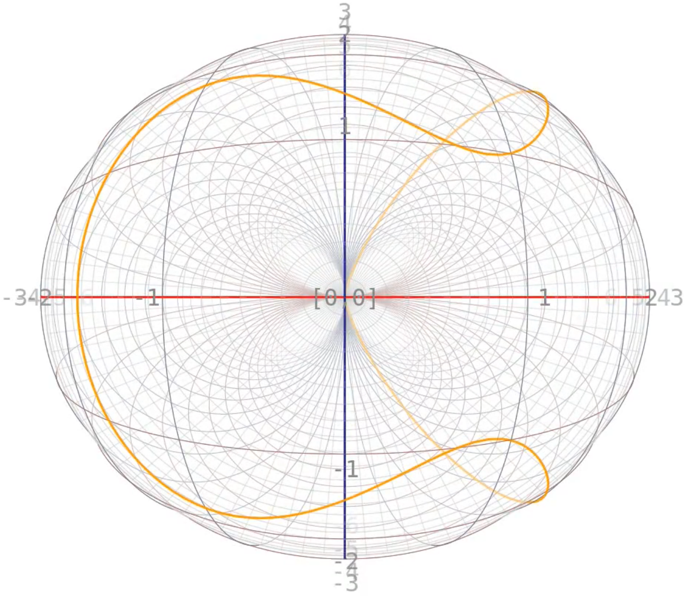
Cryptographie sur Courbes Elliptiques

Juin 2019

Numéro d'inscription : 11951



www.trustica.cz

Sommaire

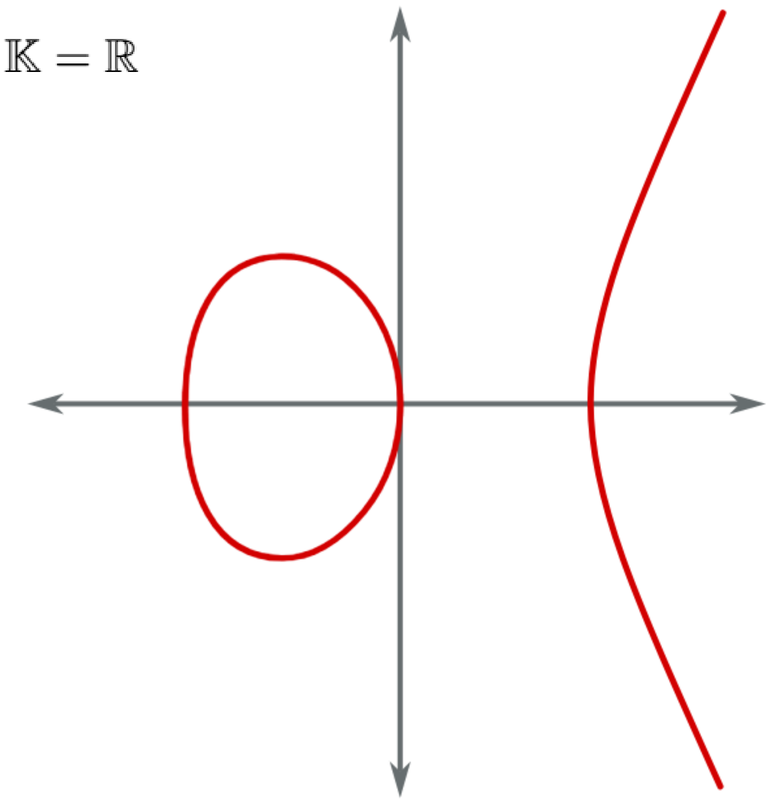
- Construction et Définitions
- Étude des Propriétés Cryptographiques
- Présentation du Cryptosystème
- Annexe

Construction et Définition

Définition d'une Courbe Elliptique

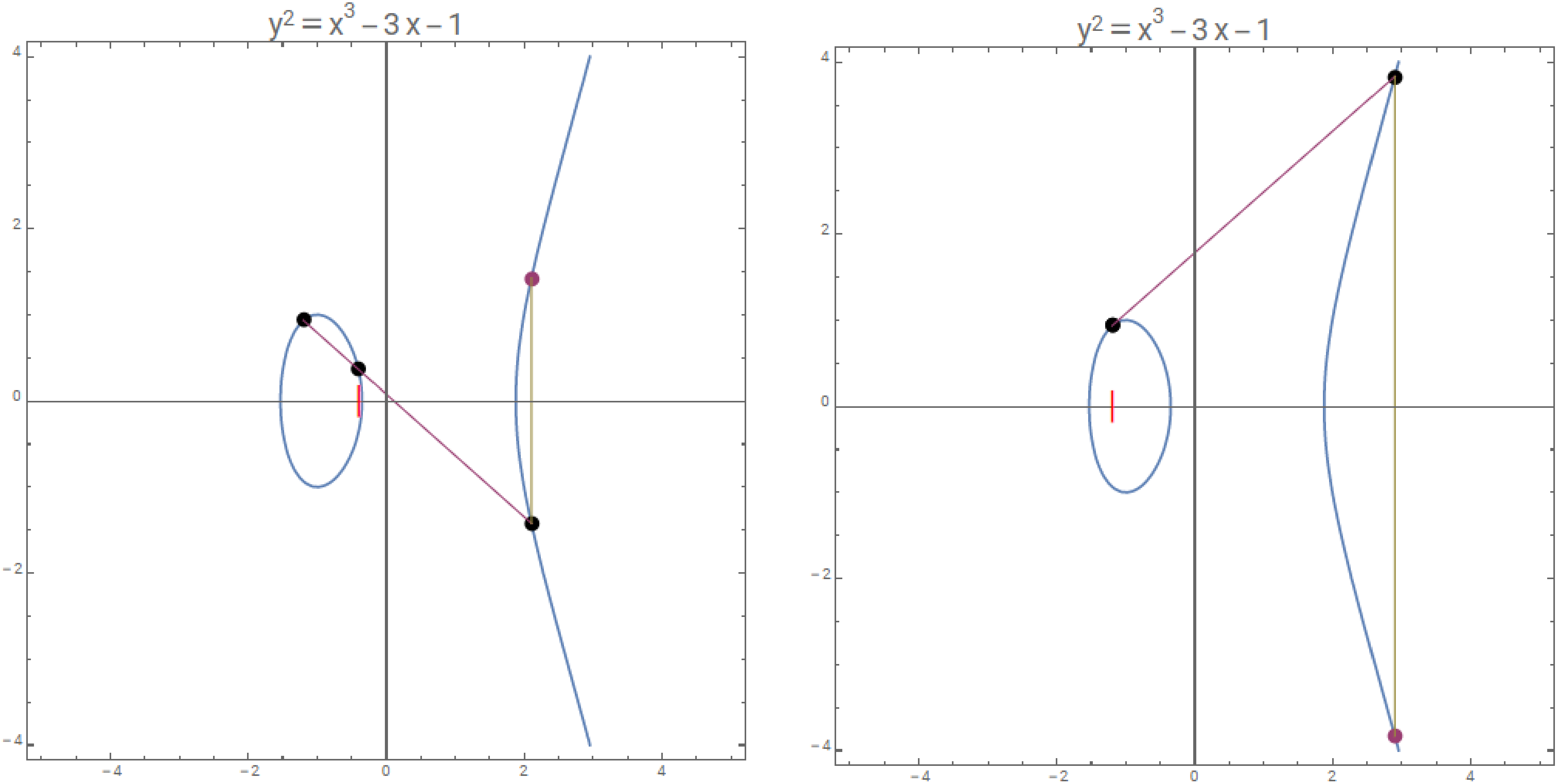
Si $\mathrm{car}(\mathbb{K}) \neq 2, 3$: $\quad E(\mathbb{K}) = \{(x, y) \in \mathbb{K}^2 \ / \ y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$

$\mathbb{K} = \mathbb{R}$



$$y^2 = x^3 - x$$

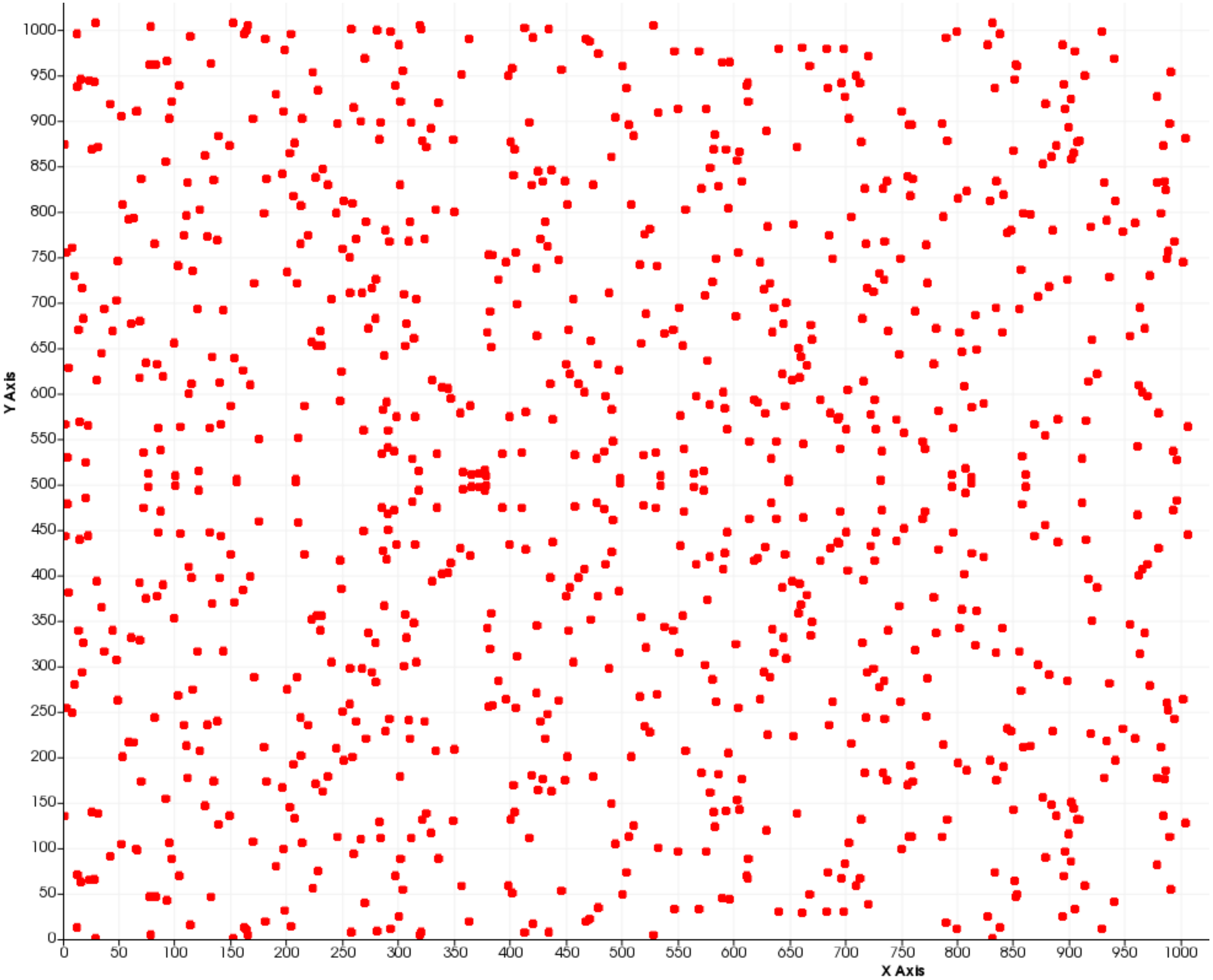
Loi de Groupe



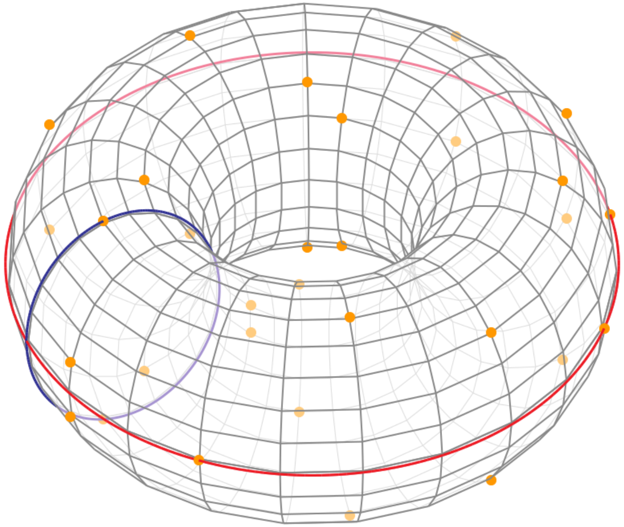
John McGee, WolframAlpha

Courbes Elliptiques sur Corps Finis

On travaille sur $\mathbb{F}_q$:



Il s'agit d'un tore :

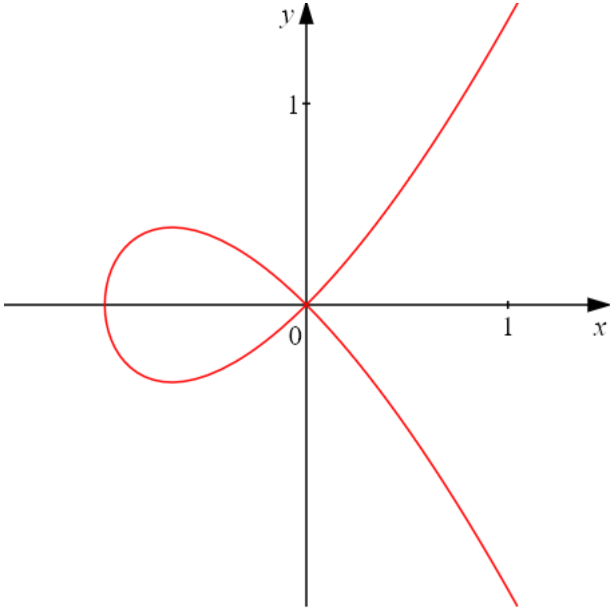


www.trustica.cz

$$y^2 = x^3 + \overline{439}x + \overline{63} \bmod 1009$$

Quelques Cas Particuliers

La courbe ne doit pas présenter de singularité : il faut $\Delta = -16(4a^3 + 27b^2) \neq 0$



La notion de j-invariant disqualifie d'autres courbes
sur lesquelles des attaques connues peuvent être menées.

$$j = -\frac{1728(4a^3)}{\Delta}$$

Préoccupations Cryptographiques

Problème du Logarithme Discret

$G =< g >$ groupe cyclique. Pour $h \in G$ donné il s'agit de trouver $k \in \mathbb{Z}$ tel que $g^k = h$.

Taille des clés à niveau de sécurité égal :

RSA	ECC	Sécurité	Rapport
1024	163	81	1:6
3072	256	128	1:12
7680	384	192	1:20
15360	512	256	1:30

Dans un sens, le calcul est simple : *(exponentiation rapide)*

$$\text{puissance}(x, n) = \begin{cases} x, & \text{si } n = 1 \\ \text{puissance}(x^2, n/2), & \text{si } n \text{ est pair} \\ x \times \text{puissance}(x^2, (n-1)/2), & \text{si } n > 2 \text{ est impair} \end{cases}$$

n	Naïf (ms)	Rapide (ms)
10 000	99,8	0,17
50 000	459,8	0,23
100 000	904,0	0,29
500 000	4 174,5	0,31

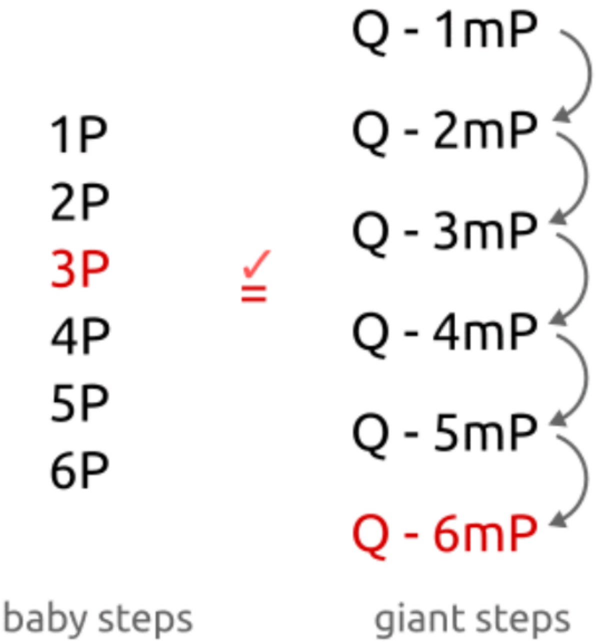
(secp512r1)

S'attaquer au DLP (Discrete Logarithm Problem)

Baby Step Giant Step

$\#G = n, \quad m = \sqrt{n}, \quad a, b \leq \sqrt{n}$

$$\begin{aligned} Q &= xP \\ Q &= (am + b)P \\ Q &= amP + bP \\ Q - amP &= bP \end{aligned}$$



1P
2P
3P
4P
5P
6P

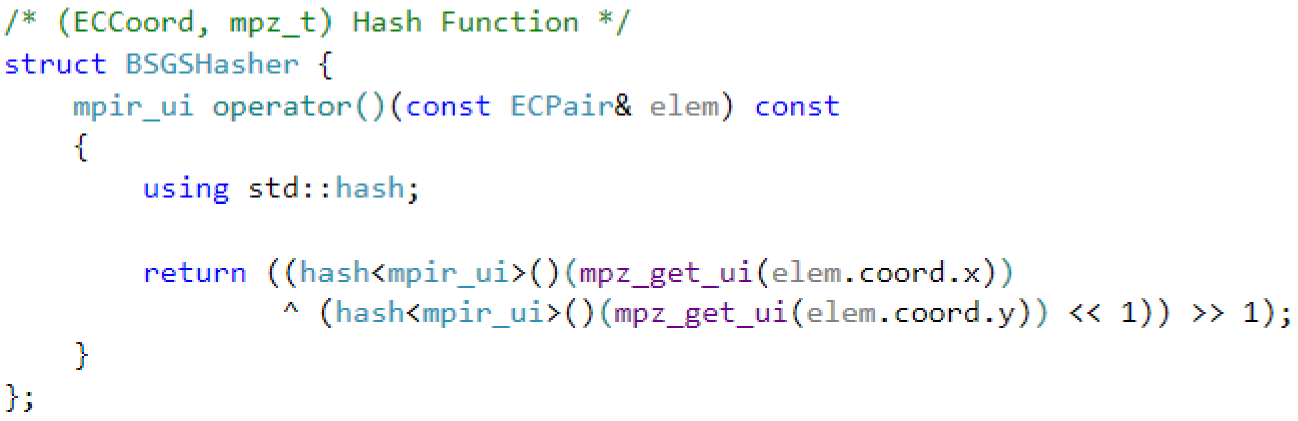
$\overset{\checkmark}{=}$

Q - 1mP
Q - 2mP
Q - 3mP
Q - 4mP
Q - 5mP
Q - 6mP

baby steps

giant steps

Tableau de Hashage : $O(1)$

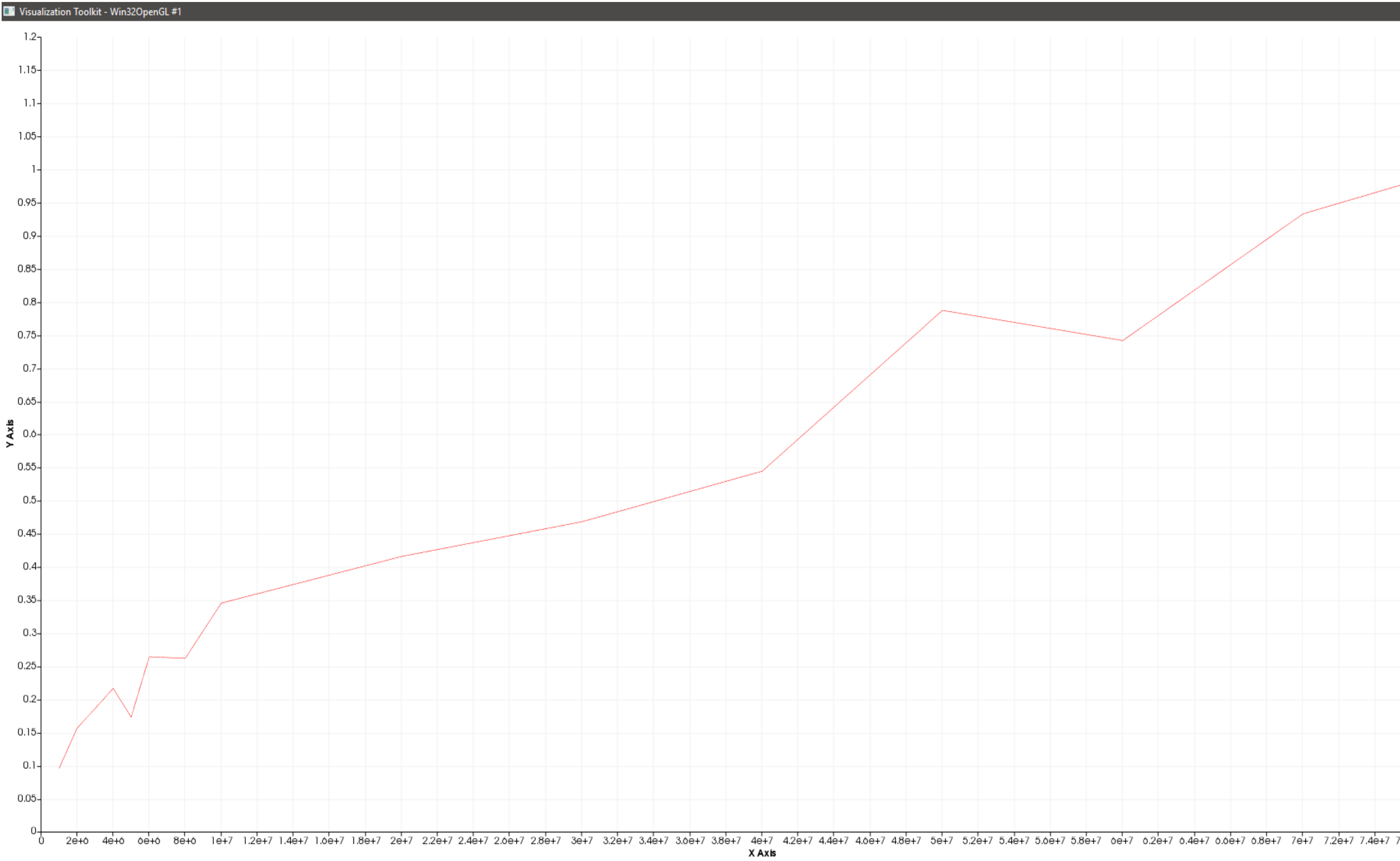


```
/* (ECCoord, mpz_t) Hash Function */
struct BSGSHasher {
    mpir_ui operator()(const ECPair& elem) const
    {
        using std::hash;

        return ((hash<mpir_ui>()(mpz_get_ui(elem.coord.x))
                 ^ (hash<mpir_ui>()(mpz_get_ui(elem.coord.y)) << 1)) >> 1);
    }
};
```

