

Fiabilité des méthodes de chiffrement pour la sécurité des données.

Les principes de base du chiffrement provenaient de notions mathématiques abstraites et élégantes qui ont suscité notre intérêt.

Internet est un immense réseau où les utilisateurs sont interconnectés échangeant continuellement des données. Les informations personnelles y transitent chaque jour cependant, ces informations peuvent être sensibles. Pour pallier aux vols d'informations, il existe des méthodes de chiffrement qui ont pour but de rendre inintelligible les données sensibles.

Ce TIPE fait l'objet d'un travail de groupe.

Liste des membres du groupe :

- BERTIN Alexandre
- DERANLOT Sacha

Positionnement thématique (phase 2)

MATHÉMATIQUES (Algèbre), MATHÉMATIQUES (Mathématiques Appliquées), INFORMATIQUE (Informatique Théorique).

Mots-clés (phase 2)

Mots-Clés (en français)	Mots-Clés (en anglais)
Cryptographie	Cryptography
Courbe elliptique	Elliptic curve
RSA	RSA
Prime number	Nombre premier
Clé Publique/Privée	Public/Private Key

Bibliographie commentée

RSA et l'ECDH sont des méthodes de chiffrements asymétriques, certifiés par la NSA. En effet, RSA (brevetée en 1983) s'appuie sur les nombres premiers et de la difficulté à factoriser un nombre composé de deux nombres premiers élevées [1]. Le RSA est un chiffrement asymétrique qui a pour particularité de différencier la clef publique de la clef privée, Alice envoie sa clé publique à Bob pour que ce dernier chiffrer son message avec cette clef. Bob envoie à Alice le message crypté et Alice peut le décoder grâce à sa clef privée qu'elle n'aura jamais divulguée. Le message est donc inintelligible lors du transport, cependant la clef publique est envoyée de façon clair. Même si ce chiffrement est considéré comme très sûr, celui-ci présente des limites d'une part par la taille des clefs et aussi par le stockage de ces dernières, ainsi que les backdoors (donnant accès aux clés) éventuelles. On remarque ainsi, qu'avec du temps, n'importe quel système de chiffrement est crackable [2], car il suffit de tester toutes les combinaisons possibles (brute force) afin de trouver la

bonne clef privée.

Pour le RSA, la taille des clés est donc proportionnelle au temps qu'il sera nécessaire pour la trouver. L'autre facteur cité est celui du stockage de ces clefs, si ces dernières ne sont pas en lieux sûrs, elles seront piratables et récupérables [3]. Les limites du RSA sont notamment dûs aux nombreux calculs de puissance et modulo de très grands nombres qui sont très gourmands en ressources.

Contrairement au RSA, le chiffrement par courbe elliptique est performant sur d'autres critères, notamment l'authentification d'un utilisateur certifié et le chiffrement de contenu. Les courbes elliptiques en cryptographie s'appuient sur une construction géométrique. Elle-même s'appuyant sur la théorie des groupes. [4]. Plus récemment, les courbes elliptiques en cryptographie sont fortement utilisées de deux façons différentes: l'ECDH et l'ECDSA, le premier étant pour crypter un contenu et l'autre étant une simple certification par signature. L'ECDSA est couramment utilisé dans les cryptomonnaies tel que le bitcoin [5]. Les transactions en attente de certification sont notées dans le block-chain. L'authentification se fait par les courbes elliptiques. Bob, lors de la création de son compte bancaire Bitcoin a obtenu aléatoirement un certain N et l'administrateur a retenu le point N^*G obtenu par "addition" de N fois le point générateur (G) que l'administrateur aura généré aléatoirement pour Bob. Pour que l'administrateur reconnaise que Bob est bien le véritable Bob, il va lui demander de calculer N^*G en lui envoyant G. Bob envoie donc son N^*G à l'administrateur et si la valeur retenue lors de la création du compte de Bob coïncide avec celle que Bob lui a envoyé, c'est que Bob est le véritable Bob car il est l'unique personne qui connaît la valeur de N. La transaction est donc authentifiée. Cela relève d'un problème de logarithme discret [6], où trouver la valeur de l'exposant (c'est-à-dire N) est impossible (du moins théoriquement très difficile) à partir de N^*G . [7]

Problématique retenue

Dans quelles mesures le problème du logarithme discret sur les courbes elliptiques est-il plus pertinent que celui du RSA ?

Objectifs du TIPE

- Développement d'un algorithme cryptant et décryptant utilisant la méthode R.S.A.
- Développement d'un algorithme cryptant et décryptant utilisant les courbes elliptiques.
- Vérifier les deux méthodes sur des critères de fiabilité, rapidité et les comparer.
- Comparer théoriquement le logarithme discret du RSA au logarithme discret des courbes elliptiques.
- Obtenir des réponses de professionnels sur l'avenir du chiffrement avec l'arrivée des ordinateurs quantiques

Références bibliographiques (phase 2)

