r} \equiv (g^m)^{aq'} g^{r'} \mod N$$

if we define $g^m = g'$ then

$$g'^{aq'} g^{r'} = g'^{aq'} g^{r+ka} \equiv g^{aq+r} \mod N \Leftrightarrow g'^{aq'} g^{ka} \equiv g^{aq} \mod N \Leftrightarrow g'^{q'} \equiv g^{q-k} \mod N$$

and so solving the discrete log problem with a bigger steps and the same remainders is equivalent to running the classical BSGS with another generator with its own remainders.

Initially I was aiming at finding a relation between the digits after the coma for the $\log_2$ in $\mathbb{R}$ and the integer values you find in $\mathbb{Z}$. Unfortunately I did not find anything. Basically $2^{-1} \equiv \frac{N+1}{2} \mod N$, so I was aiming at finding a relation between the binary expansion of $\log_2(r)$ and powers of $\frac{N+1}{2}$. If there is a link, I must have made a mistake in my calculations, because I found none. The current state of my reflexion, which does not go very far is that, if N is prime, we are searching for the smallest k' such that

$$r(N+1)^{k'} = 2^a + tN2^\beta$$

where

$\beta = 2k'$ if $N+1$ is only divisible by 4.

Say you start with $r \in [\![0, N)$, if you divide each step of size 1 into $N+1$ steps, then r becomes $r(N+1) \in [\![0, N(N+1))$ and dividing $r(N+1)$ by 2 is equivalent to multiplying r by $2^{-1} = \frac{N+1}{2}$, by repeating the process again and again, you end up on $1 + tN2^{k'}$. Multiplying by a number, here $\frac{N+1}{2}$ is thus equivalent to dividing integer intervals into $N+1$ intervals, and then changing position. Thus it is equivalent to looking at the binary digits between 0 and r and taking $N+1$ of them. Lot of sillyness, you still end up with searching for an unknown t.

All of the following reasonning is deeply flawed. basically I am stating:

We are searching for a such that

$$r \equiv 2^a \mod N \Leftrightarrow r2^{-a} \equiv 1 \mod N \Leftrightarrow r\left(\frac{N+1}{2}\right)^a \equiv 1 \mod N$$

from which I deduce

$$r(N+1)^a \equiv 2^a \mod N$$

Which is a non sense because

$$r(N+1)^a \equiv r \mod N$$

And I am running in circles.

At least we can say

$$r\frac{(N+1)^a}{2^a} = 1 + k_0 N \text{ and } 2^a = r + kN$$

so

$$r(N+1)^a = (r+kN)(1+k_0N) = r + N(k + k_0r + k_0k)$$

thus

$$\sum_{k'=1}^{a} r\binom{a}{k'} N^{k'-1} = k + k_0r + k_0k$$

Not very useful. We can say that $r|k(1+k_0)$, but that's it, 3 unkown, 1 equation...

So here is the flawed reasonning: We know that if N is a safe prime, then $N-1 = 2p$ so $N+1 = 2(p+1)$ and $p+1$ is even so $N+1$ is divisible by 4. Say $N+1$ is not divisible by more powers of 2 then

$$N+1 = 4\frac{p+1}{2} = 4\Pi_i p_i^{\alpha_i}$$

On a side note since N is a prime it is not divisible by 3 (unless $N=3$) and since $N-1=2p$ and p is a prime, $N-1$ is not neither divisible by 3, and thus $N-2$ and $N+1$ are all divisible by 3, thus $12|N+1$.

So

$$(N+1)^{k'} = 2^{2k'}\Pi_i p_i^{k'\alpha_i}$$

And we want to find the k' such that

$$\Pi_i p_i^{k'\alpha_i} \equiv r^{-1} \mod N$$

Which is just another discrete log problem, unless r^{-1} is divisible by one of the p_i.(actually, since one of the p_i is 3, at least a third of the r_i can be divided by one of the p_i). But if we know the $\log_2$ of the p_i,(i.e. $p_i \equiv 2^{\theta_i} \mod N$) then

$$(N+1)^{k'} \equiv 2^{k'(2+\sum_i \alpha_i\theta_i)} \mod N \equiv 1 \mod N$$

which means

$$k'(2 + \sum_i \alpha_i\theta_i) = K(N-1)$$

I don't know exactly what to do with that yet, I am just circling around, it's just like stating

$$2^{-2k'} \equiv r^{-1} \mod N$$

So it's completely useless.

the other expression

$$r(N+1)^{k'} = 2^a + tN2^{2k'}$$

provides even more unkowns than just the index.

$$r(N+1)^{k'} = (2^{a-2k'} + tN)2^{2k'}$$

$$\log_2\left(\frac{r(N+1)^{k'}}{(2^{a-2k'}+tN)}\right)=2k'$$

Back to sanity

Reduction

The multiplication by the inverse of a value and the division are the same only when the residue is divisible by the value.

For example, in $\mathbb{Z}_{16}$, 3 is invertible. We know that $33=32+1=16\times 2+1=3\times 11$ so $11\equiv 3^{-1} \mod 16$.

Ok, so let's see what happens when we multiply $4,5,6$ by 11 and reduce them $\mod 16$.

$$4\times 11=16\times 2+12\equiv 12 \mod 16$$

$$5\times 11=16\times 3+3\equiv 3 \mod 16$$

$$6\times 11=16\times 4+2\equiv 2 \mod 16$$

Notice that $4/3=1.\bar{3}3$, $5/3=1.\bar{6}6$ and... $6/3=2$, so multiplying by the inverse has the same effect than dividing by 3 only for 6.

We can ask ourselves if we can divide instead of having to multiply and then reduce. The reduction is not necessarily a costly operation, but the division can be faster than having to multiply and then reduce.

Back to 4, we know that

$$4\times 11\equiv 12 \mod 16$$

which means

$$\exists k, 4\times 11=12+16k$$

(omg what did I write previously here? Was I drunk?) here k is 2. So what is the link with $4/3=1.33$?

First

$$11\times 3=1+16\times 2$$

so

$$11=\frac{1}{3}+\frac{16\times 2}{3}\Rightarrow 4\times 11=\frac{4}{3}+\frac{16\times 8}{3}$$

ok, well, there is $4/3$ here but that is about it.

Now 3 and 16 are coprime, so from Bezout

$$\exists u,v\in\mathbb{Z}, 3u+16v=1$$

When we look at that relation in $\mod 16$, we only think that means that u is an inverse of $3 \mod 16$, but it also implies that v is an inverse of $16 \mod 3$. Let's explore this further.

$$16 = 3 \times 5 + 1 \Rightarrow 16 \equiv 1 \mod 3$$

So it is its own inverse. Not much fun in here. Let's look at 7 instead then. 7 is invertible $\mod 16$

$$7 \times 7 - 16 \times 3 = 1$$

So 7 is the inverse of $7 \mod 16$ and $-3 \equiv 4 \mod 7$ is the inverse of $16 \mod 7$

Let's imagine we have a residue $v \mod 16$ and we want to multiply it by $7^{-1} \mod 16$. We know that dividing and multiplying by the inverse are the same if the residue is divisible by its value, and its representative does not need to be in a specific range.