$\longrightarrow \quad O(\sqrt{N})$

Meilleure complexité asymptotique connue. On peut aussi citer *rho* de Pollard

S’attaquer au DLP (Discrete Logarithm Problem)



Calcul d’Ordre, Théorème de Hasse

Une approche naïve :

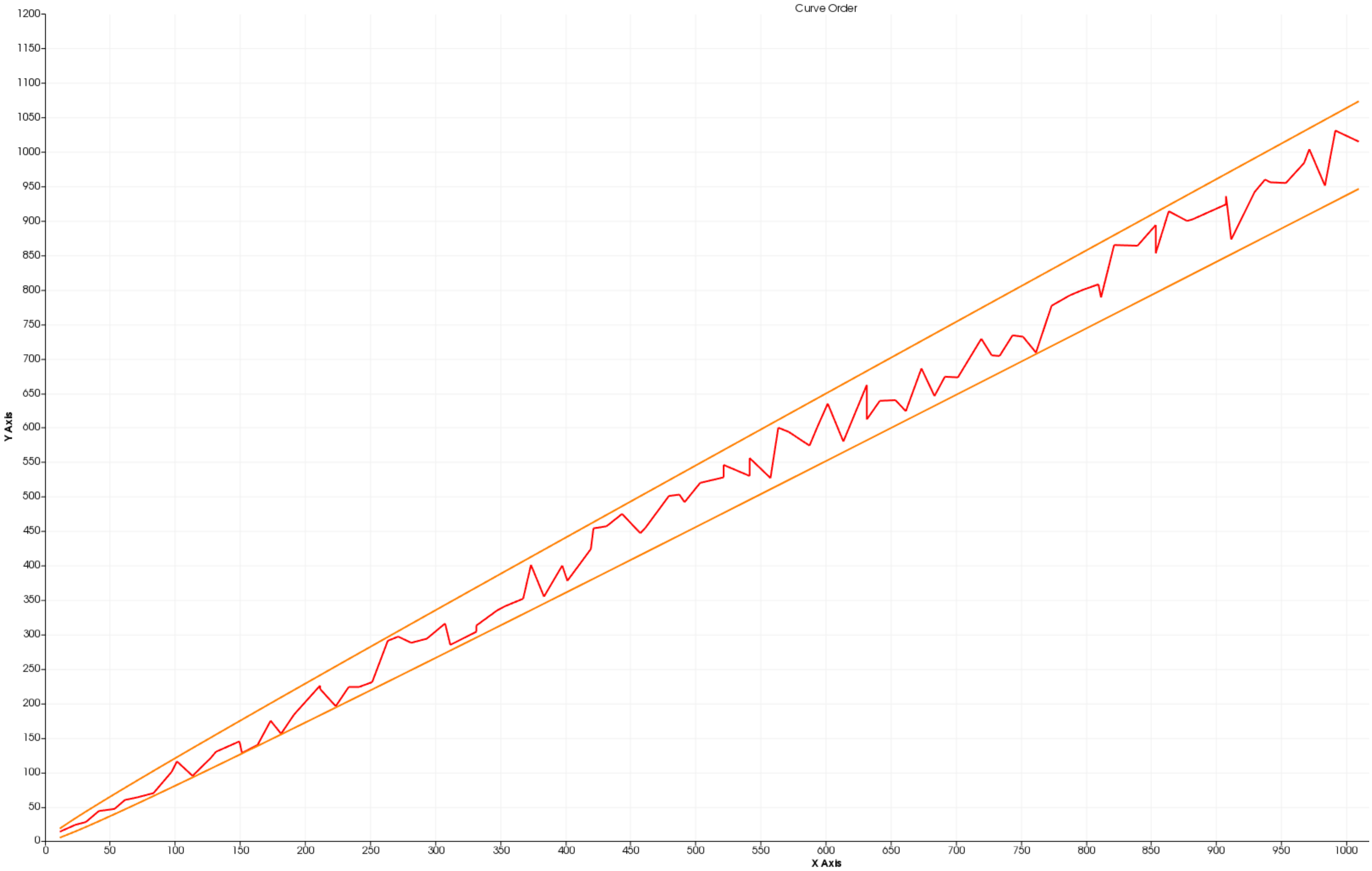
$$\#E(\mathbb{F}_q) = 1 + \sum_{x \in \mathbb{F}_q} \left(1 + \left(\frac{x^3 + ax + b}{q}\right)\right)$$

$O\left(qlog(q)^2\right)$

Théorème *Théorème de Hasse sur les Courbes Elliptiques*

$$|\#E(\mathbb{F}_q) - q - 1| \leq 2\sqrt{q}$$

Théorème de Hasse



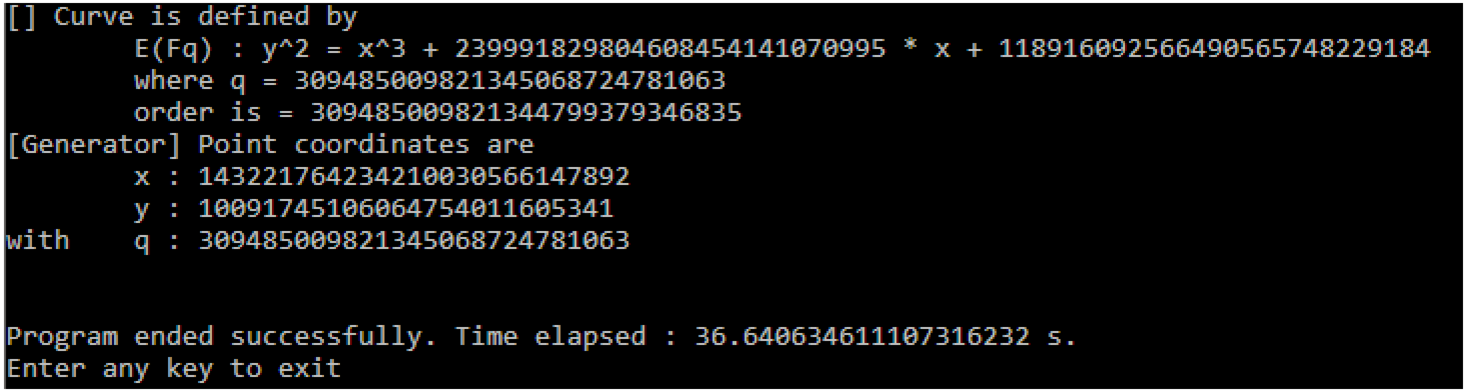
Des Améliorations au Calcul de l’Ordre

(Baby Step Giant Step)

1. $m := \lceil q^{1/4} \rceil$. Stockage de $\{jP\}_{0 \leq j \leq m}$ (Baby Steps).
2. $P \in E(\mathbb{F}_q), \quad Q := (q + 1 - 2\sqrt{q})P$.
3. Collision : $Q + i_0(2mP) = j_0P$ (Giant Steps).
4. Avec $k = q + 1 - 2\sqrt{q} + 2i_0m - j_0$, $kP = \mathcal{O}$.
   On factorise $k = p_1^{\alpha_1} ... p_n^{\alpha_2}$ et on détermine n l’ordre de P.
5. On stocke $n_1, ..., n_i$ jusqu’à ce que $ppcm(n_1, ..., n_i)$ divise un unique nombre $N \in [\![q + 1 - 2\sqrt{q} \ , \ q + 1 + 2\sqrt{q}]\!]$: c’est forcément $\#E(\mathbb{F}_q)$.

$$O(qlog(q)^2) \qquad\qquad \to O(q^{1/4})$$

Calcul envisageable jusqu’à $p \approx 2^{90}$, 1 min de calcul et 2 GB de RAM.

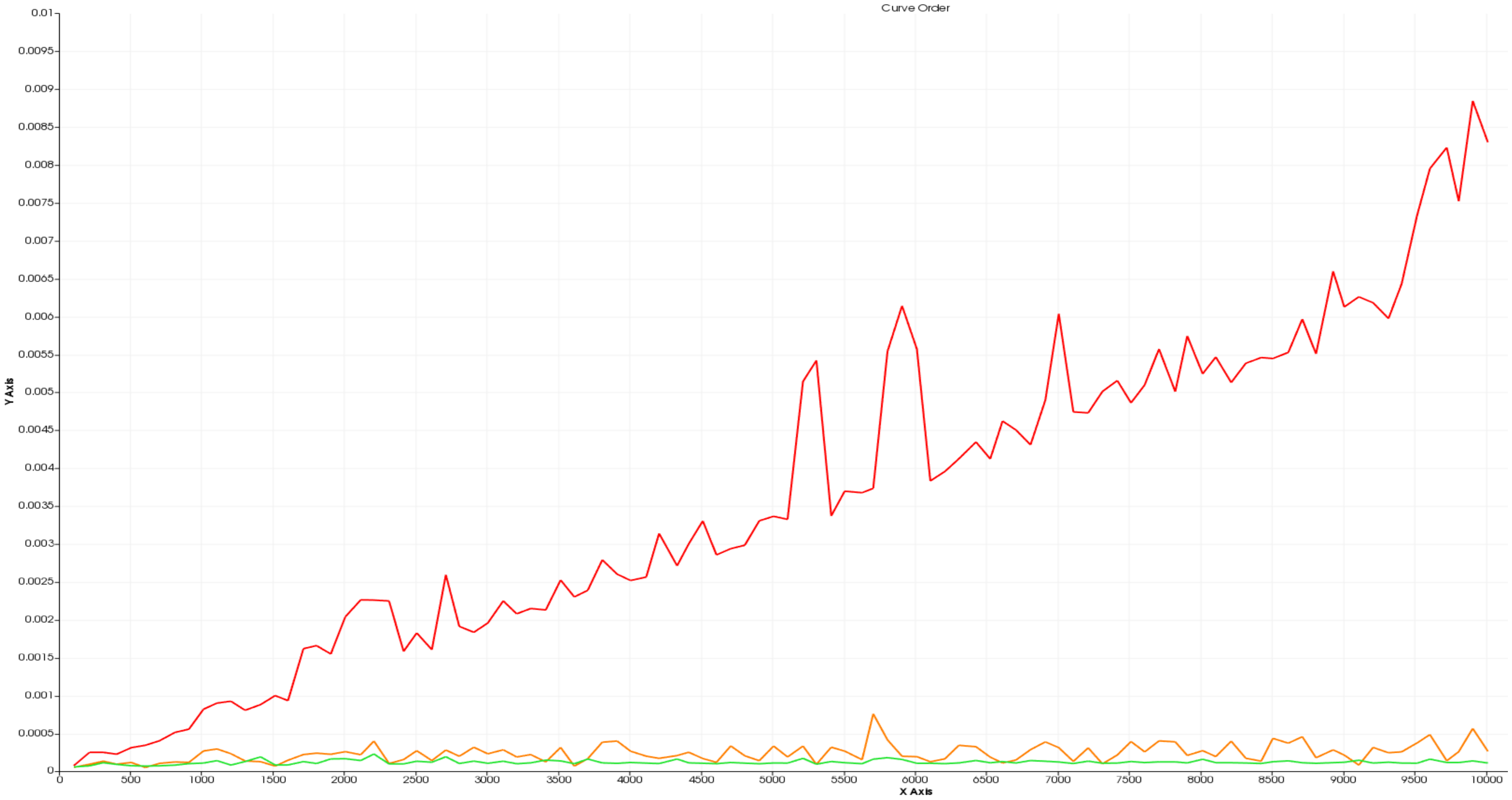


```
[] Curve is defined by
        E(Fq) : y^2 = x^3 + 239991829804608454141070995 * x + 118916092566490565748229184
        where q = 309485009821345068724781063
        order is = 309485009821344799379346835
[Generator] Point coordinates are
        x : 143221764234210030566147892
        y : 100917451060647540116053341
with    q : 309485009821345068724781063


Program ended successfully. Time elapsed : 36.640634611107316232 s.
Enter any key to exit
```

Calcul Naïf

$$|E(\mathbb{F}_q)| = 1 + \sum_{x \in \mathbb{F}_q} \left(1 + \left(\frac{x^3 + ax + b}{q}\right)\right)$$

$$O\left(qlog(q)^2\right)$$



Utilisation de Hasse avec BSGS

$$O(q^{1/4})$$

L'Algorithme de Schoof et les Polynômes de Division

D'après Hasse : $|E(\mathbb{F}_q)| = q + 1 - t \qquad |t| \leq 2\sqrt{q}$

S tel que $\prod_{l \in S} l \geq 4\sqrt{q} \longrightarrow t \bmod l\ ,\ l \in S \longrightarrow$ Restes Chinois $\longrightarrow t$

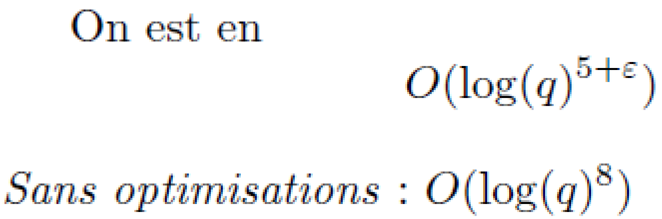
On définit

$$\begin{aligned}
\psi_0 &= 0 \\
\psi_1 &= 1 \\
\psi_2 &= 2y \\
\psi_3 &= 3x^4 + 6Ax^2 + 12Bx - A^2 \\
\psi_4 &= 4y(x^6 + 5Ax^4 + 20Bx^3 - 5A^2x^2 - 4ABx - 8B^2 - A^3) \\
&\cdots \\
\psi_{2m+1} &= \psi_{m+2}\psi_m^3 - \psi_{m-1}\psi_{m+1}^3 \text{ for } m \geq 2 \\
\psi_{2m} &= \left(\frac{\psi_m}{2y}\right) \cdot (\psi_{m+2}\psi_{m-1}^2 - \psi_{m-2}\psi_{m+1}^2) \text{ for } m \geq 2
\end{aligned}$$

Froebenius : $\phi_q : (x, y) \longmapsto (x^q, y^q).$

$$\phi_q^2 - t\phi_q + q = 0$$

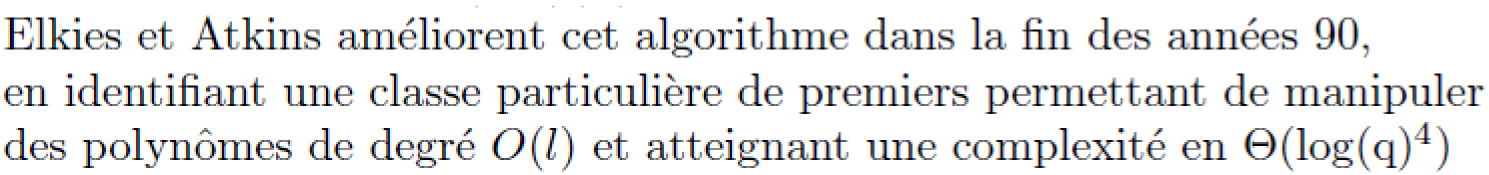
Remarques sur la Complexité



On est en

$$O(\log(q)^{5+\varepsilon})$$

Sans optimisations : $O(\log(q)^8)$



→ Elkies et Atkins améliorent cet algorithme dans la fin des années 90, en identifiant une classe particulière de premiers permettant de manipuler des polynômes de degré $O(l)$ et atteignant une complexité en $\Theta(\log(\mathrm{q})^4)$

Génération des Polynômes de Division

$\longrightarrow \mathbb{F}_q[X,Y]/_{(Y^2-aX^3-X-b)}$

$\psi_{2n} \in y\mathbb{F}_q[X], \psi_{2n+1} \in \mathbb{F}_q[X]$

$$\begin{cases} \psi_{4n} & = & \frac{1}{2}\psi_{2n}(\psi_{2n+2}\psi_{2n-1}^2 - \psi_{2n-2}\psi_{2n+1}^2) \\ \psi_{4n+1} & = & (X^3+aX+b)^2\psi_{2n+2}\psi_{2n}^3 - \psi_{2n+1}\psi_{2n-1} \\ \psi_{4n+2} & = & \frac{1}{2}\psi_{2n+1}(\psi_{2n+3}\psi_{2n}^2 - \psi_{2n-1}\psi_{2n+2}^2) \\ \psi_{4n+3} & = & \psi_{2n+3}\psi_{2n+1}^3 - (X^3+aX+b)^2\psi_{2n+2}\psi_{2n} \end{cases}$$

$$\begin{aligned} \text{Karatsuba : } PQ &= (A+X^{n/2}B)(C+X^{n/2}D) \\ &= AC + ((A-B)(D-C)+AC+BD)X^{n/2} + BDX^n \end{aligned}$$