- [1] FRANÇOIS MAUREL : Sur l'algorithme RSA :
<http://culturemath.ens.fr/math/pdf/nombres/RSA.pdf>
- [2] RICHARD CLAYTON : Brute force attacks on cryptographic keys :
<https://www.cl.cam.ac.uk/~rnc1/brute.html>
- [3] DON PURNHAGEN : Encryption Key Management and Why You Should Care :
<https://www.iri.com/blog/data-protection/encryption-key-management-and-why-you-should-care/>
- [4] KOBILITZ : A course in number theory and cryptography : 2ed., GTM 114, Springer, 1994
- [5] PIERRE NOIZET : ECDSA, technologie clé de bitcoin : <http://e-ducal.fr/links/ecdsa/>
- [6] CHRISTOPHE DELAUNAY : Le "problème" du logarithme discret en cryptographie :
<http://images.math.cnrs.fr/Le-probleme-du-logarithme-discret-en-cryptographie.html>
- [7] CÉCILE GONÇALVES : Cours sur les courbes elliptiques :
http://www.lix.polytechnique.fr/~goncalves/Downloads/Cours2_Courbes_elliptiques.pdf



Fiabilité des méthodes de **chiffrement asymétrique**

Pour la sécurité des transports de
données

DERANLOT Sacha (4603)
BERTIN Alexandre (4976)
BERTRAND Simon (12992)

Sommaire

Introduction	01	Courbes Elliptiques	05	Etude probabiliste Brute force naïf	09
Objectifs des travaux	02	Procédé Diffie-Hellman	06	Algorithmes Brute force réfléchi	10
Principe des clés asymétriques	03	Mise en évidence par des images	07	Conclusion	11
RSA	04	Résultats des tests	08	Tous les programmes	12

Introduction



Problématique

Dans quelle mesure le chiffrement par courbe elliptique est-il plus pertinent que celui du RSA ?

Objectifs des travaux

En réponse à la problématique



Développer des algorithmes efficaces et optimisés

Pour une compréhension approfondie des méthodes de chiffrement RSA et ECDH



Tester sur différents critères les algorithmes

La rapidité de génération de clé ainsi que la rapidité de chiffrement - déchiffrement des messages seront comparées



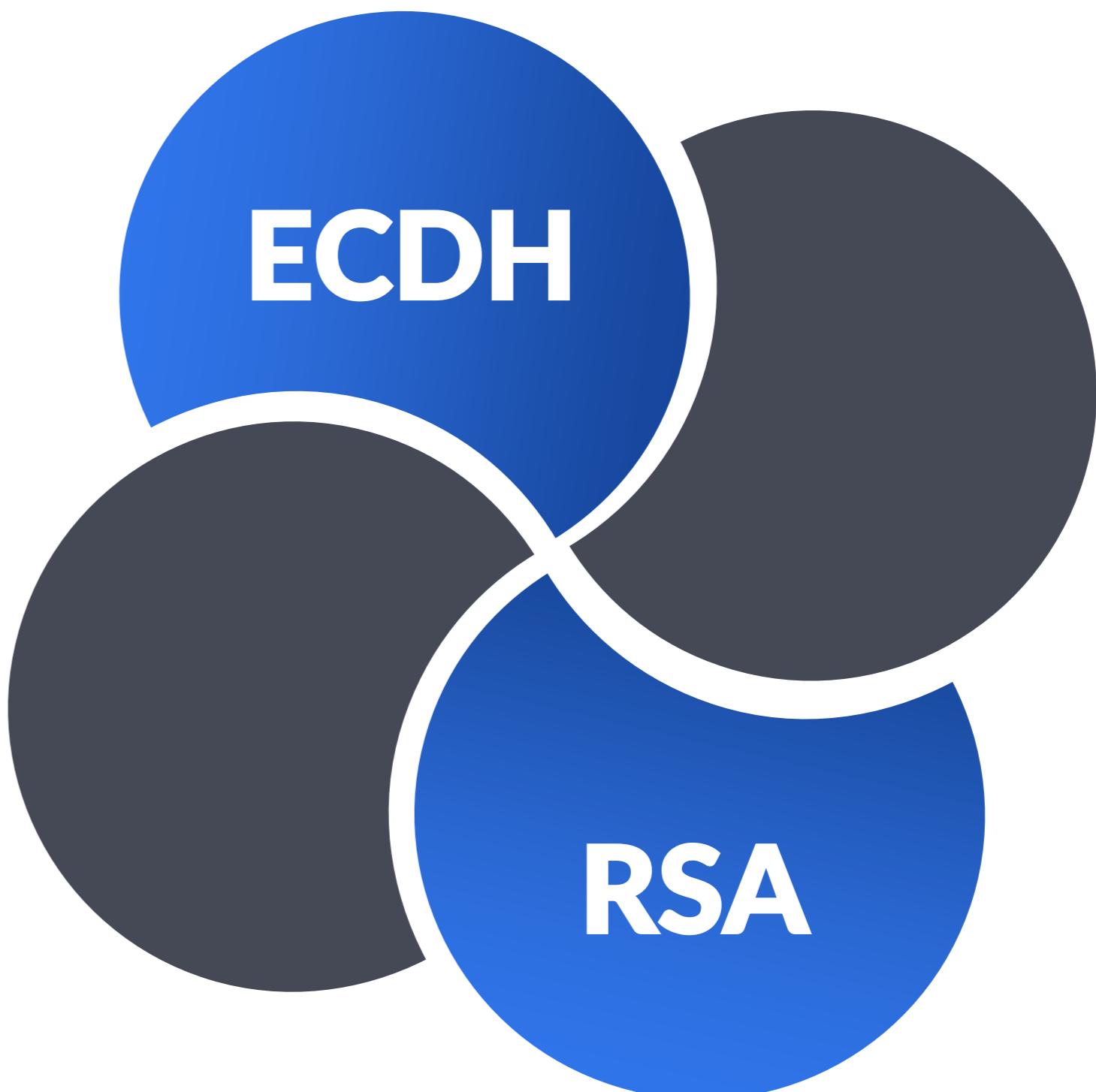
Statistique - Probabilités

Etude théorique du nombre de possibilité pour une attaque de type brute force naïve contre le RSA et l'ECDH