Therefore, because 7 and 16 are coprime, there might exist a k such that

$$7|v + k16 \Rightarrow v + k16 \equiv 0 \mod 7 \Rightarrow k \equiv -v16^{-1} \mod 7$$

$$k \equiv (7 - v)4 \mod 7$$

Then,

$$v + 16k = v + 64(7 - v) = 7 \times (64 - 9v) \Rightarrow \frac{v + k16}{7} = (64 - 9v)$$

Let's look at that formula, for $v = 4$,

$$(64 - 9v) = 28 \equiv 12 \mod 16$$

and

$$4 \times 7^{-1} = 4 \times 7 \equiv 12 \mod 16$$

Good, so we found a formula that gives us the same value, but we still used a reduction. We can use a reduction before so it will not cost as much later.

we know that

$$k \equiv (-v)4 \mod 7$$

if

$$0 \leqslant v < 7, -7 < -v \leqslant 0 \Rightarrow -28 < -4v \leqslant 0$$

for $v = 4, -4v = -16$ and so we need to add 21 to get the first non zero value 5.

Now

$$v + 16k = v + 16(21 - 4v) = 7(48 - 9v) \Rightarrow \frac{v + 16k}{7} = (48 - 9v)$$

For $v = 4$,

$$48 - 9v = 48 - 36 = 12$$

In general, if we want the we will find a k such that

$$k \equiv (-v)N^{-1} \mod n$$

Where n is invertible, we need to add the smallest multiple of n such that k is positive, thus, in the algorithm, we take

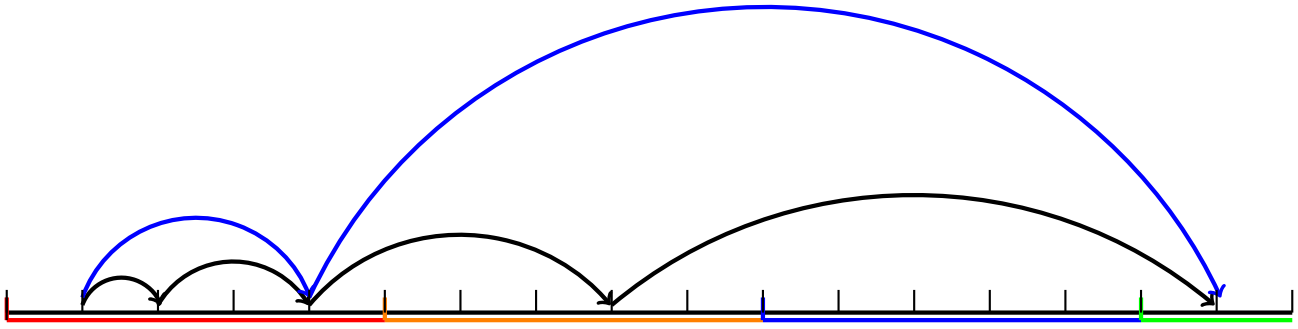
$$k = (n - v)N^{-1} \mod n$$

Then, dividing $v + kN$ by n provides the multiplication by the inverse and the reduction.

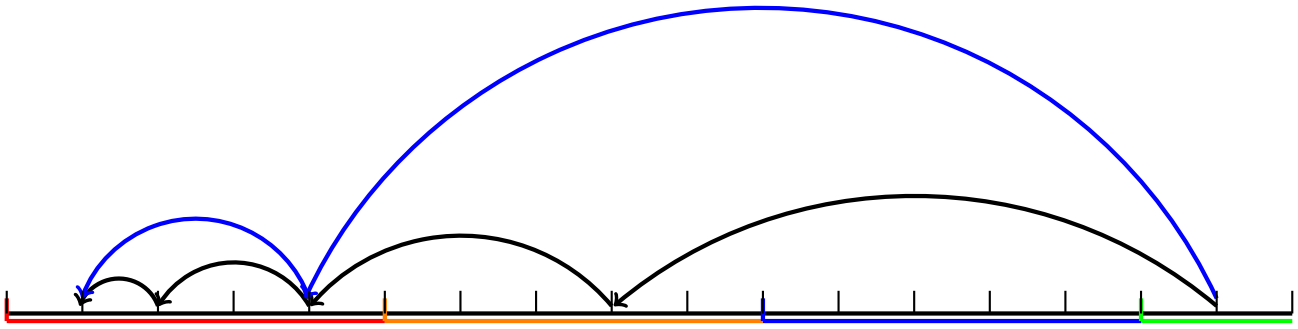
$$0 \leqslant v < N \text{ and } 0 \leqslant k < n \Rightarrow 0 \leqslant v + kN < N(1 + n) \Rightarrow 0 \leqslant \frac{v + kN}{n} < N(\frac{1}{n} + 1)$$

There is no guarantee that $N(\frac{1}{n} + 1)$ is close enough to N unless n ecomes large. This is used a lot in the algorithm where n is actually larger than N

Here are few illustrations of how the discrete logarithm works Let's take 2 as a generator and 5 as a modulus. Powers of 2 land on all values.

