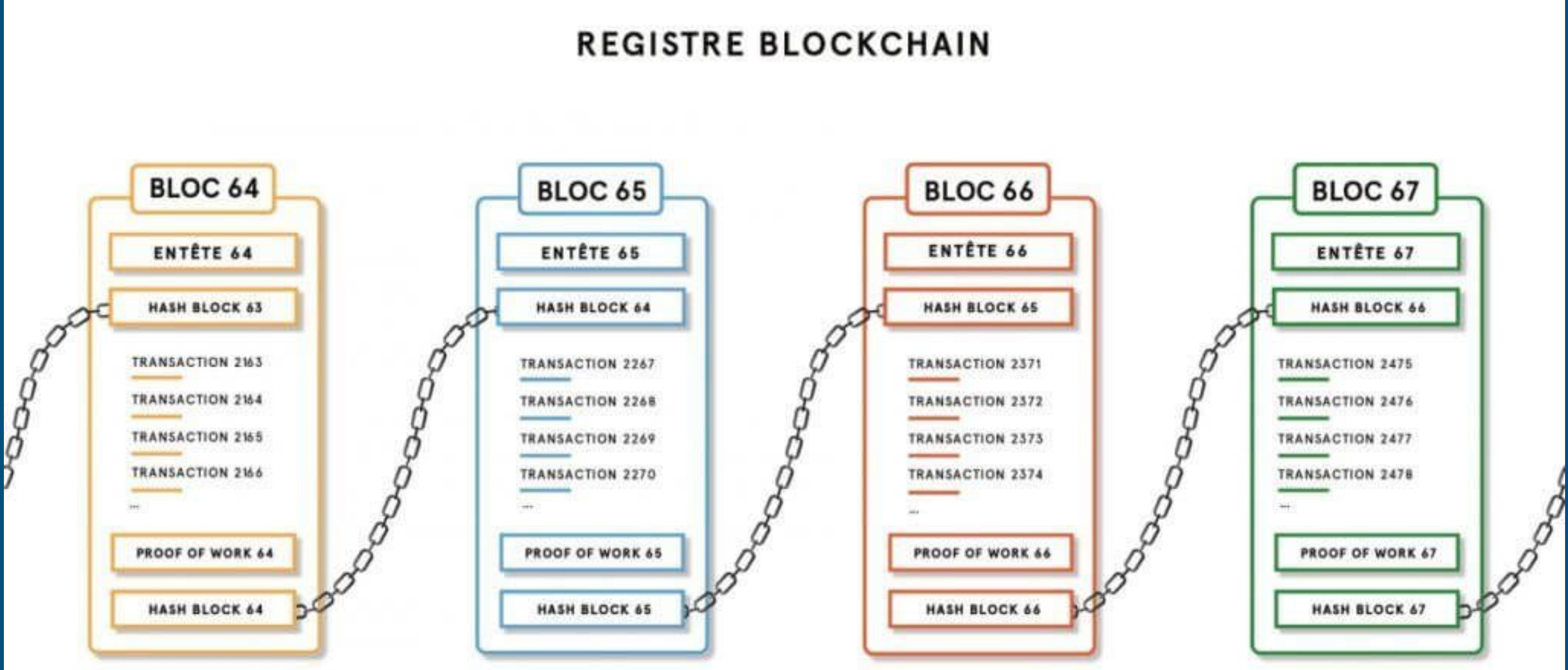
Sécuriser la blockchain grâce à la cryptographie