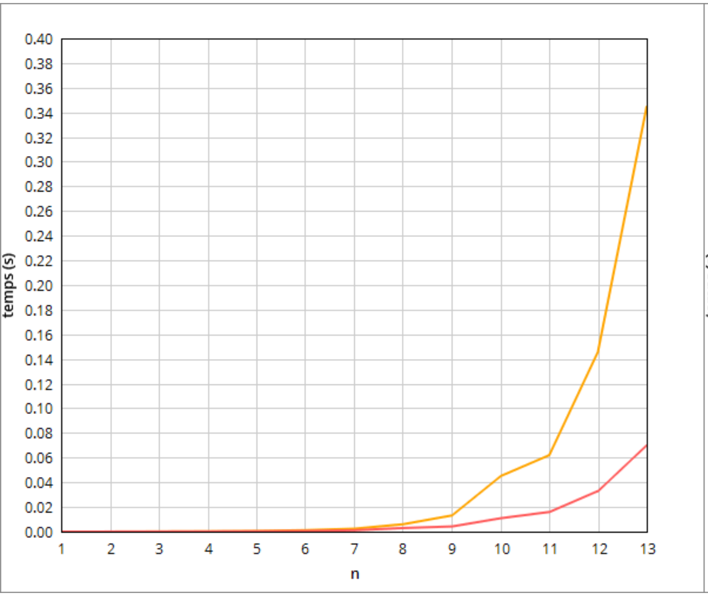
Master Theorem : $O(nlog(n))$

Génération des Polynômes de Division

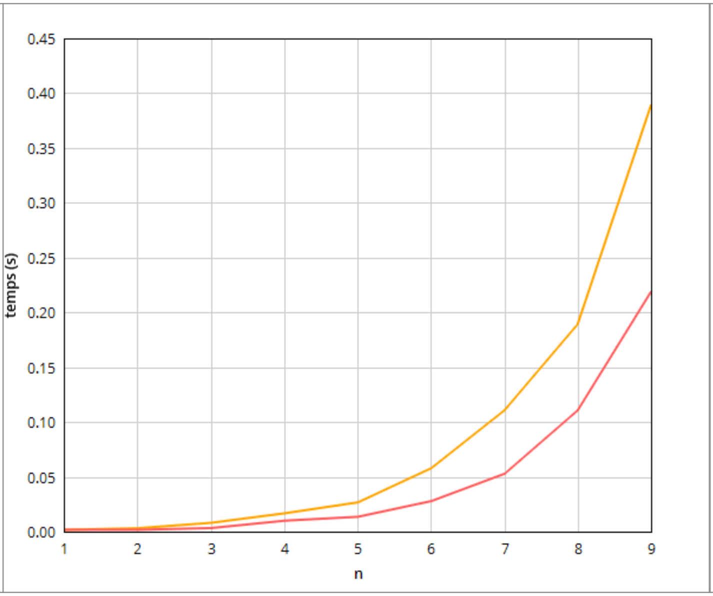
Détermination du seuil en dessous duquel utiliser l'algorithme de multiplication classique

$$Q = (X+1)^{2^d}$$

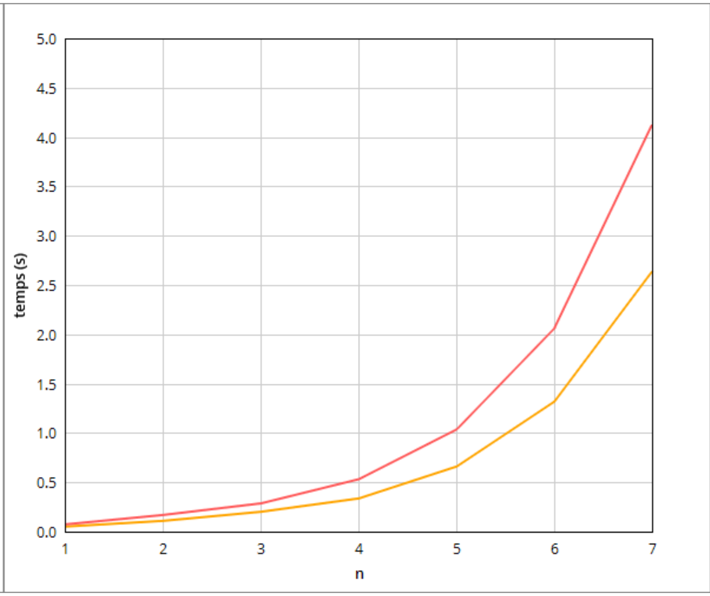
$$\longrightarrow Q * Q^{2^n}$$



$2^d = 8$



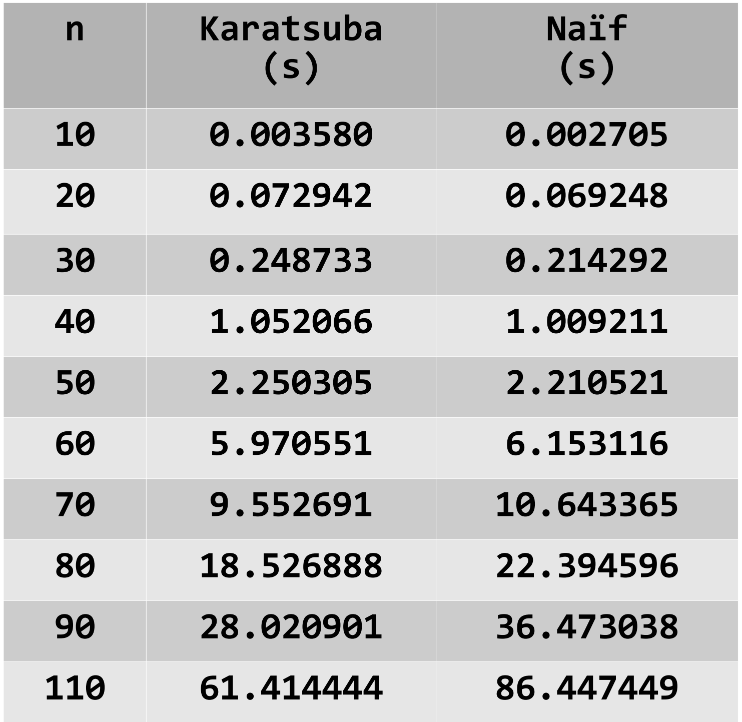
$2^d = 64$



$2^d = 256$

On fixera le seuil à 512 (il faut aussi éviter un stack overflow).

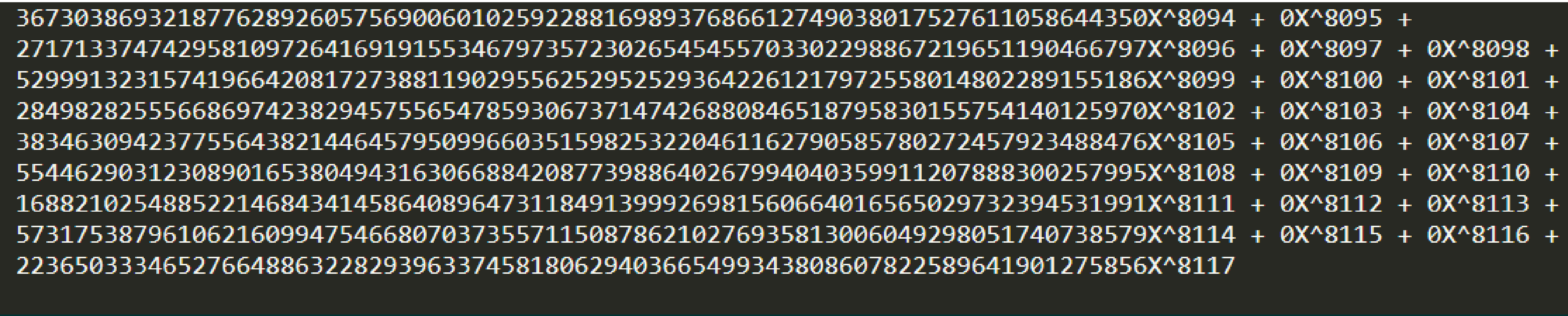
Génération des Polynômes de Division



n	Karatsuba (s)	Naïf (s)
10	0.003580	0.002705
20	0.072942	0.069248
30	0.248733	0.214292
40	1.052066	1.009211
50	2.250305	2.210521
60	5.970551	6.153116
70	9.552691	10.643365
80	18.526888	22.394596
90	28.020901	36.473038
110	61.414444	86.447449

Courbe du bitcoin : $y^2 = x^3 + 7 \bmod \mathrm{fffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f}_{16}$

$\longrightarrow \psi_{116}$



```
3673038693218776289260575690060102592288169893768661274903801752761105864435 0X^8094 + 0X^8095 +
```

Cryptosystème (C++)

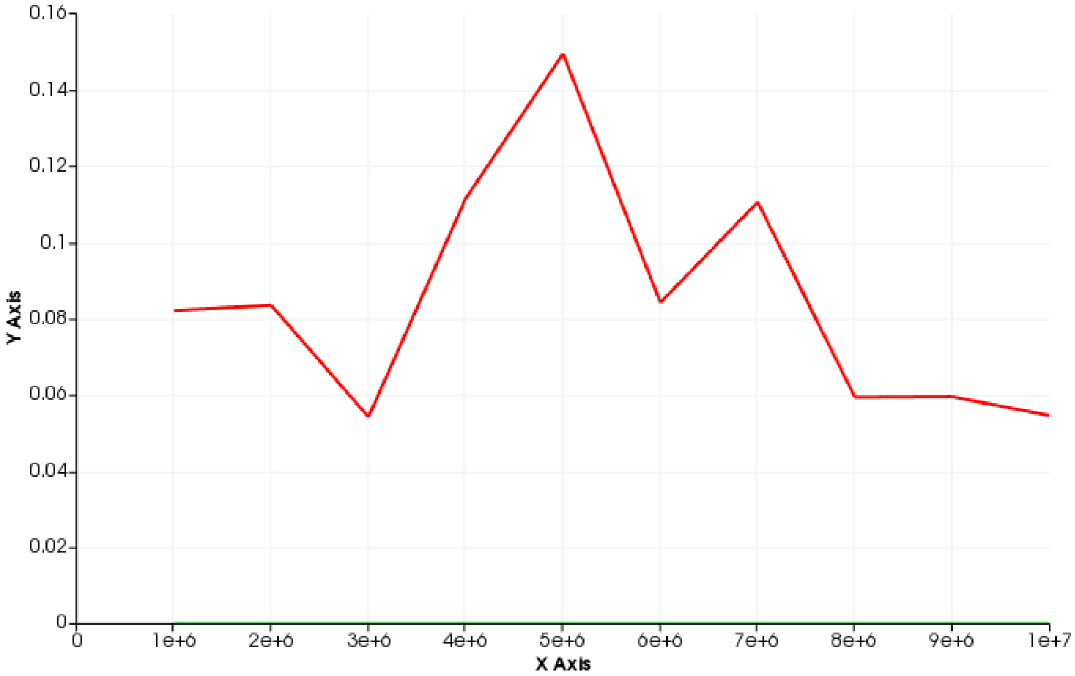
Générer une Courbe et Trouver un Certain Point

```
/* Finds new random parameter A, B and P for the curve. If an argument
P is passed, the most probable prime greater than P will be used and
new A and B will be found accordingly. Gen point is reset to neutral point*/
void setRandomParam(mpir_ui p = 0, bool findOrder = false, bool verbose = false);
void setRandomParam(const mpz_t p = NULL, bool findOrder = false, bool verbose = false);
```

Trouve une courbe valide ($\Delta \neq 0$) et prévient dans le cas où $j = 0$.

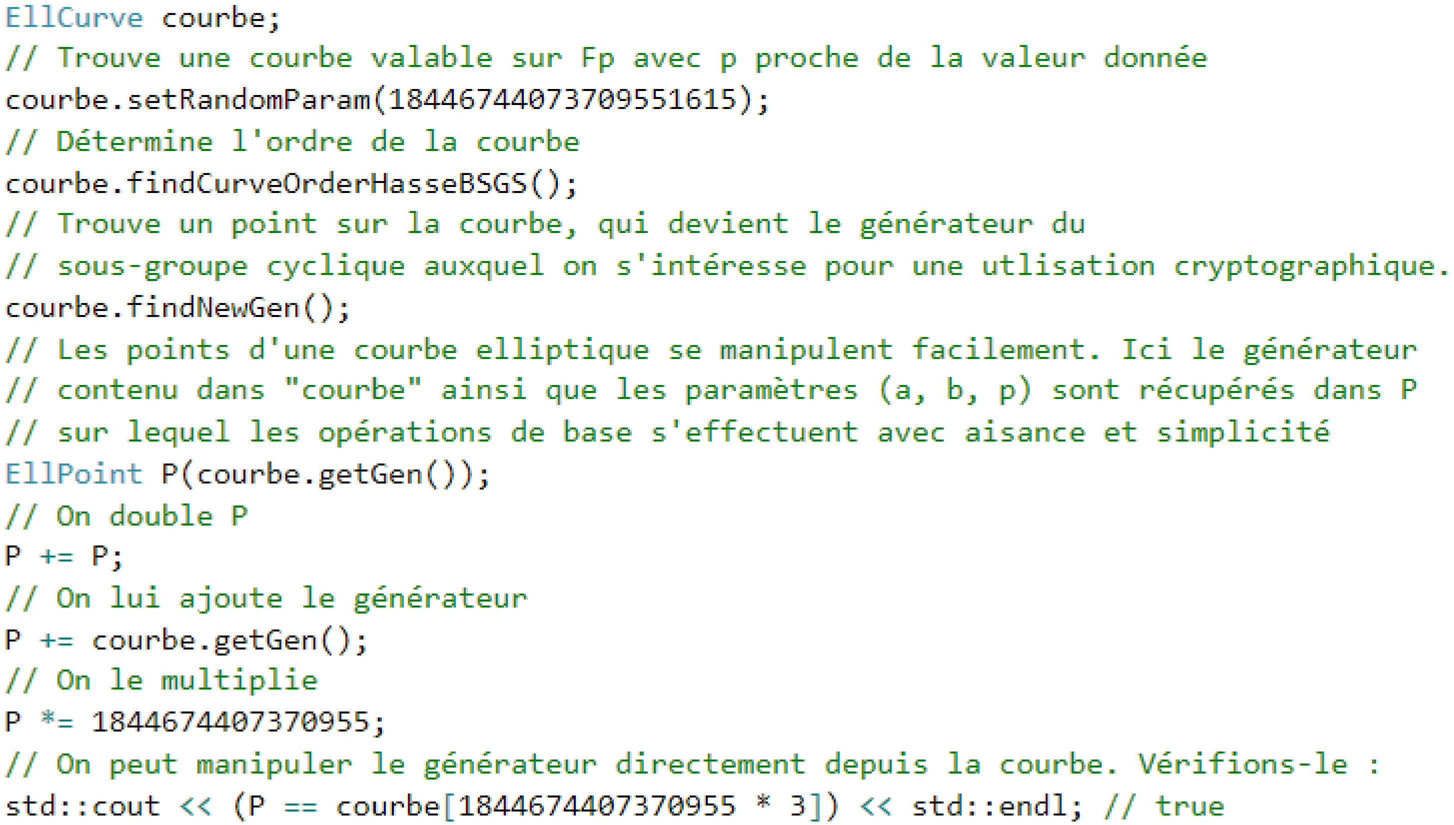
```
/*Sets new coordinates by randomly choosing x until x^3 + ax + b
is a quadratic residue mod p and setting y accordingly*/
void setRandomCoord();
```

L'algorithme de Shanks-Tonelli permet de trouver un point sur la courbe presque instantanément, plutôt que de vérifier si un point choisi au hasard est sur la courbe ou non.



Cryptosystème

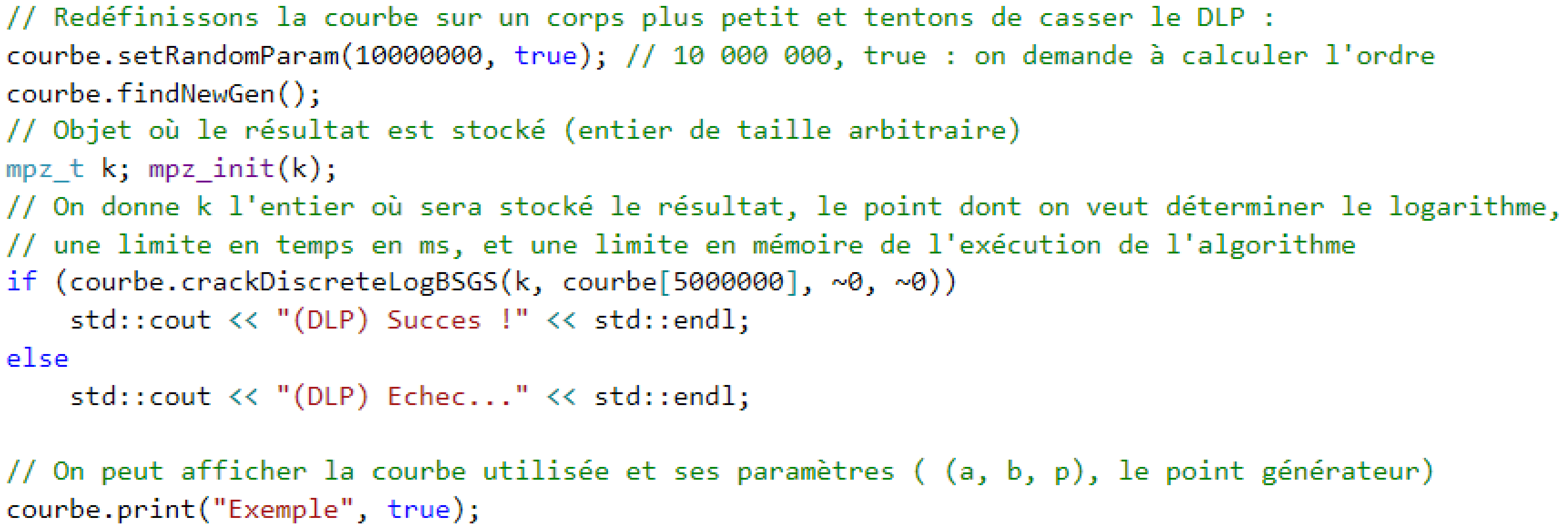
Exemple (et plot)



```
EllCurve courbe;
// Trouve une courbe valable sur Fp avec p proche de la valeur donnée
courbe.setRandomParam(18446744073709551615);
// Détermine l'ordre de la courbe
courbe.findCurveOrderHasseBSGS();
// Trouve un point sur la courbe, qui devient le générateur du
// sous-groupe cyclique auxquel on s'intéresse pour une utlisation cryptographique.
courbe.findNewGen();
// Les points d'une courbe elliptique se manipulent facilement. Ici le générateur
// contenu dans "courbe" ainsi que les paramètres (a, b, p) sont récupérés dans P
// sur lequel les opérations de base s'effectuent avec aisance et simplicité
EllPoint P(courbe.getGen());
// On double P
P += P;
// On lui ajoute le générateur
P += courbe.getGen();
// On le multiplie
P *= 1844674407370955;
// On peut manipuler le générateur directement depuis la courbe. Vérifions-le :
std::cout << (P == courbe[1844674407370955 * 3]) << std::endl; // true
```

Cryptosystème

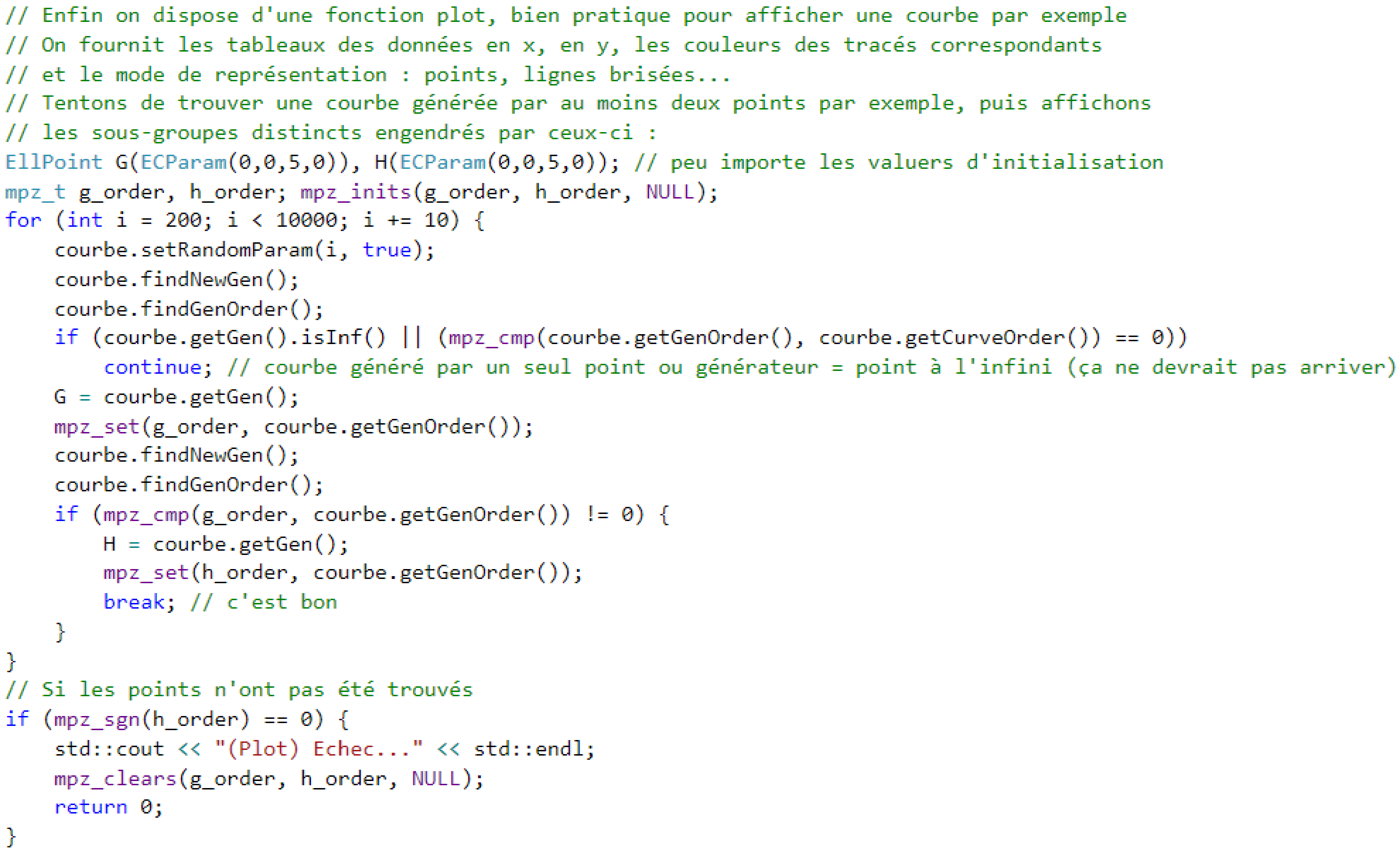
Exemple (et plot)



```
// Redéfinissons la courbe sur un corps plus petit et tentons de casser le DLP :
courbe.setRandomParam(10000000, true); // 10 000 000, true : on demande à calculer l'ordre
courbe.findNewGen();
// Objet où le résultat est stocké (entier de taille arbitraire)
mpz_t k; mpz_init(k);
// On donne k l'entier où sera stocké le résultat, le point dont on veut déterminer le logarithme,
// une limite en temps en ms, et une limite en mémoire de l'exécution de l'algorithme
if (courbe.crackDiscreteLogBSGS(k, courbe[5000000], ~0, ~0))
    std::cout << "(DLP) Succes !" << std::endl;
else
    std::cout << "(DLP) Echec..." << std::endl;

// On peut afficher la courbe utilisée et ses paramètres ( (a, b, p), le point générateur)
courbe.print("Exemple", true);
```