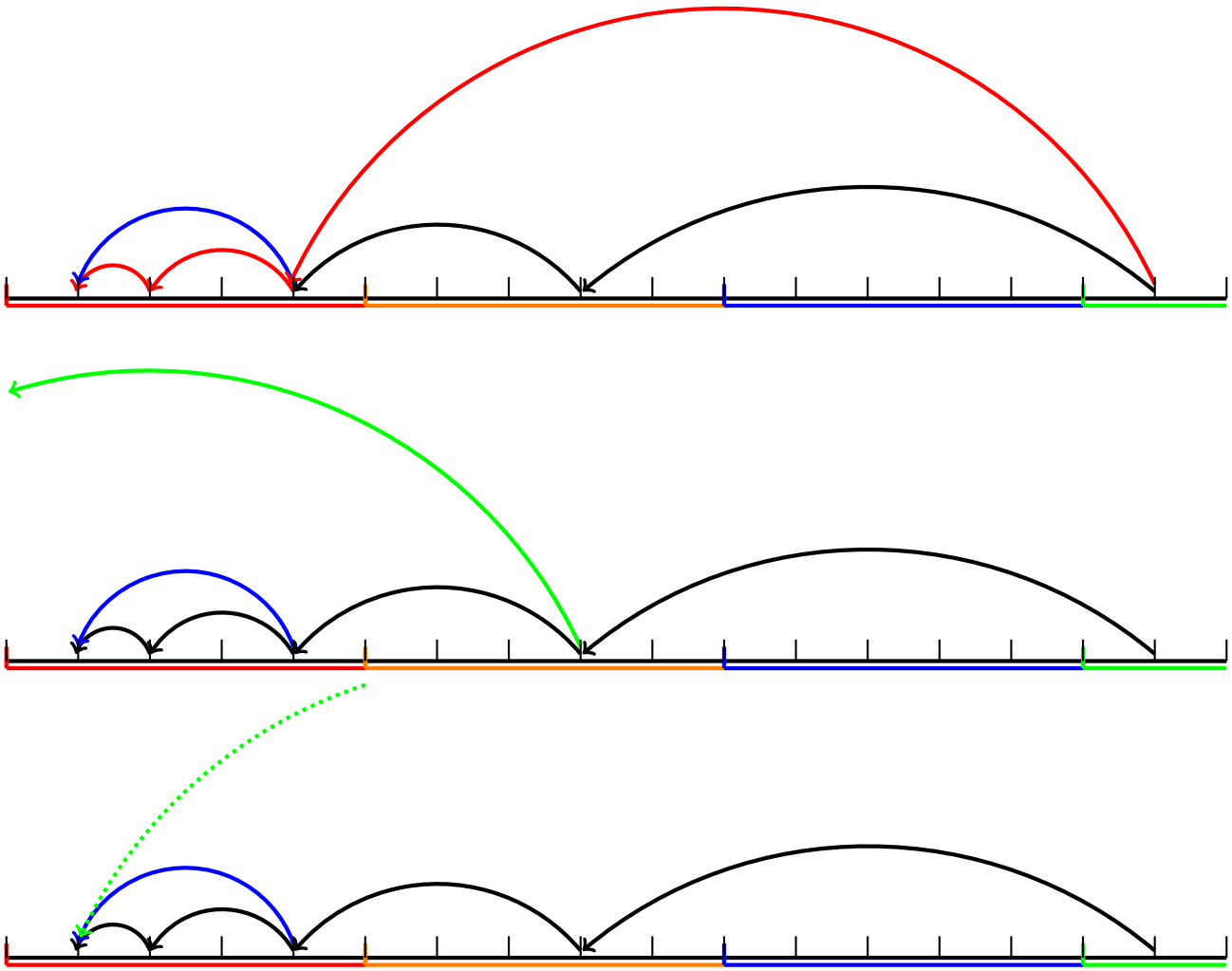


$$2^0 \equiv 1 \mod 5 \qquad 2^1 \equiv 2 \mod 5$$
$$2^2 \equiv 4 \mod 5 \qquad 2^3 \equiv 3 \mod 5$$
$$2^4 \equiv 1 \mod 5$$