Plan

Le bitcoin

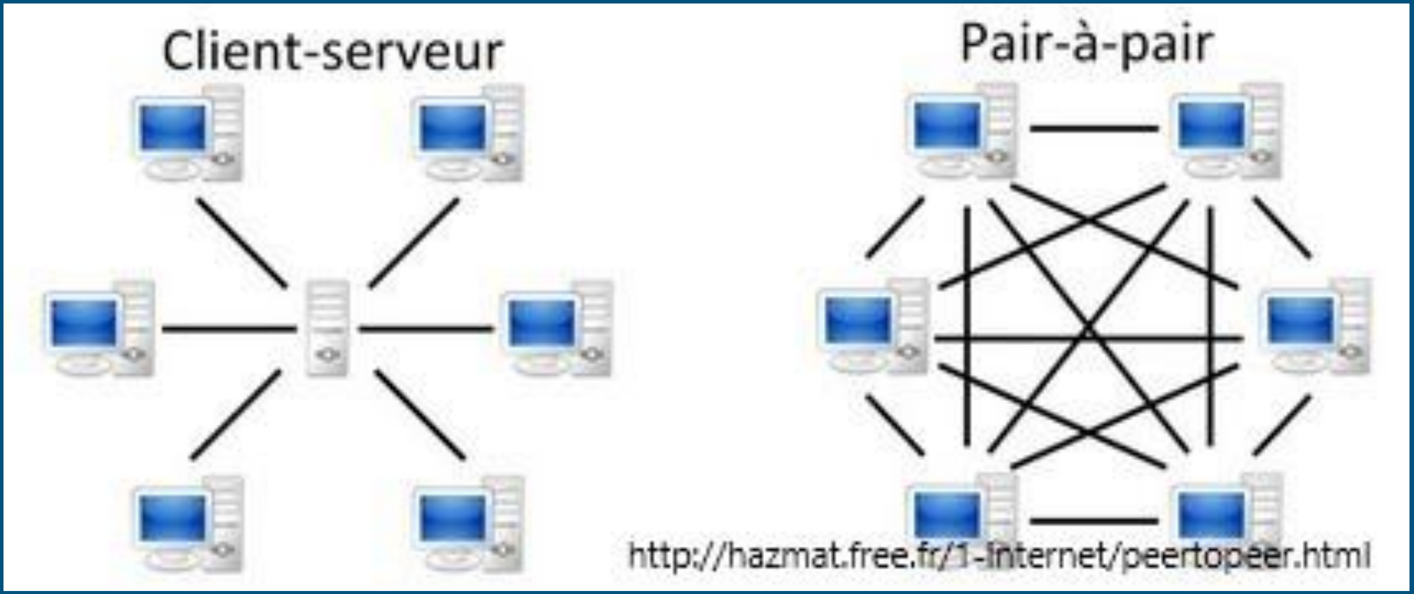
- 320 Go
- 20 millions de bitcoin en circulation
- 800 milliards d’euros
- processus décentralisé

Le principe



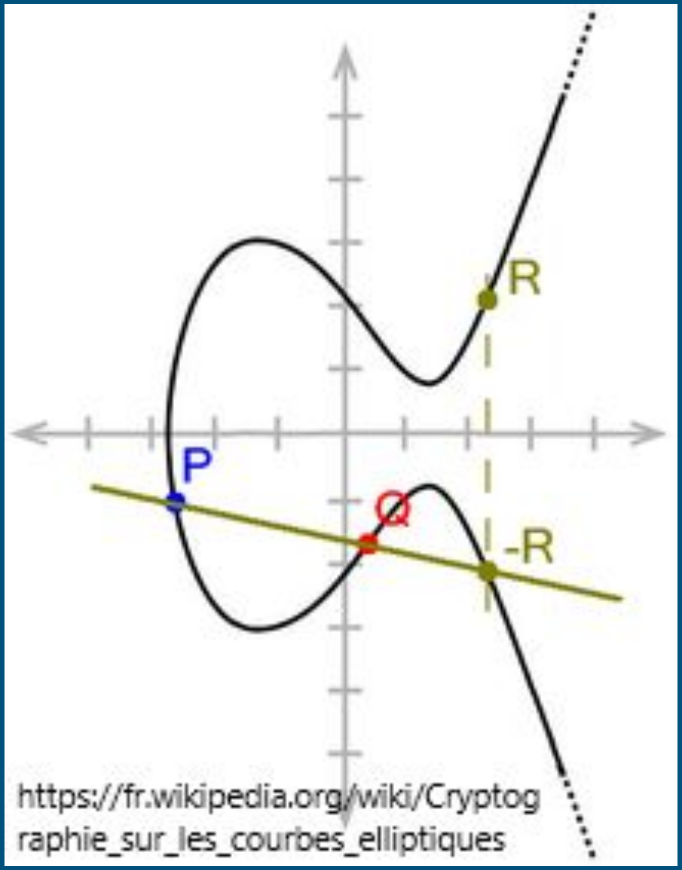
https://coin24.fr/dictionnaire/blockchain/