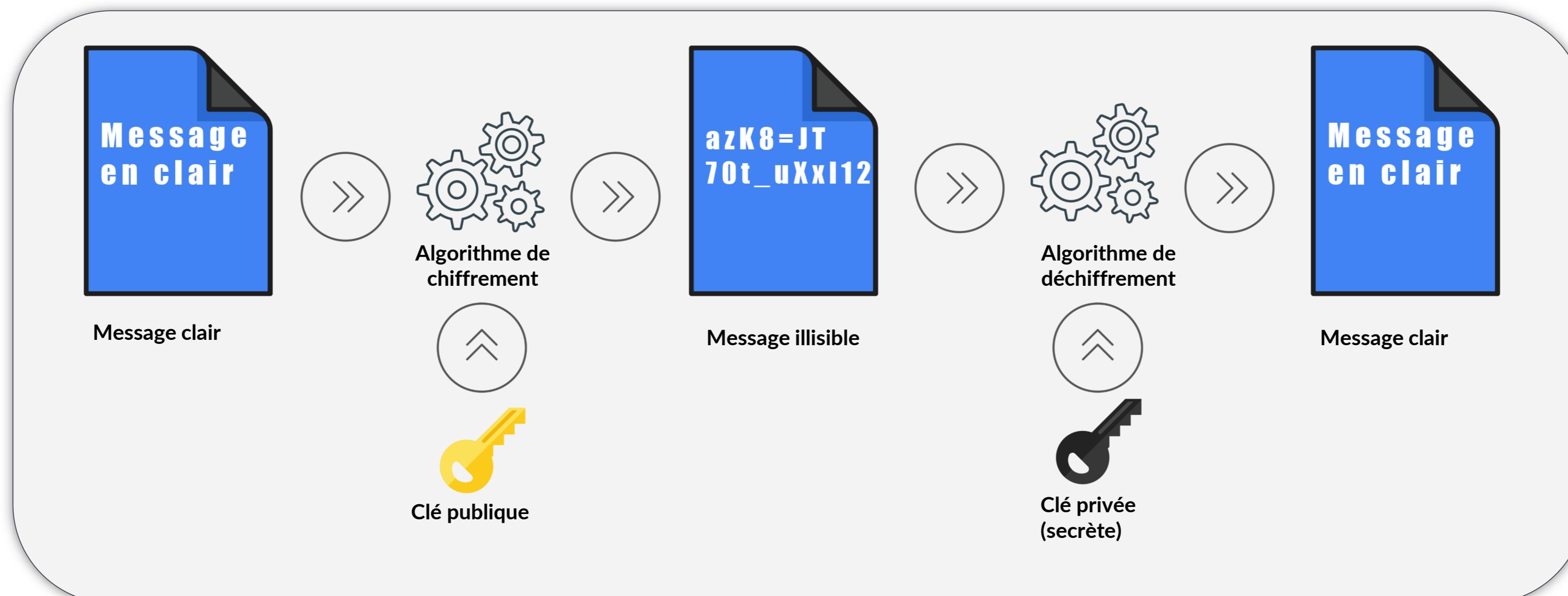
Failles

Développement des possibles solutions au logarithme discret des courbes elliptiques



Principe des clés asymétriques

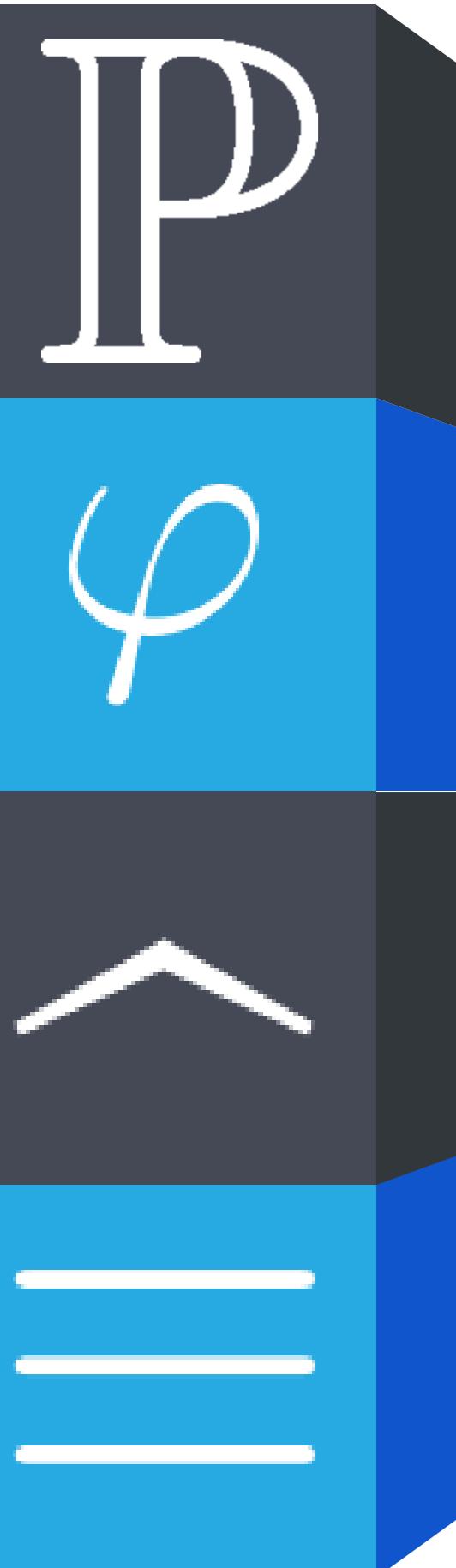
La combinaison clé publique - clé privée