Cryptosystème

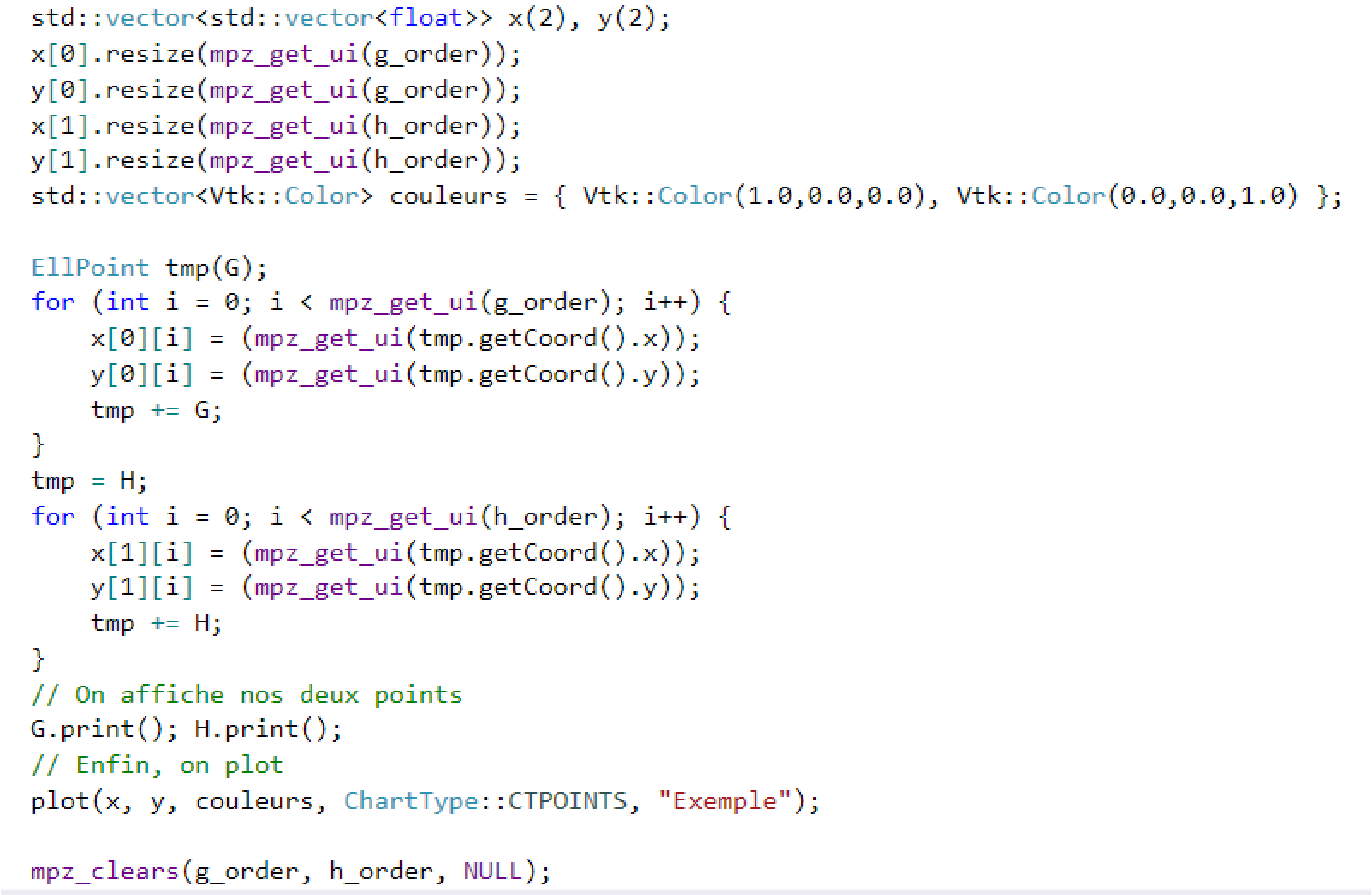
Exemple (et plot)



```
// Enfin on dispose d'une fonction plot, bien pratique pour afficher une courbe par exemple
// On fournit les tableaux des données en x, en y, les couleurs des tracés correspondants
// et le mode de représentation : points, lignes brisées...
// Tentons de trouver une courbe générée par au moins deux points par exemple, puis affichons
// les sous-groupes distincts engendrés par ceux-ci :
EllPoint G(ECParam(0,0,5,0)), H(ECParam(0,0,5,0)); // peu importe les valuers d'initialisation
mpz_t g_order, h_order; mpz_inits(g_order, h_order, NULL);
for (int i = 200; i < 10000; i += 10) {
    courbe.setRandomParam(i, true);
    courbe.findNewGen();
    courbe.findGenOrder();
    if (courbe.getGen().isInf() || (mpz_cmp(courbe.getGenOrder(), courbe.getCurveOrder()) == 0))
        continue; // courbe généré par un seul point ou générateur = point à l'infini (ça ne devrait pas arriver)
    G = courbe.getGen();
    mpz_set(g_order, courbe.getGenOrder());
    courbe.findNewGen();
    courbe.findGenOrder();
    if (mpz_cmp(g_order, courbe.getGenOrder()) != 0) {
        H = courbe.getGen();
        mpz_set(h_order, courbe.getGenOrder());
        break; // c'est bon
    }
}
// Si les points n'ont pas été trouvés
if (mpz_sgn(h_order) == 0) {
    std::cout << "(Plot) Echec..." << std::endl;
    mpz_clears(g_order, h_order, NULL);
    return 0;
}
```

Cryptosystème

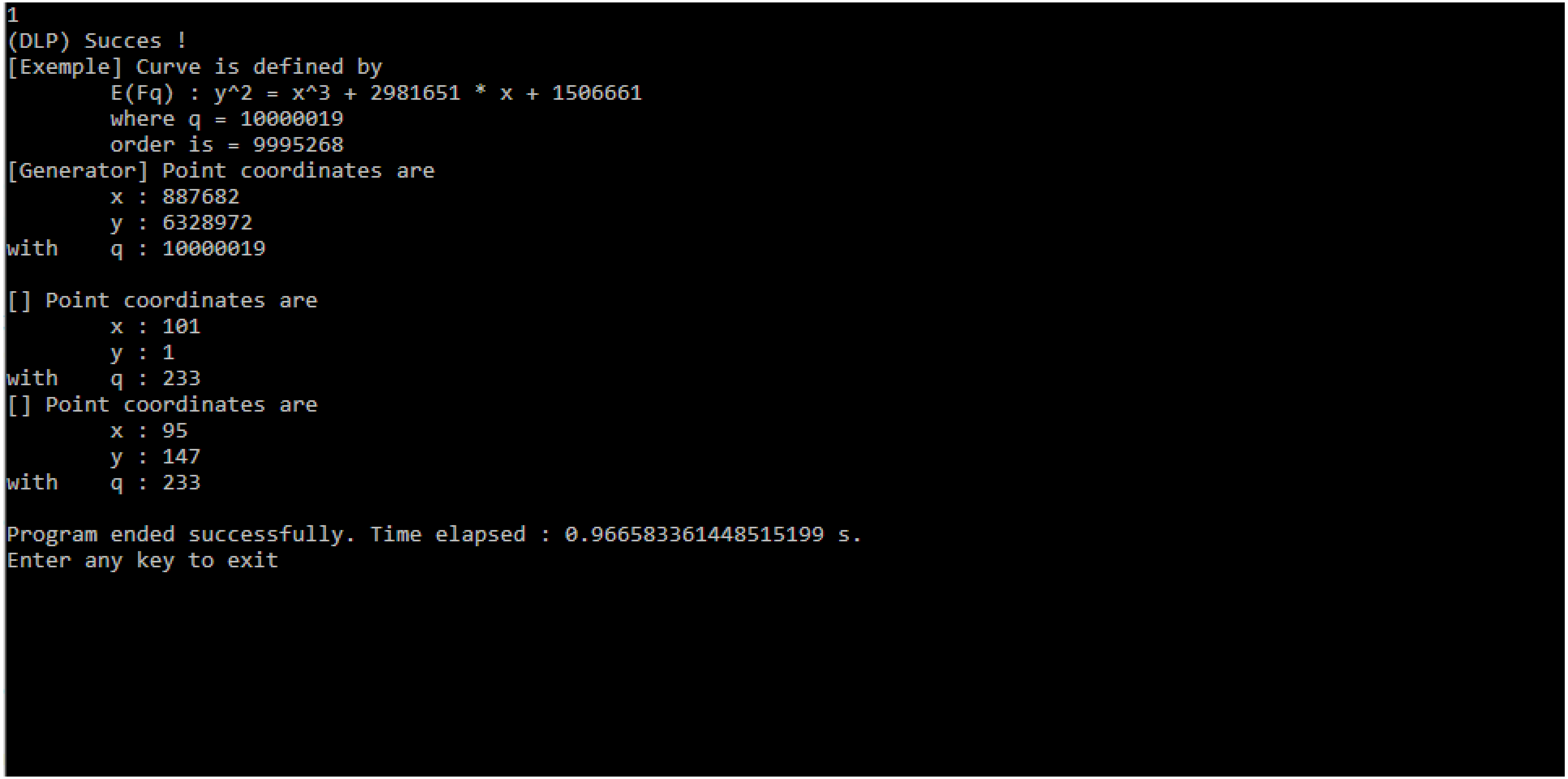
Exemple (et plot)



```cpp
std::vector<std::vector<float>> x(2), y(2);
x[0].resize(mpz_get_ui(g_order));
y[0].resize(mpz_get_ui(g_order));
x[1].resize(mpz_get_ui(h_order));
y[1].resize(mpz_get_ui(h_order));
std::vector<Vtk::Color> couleurs = { Vtk::Color(1.0,0.0,0.0), Vtk::Color(0.0,0.0,1.0) };

EllPoint tmp(G);
for (int i = 0; i < mpz_get_ui(g_order); i++) {
    x[0][i] = (mpz_get_ui(tmp.getCoord().x));
    y[0][i] = (mpz_get_ui(tmp.getCoord().y));
    tmp += G;
}
tmp = H;
for (int i = 0; i < mpz_get_ui(h_order); i++) {
    x[1][i] = (mpz_get_ui(tmp.getCoord().x));
    y[1][i] = (mpz_get_ui(tmp.getCoord().y));
    tmp += H;
}
// On affiche nos deux points
G.print(); H.print();
// Enfin, on plot
plot(x, y, couleurs, ChartType::CTPOINTS, "Exemple");

mpz_clears(g_order, h_order, NULL);
```

Cryptosystème

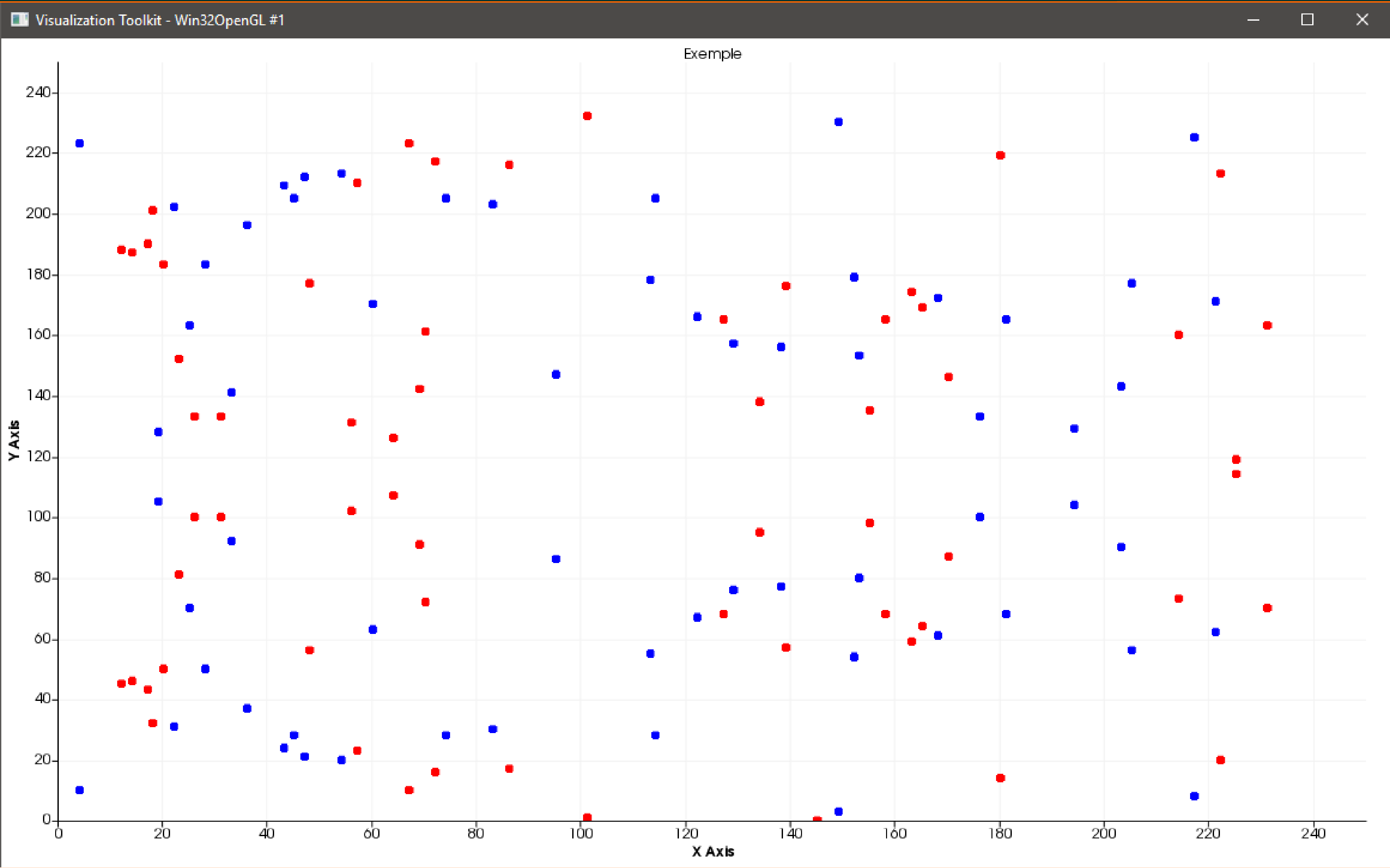
Exemple (et plot)



```
1
(DLP) Succes !
[Exemple] Curve is defined by
        E(Fq) : y^2 = x^3 + 2981651 * x + 1506661
        where q = 10000019
        order is = 9995268
[Generator] Point coordinates are
        x : 887682
        y : 6328972
with    q : 10000019

[] Point coordinates are
        x : 101
        y : 1
with    q : 233
[] Point coordinates are
        x : 95
        y : 147
with    q : 233

Program ended successfully. Time elapsed : 0.966583361448515199 s.
Enter any key to exit
```

Cryptosystème

Exemple (et plot)



Améliorations Envisageables

Actuellement : Vtk et MPIR (fork de GMP) mais :
→ Vtk étrangement gourmand en mémoire à l'affichage
→ FLINT, sous-projet de MPIR, est plus adapté à la théorie des nombres

Améliorer la gestion des polynômes (surtout d'un point de vue mémoire)
Implémenter Schoof et SEA surtout
Implémenter endomorphismes rapides