Création d’une blockchain



Courbe elliptique

$$y^2 = x^3 + ax + b$$



- Corps $\mathbb{Z}/p\mathbb{Z}$ (p premier)
- Addition de points
- O : point à l'infini
- loi de groupe

Test de Miller-Rabin

(algorithme probabiliste s'appuyant sur le petit théorème de Fermat)

$$p \text{ impair et } p = 2^e d + 1$$

$$\text{Pour } a \in [\![1, p-1]\!], \begin{cases} a^d \equiv 1[p] \\ \text{ou } \exists\, i \in [\![1, e-1]\!],\ a^{2^i d} \equiv -1[p] \end{cases}$$

$$p \text{ premier} \implies p \text{ passe le test pour tout } a$$

$$p \text{ non premier} \implies p \text{ passe le test avec au maximum } \frac{1}{4} \text{ des } a \text{ dans } [\![1, p-1]\!]$$

Trouver une courbe sur laquelle travailler

- algorithme de Schoof : calculer le nombre n de points sur la courbe
  logiciel PARI de calcul formel
- groupe cyclique possédant n points (n grands)

- Chercher un point générateur
- Prendre x arbitrairement et calculer $c = x^3 + ax + b$
- Résoudre $y^2 \equiv c[p]$ grâce à l'algorithme de Tonelli-Shanks

ECDSA (Elliptic Curve Digital Signature)

Génération de signature

Données :

- hash du message (h)
- clé privée (c)
- point générateur (G) d’ordre n

Algorithme :

$r = 0,\ s = 0$

Tant que $r = 0$ *et* $s = 0$:

 – *choisir* k *aléatoirement dans* $[\![1, n-1]\!]$

 – $(r,\ \_) = k \times G$

 – $s = k^{-1} \times (h + r \times c)$

renvoyer (r, s)

Vérification de la signature

Données :

- hash du message (h)
- clé publique (Q=c*G)
- point générateur (G) d’ordre n
- signature (r,s)

Algorithme :

$(r',\ \_) = (h \times s^{-1}) * G + (r \times s^{-1}) * Q$

renvoyer $(r = r')$

Problème du logarithme discret

Données :

- P et Q deux points de la courbe elliptique tels que P=k*Q avec k dans [1,n-1]

Objectif :

- Retrouver k

Force brute

stockage : $O(1)$

temps : $O(n)$

Tester tous les points de la courbe jusqu'à trouver k

Algorithme :

$k = 1,\ R = P$

Tant que $R \neq Q$:

$\quad R = R + P$

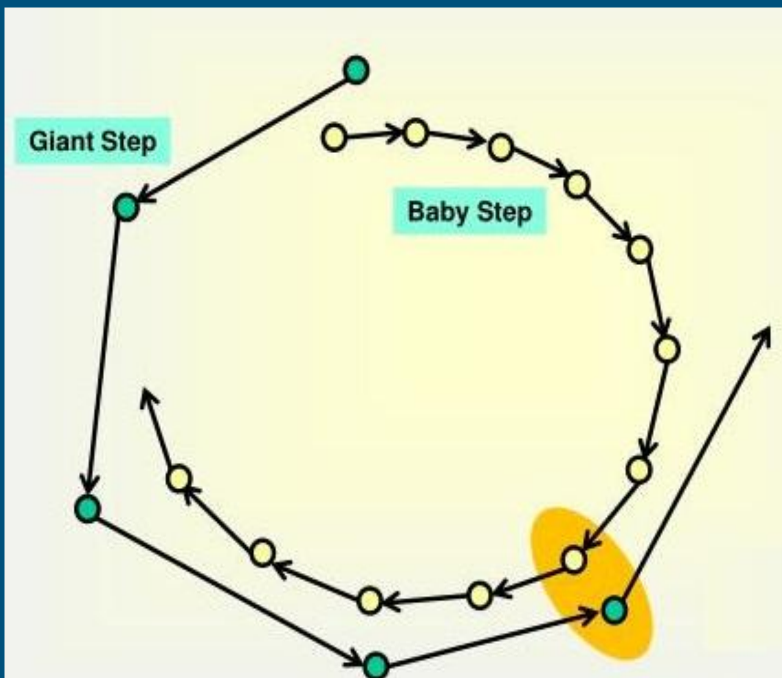
$\quad k = k + 1$

renvoyer k

Baby-step, giant-step

stockage : $O(\sqrt{n})$

temps : $O(\sqrt{n})$



https://www.ratka.net/2018/12/mencari-k-dari-ecc-gfp-menggunakan.html

Algorithme :