Source des icônes individuels : iconsfinder.com

Méthodes de chiffrement

RSA



On choisit deux nombres premiers P et Q

On calcule

$$N = P \times Q$$

et

$$\varphi(N) = (P - 1) \times (Q - 1)$$

On cherche e tel que

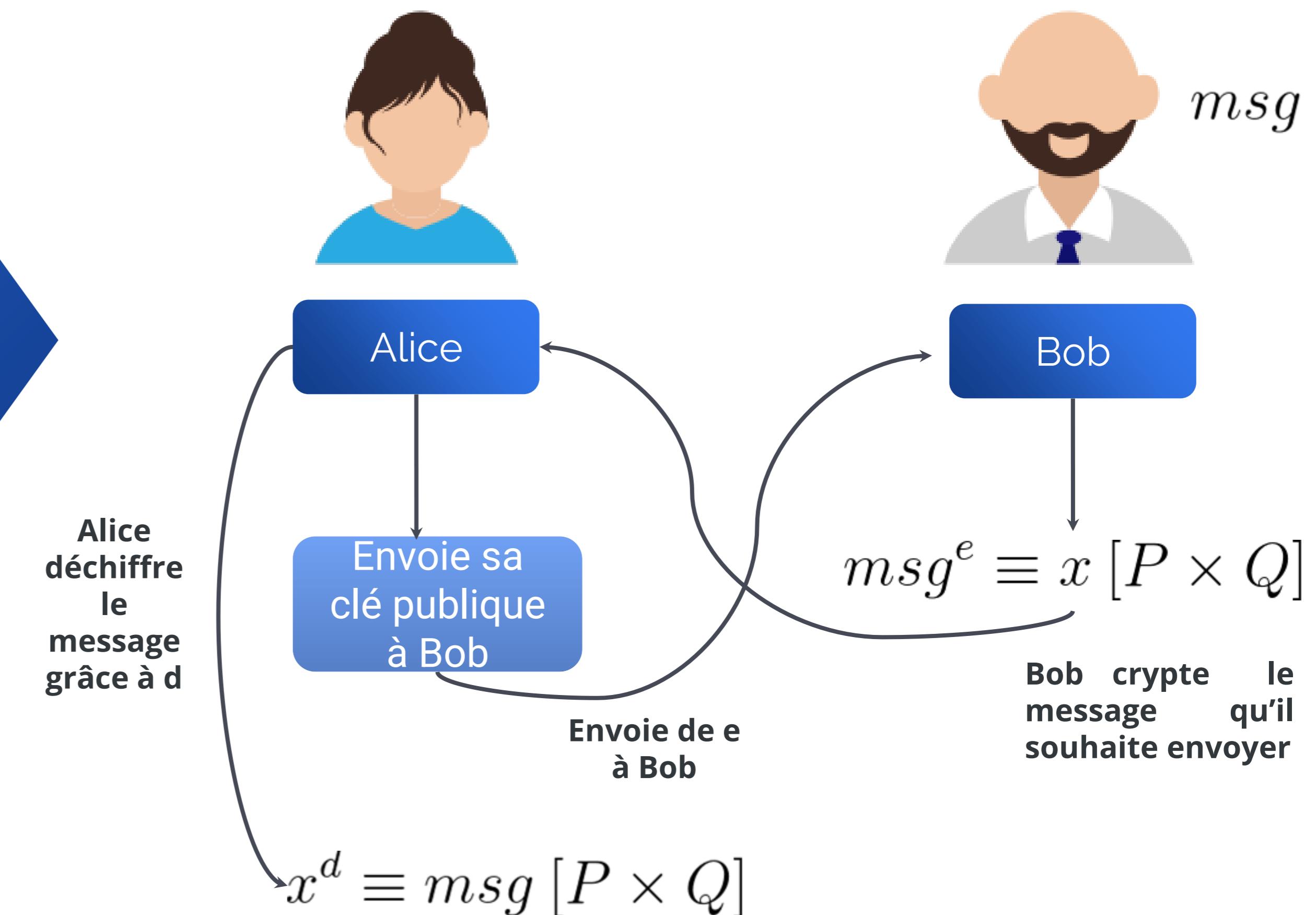
$$\text{PGCD}(e, \varphi(n)) = 1$$

On cherche d tel que

$$d \times e \equiv 1 \pmod{\varphi(N)}$$

Clef publique : (e,N) - Clef privée : (d,N)