Annexe

plot.h

```
#pragma once
//#define vtkRenderingCore_AUTOINIT 4(vtkInteractionStyle,vtkRenderingFreeType,vtkRenderingFreeTypeOpenGL,vtkRenderingOpenGL2)
//#define vtkRenderingVolume_AUTOINIT 1(vtkRenderingVolumeOpenGL2)
//#define vtkRenderingCore_AUTOINIT 2(vtkRenderingOpenGL2, vtkInteractionStyle)
#define vtkRenderingCore_AUTOINIT 3(vtkInteractionStyle,vtkRenderingFreeType,vtkRenderingOpenGL2)
#define vtkRenderingVolume_AUTOINIT 1(vtkRenderingVolumeOpenGL2)
#define vtkRenderingContext2D_AUTOINIT 1(vtkRenderingContextOpenGL2)
#include <vtkAutoInit.h>
#include <vtkVersion.h>
#include <vtkRenderer.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkRenderWindow.h>
#include <vtkSmartPointer.h>
#include <vtkChartXY.h>
#include <vtkTable.h>
#include <vtkPlot.h>
#include <vtkFloatArray.h>
#include <vtkContextView.h>
#include <vtkContextScene.h>
#include <vtkPen.h>

static enum ChartType{
CTLINE,
CTPOINTS,
CTBAR,
CTSTACKED,
CTBAG,
CTFUNCTIONALBAG,
CTAREA
};

namespace Vtk {
struct Color {
Color(double _r, double _g, double _b) : r(_r), g(_g), b(_b) {};
double r, g, b;
};
}

void extern testVtk();
void extern plot(const std::vector<std::vector<float>>& x_values, const std::vector<std::vector<float>>& y_values,
std::vector<Vtk::Color> c, ChartType&& chartType = ChartType::CTLINE, std::string name = "");
```

plot.cpp

```
#include "plot.h"
#include <cmath>
#ifndef M_PI_2
#define M_PI_2 1.57079632679489661923
#endif
#ifndef M_PI_4
#define M_PI_4 0.785398163397448309616
#endif

void plot(const std::vector<std::vector<float>>& x_values,
const std::vector<std::vector<float>>& y_values,
std::vector<Vtk::Color> c, ChartType&& chartType, std::string name) {
assert(x_values.size() == y_values.size());

// Set up the view
vtkSmartPointer<vtkContextView> view =
vtkSmartPointer<vtkContextView>::New();
view->GetRenderer()->SetBackground(1.0, 1.0, 1.0);

// Add multiple line plots, setting the colors etc
vtkSmartPointer<vtkChartXY> chart =
vtkSmartPointer<vtkChartXY>::New();
chart->SetTitle(name);
chart->SetSize(vtkRectf(150.0f, 150.0f, 1280.0f, 720.0f));
view->GetScene()->AddItem(chart);

for (int i = 0; i < x_values.size(); i++) {
std::size_t size = x_values[i].size();
// Create a table with some points in it
vtkSmartPointer<vtkTable> table =
vtkSmartPointer<vtkTable>::New();

vtkSmartPointer<vtkFloatArray> arrX =
vtkSmartPointer<vtkFloatArray>::New();
arrX->SetName("X Axis");
table->AddColumn(arrX);

vtkSmartPointer<vtkFloatArray> arrY =
vtkSmartPointer<vtkFloatArray>::New();
arrY->SetName(name == std::string("") ? "Y Axis" : name.c_str());
table->AddColumn(arrY);

// Fill in the table with some example values
table->SetNumberOfRows(size);
for (int j = 0; j < size; j++)
{
table->SetValue(j, 0, x_values[i][j]);
table->SetValue(j, 1, y_values[i][j]);
}
vtkPlot *line;
switch (chartType) {
case ChartType::CTPOINTS:
line = chart->AddPlot(vtkChart::POINTS);
break;
case ChartType::CTLINE:
line = chart->AddPlot(vtkChart::LINE);
break;
case ChartType::CTBAR:
line = chart->AddPlot(vtkChart::BAR);
break;
case ChartType::CTSTACKED:
line = chart->AddPlot(vtkChart::STACKED);
break;
case ChartType::CTBAG:
line = chart->AddPlot(vtkChart::BAG);
break;
case ChartType::CTFUNCTIONALBAG:
line = chart->AddPlot(vtkChart::FUNCTIONALBAG);
break;
case ChartType::CTAREA:
line = chart->AddPlot(vtkChart::AREA);
break;
}
line->SetInputData(table, 0, 1);

line->SetColor(c[i].r, c[i].g, c[i].b);
//line->SetWidth(0.5);
line->GetPen()->SetLineType(vtkPen::SOLID_LINE);
}
// Start interactor
view->GetInteractor()->Initialize();
view->GetInteractor()->Start();
}
```

utils.h

```
#pragma once
#include <stdarg.h>
#include <stdio.h>
#include <time.h>
#include <mpir.h>
#include <vector>
#include <algorithm>
#include <mutex>

static class RandomClass{
public :
RandomClass() { gmp_randinit_default(m_rand_state);}
~RandomClass() { gmp_randclear(m_rand_state); }

bool is_likely_prime(const mpz_t& prime)                  { return mpz_likely_prime_p(prime, m_rand_state, 0); }
void next_prime_candidate(mpz_t& dest, mpz_t src)         { mpz_next_prime_candidate(dest, src, m_rand_state); }
void randomb(mpz_t& dest, mpir_ui bitcnt)                 { mpz_urandomb(dest, m_rand_state, bitcnt); }
void randomm(mpz_t& dest, const mpz_t& mod)               { mpz_urandomm(dest, m_rand_state, mod); }
private :
gmp_randstate_t m_rand_state;
} Rand;


void get_primes(std::vector<mpir_ui>& dest, unsigned int n);
bool is_product_geq(const std::vector<mpir_ui>& src, mpz_t N);

struct PrimeFactComp {
mpz_t p;
mpir_ui exp;
};

typedef std::vector<PrimeFactComp> prime_factorization;

/* Sets dest to the number described by primeFact under its prime factorization*/
void mpz_set_from_prime_fact(mpz_t dest, prime_factorization primeFact);

/* Solve the modular equation x^2 = n (mod p) using the Shanks-Tonelli
* algorihm. x will be placed in q and 1 returned if the algorithm is
* successful.
*/
int mpz_sqrtm(mpz_t q, const mpz_t n, const mpz_t p);

/* Sets dest to rhs of the weierstrass equation for an ecc : x^3 + ax +b */
void set_weierstrass(mpz_t &dest, const mpz_t &x, const mpz_t &a, const mpz_t &b, const mpz_t &p);

/* Sets dest to the discriminant of the curve y^2 = x^3 + ax + b, and j to its j_invariant :
discr = 4a^3 + 27b^2 and j = -1728 * 64 * A^3 / discr*/
void set_weierstrass_discriminant_j_invariant(mpz_t& dest, mpz_t& j,const mpz_t &a, const mpz_t &b, const mpz_t& p);

/* Returns a^(-1) % b, assuming gcd(a, b) = 1 */
mpir_ui mod_inv(mpir_ui a, mpir_ui b);

/* Returns x such that x equiv a[i] mod n[i]*/
void chinese_remainder(mpz_t& dest, const std::vector<mpir_ui>& a, const std::vector<mpir_ui>& n);
```

utils.cpp

```cpp
#include "utils.h"

void get_primes(std::vector<mpir_ui>& dest, unsigned int n) {
std::vector<bool> vec_primes(n, true);
vec_primes[0] = false;
vec_primes[1] = false;

for (auto i = 4; i < n; i += 2) { // even numbers are handled first for efficiency
vec_primes[i] = false;
}

for (auto i = 3; i < n; i += 2) {
if (vec_primes[i]) {
//the usual is for(int j = i * 2; j < MAXIMUM; j += i) {
//but I am using a bit more optimized loop
for (auto j = i * i; j < n; j += 2 * i) {
vec_primes[j] = false;
}
}
}
dest.clear();
dest.reserve(n / log(n));
for (auto i = 2; i < n; i++) {
if (vec_primes[i])
dest.push_back(i);
}
}

/*Checks if >=*/
bool is_product_geq(const std::vector<mpir_ui>& src, mpz_t N) {
mpz_t tmp; mpz_init_set_ui(tmp, 1);
for (auto& x : src) {
mpz_mul_ui(tmp, tmp, x);
if (mpz_cmp(tmp, N) >= 0) {
mpz_clear(tmp);
return true;
}
}
mpz_clear(tmp);
return false;
}

void mpz_set_from_prime_fact(mpz_t dest, prime_factorization primeFact) {
mpz_set_ui(dest, 0);
mpz_t op;
mpz_init(op);
for (auto& x : primeFact) {
mpz_pow_ui(op, x.p, x.exp);
mpz_mul(dest, dest, op);
}
mpz_clear(op);
}
```

```
/* Sets x to rhs of the weierstrass equation for an ecc : x^3 + ax +b */
void set_weierstrass(mpz_t &dest, const mpz_t &x,
const mpz_t &a, const mpz_t &b, const mpz_t &p) {
mpz_t op1, op2;
mpz_inits(op1, op2, NULL);
mpz_mul(op1, x, a);
mpz_powm_ui(op2, x, 2, p);
mpz_addmul(op1, op2, x);
mpz_add(op1, op1, b);
mpz_mod(op1, op1, p); // op1 = x^3 + ax + b
mpz_set(dest, op1);
mpz_clears(op1, op2, NULL);
}


int mpz_sqrtm(mpz_t q, const mpz_t n, const mpz_t p) {
mpz_t w, n_inv, y;
mpir_ui i, s;
//TMP_DECL;
//TMP_MARK;

if (mpz_divisible_p(n, p)) {
mpz_set_ui(q, 0);
return 1;
}
if (mpz_legendre(n, p) != 1)
return 0;
if (mpz_tstbit(p, 1) == 1) {    /* p = 3 (mod 4) ?                 */
mpz_set(q, p);
mpz_add_ui(q, q, 1);
mpz_fdiv_q_2exp(q, q, 2);
mpz_powm(q, n, q, p);           /* q = n ^ ((p+1) / 4) (mod p)      */
return 1;
}
```

```
/*
MPZ_TMP_INIT(y, 2 * SIZ(p));
MPZ_TMP_INIT(w, 2 * SIZ(p));
MPZ_TMP_INIT(n_inv, 2 * SIZ(p));
*/
mpz_init2(y, mpz_sizeinbase(p, 2));
mpz_init2(w, mpz_sizeinbase(p, 2));
mpz_init2(n_inv, mpz_sizeinbase(p, 2));

mpz_set(q, p);
mpz_sub_ui(q, q, 1);               /* q = p-1            */
s = 0;
while (mpz_tstbit(q, s) == 0) s++;
mpz_fdiv_q_2exp(q, q, s);          /* q = q / 2^s     */
mpz_set_ui(w, 2);
while (mpz_legendre(w, p) != -1)
mpz_add_ui(w, w, 1);
mpz_powm(w, w, q, p);
mpz_add_ui(q, q, 1);
mpz_fdiv_q_2exp(q, q, 1);
mpz_powm(q, n, q, p);
mpz_invert(n_inv, n, p);
for (;;) {
mpz_powm_ui(y, q, 2, p);
mpz_mul(y, y, n_inv);
mpz_mod(y, y, p);
i = 0;
while (mpz_cmp_ui(y, 1) != 0) {
i++;
mpz_powm_ui(y, y, 2, p);
}
if (i == 0) {          /* q^2 * n^-1 = 1 (mod p), return   */
mpz_clear(w);
mpz_clear(n_inv);
mpz_clear(y);
return 1;
}
if (s - i == 1) {           /* In case the exponent to w is 1,  */
mpz_mul(q, q, w);           /* Don't bother exponentiating      */
}
else {
mpz_powm_ui(y, w, (mpir_ui)1 << (s - i - 1), p);
mpz_mul(q, q, y);
}
mpz_mod(q, q, p);          /* r = r * w^(2^(s-i-1)) (mod p)    */
}

mpz_clear(w);
mpz_clear(n_inv);
mpz_clear(y);
}
```

```
/* Sets dest to the discriminant of the curve
y^2 = x^3 + ax + b, and j to its j_invariant :
discr = -16*(4a^3 + 27b^2) and j = 1728 * 4 * A^3 / discr*/
void set_weierstrass_discriminant_j_invariant
(mpz_t& discr, mpz_t& j, const mpz_t &a,
 const mpz_t &b, const mpz_t& p) {
mpz_t op1, op2;
mpz_init(op2);
mpz_init_set(op1, b);
mpz_mul_ui(op1, b, 27);
mpz_mul(op1, op1, b); // op1 = 27b^2
mpz_powm_ui(op2, a, 3, p);
mpz_mul_ui(op2, op2, 4); // op2 = 4 * A^3
mpz_add(discr, op1, op2);
mpz_mul_ui(discr, discr, 16);
mpz_neg(discr, discr);
mpz_mod(discr, discr, p); // discr ok
if (mpz_sgn(discr) == 0) return; // discr is null
mpz_mul_ui(op2, op2, 1728); // op2 = 1728 * 4 * A^3
mpz_invert(op1, discr, p);
mpz_mul(j, op1, op2);
mpz_mod(j, j, p);// j ok
mpz_clears(op1, op2, NULL);
}

/* Returns a^(-1) % b, assuming gcd(a, b) = 1 */
mpir_ui mod_inv(mpir_ui a, mpir_ui b) {
mpir_ui b0 = b, t, q;
mpir_ui x0 = 0, x1 = 1;
if (b == 1) return 1;
while (a > 1) {
q = a / b;
t = b, b = a % b, a = t;
t = x0, x0 = x1 - q * x0, x1 = t;
}
if (x1 < 0) x1 += b0;
return x1;
}
```

```
/* Returns x such that x equiv a[i] mod n[i]*/
void chinese_remainder(mpz_t &dest,
const std::vector<mpir_ui>& a,
const std::vector<mpir_ui>& n) {
if (a.size() != n.size()) {
printf("Dimension error in chinese remainder calculation");
return;
}
mpz_t prod; mpz_init_set_ui(prod, 1);
mpz_t sum; mpz_init_set_ui(sum, 0);
mpz_t p; mpz_init(p);
mpz_t tmp_mod; mpz_init(tmp_mod);

for (size_t i = 0; i < n.size(); i++)
mpz_mul_ui(prod, prod, n[i]);

for (size_t i = 0; i < n.size(); i++) {
mpz_divexact_ui(p, prod, n[i]);
// MAYBE PROBLEM HERE WITH mpz_mod_ui :
mpz_mod_ui(tmp_mod, p, n[i]);
mpz_addmul_ui(sum, p, a[i] * mod_inv(mpz_get_ui(tmp_mod), n[i]));
}

mpz_mod(dest, sum, prod);
mpz_clears(prod, sum, p, tmp_mod, NULL);
}
```

ectypes.h

```
#pragma once
#include <stdarg.h>
#include <stdio.h>
#include <type_traits>
#include <time.h>
#include <utility>
#include <mpir.h>


#define DEFAULT_A 0
#define DEFAULT_B 0
#define DEFAULT_P 5
#define DEFAULT_ORDER 0

#define PT_INFTY 1
#define NOT_INFTY 0

void extern mpz_print(const mpz_t print);

/* Parameters of an elliptic curve in its reduced Weierstrass form
y^2 = x^3 + ax + b*/
struct ECParam {
ECParam(mpir_ui _a = DEFAULT_A, mpir_ui _b = DEFAULT_B,
       mpir_ui _p = DEFAULT_P, mpir_ui _order = DEFAULT_ORDER);
ECParam(const char* _a, const char* _b,
       const char* _p, const char* _order);
ECParam(mpz_t _a, mpz_t _b, mpz_t _p, mpz_t _order = NULL);

~ECParam();
ECParam(const ECParam& p);
void operator=(const ECParam& p);
bool operator==(const ECParam& p) const;
mpz_t a;
mpz_t b;
mpz_t p; // Z / pZ
mpz_t order;
};

/* Coordinate of a point on an ec*/
struct ECCoord {
ECCoord(int isInf = PT_INFTY, const char* x = "0", const char* y = "0");
ECCoord(int _isInf, mpz_t _x, mpz_t _y);
~ECCoord();
ECCoord(const ECCoord& p);
void operator=(const ECCoord& p);
bool operator==(const ECCoord& p) const;
int isInf;
mpz_t x;
mpz_t y;
};
```

```
/* Helper struct for hash function */
struct ECPair {
const ECCoord coord;
const mpir_ui val;
ECPair(const ECCoord& c, mpir_ui v) : coord(c), val(v) {}
bool operator==(const ECPair& ecpair) const {
return coord == ecpair.coord;
}
};

/* (ECCoord, mpz_t) Hash Function */
struct BSGSHasher {
mpir_ui operator()(const ECPair& elem) const
{
using std::hash;

return ((hash<mpir_ui>()(mpz_get_ui(elem.coord.x))
^ (hash<mpir_ui>()
(mpz_get_ui(elem.coord.y)) << 1)) >> 1);
}
};
```

ectypes.cpp

```cpp
#include "ectypes.h"

void mpz_print(const mpz_t print) {
mpz_out_str(stdout, 10, print);
printf("\n");
}

ECParam::ECParam(mpir_ui _a, mpir_ui _b ,
                 mpir_ui _p, mpir_ui _order) {
mpz_init_set_ui(a, _a);
mpz_init_set_ui(b, _b);
mpz_init_set_ui(p, _p);
mpz_init_set_ui(order, _order);
}

ECParam::ECParam(const char* _a, const char* _b,
                 const char* _p, const char* _order) {
mpz_init_set_str(a, _a, 0);
mpz_init_set_str(b, _b, 0);
mpz_init_set_str(p, _p, 0);
mpz_init_set_str(order, _order, 0);
}

ECParam::ECParam(mpz_t _a, mpz_t _b,
                 mpz_t _p, mpz_t _order) {
mpz_init_set(a, _a);
mpz_init_set(b, _b);
mpz_init_set(p, _p);
mpz_init_set(order, _order);
}

ECParam::~ECParam() {
mpz_clears(a, b, p, order, NULL);
}

ECParam::ECParam(const ECParam& param) {
mpz_init_set(a, param.a);
mpz_init_set(b, param.b);
mpz_init_set(p, param.p);
mpz_init_set(order, param.order);
}

void ECParam::operator=(const ECParam& param) {
mpz_set(a, param.a);
mpz_set(b, param.b);
mpz_set(p, param.p);
mpz_set(order, param.order);
}

bool ECParam::operator==(const ECParam& param) const {
return (mpz_cmp(a, param.a) == 0)
&&     (mpz_cmp(b, param.b) == 0)
&&     (mpz_cmp(p, param.p) == 0)
&&     (mpz_cmp(order, param.order) == 0);
}
```

```cpp
ECCoord::ECCoord(int isInf, const char* x, const char* y) : isInf(isInf) {
mpz_init_set_str(this->x, x, 0);
mpz_init_set_str(this->y, y, 0);

}

ECCoord::ECCoord(int _isInf, mpz_t _x, mpz_t _y) {
isInf = _isInf;
mpz_init_set(x, _x);
mpz_init_set(y, _y);
}

ECCoord::~ECCoord() {
mpz_clear(x);
mpz_clear(y);
}

ECCoord::ECCoord(const ECCoord& p) : isInf(p.isInf) {
mpz_init_set(x, p.x);
mpz_init_set(y, p.y);
}

void ECCoord::operator=(const ECCoord& p) {
isInf = p.isInf;
mpz_set(x, p.x);
mpz_set(y, p.y);
}

bool ECCoord::operator==(const ECCoord& p) const {
// We have to be carefull with isInf because
// it is an integer and not a boolean
if (isInf && p.isInf) return true;
if (isInf ^ p.isInf) return false;
return (mpz_cmp(x, p.x) == 0) && (mpz_cmp(y, p.y) == 0);
}
```

EllPoint.h

```cpp
#pragma once
#include <stdarg.h>
#include <stdio.h>
#include <time.h>
#include <mpir.h>
#include "utils.h"
#include "ectypes.h"


#define MPZ_PRINT(x) mpz_out_str(stdout, 10, x); printf("\n");

/* Elliptic Curve Point*/
class EllPoint {
public:
/*Constructs an EllPoint*/
EllPoint(const ECParam& param);
EllPoint(const EllPoint& p);
~EllPoint();

void operator=(const EllPoint& p);
bool operator==(const EllPoint& p) const;
bool operator+=(const EllPoint& p);
void operator*=(mpir_ui n);
void operator*=(const mpz_t& n);
void inverse();

const ECCoord& getCoord() const;
const ECParam& getECParam() const;
```

```cpp
/*Returns true and sets given coordinates if they correspond to
a point on the curve. Returns false otherwise*/
bool setCoord(const ECCoord& coord);

/*Sets new coordinates by randomly choosing x until x^3 + ax + b
is a quadratic residue mod p and setting y accordingly (Shanks-Tonelli)*/
void setRandomCoord();

/*Returns true if the point is at infinite (neutral element)*/
bool isInf() const;

/*Bruteforces by calculating nP for all n in Fp.
Can be expected to be very slow*/
void order(mpz_t& dest) const;

/*If curve order prime factorization is known,
we can compute point's order much faster
thanks to Lagrange's theorem */
void order_co(mpz_t& dest,
              prime_factorization curveOrderFactorization) const;

/*Prints the point coordinates plus its name *name* if given*/
void print(const char* name = "") const;

/*Internal function to determine whether a point is on the curve or not*/
bool isOnCurve(const ECCoord& coord) const;

/*Sets point to neutral element (point at infinity)*/
void setInf();

private:
ECParam                  m_param;
mpz_t              m_lc; // leading coefficient, used when adding etc.
ECCoord                  m_coord;
};
```