$R = O,\ m = \sqrt{n}$

Pour j *allant de* 0 à m :

$\quad table[R] = j$

$\quad R = R + P$

Pour j *allant de* 0 à m :

$\quad$ *Si* $Q - (j \times m) * P$ *est dans table* :

$\quad\quad b = table[Q - (j \times m) * P]$

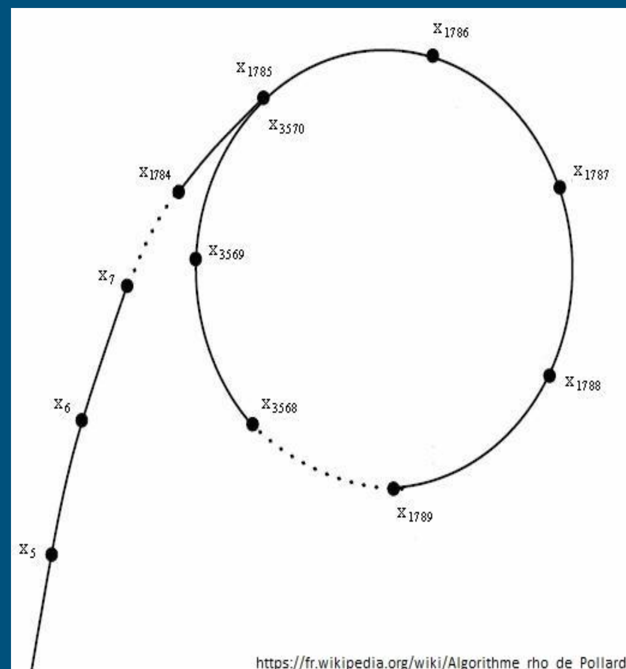
$\quad\quad$ *renvoyer* $j \times m + b$

stockage : $O(1)$

temps : $O(\sqrt{n})$

Rho de Pollard

On cherche (a_1, b_1)
différent de (a_2, b_2)
tels que
$$a_1 P + b_1 Q = a_2 P + b_2 Q$$



Algorithme :