Génération des clés publiques et privées & formule chiffrement / déchiffrement



$$(msgCrypt) = (msgDecrypt)^e \pmod{P \times Q}$$

$$(msgDecrypt) = (msgCrypt)^d \pmod{P \times Q}$$

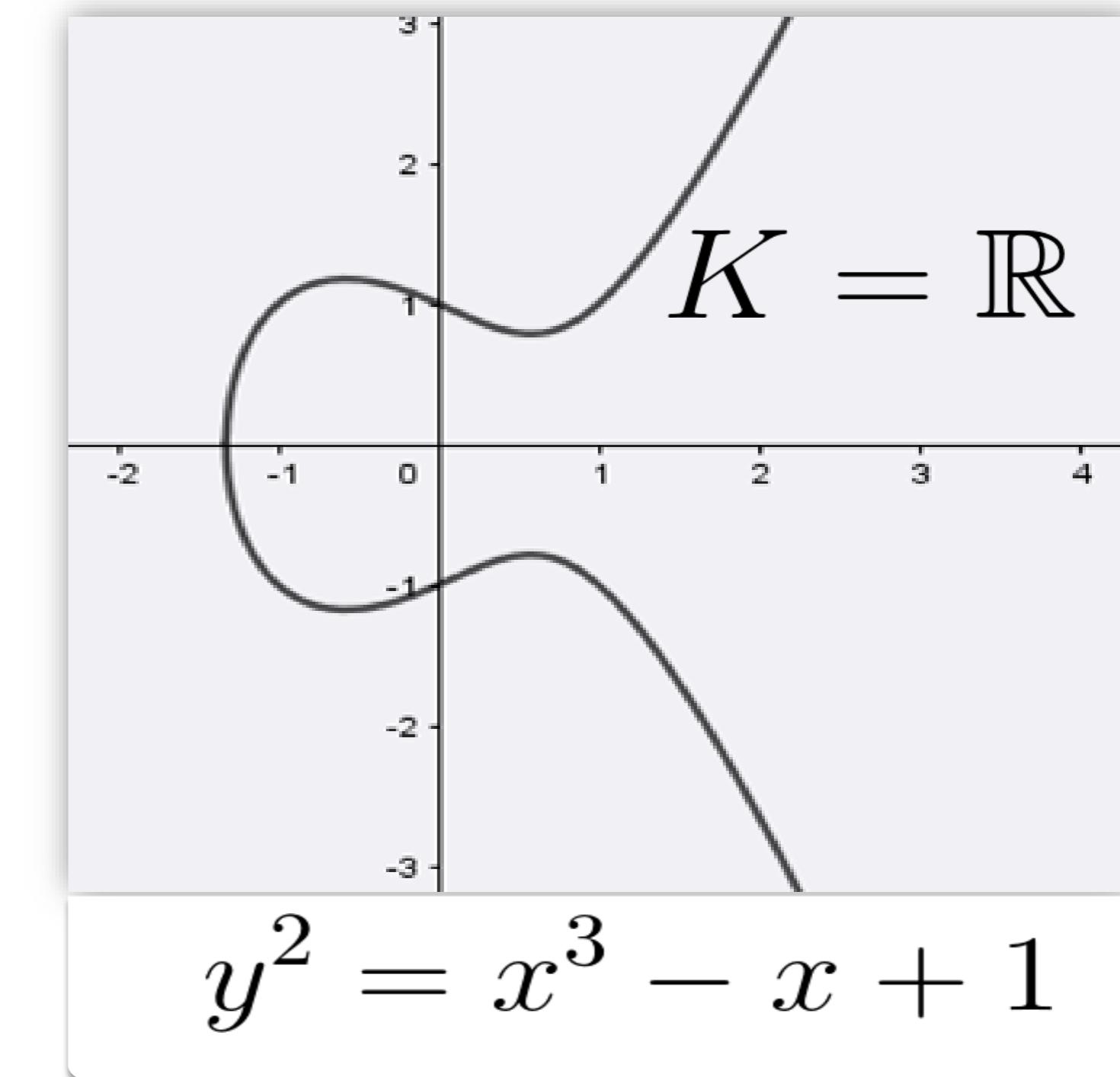
Courbe elliptique et objets mathématiques