$$1 \equiv 2^4 \mod 5 \qquad 3 \equiv 2^3 = \frac{2^4}{2} \mod 5$$

$$4 \equiv 2^2 = \frac{2^4}{2^2} \mod 5 \qquad 2 \equiv \frac{2^4}{2^3} \mod 5$$



```
use num::Unsigned;
use std::hash::Hash;
use std::time::Instant;
use std::time::Duration;
use std::fmt::Display;
use std::ops::BitAnd;
use num::PrimInt;
use mod_exp::mod_exp;
use std::collections::HashMap;

//use rand::prelude::*;

pub fn generic_extended_euclid<U>(a:U,b:U)->(U,U,U)
```

```
where
U: Unsigned + PartialOrd + Copy + std::ops::BitAnd,
{
    let one:U = U::one();
    let zero:U = U::zero();
    let two:U = one+one;
    let mut q:U;
    let ( mut r_temp, mut u_temp, mut v_temp):(U,U,U);
    let (mut u_1,mut v_1, mut u_2, mut v_2, mut r_1, mut r_2):
(U,U,U,U,U,U)=(one,zero,zero,one,a,b);
    let mut n:U = zero;
    while !r_2.is_zero(){
        q=r_1/r_2;
        (r_temp,u_temp,v_temp) = (r_1,u_1,v_1);
        (r_1,u_1,v_1)=(r_2,u_2,v_2);
        if r_temp>=q*r_2{
            r_2=r_temp-q*r_2;
        }else{
            r_2=q*r_2-r_temp;
        }
        (u_2,v_2,n)= (u_temp+q*u_2,v_temp+q*v_2,n + one);
    }
    if (n%two).is_one(){
        u_1=b-u_1;
    }else{
        v_1=a-v_1
    }
    return (r_1,u_1,v_1);
}


pub fn inverse_mod<U>(n:U,modulus:U)->U
where
U: Unsigned + PartialOrd + Copy + std::ops::BitAnd,
{
    //to do: proper error handling
    let inv:U;
    println!("searching for the inverse");
    let result = generic_extended_euclid(modulus,n);
    println!("inverse found");
    if result.0.is_one()
    {
        inv = result.2;
        return inv;
    }else
```

```
    {
        panic!("element non inversible");
    }
}

pub fn biggest_g_power_of_two<U>(g:U,v:U)->(U,U,U)
where U: Unsigned + PartialOrd + Copy +Display + std::ops::BitAnd,
{
    let one:U=U::one();
    let mut g_past:U;
    let mut g_temp :U = g;
    let mut exp :U = one;
    let two = one+one;
    if g < v
    {
        while
        {
            g_past=g_temp;
            g_temp=g_temp*g_temp;
            exp = two*exp;
            g_temp < v
        }{}
    }else{
        (g_temp,g_past,exp)=(U::one(),U::one(),U::zero());
    }
    return (g_temp,g_past,exp);
}


//*
pub fn find_power<U>( g:U, max:U)->(U,U)
where U: Unsigned + PartialOrd + Copy + Display+ std::ops::BitAnd,
{
    let one :U = U::one();
    let zero:U = U::zero();
    let mut numb:U=one;
    let mut pow:U=zero;
    while numb < max
    {
        numb = numb*g;
        pow = pow +one;
    }
    if numb==max
    {
        return (numb,pow);
```

```
    }else
    {
        return (numb/g,pow-one);
    }
}

// */
//*
pub fn is_a_generator<U>(val:U,modulus:U)-> bool
where
U: Unsigned + PartialOrd + Copy + Display + std::ops::BitAnd<Output = U> +
std::ops::Shr<Output = U> + std::ops::Shl<Output = U> + PrimInt, <U as
BitAnd>::Output: PartialEq<U>
{
    let one :U = U::one();
    //let zero:U = U::zero();
    let two = one +one;
    let div_1:U = (modulus-one)/two;
    if !mod_exp(val,div_1, modulus).is_one()
    {
        return true;
    }
    return false;
}

// */
pub fn max_parameters<U>(g:U,modulus:U)->(U,U,U,U,U,U,U,U,U,U,U,U,U,U,U,U)
where
U: Unsigned + PartialOrd + Copy + Display+ std::ops::BitAnd,
{
    let one :U = U::one();
    let zero:U = U::zero();
    let (mut g_max,mut pow_max):(U,U)=find_power(g,modulus);
    let (mut g_n_3,mut pow_n_3):(U,U)=find_power(g,modulus*modulus*modulus);
    let (mut g_n_2,mut pow_n_2):(U,U)=find_power(g,modulus*modulus);

    g_n_3=g_n_3/g;
    pow_n_3= pow_n_3 -one;

    println!("pow_n_2: {}",pow_n_2);

    let q_max:U = modulus/g_max;
    let q:U = modulus/g;
    let r_g_max :U = modulus -g_max*q_max;
    let r_g:U = modulus - g*q;
```

```
    let q_3:U = g_n_3/modulus;
    let r_g_3:U= g_n_3 - q_3*modulus;

    let inv_rg_max:U= inverse_mod(r_g_max,g_max);
    let inv_rg:U= inverse_mod(r_g,g);
    let inv_modulus_3:U = inverse_mod(modulus,g_n_3);
    let inv_modulus_2:U = inverse_mod(modulus,g_n_2);

    return
(g_max,pow_max,q_max,q,r_g_max,r_g,inv_rg_max,inv_rg,inv_modulus_3,g_n_3,pow_
n_3,inv_modulus_2,g_n_2,pow_n_2,q_3,r_g_3);
}


//*
pub fn
discrete_log_rem_inverse<U>(g:U,r:U,modulus:U,g_n_3:U,pow_n_3:U,inv_modulus_3
:U)->(U,U,U)
where
U: Unsigned + PartialOrd + Copy + Display + std::ops::BitAnd<Output = U> +
std::ops::Shr<Output = U> + std::ops::Shl<Output = U> + PrimInt, <U as
BitAnd>::Output: PartialEq<U>
{

    let one :U = U::one();
    let zero:U = U::zero();

    let mut k:U;
    let mut k_i:U;
    let mut v_temp:U= r;
    let mut v_aux:U = v_temp;

    let mut v_inverse_temp:U= inverse_mod(r,modulus);
    let mut v_inverse_aux:U = v_inverse_temp;

    let mut a:U = zero;
    let mut a_i:U = zero;
    let mut c=zero;
    let mut c_i=zero;
    let mut c_2=zero;
    let mut c_2_i=zero;
    let mask_3 = g_n_3-one;

    //*
    if (v_temp&(v_temp-one)).is_zero()
```

```
    {
        while (v_temp & one) == zero
        {

            a = a + one;
            v_temp = v_temp>>one;
            c_2 = c_2 +one;

        }
    }
    // */

    //*
    if (v_inverse_temp&(v_inverse_temp-one)).is_zero()
    {
        while (v_inverse_temp & one) == zero
        {

            a_i = a_i + one;
            v_inverse_temp = v_inverse_temp>>one;
            c_2_i = c_2_i +one;

        }
    }
    // */

    while !v_temp.is_one() && !v_inverse_temp.is_one()
    {


            c=c+one;
            c_i=c_i+one;
            k = ((g_n_3-(v_temp&mask_3))*inv_modulus_3)&mask_3;
            k_i = ((g_n_3-(v_inverse_temp&mask_3))*inv_modulus_3)&mask_3;

            v_aux = v_temp;
            v_temp = (v_temp +k*modulus)>>pow_n_3;
            a = a + pow_n_3;

            v_inverse_aux = v_inverse_temp;
            v_inverse_temp = (v_inverse_temp +k_i*modulus)>>pow_n_3;
            a_i = a_i + pow_n_3;

            //*
            if (v_temp&(v_temp-one)).is_zero()
```

```
            {
                while (v_temp & one) == zero
                {

                    a = a + one;
                    v_temp = v_temp>>one;
                    c_2 = c_2 +one;

                }
            }else if  (v_aux&one).is_zero()  && (v_temp & one).is_zero() //
need to search more for that
            {