f *une marche aléatoire*

$R_1, a_1, b_1 = f(Q, 0, 1)$ # *tortue*

$R_2, a_2, b_2 = f(R_1, a_1, b_1)$ # *lièvre*

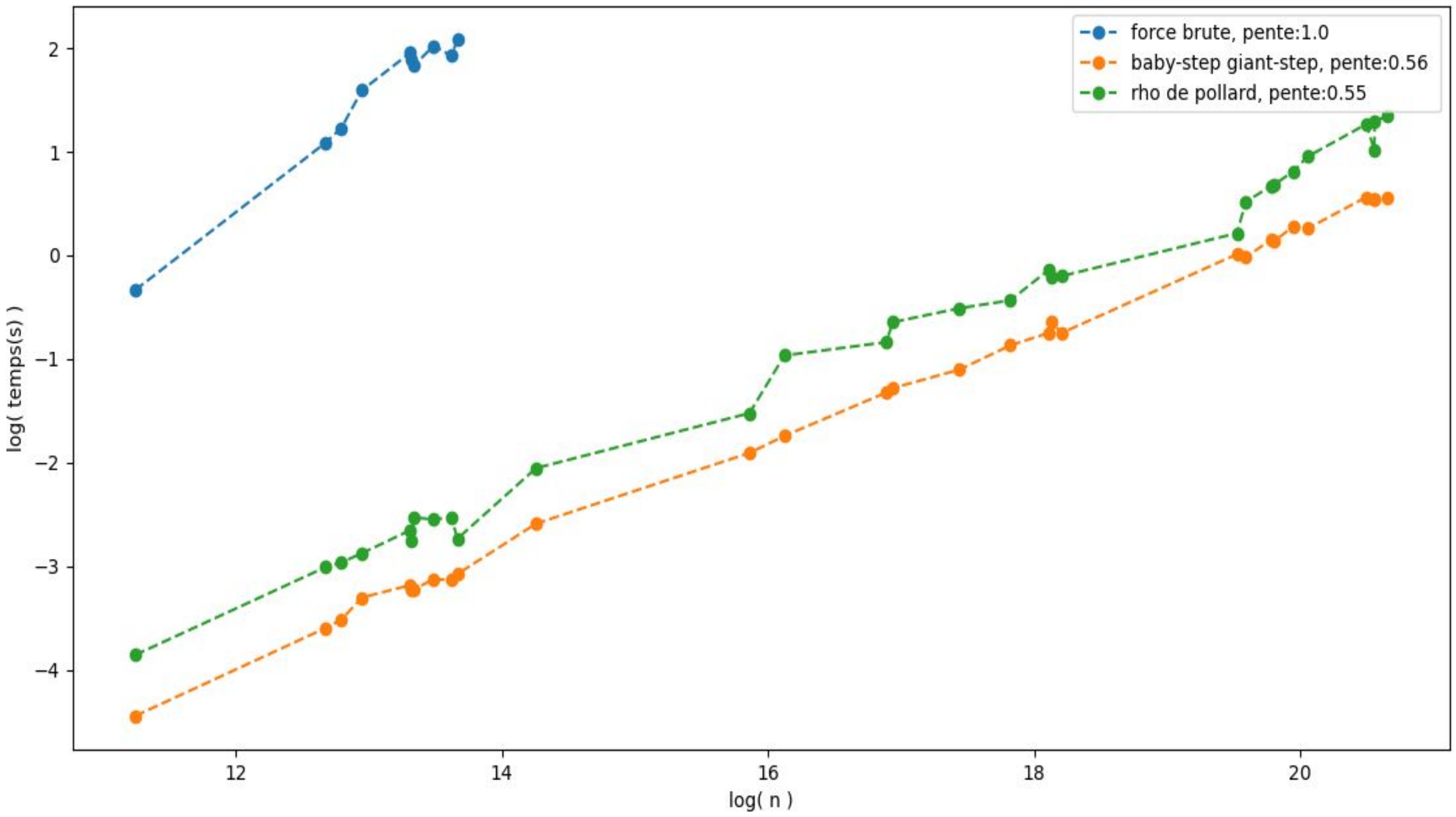
Tant que $R_1 \neq R_2$:

  $R_1, a_1, b_1 = f(R_1, a_1, b_1)$

  $R_2, a_2, b_2 = f^2(R_2, a_2, b_2)$

renvoyer $(a_1 - a_2) \times (b_2 - b_1)^{-1}$

Résultats



force brute, pente:1.0
baby-step giant-step, pente:0.56
rho de pollard, pente:0.55
log(temps(s))
log(n)
2
1
0
−1
−2
−3
−4
12
14
16
18
20

Conclusion

- Transactions sécurisées grâce aux signatures numériques
- Vulnérabilité cryptographique
- Autres problèmes liés à la blockchain