EllPoint.cpp

```
#include "EllPoint.h"

EllPoint::EllPoint(const ECParam& param)
:       m_param(param)
{
// Exepcting same size as p
mpz_init2(m_lc, mpz_sizeinbase(m_param.p, 2));
}

EllPoint::EllPoint(const EllPoint& p)
:       m_param(p.m_param)
,       m_coord(p.m_coord)
{
// Exepcting same size as p
mpz_init2(m_lc, mpz_sizeinbase(m_param.p, 2));
}

EllPoint::~EllPoint() { mpz_clear(m_lc);}

const ECCoord& EllPoint::getCoord() const {return m_coord;}

const ECParam& EllPoint::getECParam() const {return m_param;}

bool EllPoint::setCoord(const ECCoord& coord) {
if (isOnCurve(coord)) {
this->m_coord = coord;
return true;
}
return false;
}

bool EllPoint::isInf() const {return m_coord.isInf;}

void EllPoint::setInf() { m_coord.isInf = PT_INFTY;}

void EllPoint::inverse() {mpz_neg(m_coord.y, m_coord.y);}

void EllPoint::operator=(const EllPoint& p) {
m_param = p.m_param;
m_coord = p.m_coord;
}

bool EllPoint::operator==(const EllPoint& p) const {
return (m_coord == p.m_coord);
}
```

```cpp
bool EllPoint::operator+=(const EllPoint& p) {
// Adding 0
if (p.m_coord.isInf)
return true;
if (m_coord.isInf) {
*this = p;
return true;
}
//Else
//mpz_cmp returns sign of op1 - op2
if (mpz_cmp(m_coord.x, p.m_coord.x)) {
mpz_t op1, op2; // temp variables
mpz_sub(m_lc, p.m_coord.y, m_coord.y);       // m = y2 - y1
mpz_init_set(op1, m_lc); // op1 = m
mpz_sub(m_lc, p.m_coord.x, m_coord.x);       // m = x2 - x1
if(mpz_invert(m_lc, m_lc, m_param.p) == 0)   // m = 1 / m [p]
return false; // inversion failed
mpz_mul(m_lc, m_lc, op1);                    // m = m * op1
// m_lc is now set to (y2 - y1) / (x2 - x1) ;
mpz_powm_ui(op1, m_lc, 2, m_param.p);        // op1 = m^2 [p]
mpz_sub(op1, op1, p.m_coord.x);              // op1 = op1 - x2
mpz_sub(op1, op1, m_coord.x);                // op1 = op1 - x1
// x3 is now set in op1 = m^2 - x1 - x2;
mpz_init_set(op2, m_coord.x);                // op2 = x1
mpz_sub(op2, op2, op1);                // op2 = op2 - op1 = x1 - x3
mpz_mul(op2, op2, m_lc);                     // op2 = op2 * m
mpz_sub(m_coord.y, op2, m_coord.y);          // y3 = op2 - y1

mpz_mod(m_coord.x, op1, m_param.p);          // x3 = op1 % p
mpz_mod(m_coord.y, m_coord.y, m_param.p);    // y3 = y3 % p
//Done
mpz_clear(op1);
mpz_clear(op2);
}
```

```cpp
else {
if (mpz_cmp(m_coord.y, p.m_coord.y)) {
m_coord.isInf = PT_INFTY;
}
else {
if (mpz_sgn(m_coord.y)) { // != 0
mpz_t op1, op2; // temp variables
mpz_init_set(op1, m_coord.x);                // op1 = x1
mpz_init_set(op2, m_coord.x);                // op2 = x1
mpz_powm_ui(op1, op1, 2, m_param.p);   // op1 = op1^2
mpz_mul_ui(op1, op1, 3);                     // op1 *= 3
mpz_add(op1, op1, m_param.a);                // op1 += a
mpz_mod(op1, op1, m_param.p);                // op1 = op1%p
// op1 = 3 * x1 * x1 + a

mpz_add(op2, m_coord.y, m_coord.y);          // op2 = 2* y1
if(mpz_invert(op2, op2, m_param.p) == 0)// op2 = 1/op2 % p
return false; // inversion failed
mpz_mul(m_lc, op1, op2);                     // m = op1 * op2
mpz_mod(m_lc, m_lc, m_param.p);              // m = m % p
//m_lc now set
mpz_powm_ui(op1, m_lc, 2, m_param.p);  // op1 = m^2 % p
mpz_sub(op1, op1, m_coord.x);
mpz_sub(op1, op1, m_coord.x);                // op1 = op1 - 2*x1
// x3 now in op1
mpz_sub(op2, m_coord.x, op1);                // op2 = x1 - x3
mpz_mul(op2, op2, m_lc);                     // op2 = m * op2
mpz_sub(op2, op2, m_coord.y);                // y3 = op2 - y1

mpz_mod(m_coord.x, op1, m_param.p);          // x3 = x3 % p
mpz_mod(m_coord.y, op2, m_param.p);          // y3 = y3 % p
//Done

mpz_clears(op1, op2, NULL);
}
else {
m_coord.isInf = PT_INFTY;
}
}
}
return true;
}
```

```
void EllPoint::operator*=(mpir_ui n) {
if (this->m_coord.isInf) return;
if (n == 0) {
this->m_coord.isInf = PT_INFTY;
return;
}
EllPoint tmp(*this);
// set *this to neutral element
this->setCoord(ECCoord());
while (n != 0) {
if (n & 1) *this += tmp;
tmp += tmp;
n >>= 1;
}
}

void EllPoint::operator*=(const mpz_t& n) {
if (this->m_coord.isInf) return;
if (mpz_sgn(n) == 0) {
this->m_coord.isInf = PT_INFTY;
return;
}
EllPoint tmp(*this);
mpz_t t;
mpz_init_set(t, n);
this->setCoord(ECCoord());
while (mpz_sgn(t) != 0) { // While t != 0
if(mpz_odd_p(t)) *this += tmp;
tmp += tmp;
mpz_tdiv_q_2exp(t, t, 1);
}
mpz_clear(t);
}

void EllPoint::setRandomCoord() {
mp_bitcnt_t n = mpz_sizeinbase(m_param.p, 2);
mpz_t op1;
mpz_init(op1);
while (true) {
Rand.randomb(m_coord.x, n);
mpz_mod(m_coord.x, m_coord.x, m_param.p); // Even if m_coord.x has same bitcount than m_param.p, it could come out larger
set_weierstrass(op1, m_coord.x, m_param.a, m_param.b, m_param.p); // op1 = x^3 + ax + b
if (mpz_sqrtm(m_coord.y, op1, m_param.p))
break;
}
m_coord.isInf = NOT_INFTY;
mpz_clear(op1);
}
```

```
void EllPoint::order(mpz_t& dest) const {
// Naively brute-forcing is so unefficient
//that we won't need smthng larger than mpir_ui
mpir_ui n = 0;
EllPoint Q(*this);
while (!Q.isInf()) {
Q += *this; n++;
}
mpz_set_ui(dest, n);
}

void EllPoint::order_co(mpz_t& dest,
      prime_factorization curveOrderFactorization) const{
mpz_t m, op1;
mpz_set_from_prime_fact(m, curveOrderFactorization);
mpz_init(op1);
for (auto& comp : curveOrderFactorization) {
mpz_pow_ui(op1, comp.p, comp.exp);
mpz_divexact(m, m, op1);
EllPoint Q(*this);
Q *= m;
while (!Q.isInf()) {
Q *= comp.p;
mpz_mul(m, m, comp.p);
}
}
mpz_clear(op1);
mpz_set(dest, m);
mpz_clear(m);
}

void EllPoint::print(const char* name) const {
if (m_coord.isInf) {
printf("[%s] Point is at infinty (neutral element)\n", name);
return;
}
printf("[%s] Point coordinates are \n\tx : ", name);
mpz_out_str(stdout, 10, m_coord.x);
printf("\n\ty : ");
mpz_out_str(stdout, 10, m_coord.y);
printf("\nwith \tq : ");
mpz_out_str(stdout, 10, m_param.p);
printf("\n");
}
```

```
bool EllPoint::isOnCurve(const ECCoord& coord) const {
if (coord.isInf) return true;
mpz_t op1, op2, op3;
mpz_init_set(op1, coord.y);
mpz_powm_ui(op1, op1, 2, m_param.p);
//op1 = y^2
mpz_init_set(op2, coord.x);
mpz_init_set(op3, coord.x);
mpz_mul(op2, coord.x, m_param.a);
mpz_mul(op3, coord.x, coord.x);
mpz_addmul(op2, op3,coord.x);
mpz_add(op2, op2, m_param.b);
mpz_mod(op2, op2, m_param.p);
//op2 = x^3 + ax + b

int res = mpz_cmp(op1, op2);
mpz_clear(op1);
mpz_clear(op2);
mpz_clear(op3);
return res == 0;
}
```

Poly.h

```
#pragma once
#include "EllPoint.h"
#include <vector>
#include <fstream>
#include <iostream>

#define NEW_ALLOC_SIZE(x) (1.5*x) // soft
//#define NEW_ALLOC_SIZE(x) (1.5*x*x + 7*x + 1) // HARD

#define BASE_n 10

struct Poly_p {
Poly_p(const mpz_t& p, mpir_ui size = 1);
Poly_p(const Poly_p& p);
~Poly_p();

void resize(mpir_ui n);
mpir_ui size() const;

void operator+=(const Poly_p& p);
void operator-=(const Poly_p& p);
void operator*=(const Poly_p& p);
void operator<<=(mpir_ui n);
void operator=(Poly_p&& p);
void operator=(const Poly_p& p);
mpz_t& operator[](mpir_ui i);
const mpz_t& operator[](mpir_ui i) const;

void print() const;
void writeOut(std::ofstream& out);

// returns false fi there has been no cut
bool cut(Poly_p& A, Poly_p& B, mpir_ui deg) const;
private:
// nb_coef must be <= size.
void replace(mpir_ui size, mpir_ui nb_coef = 0, mpz_t* coef = nullptr);

mpir_ui            m_size;
mpz_t*       m_coef;
mpir_ui            m_alloc_size;
mpz_t        m_p;
};
```

Poly.cpp

```
#include "Poly.h"

Poly_p::Poly_p(const mpz_t& p, mpir_ui size) :
      m_size(size),
      m_alloc_size(NEW_ALLOC_SIZE(size)),
      m_coef(nullptr)
{
mpz_init_set(m_p, p);
m_coef = new mpz_t[m_alloc_size];
for (int i = 0; i < m_alloc_size; i++)
mpz_init_set_ui(m_coef[i], 0);
}

Poly_p::Poly_p(const Poly_p& p) :
      m_size(p.m_size),
      m_alloc_size(p.m_alloc_size),
      m_coef(nullptr)
{
mpz_init_set(m_p, p.m_p);
m_coef = new mpz_t[m_alloc_size];
for (int i = 0; i < m_alloc_size; i++)
mpz_init_set(m_coef[i], p[i]);
}

Poly_p::~Poly_p() {
for (int i = 0; i < m_alloc_size; i++)
mpz_clear(m_coef[i]);
delete[] m_coef;
mpz_clear(m_p);
}

void Poly_p::resize(mpir_ui n) {
replace(n, 0, nullptr);
}

mpir_ui Poly_p::size() const {
return m_size;
}

void Poly_p::operator+=(const Poly_p& p) {
mpir_ui m = std::min(m_size, p.size());
mpir_ui M = std::max(m_size, p.size());
replace(M, m_size, m_coef);

for (mpir_ui i = 0; i < m; i++) {
mpz_add(m_coef[i], m_coef[i], p[i]);
mpz_mod(m_coef[i], m_coef[i], m_p);
}
if (p.size() == M) {
for (mpir_ui i = m; i < M; i++)
mpz_set(m_coef[i], p[i]);
}
}
```

```
void Poly_p::operator-=(const Poly_p& p) {
mpir_ui m = std::min(m_size, p.size());
mpir_ui M = std::max(m_size, p.size());
replace(M, m_size, m_coef);

for (mpir_ui i = 0; i < m; i++) {
mpz_sub(m_coef[i], m_coef[i], p[i]);
mpz_mod(m_coef[i], m_coef[i], m_p);
}
if (p.size() == M) {
for (mpir_ui i = m; i < M; i++) {
mpz_neg(m_coef[i], p[i]);
mpz_mod(m_coef[i], m_coef[i], m_p);
}
}
}

void Poly_p::operator<<=(mpir_ui n) {
mpz_t* coef_new = new mpz_t[m_alloc_size + n];
for (mpir_ui i = 0; i < n; i++)
mpz_init_set_ui(coef_new[i],0);
for (mpir_ui i = 0; i < m_alloc_size; i++)
mpz_init_set(coef_new[i + n],m_coef[i]);
for (mpir_ui i = 0; i < m_alloc_size; i++)
mpz_clear(m_coef[i]);
delete[] m_coef;
m_coef = coef_new;
m_size += n;
m_alloc_size += n;
}
```

```
void Poly_p::operator*=(const Poly_p& p) {
if (m_size >= 500 || p.size() >= 500) {
Poly_p A(m_p), B(m_p), C(m_p), D(m_p);
mpir_ui deg = m_size / 2;
cut(A, B, deg);
if (!p.cut(C, D, deg)) { // C = 0
A *= D; B *= D;
*this = A;
*this <<= deg;
*this += B;
return;
};
Poly_p tmp_1(A);
Poly_p tmp_2(C);
tmp_1 -= B;
tmp_2 -= D;
A *= C;
B *= D;
tmp_2 *= tmp_1;
tmp_1 = A;
tmp_1 += B;
tmp_1 -= tmp_2;
// *this = AC, B = BD, tmp_2 = (A - B)(C - D),
// tmp_1 = AC + BD - (A-B)(C-D) = AD + BC
A <<= m_size;
tmp_1 <<= deg;
A += tmp_1;
A += B;
*this = A;
return;
}

mpir_ui m = m_size; mpir_ui M = p.size();
mpir_ui d = m + M - 1;
mpz_t* coef = new mpz_t[d];
for (mpir_ui i = 0; i < d; i++)
mpz_init_set_ui(coef[i], 0);
for (int i = 0; i < m; i++)
for (int j = 0; j < M; j++) {
mpz_addmul(coef[i + j], m_coef[i], p[j]);
mpz_mod(coef[i + j], coef[i + j], m_p);
}

replace(d, d, coef);
for (mpir_ui i = 0; i < d; i++)
mpz_clear(coef[i]);
delete[] coef;
}
```

```
void Poly_p::operator=(Poly_p&& p)
      {replace(p.size(), p.size(), p.m_coef);}
void Poly_p::operator=(const Poly_p& p)
      {replace(p.size(), p.size(), p.m_coef);}
mpz_t& Poly_p::operator[](mpir_ui i)
      { return m_coef[i]; }
const mpz_t& Poly_p::operator[](mpir_ui i) const
      { return m_coef[i]; }

void Poly_p::print() const {
char str[16384];
mpz_get_str(str, BASE_n, m_coef[0]);
printf("%s", str);
for (int i = 1; i < m_size; i++) {
mpz_get_str(str, BASE_n, m_coef[i]);
if (mpz_sgn(m_coef[i]) >= 0)
printf(" + %s X^%d", str, i);
else
printf(" %sX^%d", str, i);
}
printf("\n");
}

void Poly_p::writeOut(std::ofstream& out) {
char str[1 << 18];
mpz_get_str(str, 10, m_coef[0]);
out << str;
for (int i = 1; i < m_size; i++) {
mpz_get_str(str, 10, m_coef[i]);
if (mpz_sgn(m_coef[i]) >= 0)
out << " + " << str << "X^" << i;
else
out << " " << str << "X^" << i;
}
out << "\n";
}

bool Poly_p::cut(Poly_p& A, Poly_p& B, mpir_ui deg) const {
signed long long size_a = m_size - deg;
if (size_a <= 0) {
A.replace(1);
B = *this;
return false;
}
A.replace(size_a, size_a, &m_coef[deg]);
B.replace(deg, deg, m_coef);
return true;
}
```

```
void Poly_p::replace(mpir_ui size, mpir_ui nb_coeff, mpz_t* coef) {
if (size > m_alloc_size) {
mpir_ui m_alloc_size_new = NEW_ALLOC_SIZE(size);
mpz_t* m_coef_new = new mpz_t[m_alloc_size_new];
if (coef) {
for (mpir_ui i = 0; i < nb_coeff; i++)
mpz_init_set(m_coef_new[i], coef[i]);
for (mpir_ui i = nb_coeff; i < size; i++)
mpz_init_set_ui(m_coef_new[i], 0);
}
else {
for (mpir_ui i = 0; i < size; i++)
mpz_init_set_ui(m_coef_new[i], 0);
}
for (mpir_ui i = size; i < m_alloc_size_new; i++)
mpz_init_set_ui(m_coef_new[i], 0);

for (mpir_ui i = 0; i < m_alloc_size; i++)
mpz_clear(m_coef[i]);
delete[] m_coef;
m_coef = m_coef_new;
m_size = size;
m_alloc_size = m_alloc_size_new;
}
else {
if (coef && (m_coef != coef)) {
for (mpir_ui i = 0; i < size; i++)
mpz_set(m_coef[i], coef[i]);
for (mpir_ui i = size; i < m_size; i++)
mpz_set_ui(m_coef[i], 0);
}
else if (!coef) { // reset
for (mpir_ui i = 0; i < m_size; i++)
mpz_set_ui(m_coef[i], 0);
}
else {
for (mpir_ui i = size; i < m_size; i++)
mpz_set_ui(m_coef[i], 0);
}
m_size = size;
}
}
```

EllCurve.h

```
#pragma once
#include <stdarg.h>
#include <stdio.h>
#include <time.h>
#include <mpir.h>
#include <string>
#include <vector>
#include <ctime>
#include <algorithm>
#include <unordered_set>
#include "EllPoint.h"
#include "Poly.h"

#define BSGS_MEMORY 3*1024*1024*1024
#define MAX_RANDOM_BITCOUNT 512

/*Elliptic Curve*/
class EllCurve
{
public:
/*Constructs a new elliptic curve with its generator set to 0*/
EllCurve();
EllCurve(const ECParam& param);
EllCurve(ECParam&& param);
EllCurve(const EllCurve& curve);
~EllCurve();

/*Returns i*P where P is the generator in use*/
EllPoint operator[](const mpz_t& i) const;

/*Returns i*P where P is the generator in use*/
EllPoint operator[](mpir_ui i) const;

/*Sets new curve parameters, finds new generator and then returns true
if parameters are usable (i.e no problems such as curve singularity or
weak choice of p (?)). If param order is null, finds curve order.
Returns false otherwise and resets curve parameters.*/
bool setECParam(ECParam&& param, bool verbose = false);
bool setECParam(const ECParam& param, bool verbose = false);

/*Returns the parameters of the curve in use*/
const ECParam& getECParam() const;

/*Returns generator in use*/
const EllPoint& getGen() const;

/* Sets a new generator. Returns true if valid order and point on curve, false otherwise*/
bool setGen(const ECCoord& coord, const mpz_t& order);

/*Finds a new generator by efficiently randomly finding a new point on
the elliptic curve in use.*/
void findNewGen();

/* Finds order of the generator*/
void findGenOrder();

/*Returns order of the subgroup generated by generator. For now, since order is prime, same as curve order.*/
const mpz_t& getGenOrder() const;
```

```
/*Finds a point on the elliptic curve in use, by random trials.
Because this can be very slow for large Fp, user can specify
a time threshold in ms after which the function will fail,
returning false.
On success returns true with found point stored in dest*/
bool getRandomPointRandomTrials(EllPoint& dest,
                                mpir_ui time_threshold_ms = ~0);

/*Returns kP with random k which will then be stored in random_k parameter*/
EllPoint getRandomPoint(mpz_t& random_k);

/* Finds new random parameter A, B and P for the curve. If an argument
P is passed, the most probable prime greater than P will be used and
new A and B will be found accordingly. Gen point is reset to neutral point*/
void setRandomParam(mpir_ui p = 0,
                        bool findOrder = false, bool verbose = false);
void setRandomParam(const mpz_t p = NULL,
                        bool findOrder = false, bool verbose = false);

/*Finds curve order using naive formula.
Result will be stored in internal ECParam.order*/
void findCurveOrderNaive();

/*Finds curve order using Hasse Theorem naively.
Result will be stored in internal ECParam.order*/
void findCurveOrderHasseNaive();

/*Finds curve order using Hasse Theorem and baby-steps giant-steps.
Result will be stored in internal ECParam.order*/
bool findCurveOrderHasseBSGS();

/*Finds curve order using basic Schoof algorithm.
Result will be stored in internal ECParam.order*/
void findCurveOrderSchoof(); // TODO

/*Finds curve order using improved Schoof algorithm
(Schoof-Elkies-Atkin aka SEA).
Result will be stored in internal ECParam.order*/
void findCurveOrderSEA(); // TODO

/*Returns curve order.*/
const mpz_t& getCurveOrder() const;
```

```
/*Finds k satisfying K = k*P where P is the generator in use.
Cracking the discrete logarithm problem with baby step giant
step method. max_memory_usage is the maximum amount of memory
the function will be able to use (default being BSGS_MEMEORY
Bytes). Returns true if found before time threshold,
false otherwise*/
bool crackDiscreteLogBSGS(mpz_t& k, const EllPoint& K,
        mpir_ui time_threshold_ms = -1,
        mpir_ui max_memory_usage = BSGS_MEMORY) const;

/*Finds k satisfying K = k*P where P is the generator in use.
Cracking the discrete logarithm problem naively.
Returns true if found before time threshold, false otherwise*/
bool crackDiscreteLogNaive(mpz_t& k, const EllPoint& K ,
                        mpir_ui time_threshold_ms = -1) const;

/*Prints the elliptic curve parameters with its name if given,
and the generator in use*/
void print(const char* name = "", bool printGen = false) const;

// Finds exact order of G from k, knowing that k * G = 0.
// k is set to the lowest number such that k * G = 0;
assumes sqrt(k) can be stored in mpir_ui
void static find_exact_order(mpz_t& k, const EllPoint& G);

// pre compute j*gen for 0<= j <= m
void static pre_compute_bsgs(
        std::unordered_set<ECPair, BSGSHasher>& data,
        const EllPoint& gen, mpir_ui m);

// find k != 0 such that k * G = K using baby step giant step.
//m is such that one can write k = (am + b), 0<= a,b <= m.
// data must hold j*gen for 0<= j <= m
void static find_k_bsgs(mpz_t& k,
        const std::unordered_set<ECPair, BSGSHasher>& data,
        const EllPoint& gen, const EllPoint& K, mpir_ui m);

private :

EllPoint                        m_gen; // subgroup generator
mpz_t                           m_genOrder;
ECParam                         m_param;
};
```