                a = a + one;
                v_temp = v_temp>>one;
                c_2 = c_2 +one;

            }
            // */

            //*
            if (v_inverse_temp&(v_inverse_temp-one)).is_zero()
            {
                while (v_inverse_temp & one) == zero
                {

                    a_i = a_i + one;
                    v_inverse_temp = v_inverse_temp>>one;
                    c_2_i = c_2_i +one;

                }
            }else if  (v_inverse_aux&one).is_zero()  && (v_inverse_temp &
one).is_zero() // need to search more for that
            {
                a_i = a_i + one;
                v_inverse_temp = v_inverse_temp>>one;
                c_2_i = c_2_i +one;

            }
            // */

    }
    if v_temp.is_one()
    {
        a= a%(modulus-one);
    }else
```

```
    {
        a= (modulus - one - a_i)%(modulus-one);
        c=c_i;
        c_2=c_2_i;
    }

    return (a,c,c_2);

}
// */

//*
pub fn
discrete_log_rem<U>(g:U,r:U,modulus:U,g_n_3:U,pow_n_3:U,inv_modulus_3:U)-
>(U,U,U)
where
U: Unsigned + PartialOrd + Copy + Display + std::ops::BitAnd<Output = U> +
std::ops::Shr<Output = U> + std::ops::Shl<Output = U> + PrimInt, <U as
BitAnd>::Output: PartialEq<U>
{

    let one :U = U::one();
    let zero:U = U::zero();

    let mut k:U;
    let mut v_temp:U= r;
    let mut v_aux:U = v_temp;
    let mut a:U = zero;
    let mut c=zero;
    let mut c_2=zero;
    let mask_3 = g_n_3-one;
    //*
    if (v_temp&(v_temp-one)).is_zero()
    {
        while (v_temp & one) == zero
        {

            a = a + one;
            v_temp = v_temp>>one;
            c_2 = c_2 +one;

        }
    }
    // */
    while !v_temp.is_one()
```

```
    {


            c=c+one;

            k = ((g_n_3-(v_temp&mask_3))*inv_modulus_3)&mask_3;

            v_aux = v_temp;
            v_temp = (v_temp +k*modulus)>>pow_n_3;
            a = a + pow_n_3;


            //*
            if (v_temp&(v_temp-one)).is_zero()
            {
                while (v_temp & one) == zero
                {

                    a = a + one;
                    v_temp = v_temp>>one;
                    c_2 = c_2 +one;

                }
            }else if  (v_aux&one).is_zero()  && (v_temp & one).is_zero() //
need to search more for that
            {
                a = a + one;
                v_temp = v_temp>>one;
                c_2 = c_2 +one;

            }
            // */

    }

    a= a%(modulus-one);
    return (a,c,c_2);

}
// */

pub fn discrete_log_hash<U>(r:U,modulus:U,g_n:U,inv_g_n:U,pow_n:U,
rest_hash:&HashMap<U,U>)->(U,U,U)
where
U: Unsigned + PartialOrd + Copy + Display + std::ops::BitAnd<Output = U> +
```

```
std::ops::Shr<Output = U> + std::ops::Shl<Output = U> + PrimInt, <U as
BitAnd>::Output: PartialEq<U>, U:Hash
{

    let one :U = U::one();
    let zero:U = U::zero();

    let mut v_temp:U = r;
    let mut a: U = zero;
    let mut c: U =zero;
    let mut c_2:U=zero;

    match rest_hash.get(&r)
    {
        Some(pow) => {a = *pow;v_temp=one;c_2=*pow;},
        None =>{},
    }


    while !v_temp.is_one()
    {
        c=c+one;
        v_temp = (v_temp*inv_g_n)%modulus;
        a = a + pow_n;
        match rest_hash.get(&v_temp)
        {
            Some(pow) => {a = a + *pow; v_temp=one;c_2=*pow;},
            None =>{},
        }
    }

    a= a%(modulus-one);

    return (a,c,c_2);

}
// */
pub fn discrete_log_hash_8<U>(r:U,modulus:U,g_n:U,inv_g_n:U,pow_n:U,
rest_hash:&HashMap<U,U>)->(U,U,U)
where
U: Unsigned + PartialOrd + Copy + Display + std::ops::BitAnd<Output = U> +
std::ops::Shr<Output = U> + std::ops::Shl<Output = U> + PrimInt, <U as
BitAnd>::Output: PartialEq<U>, U:Hash
{
```

```
    let one :U = U::one();
    let zero:U = U::zero();
    let three:U = one+one+one;
    let seven:U = three + three + one;
    let mut v_temp:U = r;
    let mut a: U = zero;
    let mut c: U = zero;
    let mut c_2:U = zero;

    match rest_hash.get(&r)
    {
        Some(pow) => {a = *pow;v_temp=one;c_2=*pow;},
        None =>{},
    }


    while !v_temp.is_one()
    {

        c=c+one;
        v_temp = (v_temp*inv_g_n)%modulus;
        a = a + pow_n;
        //*
        while (v_temp&seven) == zero
        {
            a= a+three;
            v_temp = v_temp>>three;
            match rest_hash.get(&v_temp)
            {
                Some(pow) => {a = a + *pow; v_temp=one;c_2=*pow;},
                None =>{},
            }
        }
        // */
        /*
        while (v_temp&one) == zero
        {
            println!("loop {}",v_temp);
            a= a+one;
            v_temp = v_temp>>one;
            match rest_hash.get(&v_temp)
            {
                Some(pow) => {a = a + *pow; v_temp=one;c_2=*pow;},
                None =>{},
            }
```

```
        }
        // */
        if v_temp != one
        {
            match rest_hash.get(&v_temp)
            {
                Some(pow) => {a = a + *pow; v_temp=one;c_2=*pow;},
                None =>{},
            }
        }

    }

    a= a%(modulus-one);

    return (a,c,c_2);

}

// */
pub fn discrete_log_hash_16<U>(r:U,modulus:U,g_n:U,inv_g_n:U,pow_n:U,
rest_hash:&HashMap<U,U>)->(U,U,U)
where
U: Unsigned + PartialOrd + Copy + Display + std::ops::BitAnd<Output = U> +
std::ops::Shr<Output = U> + std::ops::Shl<Output = U> + PrimInt, <U as
BitAnd>::Output: PartialEq<U>, U:Hash
{

    let one :U = U::one();
    let zero:U = U::zero();
    let three:U = one+one+one;
    let four:U = three + one;
    let seven:U = three + three + one;
    let fifteen:U = seven + seven + one;
    let mut v_temp:U = r;
    let mut a: U = zero;
    let mut c: U = zero;
    let mut c_2:U = zero;

    match rest_hash.get(&r)
    {
        Some(pow) => {a = *pow;v_temp=one;c_2=*pow;},
        None =>{},
    }
```

```
    while !v_temp.is_one()
    {

        c=c+one;
        v_temp = (v_temp*inv_g_n)%modulus;
        a = a + pow_n;
        while (v_temp&fifteen) == zero
        {
            a= a+four;
            v_temp = v_temp>>four;
            match rest_hash.get(&v_temp)
            {
                Some(pow) => {a = a + *pow; v_temp=one;c_2=*pow;},
                None =>{},
            }