Annexe

```python
from matplotlib.pyplot import *
from random import randint
import numpy as np
import math
from hashlib import sha256

def mod_inverse(x, n):
    """
    Effectue l'algorithme d'Euclide augmentée pour trouver l'inverse modulaire
    """
    modulo = n
    x0, x1, y0, y1 = 1, 0, 0, 1
    while n!=0:
        q, x, n = x // n, n, x % n
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1
    return x0 % modulo

def puissance(a,d,n):
    # calcul p tel que p = a^d mod n
    p=1
    while d > 0:
        if d%2 != 0 :
            p = p*a %n
        a = a*a %n
        d = d//2
    return p

def sqrt(a,p):
    # Resout x^2 = a mod p avec l'algorithme de Tonelli-Shanks
    # 1. Claculer e et n tel que p = 2^e*n + 1
    if puissance(a, (p-1)//2, p) == 1 :
        e=0
        n=p-1
        while True :
            if not n&1 : # si n termine par un 0 (n pair)
                e+=1
                n >>= 1
            else :
                break
        # 2. Trouver u non résidu quadratique mod p
        # si u^k = 1 mod p alors résidu quadratique
        l = 1
        k = (p-1)//2
```

```python
        l = 1
        k = (p-1)//2
        while l==1 :
            u = randint(0, p-1)
            l = puissance(u, k, p)

        # 3. Calcul de x
        k = e
        z = puissance(u,n,p)
        x = puissance(a, (n+1)//2,p)
        b = puissance(a,n,p)

        while b != 1 :
            # trouver le plus petit m tel que b^2^m = 1 mod p
            m=0
            r=b
            while r != 1 :
                r = (r*r)% p
                m+=1
            t = puissance(z,puissance(2,k-m-1,p), p)
            z = (t*t)% p
            b = (b*z)% p
            x = (x*t)% p
            k = m
        return x
    else :
        raise Exception("Pas trouve pour " + str(a))

class Courbe_elliptique(object):
    """
    y^2 = x^3 + ax + b mod p
    p nombre premier
    """
    def __init__(self, a, b, p):
        self.a = a
        self.b = b
        self.p = p

    def has_point(self, x, y): # dit si le pt appartient à la courbe
        return (y ** 2) % self.p == (x ** 3 + self.a * x + self.b) % self.p

    def __str__(self):
        return f'y^2 = x^3 + {self.a}x + {self.b} mod {self.p}'
```

```python
        return f'y^2 = x^3 + {self.a}x + {self.b} mod {self.p}'

    def afficher(self):
        x = []
        y = []
        for i in range(0,self.p):
            try :
                P1, P2 = self.gen(i)
                x.append(P1.x)
                x.append(P2.x)
                y.append(P1.y)
                y.append(P2.y)
            except :
                pass
        plot(x,y, ".")
        xlim(0, self.p)
        ylim(0, self.p)
        show()

    def gen(self, x) :
        # Génère un point à partir de l'abscisse donné
        x = x % self.p
        y2 = (x ** 3 + self.a * x + self.b) % self.p
        try :
            y = sqrt(y2, self.p)

            return Point(self, x, y), Point(self, x, self.p - y)
        except Exception as e :
            raise e

    def intervalle(self):
        # Donne selon le théorème de Hasse un encadrement du cardinal
        rp = math.ceil(math.sqrt(self.p))
        mini = self.p+1-2*rp
        maxi = self.p+1+2*rp
        return mini, maxi

    def cardinal(self):
        # Calcul le nombre de points appartenant à la courbe elliptique
        mini, maxi = self.intervalle()
        nb = maxi - mini
        possible = []
        i=0
        while i < 20 :
```

```python

    def cardinal(self):
        # Calcul le nombre de points appartenant à la courbe elliptique
        mini, maxi = self.intervalle()
        nb = maxi - mini
        possible = []
        i=0
        while i < 20 :
            try :
                P,_ = self.gen(i)

                m = P.order() # m divise N
                if m != 0 and nb//m < 10000 : # Si m est intéressant
                    a = math.ceil(mini/m) # premier entier dans l'intervalle
                    b = math.floor(maxi/m)
                    if possible != [] :
                        possible = intersection(possible, np.arange(a*m,b*m+1,m, dtype=int))
                    else :
                        possible = np.arange(a*m,b*m+1,m, dtype=int)
                    if len(possible)==1:
                        return possible[0]
            except :
                pass
            finally :