EllCurve.cpp

```
#include "EllCurve.h"

EllCurve::EllCurve() : m_gen(ECParam())
{
mpz_init(m_genOrder);
}

EllCurve::EllCurve(const ECParam& param) : m_gen(ECParam())
{
mpz_init(m_genOrder);
setECParam(param);
}

EllCurve::EllCurve(ECParam&& param) : m_gen(ECParam())
{
mpz_init(m_genOrder);
setECParam(param);
}

EllCurve::EllCurve(const EllCurve& curve)
: m_gen(curve.getECParam()) // Generator is initialiazed to 0 (inty)
, m_param(curve.getECParam())
{
mpz_init(m_genOrder);
}


EllCurve::~EllCurve() {
mpz_clear(m_genOrder);
}

/*Returns i*P where P is the generator in use*/
EllPoint EllCurve::operator[](const mpz_t& i) const{
EllPoint tmp(m_gen);
tmp *= i;
return tmp;
}

/*Returns i*P where P is the generator in use*/
EllPoint EllCurve::operator[](mpir_ui i) const {
EllPoint tmp(m_gen);
tmp *= i;
return tmp;
}

/*Returns the parameters of the curve in use*/
const ECParam& EllCurve::getECParam() const {
return m_param;
}

/*Sets new curve parameters, finds new generator and then returns true
if parameters are usable (i.e no problems such as curve singularity,
null discriminant or weak choice of p (?)). Returns false otherwise.*/
bool EllCurve::setECParam(ECParam&& param, bool verbose) {
return setECParam(param, verbose);
}
```

```
/*Sets new curve parameters, sets geenrator to 0 and then returns true
if parameters are usable (i.e no problems such as curve singularity,
null discriminant or weak choice of p (?)). Returns false otherwise.*/
bool EllCurve::setECParam(const ECParam& param, bool verbose) {
if (verbose) {
if (mpz_divisible_ui_p(param.p, 2) || mpz_divisible_ui_p(param.p, 3)) {
printf("Error : elliptic curves defined on field of \
characteristic 2 or 3 aren't supported.\n");
return false;
}
mpz_t discr, j; mpz_inits(discr, j, NULL);
if (!Rand.is_likely_prime(param.p)) {
printf("Warning : field order is not prime.\n");
return false;
}
else {
int shifts = 0;
mpz_set(discr, param.p);
while (mpz_sgn(discr)) {
mpz_fdiv_q_2exp(discr, discr, 1);
shifts++;
}
printf("Curve security level : %d bits\n", shifts / 2);
}
set_weierstrass_discriminant_j_invariant
       (discr, j, param.a, param.b, param.p);
if (mpz_sgn(discr) == 0) {
printf("Error : new curve parameters yield null discriminant \
: curve is singular and cannot be used.\n");
mpz_clears(discr, j, NULL);
return false;
}
if (mpz_sgn(j) == 0) {
printf("Warning : new curve parameters yield null j_invariant \
: curve is super-singular, potentially weak\n");
}
else if (mpz_cmp_ui(j, 1728) == 0) {
printf("Warning : new curve parameters yield j_invariant = 1728 \
: curve is super-singular, potentially weak\n");
}
m_param = param;
if (mpz_sgn(param.order) == 0) {
printf("Order not specified.\n");
}
// if order is given, assume it is right
printf("New curve set with :\n\tDiscriminant = "); mpz_print(discr);
printf("\tj_invariant = "); mpz_print(j);
m_gen = EllPoint(param); // Resetting gen point to inf
mpz_set_ui(m_genOrder, 1);
mpz_clears(discr, j, NULL);
print("new curve", true);
return true;
}
```

```
//else
if (mpz_divisible_ui_p(param.p, 2) || mpz_divisible_ui_p(param.p, 3))
return false;
mpz_t discr, j; mpz_inits(discr, j, NULL);
if (!Rand.is_likely_prime(param.p))
return false;

set_weierstrass_discriminant_j_invariant
(discr, j, param.a, param.b, param.p);
if (mpz_sgn(discr) == 0) {
mpz_clears(discr, j, NULL);
return false;
}
m_param = param;

m_gen = EllPoint(param); // Resetting gen point to inf
mpz_set_ui(m_genOrder, 1);
mpz_clears(discr, j, NULL);
return true;
}
```

```
/* Finds new random parameter A, B and P for the curve. If an argument
P is passed, the most probable prime greater than P will be used and
new A and B will be found accordingly. Gen point
is reset to neutral point*/
void EllCurve::setRandomParam(mpir_ui p,
                       bool findOrder, bool verbose) {
mpz_t tmp; mpz_init_set_ui(tmp, p);
setRandomParam(tmp, findOrder, verbose);
mpz_clear(tmp);
}

/* Finds new random parameter A, B and P for the curve. If an argument
P is passed, the most probable prime greater than P will be used and
new A and B will be found accordingly. Gen point
is reset to neutral point. */
void EllCurve::setRandomParam(const mpz_t p,
                       bool findOrder, bool verbose) {
if (p != NULL) {
mpz_set(m_param.p, p);
while (Rand.is_likely_prime(m_param.p) == 0)
Rand.next_prime_candidate(m_param.p, m_param.p);
}
else {
do {
Rand.randomb(m_param.p, rand() % MAX_RANDOM_BITCOUNT);
} while (!Rand.is_likely_prime(m_param.p));
}
ECParam param(m_param);
//mpz_set_ui(param.order, 0); // so curve order is found
while (true) {
if (verbose) {
printf("Constructing curve in E(Fp) where p = ");
mpz_print(param.p);
}
for (int i = 0; i < 3; i++) {
Rand.randomm(param.a, param.p);
Rand.randomm(param.b, param.p);
if (setECParam(param)) {
if (verbose)
printf("Finding curve order...\n");
if (!findCurveOrderHasseBSGS())
continue;
return;
}
}
do { Rand.next_prime_candidate(param.p, param.p); }
while (Rand.is_likely_prime(param.p) == 0);
}

}
```

```
/*Returns generator in use*/
const EllPoint& EllCurve::getGen() const {
return m_gen;
}

bool EllCurve::setGen(const ECCoord& coord, const mpz_t& order) {
mpz_set(m_genOrder, m_param.order);
if (m_gen.setCoord(coord)) {
if (order) {
mpz_set(m_genOrder, order);
return true;
}
else {
find_exact_order(m_genOrder, getGen());
return true;
}
}
return false;
}

/*Finds a new generator by efficiently
randomly finding a new point on the elliptic curve in use.*/
void EllCurve::findNewGen() {
m_gen.setRandomCoord();
}

void EllCurve::findGenOrder() {
mpz_set(m_genOrder, m_param.order);
find_exact_order(m_genOrder, m_gen);
}

/*Returns curve order.*/
const mpz_t& EllCurve::getCurveOrder() const {
return m_param.order;
}

/*Returns order of the subgroup generated by generator.
For now, since order is prime, same as curve order.*/
const mpz_t& EllCurve::getGenOrder() const {
return m_genOrder;
}
```

```
/*Finds a point on the elliptic curve in use, by random trials.
Because this can be very slow for large Fp, user can specify a time threshold
in ms after which the funciton will fail, returning false. 0 means no threshold,
which may lead to freezing the program.
On success returns true with found point stored in dest*/
bool EllCurve::getRandomPointRandomTrials(EllPoint& dest, mpir_ui time_threshold_ms) {
mpz_t op1, op2;
mpz_inits(op1, op2, NULL);
clock_t begin = clock();
while (true) {
for (int i = 0; i < 1024; i++) {
Rand.randomm(op1, m_param.p);
Rand.randomm(op2, m_param.p);
if (dest.setCoord(ECCoord(NOT_INFTY, op1, op2))) {
mpz_clears(op1, op2, NULL);
return true;
}
}
if (1000.0 * double(clock() - begin) / (double)CLOCKS_PER_SEC > time_threshold_ms) break;
}
mpz_clears(op1, op2, NULL);
return false;
}

/*Returns kP with random k which will then be stored in random_k parameter*/
EllPoint EllCurve::getRandomPoint(mpz_t& random_k) {
Rand.randomb(random_k, mpz_sizeinbase(m_param.p, 2));
EllPoint tmp(m_gen);
tmp *= random_k;
return tmp;
}

/*Finds curve order using simple formula |E(Fq)| = 1 + sum for x in Fq (1  + legendre(x^3 + ax + b, q) )
Result will be stored in internal ECParam.order*/
void EllCurve::findCurveOrderNaive() {
// using simple formula |E(Fq)| = 1 + sum for x in Fq (1  + legendre(x^3 + ax + b, q) )
// or |E(Fq)| = 1 + q + sum for x in Fq legendre(x^3 + ax + b, q) but we'll use the former
// because we can only easily add unsigned integers with mpir.
mpz_set_ui(m_param.order, 1);
mpz_t x, weier_x;
mpz_init(weier_x);
mpz_init_set_ui(x, 0);
while (mpz_cmp(x, m_param.p) < 0) {
set_weierstrass(weier_x, x, m_param.a, m_param.b, m_param.p);
mpz_add_ui(m_param.order, m_param.order, 1 + mpz_legendre(weier_x, m_param.p));
mpz_add_ui(x, x, 1);
}
mpz_clears(weier_x, x, NULL);

}
```

```
/*Finds curve order naively using Hasse Theorem
Result will be stored in internal ECParam.order*/
void EllCurve::findCurveOrderHasseNaive() {
std::vector<mpir_ui> orders;
mpz_t ppcm; mpz_init_set_ui(ppcm, 1);

mpz_t inf; mpz_init(inf);
mpz_sqrt(inf, m_param.p);
mpz_mul_ui(inf, inf, 2); // inf = 2sqrt(p)
mpz_sub(inf, m_param.p, inf);
mpz_add_ui(inf, inf, 1);
// inf = p + 1 - 2sqrt(p)

mpz_t width; mpz_init_set_ui(width, 0);
mpz_sqrt(width, m_param.p);
mpz_mul_ui(width, width, 4);
// width = 4sqrt(p)
mpir_ui w = mpz_get_ui(width) + 1;

mpz_t k, m; mpz_inits(k, m, NULL);
 // Finding k such that k*gen = 0,
//with p + 1 - 2sqrt(p) <= k <= p + 1 + 2sqrt(p)
do {
do {
findNewGen();
} while (getGen().isInf());
EllPoint P(getGen());
EllPoint Q(getGen());
Q *= inf;

mpz_set(k, inf);
while(!Q.isInf()) {
Q += P;
mpz_add_ui(k, k, 1);
}
```

```
// prime factorization of k now

// if k is prime we're done
if (Rand.is_likely_prime(k)) {
if (mpz_cmp(k, inf) < 0 // otherwise curve order is prime (k > p + 1 - 2sqrt(p))
&& std::find(orders.begin(), orders.end(), mpz_get_ui(k)) == orders.end()) {
// hoping k is sufficiently small
orders.push_back(mpz_get_ui(k));
}
mpz_set(m_genOrder, k);
mpz_lcm(ppcm, ppcm, k);
continue;
}

// k = p1^a1 * .. * pr^ar. Finding k' such that
//ord(gen) = k' = p1^a1' * ... * pr^ar'
find_exact_order(k, getGen());

if (std::find(orders.begin(), orders.end(), mpz_get_ui(k)) == orders.end()) {
// so we are sure it is not an element already seen
orders.push_back(mpz_get_ui(k));
mpz_lcm(ppcm, ppcm, k);
}
mpz_set(m_genOrder, k);

} while (mpz_cmp(ppcm, width) <= 0);

// Finding unique N in right range : N = n*ppcm, ppcm > 4sqrt(p),
//p + 1 - 2sqrt(p) <= n*ppcm <= p + 1 + 2sqrt(p)
// so inf <= ppcm*n <= inf + 4sqrt(p) and
//inf / ppcm <= n <= inf / ppcm + e where e < 1 : n = ceil(inf / ppcm)
mpz_t n; mpz_init(n);
mpz_cdiv_q(n, inf, ppcm); // n = ceil(inf / ppcm)
mpz_mul(m_param.order, n, ppcm); // ok : N = n * ppcm

mpz_clears(n, ppcm, inf, width, k, m, NULL);
}
```

```
/*Finds curve order using baby-steps giant-steps.
Result will be stored in internal ECParam.order*/
bool EllCurve::findCurveOrderHasseBSGS() {

std::vector<mpir_ui> orders;
mpz_t ppcm; mpz_init_set_ui(ppcm, 1);

mpz_t sup; mpz_init(sup);
mpz_sqrt(sup, m_param.p);
mpz_mul_ui(sup, sup, 2);
mpz_add(sup, sup, m_param.p);
mpz_add_ui(sup, sup, 1); // sup =  p + 1 + 2sqrt(p)

mpz_t width; mpz_init_set_ui(width, 0);
mpz_sqrt(width, m_param.p);
mpz_mul_ui(width, width, 4); // width =  4sqrt(p)

mpz_t m, k, l, tmp;
mpz_inits(m, k, l, tmp, NULL);
mpz_sqrt(m, width); // m = sqrt(4sqrt(p))

do {
do {
findNewGen();
} while (getGen().isInf());

// If space needed is too big for memory allocation
// (let's say 4*sizeof(ECPair) , if hash function isn't too good)
mpz_mul_ui(tmp, m, 4*sizeof(ECPair));
if (mpz_cmp_ui(tmp, BSGS_MEMORY) > 0) {
printf("Cannot run bsgs : too much memory needed : ");
mpz_print(tmp);
mpz_clears(m, k, l, tmp, ppcm, sup, width, NULL);
return false;
}

// Finding k such that k*gen = 0,
// with p + 1 - 2sqrt(p) <= k <= p + 1 + 2sqrt(p)
// k =(p + 1 -2sqrt(p)) + am +b where m = sqrt(4sqrt(p))
// and -(p + 1)gen = (am + b)gen

// Pre-computing j*gen fo j in 0, m
std::unordered_set<ECPair, BSGSHasher> data;
pre_compute_bsgs(data, getGen(), mpz_get_ui(m)+1);

// Now finding k
EllPoint Q(getGen());
mpz_divexact_ui(tmp, width, 2); // tmp = 2sqrt(p)
mpz_sub(tmp, m_param.p, tmp);
mpz_add_ui(tmp, tmp, 1); // tmp = p + 1 - 2sqrt(p)
Q *= tmp;
Q.inverse();
mpz_set_ui(k, 0);
if (!Q.isInf())
find_k_bsgs(k, data, getGen(), Q, mpz_get_ui(m));

mpz_add(k, k, tmp); // ok
data.clear();
EllPoint P = getGen();
P *= k;
if (!P.isInf()) {
printf("problem, with k = "); mpz_print(k);
P.print();
mpz_clears(m, k, l, tmp, ppcm, sup, width, NULL);
return false;
}

// prime fact of k now
std::vector<mpir_ui> primes;
EllPoint T(getGen());

// If k is big enough
if (mpz_cmp(k, width) > 0) {
// gen order is still undefined
mpz_lcm(ppcm, ppcm, k);
continue;
}
// if k is prime we're done
if (Rand.is_likely_prime(k)) {
mpz_sub(tmp, sup, width);
if (mpz_cmp(k, tmp) < 0 // otherwise curve order is prime
&& std::find(orders.begin(), orders.end(),
mpz_get_ui(k)) == orders.end()) {
// hoping k is sufficiently small
orders.push_back(mpz_get_ui(k));
}
mpz_set(m_genOrder, k);
mpz_lcm(ppcm, ppcm, k);
continue;
}

// k = p1^a1 * .. * pr^ar. Finding k' such that
ord(gen) = k' = p1^a1' * ... * pr^ar'
// this function can take a lot of memory (prime table)
// only about sqrt(sqrt(p)) since k < width. ok
find_exact_order(k, getGen());

if (std::find(orders.begin(), orders.end(),
mpz_get_ui(k)) == orders.end()){ // so we are sure not already seen
orders.push_back(mpz_get_ui(k));
mpz_lcm(ppcm, ppcm, k);
}
mpz_set(m_genOrder, k);

} while (mpz_cmp(ppcm, width) <= 0);

// Finding unique N in right range : N = n*ppcm, ppcm > 4sqrt(p),
//p + 1 - 2sqrt(p) <= n*ppcm <= p + 1 + 2sqrt(p)
// so sup - 4sqrt(p) <= ppcm*n <= sup and s
//up / ppcm - e<= n <= sup / ppcm where e < 1 : n = floor(sup / ppcm)
mpz_t n; mpz_init(n);
mpz_fdiv_q(n, sup, ppcm); // n = floor(sup / ppcm)
mpz_mul(m_param.order, n, ppcm); // ok : N = n * ppcm

mpz_clears(n, m, k, l, tmp, ppcm, sup, width, NULL);
return true;
}
```

```
void EllCurve::print(const char* name, bool printGen) const {
printf("[%s] Curve is defined by ", name);
printf("\n\tE(Fq) : y^2 = x^3 + ");
mpz_out_str(stdout, 10, m_param.a);
printf(" * x + ");
mpz_out_str(stdout, 10, m_param.b);
printf("\n\twhere q = ");
mpz_out_str(stdout, 10, m_param.p);
printf("\n\torder is = ");
mpz_out_str(stdout, 10, m_param.order);
if (printGen) {
printf("\n");
m_gen.print("Generator");
}
printf("\n");
}

void EllCurve::find_exact_order(mpz_t& k,
                   const EllPoint& G) {
std::vector<mpir_ui> primes;
EllPoint T(G);

mpz_t m; mpz_init(m);
mpz_sqrt(m, k);
mpir_ui s = mpz_get_ui(m);
get_primes(primes, s);

// k = p1^a1 * .. * pr^ar. Finding k' such that
//ord(gen) = k' = p1^a1' * ... * pr^ar'
for (auto& i : primes) {
if (!mpz_divisible_ui_p(k, i)) continue;
while (mpz_divisible_ui_p(k, i)) {
mpz_divexact_ui(k, k, i);
T = G;
T *= k;
if (!T.isInf()) {
mpz_mul_ui(k, k, i);
break;
}
}
}
mpz_clear(m);
}

void EllCurve::pre_compute_bsgs
       (std::unordered_set<ECPair, BSGSHasher>& data,
       const EllPoint& G, mpir_ui m) {
EllPoint P(G);
data.reserve(m + 1); // so there is no rehash during computation
EllPoint InvGen(P); InvGen.inverse();
P *= m;
while (m > 0) {
data.emplace(ECPair(P.getCoord(), m));
P += InvGen;
m--;
}
data.emplace(ECPair(P.getCoord(), m)); // m = 0
}
```

```
// find k != such that k * G = K using baby step giant step.
//m is such that it is possible to write k = (am + b), 0<= a,b <= m.
// data must hold j*gen for 0<= j <= m
void EllCurve::find_k_bsgs(mpz_t& k,
                   const std::unordered_set<ECPair, BSGSHasher>& data,
                   const EllPoint& gen, const EllPoint& K, mpir_ui m) {
// assumes m can be stored in mpir_ui but m^2 might not
mpz_set_ui(k, 1);
mpir_ui l=0; // l <= m can be stored in mpir_ui
EllPoint _mPoint(gen);
_mPoint *= m;
_mPoint.inverse();
EllPoint R(K);
// beginning at l = 1 so k is not set to 0 if R.isInf()
R.isInf() ? l = 1 : l = 0;

while (true) {
std::unordered_set<ECPair, BSGSHasher>::const_iterator it =
                          data.find(ECPair(R.getCoord(), 0));
                          // Since second value doesn't count in hash
if (it != data.end()) {
mpir_ui j_value = (*it).val;
mpz_mul_ui(k, k, m);
mpz_mul_ui(k, k, l);
mpz_add_ui(k, k, j_value); // k = ml + j
break;
}
R += _mPoint;
l++;
}
}
```