        }
        if v_temp != one
        {
            match rest_hash.get(&v_temp)
            {
                Some(pow) => {a = a + *pow; v_temp=one;c_2=*pow;},
                None =>{},
            }
        }

    }

    a= a%(modulus-one);

    return (a,c,c_2);

}


//*
pub fn
discrete_log_g_n_3<U>(g:U,inv_g:U,r:U,modulus:U,g_n_3:U,g_n_2:U,g_min:U,pow_n
_3:U,pow_n_2:U,pow_min:U,q_min:U,q:U,r_g_min:U,inv_modulus_3:U,inv_modulus_2:
U,inv_rg_min:U,inv_rg:U,r_g:U)->(U,U,U)
where
U: Unsigned + PartialOrd + Copy + Display + std::ops::BitAnd<Output = U> +
std::ops::Shr<Output = U> + std::ops::Shl<Output = U> + PrimInt, <U as
BitAnd>::Output: PartialEq<U>
{
```

```
let one :U = U::one();
let zero:U = U::zero();
let mut limit = zero;
let mut k:U;
let mut v_temp:U= r;
let mut v_aux:U= r;
let mut a:U = zero;
let mut c=zero;
let mut c_2=zero;
let mask_3 = g_n_3-one;
let mask_2 = g_n_2-one;
let mask_min = g_min-one;

while !v_temp.is_one()
//while  !(v_temp&(v_temp-one)).is_zero()
{


        c=c+one;

        /*
        k = ((g_n_3-(v_temp&mask_3))*inv_modulus_3)&mask_3;
        v_temp = (v_temp+k*modulus)>>pow_n_3;
        a = a + pow_n_3;

        // */
        //*

        v_aux = v_temp;
        k = ((g_n_2-(v_temp&mask_2))*inv_modulus_2)&mask_2;
        v_temp = (v_temp+k*modulus)>>pow_n_2;
        a = a + pow_n_2;

        // */

        /*
        k = ((g_min-(v_temp&mask_min))*inv_rg_min)&mask_min;
        v_temp = (v_temp+k*modulus)>>pow_min;
        a = a + pow_min;
        // */



        //*
```

```
            if (v_temp&(v_temp-one)).is_zero()
            {

                //panic!("v_temp {} a:{} c:{} c_2:{}",v_temp,a,c,c_2);
                while (v_temp & one) == zero
                {

                    a = a + one;
                    v_temp = v_temp>>one;
                    //c_2 = c_2 +one;

                }
            }// */
            else if ((v_temp+modulus)&(v_temp + modulus-one)).is_zero()
            {
                v_temp= v_temp+modulus;
                //panic!("v_temp {} a:{} c:{} c_2:{}",v_temp,a,c,c_2);
                while (v_temp & one) == zero
                {
                    a = a + one;
                    v_temp = v_temp>>one;
                    c_2 = c_2 +one;
                }
            }
            //*
            else if (v_temp&one) != zero
            {
                a = a + one;
                v_temp = (inv_g*v_temp)%modulus;
                c_2 = c_2 +one;

            }
            // */
            /*
            else if (v_temp&one) == zero
            {
                a = a + one;
                //v_temp = v_temp>>one;
                v_temp = (inv_g*v_temp)%modulus;
                c_2 = c_2 +one;

            }
            // */

    }
```

```
    a= a%(modulus-one);

    return (a,c,c_2);

}
// */

pub fn super_simple_discrete_log<U>(g:U,r:U, modulus:U)->U
where
U: Unsigned + PartialOrd + Copy + Display + std::ops::BitAnd<Output = U> +
std::ops::Shr<Output = U> + std::ops::Shl<Output = U> + PrimInt, <U as
BitAnd>::Output: PartialEq<U>
{
    let one :U = U::one();
    let zero:U = U::zero();
    let mut v = r;
    let mut x = zero;
    let mut vmask= zero;
    while v != one
    {
        vmask = v&one;
        v = (v+vmask*modulus)>>one;
        x = x +one;
    }
    return x;
}

pub fn
discrete_log_n<U>(g:U,r:U,g_max:U,pow_max:U,q_max:U,r_g_max:U,inv_rg_max:U)-
>U
where
U: Unsigned + PartialOrd + Copy + Display + std::ops::BitAnd,
{
    let one :U = U::one();
    let zero:U = U::zero();


    let mut k:U;
    let mut v_temp:U=r;
    let mut a:U = zero;
    while !v_temp.is_one()
    {

        k=( (g_max-(v_temp%g_max))*inv_rg_max)%g_max;
```

```
        v_temp =(v_temp+k*r_g_max)/g_max + k*q_max;
        a = a + pow_max;

        while (v_temp%g).is_zero()
        {
            a = a + one;
            v_temp = v_temp/g;
            //println!("v_temp%g in :{}",v_temp%g);
            //println!("a{}",a);
        }
    }
    //println!("loop_counter:{}",loop_counter);
    return a;

}



fn main() {


    //let modulus_2:u128 =modulus*modulus*modulus;
    //let g:u128=911;
    //let g:u128=5;
    let g:u128=2;
    let mut r:u128;
    let mut x:u128;
    let mut c:u128;
    let mut c_2:u128;
    let mut vec_pair:Vec<(u128,u128,f64)>=vec![(0,0,0.0);0];
    let mut vec_pair_rem:Vec<(u128,u128,f64)>=vec![(0,0,0.0);0];
    let mut vec_pair_rem_inverse:Vec<(u128,u128,f64)>=vec![(0,0,0.0);0];


    /*
    let mut start_time_n_3;
    let mut elapsed_time_n_3=Duration::ZERO;
    let mut elapsed_time_temp=Duration::ZERO;
    // */

    /*
    let mut start_time_rem;
    let mut elapsed_time_rem=Duration::ZERO;
    let mut elapsed_time_rem_temp=Duration::ZERO;
    // */
```

```
    /*
    let mut start_time_rem_inverse;
    let mut elapsed_time_rem_inverse=Duration::ZERO;
    let mut elapsed_time_rem_inverse_temp=Duration::ZERO;
    // */


    /*
    let mut start_time_mul_inverse;
    let mut elapsed_time_mul_inverse=Duration::ZERO;
    //let mut elapsed_time_mul_inverse_temp=Duration::ZERO;
    // */

    //*
    let mut start_time_hash;
    let mut elapsed_time_hash=Duration::ZERO;
    let mut elapsed_time_hash_8=Duration::ZERO;
    let mut elapsed_time_hash_16=Duration::ZERO;
    let mut elapsed_time_hash_temp=Duration::ZERO;
    // */

    //*
    let mut start_time_super_simple;
    let mut elapsed_time_super_simple=Duration::ZERO;
    let mut elapsed_time_super_simple_temp=Duration::ZERO;
    // */



    //let mut modulus:u128 =1048343;

    /*
    let
(g_max,pow_max,q_max,q,r_g_max,r_g,inv_rg_max,inv_rg)=max_parameters(g,modulu
s);

    start_time_n = Instant::now();


    for i in 1047342..1048343{

    r=i as u128;

            x=
```

```
discrete_log_n(g,r,modulus,g_max,pow_max,q_max,q,r_g_max,r_g,inv_rg_max,inv_rg)

    }

    elapsed_time_n += start_time_n.elapsed();
    println!("Average Elapsed time n for modulus {:?} : {:?}",modulus,
elapsed_time_n.as_secs_f64()/1000.);

    // */

    let modulus = 2697803;

    //let
(g_max,pow_max,q_max,q,r_g_max,r_g,inv_rg_max,inv_rg,inv_modulus_3,g_n_3,pow_
n_3,inv_modulus_2,g_n_2,pow_n_2,q_3,r_g_3)=max_parameters(g,modulus);