                i+=1
        return 4

    def fast_cardinal(self):
        # On veut trouver un cardinal premier
        mini, maxi = self.intervalle()
        i=1
        while i< 20 :
            try :
                P,_ = self.gen(i)
                m = P.order() # m divise N
                if m != 0 :
                    if m > mini and m < maxi :
                        return m
                    else :
                        return 4
            except :
                pass
            finally :
                i+=1
```

```python
class Point(object):
    """
    Definie un point d'une courbe elliptique
    """
    def __init__(self, courbe, x, y):
        self.courbe = courbe
        self.x = x % courbe.p
        self.y = y % courbe.p
        if not isinstance(self, Inf) and not self.courbe.has_point(x, y):
            raise ValueError(f"{self} n\'est pas sur la courbe {self.courbe}")

    def __str__(self):
        if isinstance(self, Inf) :
            return 'Point à l\'infini'
        else :
            return '({}, {})'.format(self.x, self.y)
    """
    def __getitem__(self, index):
        return [self.x, self.y][index]
    """
    def __eq__(self, Q):
        return (self.courbe, self.x, self.y) == (Q.courbe, Q.x, Q.y)

    def __ge__(self,Q):
        if isinstance(Q, Inf) :
            return True
        elif isinstance(self, Inf) :
            return False
        else :
            return  (self.x > Q.x) or (self.x==Q.x and self.y>Q.y)

    def __lt__(self,Q):
        if isinstance(Q, Inf) :
            return False
        elif isinstance(self, Inf) :
            return True
        else :
            return  (self.x < Q.x) or (self.x==Q.x and self.y<Q.y)

    def __neg__(self):
        return Point(self.courbe, self.x, -self.y)

    def __add__(self, Q):
```

```python
        assert self.courbe == Q.courbe # Check si les pts sont sur la meme courbe

        # 0 + P = P
        if isinstance(Q, Inf):
            return self

        xp, yp, xq, yq = self.x, self.y, Q.x, Q.y
        m = None

        # P == Q
        if self == Q:
            if self.y == 0:
                R = Inf(self.courbe)
            else:
                m = ((3 * xp * xp + self.courbe.a) * mod_inverse(2 * yp % self.courbe.p, self.courbe.p)) % self.courbe.p

        # ligne vertical
        elif xp == xq:
            R = Inf(self.courbe)

        # En general
        else:
            m = ((yq - yp) * mod_inverse(xq - xp % self.courbe.p, self.courbe.p)) % self.courbe.p

        if m is not None:
            xr = (m ** 2 - xp - xq) % self.courbe.p
            yr = (m * (xp - xr) - yp) % self.courbe.p
            R = Point(self.courbe, xr, yr)

        return R

    def __mul__(self, n):
        """
        Ne se fait pas en O(n) car sinon cela ne sert à rien
        Se fait en O(log(n)) en decomposant n en base 2
        le problème du log discret repose sur cette difference
        """
        assert isinstance(n, int)

        if n == 0:
            return Inf(self.courbe)
```

```python
        """
        Ne se fait pas en O(n) car sinon cela ne sert à rien
        Se fait en O(log(n)) en decomposant n en base 2
        le problème du log discret repose sur cette difference
        """
        assert isinstance(n, int)

        if n == 0:
            return Inf(self.courbe)
        else:
            Q = self
            R = Inf(self.courbe)
            i = 1
            while i <= n: # Pour optimiser l'addition -> decompose en base 2
                if n & i == i:
                    R = R + Q
                Q = Q + Q
                i = i << 1 # multiplie par 2
        return R

    def __rmul__(self, n):
        return self * n

    def order(self):
        # Algorithme baby-giant step
        Q = Inf(P.courbe)
        pt = Inf(P.courbe)
        m = math.ceil(math.sqrt(P.courbe.p +1 + 2*math.sqrt(P.courbe.p)))
        table_hachage = {}
        for j in range(m+1):
            table_hachage[hash(str(pt))] = (j,pt)
            pt = pt + P
        for j in range(0, m+1):
            Q2 = Q + (-(j*m*P))
            h = hash(str(Q2))
            if h in table_hachage :
                c = table_hachage[h][0]
                return j*m + c
        raise Exception("Pas reussi")
```

```python
class Inf(Point):
    """
    Point infini
    """
    def __init__(self, courbe):