test.h

```
#pragma once
#include <thread>
#include <algorithm>
#include <mutex>
#include "Constants.h"
#include "plot.h"
#include "ECDH.h"
#include "ECDSA.h"
#include "Poly.h"

#define THREADS 4

void compute_formula_1_p(std::vector<Poly_p>& P,
      const Poly_p& Q, const mpz_t& p, mpir_ui i) {
Poly_p R(p);
//5
P[i] += P[(i / 2) + 2];
P[i] *= P[i / 2];
P[i] *= P[i / 2];
P[i] *= P[i / 2];
P[i] *= Q;
R += P[(i / 2) - 1];
R *= P[(i / 2) + 1];
R *= P[(i / 2) + 1];
R *= P[(i / 2) + 1];
P[i] -= R;
}

void compute_formula_2_p(std::vector<Poly_p>& P,
             const Poly_p& Q, const mpz_t& p,  mpir_ui i) {
Poly_p R(p);
P[i + 1] += P[((i + 1) / 2) + 2];
P[i + 1] *= P[((i + 1) / 2) - 1];
P[i + 1] *= P[((i + 1) / 2) - 1];
R += P[((i + 1) / 2) - 2];
R *= P[((i + 1) / 2) + 1];
R *= P[((i + 1) / 2) + 1];
P[i + 1] -= R;
P[i + 1] *= P[(i + 1) / 2];
for (int j = 0; j < P[i + 1].size(); j++)
mpz_divexact_ui(P[i + 1][j], P[i + 1][j], 2);
}

void compute_division_polynomials_p(int n,
             const ECParam& p, const std::string& name) {
n = max(5, n);
n += 4 - (n % 4);
std::vector<Poly_p> P;
Poly_p R(p.p);
for (int i = 0; i < n+1; i++) P.push_back(Poly_p(R));

P[0].resize(1);
mpz_init_set_ui(P[0][0], 0);

P[1].resize(1);
mpz_init_set_ui(P[1][0], 1);

P[2].resize(1);
mpz_init_set_ui(P[2][0], 2);
P[3].resize(5);
mpz_init_set_ui(P[3][0], 0);
mpz_submul(P[3][0], p.a, p.a);
mpz_init_set_ui(P[3][1], 0);
mpz_addmul_ui(P[3][1], p.b, 12);
mpz_init_set_ui(P[3][2], 0);
mpz_addmul_ui(P[3][2], p.a, 6);
mpz_init_set_ui(P[3][3], 0);
mpz_init_set_ui(P[3][4], 3);
//P[3] =  -a^2 + 12bx + 6ax^2 + 3x^4

mpz_t tmp; mpz_init_set_ui(tmp, 0);
P[4].resize(7);
mpz_init_set_ui(P[4][0], 0);
mpz_submul(P[4][0], p.a, p.a);
mpz_submul_ui(P[4][0], p.a, 4);
mpz_submul(P[4][0], p.a, p.a);
mpz_submul(tmp, p.b, p.b);
mpz_submul_ui(P[4][0], tmp, 32);
mpz_init_set_ui(P[4][1], 0);
mpz_mul(tmp, p.a, p.b);
mpz_submul_ui(P[4][1], tmp, 16);
mpz_init_set_ui(P[4][2], 0);
mpz_mul(tmp, p.a, p.a);
mpz_submul_ui(P[4][1], tmp, 20);
mpz_init_set_ui(P[4][3], 0);
mpz_addmul_ui(P[4][3], p.b, 80);
mpz_init_set_ui(P[4][4], 0);
mpz_addmul_ui(P[4][4], p.a, 20);
mpz_init_set_ui(P[4][5], 0);
mpz_init_set_ui(P[4][6], 4);
// P[4] = -4a^3-32b^2 -16abx -20a^2x^2 -80bx^3+20ax^4 +4x^6
mpz_clear(tmp);

Poly_p Q(p.p, 4); // Q = Y^2 = X^3 + aX + b
mpz_set(Q[0], p.b);
mpz_set(Q[1], p.a);
mpz_set_ui(Q[3], 1);
Q *= Q; // Q = Q^2
// Now computing, 4 different formulas depending on i mod 4
std::thread t[4];

for (int i = 5; i < n;) {
compute_formula_1_p(P, Q, p.p,  i);
compute_formula_2_p(P, Q, p.p, i);
i += 2;
compute_formula_1_p(P, Q, p.p, i);
compute_formula_2_p(P, Q, p.p, i);
i += 2;
}

ofstream f(name, ios::out);
std::vector<char> vec(1 << 20);
f.rdbuf()->pubsetbuf(&vec.front(), vec.size());
for (int i = 0; i <= n; i++) {
f << "\nP_" << i << " = "; P[i].writeOut(f);
}
f.close();
}
```

Constants.h

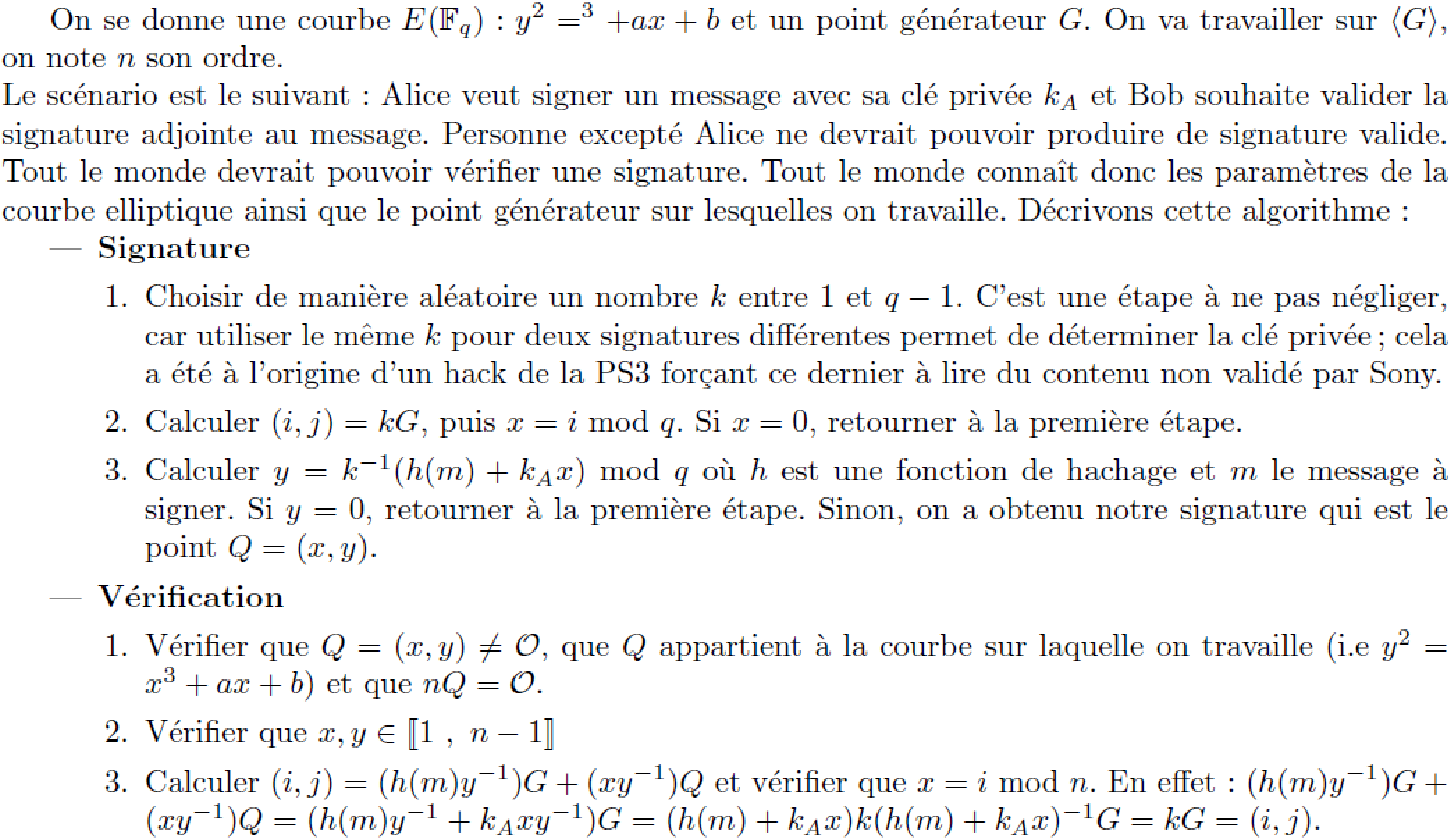
```
#pragma once
#include "EllCurve.h"

#define BIG_A "3626830188090040170470538223632857507340287510495561807368105032792956752466460499091754944292670519"
#define BIG_B "6504080042008251533437753757878856267866982998831103001251197043718345565337995842176672637937899949"
#define BIG_P "35265075282301222022179652071920843985804547105655617470846478149269603109205365857799131187032150 \
238941117870812985999909923480249871256893147030937370668858201140634615843266527731318894294518024331"

#define DEFAULT_A "11"
#define DEFAULT_B "17"
#define DEFAULT_P "1000003"

// Bitcoin chain parameters
#define BITCOIN_A "0"
#define BITCOIN_B "7"
// P to be defined later
// Gen Point
#define BITCOIN_Gx "0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798"
#define BITCOIN_Gy "0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8"
// Number of points in the field
#define BITCOIN_N "0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141"

// secp112r1
#define secp112_A  "0xDB7C2ABF62E35E668076BEAD2088"
#define secp112_B  "0x659EF8BA043916EEDE8911702B22"
#define secp112_Gx "0x09487239995A5EE76B55F9C2F098"
#define secp112_Gy "0xA89CE5AF8724C0A23E0E0FF77500"
#define secp112_P  "0xDB7C2ABF62E35E668076BEAD208B"
#define secp112_n  "0xDB7C2ABF62E35E7628DFAC6561C5"

// secp521r1
#define secp521_A
"0x01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC"
#define secp521_B
"0x0051953EB9618E1C9A1F929A21A0B68540EEA2DA725B99B315F3B8B489918EF109E156193951EC7E937B1652C0BD3BB1BF073573DF883D2C34F1EF451FD46B503F00"
#define secp521_Gx
"0x00C6858E06B70404E9CD9E3ECB662395B4429C648139053FB521F828AF606B4D3DBAA14B5E77EFE75928FE1DC127A2FFA8DE3348B3C1856A429BF97E7E31C2E5BD66"
#define secp521_Gy
"0x011839296A789A3BC0045C8A5FB42C7D1BD998F54449579B446817AFBD17273E662C97EE72995EF42640C550B9013FAD0761353C7086A272C24088BE94769FD16650"
#define secp521_P
"0x01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
#define secp521_n
"0x01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFA51868783BF2F966B7FCC0148F709A5D03BB5C9B8899C47AEBB6FB71E91386409"
```

```
/* Initializing with bitcoin chain parameters */
static void curve_init_bitcoin(EllCurve& curve) {
mpz_t p; mpz_init(p);
mpz_t op1;
mpz_set_ui(p, 1);
mpz_mul_2exp(p, p, 256);
mpz_init_set_ui(op1, 1);
mpz_mul_2exp(op1, op1, 4); mpz_sub(p, p, op1);
mpz_mul_2exp(op1, op1, 2); mpz_sub(p, p, op1);
mpz_mul_2exp(op1, op1, 1); mpz_sub(p, p, op1);
mpz_mul_2exp(op1, op1, 1); mpz_sub(p, p, op1);
mpz_mul_2exp(op1, op1, 1); mpz_sub(p, p, op1);
mpz_mul_2exp(op1, op1, 23); mpz_sub(p, p, op1);
mpz_sub_ui(p, p, 1);
// p == 2^256 - 2^32 - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1

char* s = mpz_get_str(NULL, 10, p);
curve.setECParam(ECParam(BITCOIN_A, BITCOIN_B, s, BITCOIN_N));

if (!curve.setGen(ECCoord(NOT_INFTY, BITCOIN_Gx, BITCOIN_Gy),
                    curve.getCurveOrder())) {
printf("Generator given is not on curve.\n");
curve.findNewGen();
return;
}
mpz_clears(p, op1, NULL);
}

/* Initializing with some parameters and a generator point*/
static void curve_init_default(EllCurve& curve) {
ECParam p("968113241544", "19232197347131",
             "23640121541677", "23640123950704");
curve.setECParam(p);
curve.findNewGen();
}

/* Initializing with some Big parameters */
static void curve_init_bignum(EllCurve& curve) {
curve.setECParam(ECParam(BIG_A, BIG_B, BIG_P, "0"));
curve.findNewGen();
}

static void curve_init_secp112r1(EllCurve& curve) {
curve.setECParam(ECParam(secp112_A, secp112_B, secp112_P, secp112_n));
if (!curve.setGen(ECCoord(NOT_INFTY, secp112_Gx, secp112_Gy), curve.getCurveOrder())) {
printf("Generator given is not on curve.\n");
curve.findNewGen(); // oupsi
return;
}
}

static void curve_init_secp521r1(EllCurve& curve) {
curve.setECParam(ECParam(secp521_A, secp521_B, secp521_P, secp521_n));
if (!curve.setGen(ECCoord(NOT_INFTY, secp521_Gx, secp521_Gy), curve.getCurveOrder())) {
printf("Generator given is not on curve.\n");
curve.findNewGen(); // oupsi
return;
}
}
```

```
// Some other curve i've found
#define ds1a "118916092566490565748229184"
#define ds1b "66274645112519462583913069"
#define ds1p "2475880078570760549798248507"
#define ds1n "2475880078570690266226895996"

#define ds2a "2972362603913098775400772217"
#define ds2b "5191751986946129553737567891"
#define ds2p "9903520314283042199192993897"
#define ds2n "9903520314282870095649001203"
```

Exemple d’utilisation : ECDSA

(Elliptic Curve Digital Signature Algorithm)



On se donne une courbe $E(\mathbb{F}_q) : y^2 =^3 +ax + b$ et un point générateur G. On va travailler sur $\langle G \rangle$, on note n son ordre.
Le scénario est le suivant : Alice veut signer un message avec sa clé privée k_A et Bob souhaite valider la signature adjointe au message. Personne excepté Alice ne devrait pouvoir produire de signature valide. Tout le monde devrait pouvoir vérifier une signature. Tout le monde connaît donc les paramètres de la courbe elliptique ainsi que le point générateur sur lesquelles on travaille. Décrivons cette algorithme :

- **Signature**
  1. Choisir de manière aléatoire un nombre k entre 1 et $q-1$. C’est une étape à ne pas négliger, car utiliser le même k pour deux signatures différentes permet de déterminer la clé privée ; cela a été à l’origine d’un hack de la PS3 forçant ce dernier à lire du contenu non validé par Sony.
  2. Calculer $(i, j) = kG$, puis $x = i \mod q$. Si $x = 0$, retourner à la première étape.
  3. Calculer $y = k^{-1}(h(m) + k_A x) \mod q$ où h est une fonction de hachage et m le message à signer. Si $y = 0$, retourner à la première étape. Sinon, on a obtenu notre signature qui est le point $Q = (x, y)$.
- **Vérification**
  1. Vérifier que $Q = (x, y) \neq \mathcal{O}$, que Q appartient à la courbe sur laquelle on travaille (i.e $y^2 = x^3 + ax + b$) et que $nQ = \mathcal{O}$.
  2. Vérifier que $x, y \in [\![1\ ,\ n-1]\!]$
  3. Calculer $(i, j) = (h(m)y^{-1})G + (xy^{-1})Q$ et vérifier que $x = i \mod n$. En effet : $(h(m)y^{-1})G + (xy^{-1})Q = (h(m)y^{-1} + k_A xy^{-1})G = (h(m) + k_A x)k(h(m) + k_A x)^{-1}G = kG = (i, j)$.

ECDSA.h

```
#pragma once
#include "EllCurve.h"
#include "Constants.h"

/*
The scenario is the following : Alice wants to sign a message
with her private key (dA), and Bob wants to validate the signature
using Alice's public key (HA = dA*G).
Nobody but Alice should be able to
produce valid signatures. Everyone should be able to check
signatures. Some pre-requisite :
- Order of the subgroup must be prime
  (<G> is a subgroup of prime order ok)
- The hash of the message to be signed
  should be the same bit length as n
- The random k used to generate signature must be changed
  for each signature (c.f Sony hack)*/

/*Signs messages given :
- Publicly knwon elliptic curve
- priv_key the private key of the sender (that is dA defined earlier)
- Hash hash of the message to be sent
Returns (r, s), r the x-coord of P = kG with random k
and s the variable where are "melted" r,
the message hash, priv_key and k*/
void SignMessage(mpz_t& r, mpz_t& s,
EllCurve& curve, const mpz_t& priv_key, const mpz_t& hash) {
printf("Signing message of hash "); mpz_print(hash);
curve.print("ECDSA Curve", true);
printf("Private key is "); mpz_print(priv_key);
printf("Corresponding public key is ");
mpz_t k; mpz_init(k);
EllPoint P(curve.getGen());
do {
do { P = curve.getRandomPoint(k); } while (P.isInf());
mpz_mod(r, P.getCoord().x, curve.getGenOrder()); // r ok
mpz_set(s, hash);
mpz_addmul(s, r, priv_key);
mpz_invert(k, k, curve.getGenOrder());
mpz_mul(s, s, k);
mpz_mod(s, s, curve.getGenOrder()); // s ok
mpz_print(curve.getGenOrder());
} while (mpz_sgn(s) == 0);
mpz_clear(k);

printf("Signature (r, s) is : \n\tr = "); mpz_print(r);
printf("\ts = "); mpz_print(s);
}
```

```
bool CheckSignature(const mpz_t& r, const mpz_t& s,
const EllCurve& curve, const EllPoint& publicKey, mpz_t hash) {
if (!(curve.getECParam() == publicKey.getECParam())) {
printf("(ECDSA)Point given isn't defined over the same curve\n");
return false;
}
if (!publicKey.isOnCurve(publicKey.getCoord())) {
printf("(ECDSA)Point given is not on curve\n");
return false;
}
if (mpz_sgn(r) == 0 || mpz_sgn(s) == 0) {
printf("(ECDSA)Signature given is invalid \
(at least one parameter is null\n");
return false;
}

mpz_t u, v; mpz_inits(u, v, NULL);
mpz_invert(u, s, curve.getGenOrder());
mpz_mul(v, u, r);
mpz_mod(v, v, curve.getGenOrder()); // v ok
mpz_mul(u, u, hash);
mpz_mod(u, u, curve.getGenOrder()); // u ok

EllPoint P(curve.getGen()), tmp(publicKey);
tmp *= v; P *= u; P += tmp;

mpz_sub(u, r, P.getCoord().x); // using u as tmp variable
mpz_mod(u, u, curve.getGenOrder());
if (mpz_sgn(u) == 0) {
printf("(ECDSA)Signature is valid.\n");
mpz_clears(u, v, NULL);
return true;
}
printf("(ECDSA)Signature is invalid.\n");
mpz_clears(u, v, NULL);
return false;
}
```

ECDH.h

```
#pragma once
#include "EllCurve.h"
#include "Constants.h"


/*Elliptic Curve Diffie-Hellman lets two users securely generate a secret key
over an insecure channel. The private key obtained can be used to encrypt/decrypt
a file by both parties with AES for instance.*/
void ECDH(EllCurve& curve) {
// Publicly known parameters
printf("Public parameters used for ECDH : \n");
curve.print("ECDH", true);

//Alice
mpz_t k_alice;
mpz_init(k_alice);
EllPoint P_Alice = curve.getRandomPoint(k_alice);
printf("Alice private key : "); mpz_print(k_alice);

//Bob
mpz_t k_bob;
mpz_init(k_bob);
EllPoint P_Bob = curve.getRandomPoint(k_bob);
printf("Bob private key : "); mpz_print(k_bob);

// Sent over insecure channel
printf("Alice sends to Bob : \n");
P_Alice.print("Alice");
printf("Bob sends to Alice : \n");
P_Bob.print("Bob");

// Private key for Alice, received P_Bob
EllPoint P_Alice_private(P_Bob);
P_Alice_private *= k_alice;

// Private key for Bob, received P_Alice
EllPoint P_Bob_private(P_Alice);
P_Bob_private *= k_bob;

// Shared private key is now the same for both Alice and Bob
if (!(P_Alice_private == P_Bob_private)) {
// Should never happen
printf("Error : private keys don't match !\n");
return;
}

P_Alice.print("Shared private key");

// Now private key can be used for securing communication between
// and bob, for instance using the x coordinate stripped of its first
// 16 bytes to encrypt/ decrypt file using AES/DES etc.

mpz_clears(k_alice, k_bob, NULL);
}
```