    //let inv_g = inverse_mod(g,modulus);

    //let pow_n = 2697801; //100097;
    //let pow_n =  101097;
    let pow_n = 189;

    let mut r;
    let g_n = mod_exp(g,pow_n,modulus) as u128;
    let g_n_8 = mod_exp(g,8*pow_n,modulus) as u128;
    let g_n_16 = mod_exp(g,16*pow_n,modulus) as u128;
    let inv_g_n = inverse_mod(g_n,modulus);
    let inv_g_n_8 = inverse_mod(g_n_8,modulus);
    let inv_g_n_16 = inverse_mod(g_n_16,modulus);
    let mut rest_hash:HashMap<u128,u128> = HashMap::new();
    let mut rest_hash_8:HashMap<u128,u128> = HashMap::new();
    let mut rest_hash_16:HashMap<u128,u128> = HashMap::new();

    for i in 1..pow_n/3
    {
        rest_hash.insert(mod_exp(g,i,modulus),i,);
    }
    for i in 1..22
    {
        let v = mod_exp(g,i,modulus);
        rest_hash_8.insert(v,i,);
        rest_hash_16.insert(v,i,);
    }
    for i in 22..3*pow_n
    {
```

```
        let v = mod_exp(g,i,modulus);
        if v < modulus/8
        {
            rest_hash_8.insert(v,i,);
        }
    }
    for i in 22..4*pow_n
    {
        let v = mod_exp(g,i,modulus);
        if v < modulus/16
        {
            rest_hash_16.insert(v,i,);
        }
    }
    for i in modulus-201..modulus
    {
        r= i as u128;
        start_time_hash=Instant::now();
        let (x,c,c_2)=discrete_log_hash(r,modulus,g_n,inv_g_n,pow_n,
&rest_hash);
        elapsed_time_hash += start_time_hash.elapsed();
        elapsed_time_hash_temp = elapsed_time_hash-elapsed_time_hash_temp;

        println!("{}^{} = {} [{}] ",g,x,r,modulus);
        println!("g_n :2^{} ",pow_n);
        println!("c*pow_n+c_2 :{} ",c*pow_n+c_2);
        println!("c:{} ",c);
        assert_eq!(mod_exp(g, x, modulus),r);
        println!("Elapsed time hash temp: {:?}",elapsed_time_hash_temp);
        elapsed_time_hash_temp=elapsed_time_hash;
        //*
        start_time_super_simple=Instant::now();
        let x=super_simple_discrete_log(g,r,modulus);
        elapsed_time_super_simple += start_time_super_simple.elapsed();
        elapsed_time_super_simple_temp = elapsed_time_super_simple-
elapsed_time_super_simple_temp;

        println!("{}^{} = {} [{}] ",g,x,r,modulus);
        assert_eq!(mod_exp(g, x, modulus),r);
        println!("Elapsed time super simple:
{:?}",elapsed_time_super_simple_temp);
        elapsed_time_super_simple_temp=elapsed_time_super_simple;
        // */
    }
    elapsed_time_hash_temp=Duration::ZERO;
```

```
    for i in modulus-201..modulus
    {
        r= i as u128;
        start_time_hash=Instant::now();
        let (x,c,c_2)=discrete_log_hash_8(r,modulus,g_n_8,inv_g_n_8,8*pow_n,
&rest_hash_8);
        elapsed_time_hash_8 += start_time_hash.elapsed();
        elapsed_time_hash_temp = elapsed_time_hash_8-elapsed_time_hash_temp;

        println!("{}^{} = {} [{}] ",g,x,r,modulus);
        println!("g_n :2^{} ",8*pow_n);
        println!("c*pow_n+c_2 :{} ",8*c*pow_n+c_2);
        println!("c:{} ",c);
        assert_eq!(mod_exp(g, x, modulus),r);
        println!("Elapsed time hash temp: {:?}",elapsed_time_hash_temp);
        elapsed_time_hash_temp=elapsed_time_hash_8;
    }
    elapsed_time_hash_temp=Duration::ZERO;
    /*
    for i in modulus-201..modulus
    {
        r= i as u128;
        start_time_hash=Instant::now();
        let
(x,c,c_2)=discrete_log_hash_16(r,modulus,g_n_16,inv_g_n_16,16*pow_n,
&rest_hash_16);
        elapsed_time_hash_16 += start_time_hash.elapsed();
        elapsed_time_hash_temp = elapsed_time_hash_16-elapsed_time_hash_temp;

        println!("{}^{} = {} [{}] ",g,x,r,modulus);
        println!("g_n :2^{} ",16*pow_n);
        println!("c*pow_n+c_2 :{} ",16*c*pow_n+c_2);
        println!("c:{} ",c);
        assert_eq!(mod_exp(g, x, modulus),r);
        println!("Elapsed time hash temp: {:?}",elapsed_time_hash_temp);
        elapsed_time_hash_temp=elapsed_time_hash_16;
    }
    // */
    println!("Average Elapsed time hash {:?} : {:?}",modulus,
elapsed_time_hash.as_secs_f64()/200.);
    println!("Average Elapsed time hash _8 {:?} : {:?}",modulus,
elapsed_time_hash_8.as_secs_f64()/200.);
    println!("Average Elapsed time hash _8 {:?} : {:?}",modulus,
elapsed_time_hash_16.as_secs_f64()/200.);
```

```
    println!("Average Elapsed time super simple {:?} : {:?}",modulus,
elapsed_time_super_simple.as_secs_f64()/20000.);

    /*
        let mut pow_n = modulus -2;
        let mut g_n = mod_exp(g,pow_n,modulus) as u128;
        if is_a_generator(g_n,modulus)
        {
            let mut g_n_inv = inverse_mod(g_n,modulus);
            let mut parity = 0;
            let r = modulus-10001;
            start_time_mul_inverse = Instant::now();
            let (mut x,mut c,mut c_2, mut p) =
discrete_log_mult_by_inverse(g,inv_g,r,modulus,g_n,g_n_inv,pow_n,parity);
            while c*pow_n + c_2 > 2*modulus -2
            {
                //pow_n= (pow_n*10)/11;
                pow_n = pow_n -1001;
                g_n = mod_exp(g,pow_n,modulus) as u128;
                g_n_inv = inverse_mod(g_n,modulus);
                (x,c,c_2, p) =
discrete_log_mult_by_inverse(g,inv_g,r,modulus,g_n,g_n_inv,pow_n,parity);
            }
            elapsed_time_mul_inverse = start_time_mul_inverse.elapsed();

            /*
            if c*pow_n +c_2 > 2*modulus-2
            {
                parity = 1;
                ( x, c,c_2, p) =
discrete_log_mult_by_inverse(g,inv_g,r,modulus,g_n,g_n_inv,pow_n,parity);
            }
            // */
            println!("pow_n:{}",pow_n);
            if mod_exp(g,x, modulus) == r
            {
                println!("{}^{} = {} [{}] ",g,x,r,modulus);
                println!("g_n :2^{} ",pow_n);
                println!("g_n_inv :{}",g_n_inv);
                println!("elapsed_time_mul_inverse:
{:?}",elapsed_time_mul_inverse);
                println!("parity {} ",p);
                println!("compteur = {} ",c);
                println!("compteur 2= {} ",c_2);
```

```
                println!("c*pow_n+c_2 = {} ",c*pow_n+c_2);
                println!("c*pow_n+c_2%modulus = {} ",(c*pow_n+c_2)%
(modulus-1));

            }
        }
    // */



    /*
    //for i in 2697664..2697665{
    for i in (modulus-10001)..modulus{