        Point.__init__(self, courbe, 0,0)

    def __eq__(self, Q):
        return isinstance(Q, Inf)

    def __neg__(self):
        """-0 = 0"""
        return self

    def __add__(self, Q):
        """P + 0 = P"""
        return Q

class ECDSA(object):
    def __init__(self, courbe, generator, order):
        self.courbe = courbe
        self.G = generator # point de la courbe choisi
        self.n = order # tel que G.n = 0 (n premier)

    def sign(self, msghash, prK):
        r=0
        s=0
        while r==0 or s==0 :
            k = randint(1, self.n - 1) # on choisit k entre 1 - n-1 (important car si on prend le même ça ne marche plus ex Sony PlayStation
            r = (k * self.G).x # point Q
            s = (mod_inverse(k, self.n) * (msghash + r * prK)) % self.n
        return r, s

    def verify(self, msghash, r, s, puK):
        assert 0<r and r<self.n and 0<s and s<self.n
        w = mod_inverse(s, self.n)
        c = msghash*w %self.n
        d = r*w %self.n
        X = c*self.G + d*puK
        print(r == X.x)
```

```python
def rabin(n,k):
    # Test de primalité probabiliste (échec : 4**(-k))
    #https://fr.wikipedia.org/wiki/Test_de_primalit%C3%A9_de_Miller-Rabin#:~:text=Le%20th%C3%A9or%C3%A8me%20de%20Rabin%20permet,est%20inf%C3
    # calcul de s et d tel que n-1 = 2**s *d
    s=0
    d=n-1
    while True :
        if not d&1 : # si d termine par un 0 (d pair)
            s+=1
            d >>= 1
        else :
            break

    def temoin_miller(n,a):
        x = puissance(a,d,n)
        if x==1 or x==n-1 :
            return False

        for i in range(s):
            x = x*x % n
            if x == n-1 :
                return False
        return True

    for i in range(k): # On effectue k fois le test de miller
        a = randint(2, n-2)
        if temoin_miller(n, a) :
            return False
    return True

def card(c, methode):
    if methode == 1 :
        pari('E = ellinit(['+str(c.a)+','+str(c.b)+'],{D='+str(c.p)+'})')
        return int(pari('ellcard(E)'))
    else :
        return c.fast_cardinal()
```

```python
    Nothing-up-my-sleeve number
    """
    found = False
    while not found :
        a = randint(0,p-1)
        b = randint(0, p-1)
        if 4*int((a**3)) + 27*int((b**2)) :
            return Courbe_elliptique(a, b, p)

def search_curve(p, maxi):
    i=0
    while i<maxi :
        c = random_curve(p)
        N = card(c,1)
        if N%2 and rabin(N, 3): # Si N est premier
            j=0
            while j<50 :
                try :
                    P,_ = c.gen(j)
                    print("Courbe "+str(c)+ " de cardinal "+str(N)+" trouvée"+" Point generateur : "+str(P))
                    return c,P,N
                except :
                    j+=1
            break
        i+=1
    raise Exception("pas trouve")
```

```python
def hash(mot):
    return sha256(mot.encode()).hexdigest()


def force_brute(P, Q):
    i=1
    P2 = P
    while P2 != Q :
        P2 += P
        i += 1
    return i


def baby_giant_step(P, Q):
    pt = Inf(P.courbe)
    m = math.ceil(math.sqrt(P.courbe.p +1 + 2*math.sqrt(P.courbe.p)))
    table_hachage = {}

    for j in range(m+1):
        table_hachage[hash(str(pt))] = (j,pt)
        pt = pt + P

    P2 = -(m*P)
    Q2 = Q +(-P2)
    for j in range(0, m+1):
        Q2 += P2
        #Q2 = Q + (-(j*m*P)) # moins efficace
        h = hash(str(Q2))
        if h in table_hachage :
            c = table_hachage[h][0]
            return j*m+c

    raise Exception("Pas reussi")
```

```python
def pollard(P,Q, n):
    """
    Technique du rho de pollard O(sqrt(n)) asymptote pour le temps et O(1) pour l'espace
    Principe : Trouver a1,b1,a2,b2 tel que a1P + b1Q = a2P + b2Q
    pour Q = xP, on a donc x = (a1-a2)*(b2-b1)^-1
    """
    def f(R, a, b): # marche aléatoire
        if R.x < n//3 :
            return R+Q, a, (b+1)%n
        elif R.x > 2*n//3 :
            return R+P, (a+1)%n, b
        else :
            return R+R, (a+a)%n, (b+b)%n

    R1, a1, b1 = f(Q, 0, 1) # R1 = a1P + b1Q
    R2, a2, b2 = f(R1, a1, b1)

    i = 1
    while R1 != R2 and i < 100000 :
        R1, a1, b1 = f(R1, a1, b1)
        R2, a2, b2 = f(R2, a2, b2) # on le fait 2 fois
        R2, a2, b2 = f(R2, a2, b2)
        i += 1

    x = ((a1-a2)*mod_inverse(b2-b1, n))% n # Q = xP
    return x,i

```