 K Δ

$$E = \{(x, y) \in K^2 \mid y^2 = x^3 + ax + b\}$$

$$\Delta = -16(4a^3 + 27b^2)$$

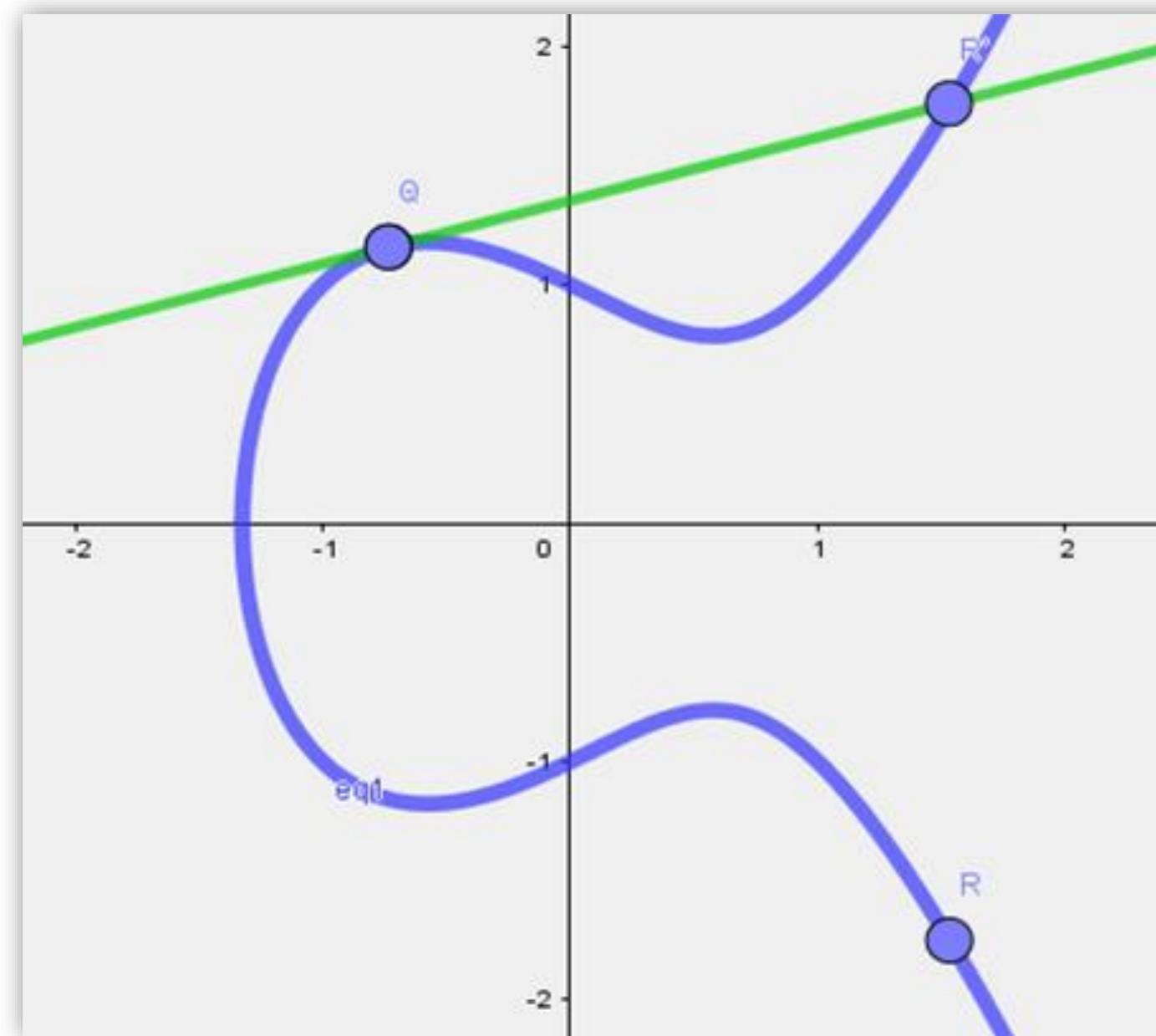
Soit K un corps, $(a, b) \in K^2$ tel que $\Delta \neq 0$