    //for i in (modulus-1)..modulus{
    //for i in 3..modulus{
        r=i as u128;
        //*
        start_time_n_3 = Instant::now();
        (x,c,c_2)=
discrete_log_g_n_3(g,inv_g,r,modulus,g_n_3,g_n_2,g_max,pow_n_3,pow_n_2,pow_max,
        elapsed_time_n_3 += start_time_n_3.elapsed();
        elapsed_time_temp = elapsed_time_n_3-elapsed_time_temp;
        println!("{}^{} = {} [{}] ",g,x,r,modulus);
        println!("compteur = {} ",c);
        println!("compteur 2= {} ",c_2);
        // block 63
        //*
        println!("c*63+c_2 = {} ",63*c+c_2);
        println!("c*63+c_2%(modulus-1)) = {} ",(63*c+c_2)%(modulus-1));
        //assert_eq!((63*c+c_2)%(modulus-1),x);
        // */

        // block 45

        /*
        println!("c*45+c_2 = {} ",45*c+c_2);
        println!("c*45+c_2%(modulus-1)) = {} ",(45*c+c_2)%(modulus-1));
        assert_eq!((45*c+c_2)%(modulus-1),x);
        // */

        // block 21
        /*
        println!("c*21+c_2 = {} ",21*c+c_2);
        println!("c*21+c_2%(modulus-1)) = {} ",(21*c+c_2)%(modulus-1));
        assert_eq!((21*c+c_2)%(modulus-1),x);
        // */
```

```
        // block 11
        /*
        println!("c*11+c_2 = {} ",11*c+c_2);
        println!("c*11+c_2%(modulus-1)) = {} ",(11*c+c_2)%(modulus-1));
        assert_eq!((11*c+c_2)%(modulus-1),x);
        // */

        println!("x/compteur = {} ",x/c);
        println!("x%compteur = {} ",x%c);
        //vec_pair.push((x,r,elapsed_time_temp.as_secs_f64()));
        vec_pair.push((x,r,(c+c_2) as f64));
        println!("Elapsed time temp: {:?}",elapsed_time_temp);
        elapsed_time_temp=elapsed_time_n_3;
        assert_eq!(mod_exp(g,x,modulus),r);
        // */

        //*
        start_time_rem = Instant::now();
        (x,c,c_2)= discrete_log_rem(g,r,modulus,g_n_3,pow_n_3,inv_modulus_3);
        elapsed_time_rem += start_time_rem.elapsed();
        elapsed_time_rem_temp = elapsed_time_rem-elapsed_time_rem_temp;
        println!("{}^{} = {} [{}] ",g,x,r,modulus);
        println!("compteur = {} ",c);
        println!("compteur 2= {} ",c_2);
        println!("c*63+c_2 = {} ",63*c+c_2);
        println!("c*63+c_2%(modulus-1)) = {} ",(63*c+c_2)%(modulus-1));
        //assert_eq!((63*c+c_2)%(modulus-1),x);
        //println!("c*63+c_2 = {} ",63*c+c_2);
        //assert_ne!(63*c+c_2,x);
        println!("x/compteur = {} ",x/c);
        println!("x%compteur = {} ",x%c);
        //vec_pair.push((x,r,elapsed_time_temp.as_secs_f64()));
        vec_pair_rem.push((x,r,(c+c_2) as f64));
        println!("Elapsed time rem temp: {:?}",elapsed_time_rem_temp);
        elapsed_time_rem_temp=elapsed_time_rem;
        assert_eq!(mod_exp(g,x,modulus),r);
        // */

        //*
        start_time_rem_inverse = Instant::now();
        (x,c,c_2)=
discrete_log_rem_inverse(g,r,modulus,g_n_3,pow_n_3,inv_modulus_3);
        elapsed_time_rem_inverse += start_time_rem_inverse.elapsed();
        elapsed_time_rem_inverse_temp = elapsed_time_rem_inverse-
```

```
elapsed_time_rem_inverse_temp;
        println!("{}^{} = {} [{}] ",g,x,r,modulus);
        println!("compteur = {} ",c);
        println!("compteur 2= {} ",c_2);
        println!("c*63+c_2 = {} ",63*c+c_2);
        println!("c*63+c_2%(modulus-1)) = {} ",(63*c+c_2)%(modulus-1));
        //assert_eq!((63*c+c_2)%(modulus-1),x);
        //println!("c*63+c_2 = {} ",63*c+c_2);
        //assert_ne!(63*c+c_2,x);
        println!("x/compteur = {} ",x/c);
        println!("x%compteur = {} ",x%c);
        //vec_pair.push((x,r,elapsed_time_temp.as_secs_f64()));
        vec_pair_rem_inverse.push((x,r,(c+c_2) as f64));
        println!("Elapsed time rem inverse temp:
{:?}",elapsed_time_rem_inverse_temp);
        elapsed_time_rem_inverse_temp=elapsed_time_rem_inverse;
        assert_eq!(mod_exp(g,x,modulus),r);
        // */


    }
    // */



    //vec_pair.sort_by(|a,b| a.0.cmp(&b.0));
    //vec_pair_rem.sort_by(|a,b| a.0.cmp(&b.0));
    /*
    vec_pair_rem_inverse.sort_by(|a,b| a.0.cmp(&b.0));

    for p in
vec_pair.iter().zip(vec_pair_rem.iter()).zip(vec_pair_rem_inverse.iter())
    {
        let ((pair,pair_rem),pair_rem_inverse) = p;
        print!("2^{}={}",pair.0,pair.1);
        let mut i:usize = 0;
        let j = ((pair.2)/1000.0).round() as usize;
        //let j = ((pair.2)*2000.0).round() as usize;
        while i < j
        {
            print!("o");
            i+=1;
        }
        println!();

        print!("2^{}={}",pair_rem.0,pair_rem.1);
```

```
        let mut i:usize = 0;
        let j = ((pair_rem.2)/1000.0).round() as usize;
        //let j = ((pair_rem.2)*2000.0).round() as usize;
        while i < j
        {
            print!("*");
            i+=1;
        }
        println!();

        print!("2^{}={}",pair_rem_inverse.0,pair_rem_inverse.1);
        let mut i:usize = 0;
        let j = ((pair_rem.2)/1000.0).round() as usize;
        //let j = ((pair_rem.2)*2000.0).round() as usize;
        while i < j
        {
            print!("#");
            i+=1;
        }
        println!();

    }
    // */
    //two vectors visualisation
    /*
    for p in vec_pair.iter().zip(vec_pair_rem.iter())
    {
        let (pair,pair_rem) = p;
        print!("2^{}={}",pair.0,pair.1);
        let mut i:usize = 0;
        let j = ((pair.2)/1000.0).round() as usize;
        //let j = ((pair.2)*2000.0).round() as usize;
        while i < j
        {
            print!("o");
            i+=1;
        }
        println!();

        print!("2^{}={}",pair_rem.0,pair_rem.1);
        let mut i:usize = 0;
        let j = ((pair_rem.2)/1000.0).round() as usize;
        //let j = ((pair_rem.2)*2000.0).round() as usize;
        while i < j
        {
```

```
            print!("*");
            i+=1;
        }
        println!();

    }
    // */
    /*

    println!("Average Elapsed time n_3 for modulus {:?} : {:?}",modulus,
elapsed_time_n_3.as_secs_f64()/10000.);
    println!("Average Elapsed time rem for modulus {:?} : {:?}",modulus,
elapsed_time_rem.as_secs_f64()/10000.);
    println!("Average Elapsed time rem inverse for modulus {:?} :
{:?}",modulus, elapsed_time_rem_inverse.as_secs_f64()/10000.);
    // */

}
```