Définition de la loi de groupe

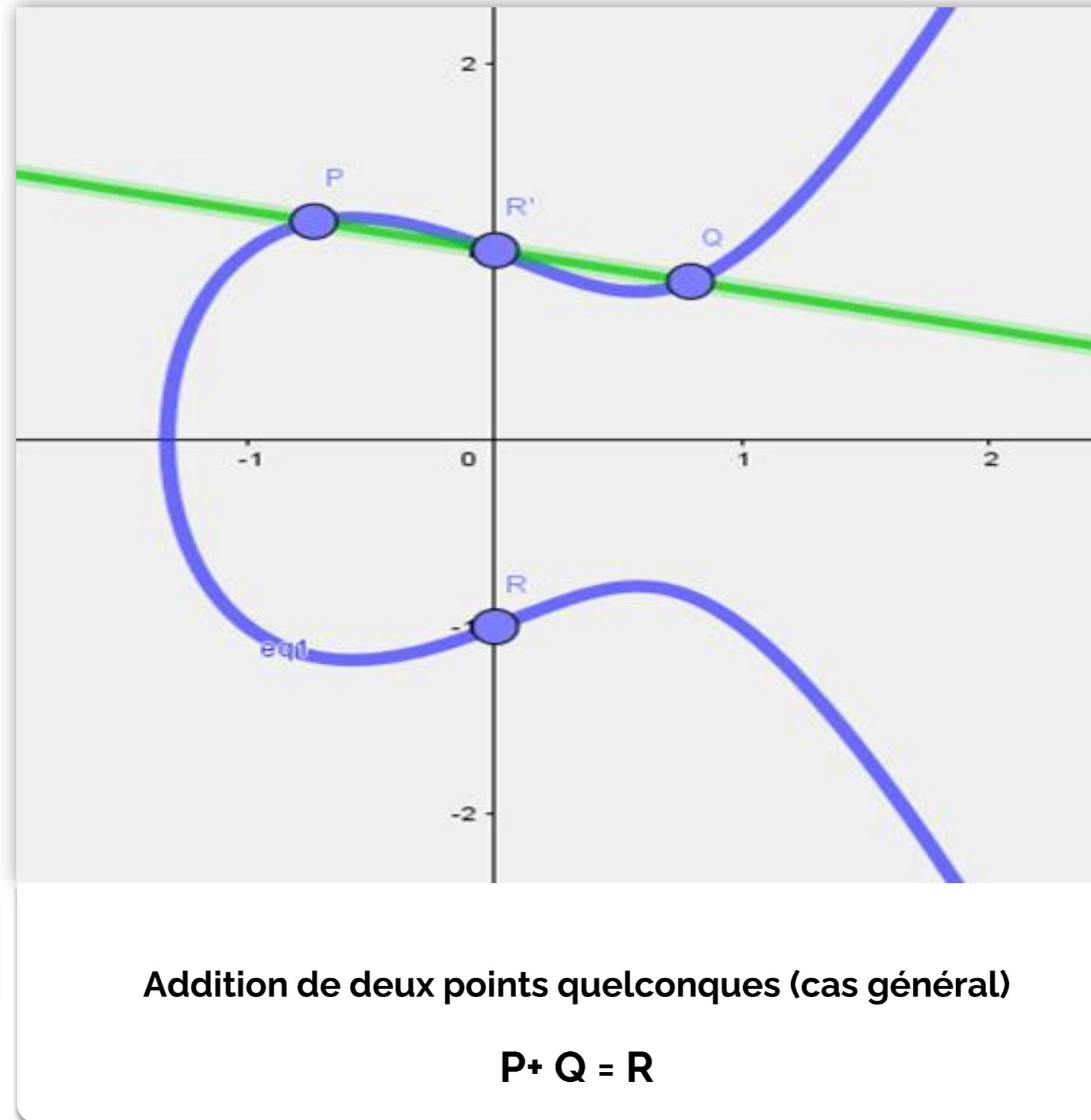
Additions de points sur la courbe

Soit $(P, Q) \in E^2$



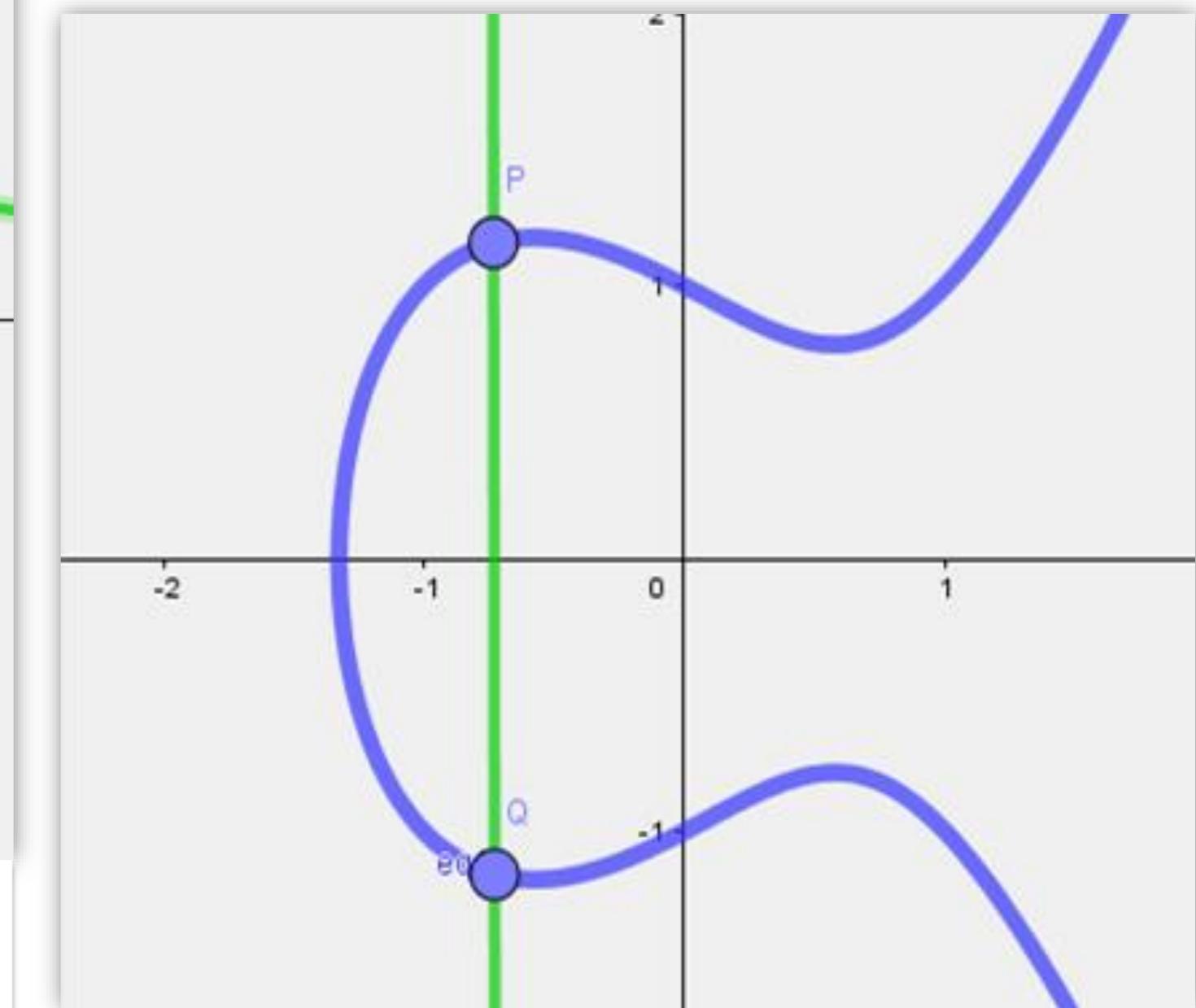
Addition de deux points possédant les mêmes coordonnées

$$Q + Q = R' (=2P)$$



Addition de deux points opposés pour obtention du point l'infini. (Ici, $Q = -P$)

$$P + (-P) = 0$$

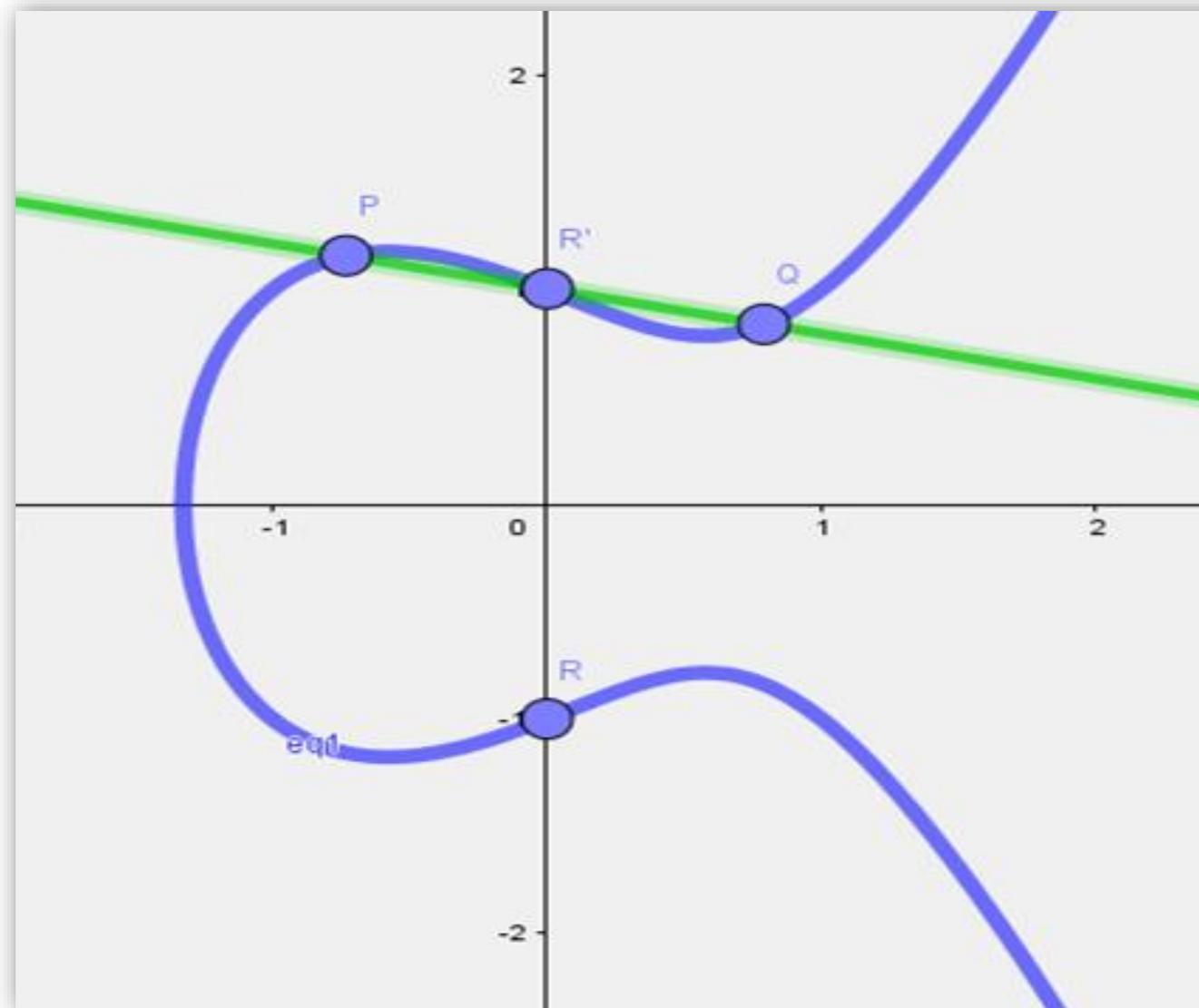


Construction de la loi +

$$E = \{(x, y) \in K^2 \mid y^2 = x^3 + ax + b\}$$

avec $(a, b) \in \mathbb{R}^2$ et $\Delta = -16(4a^3 + 27b^2) \neq 0$

Soit $(P, Q) \in E^2$



$$y = \lambda x + \beta \quad \lambda = \frac{Q_y - P_y}{Q_x - P_x}$$

$$R_y = -\lambda(\lambda^2 - P_x - Q_y) - \beta$$

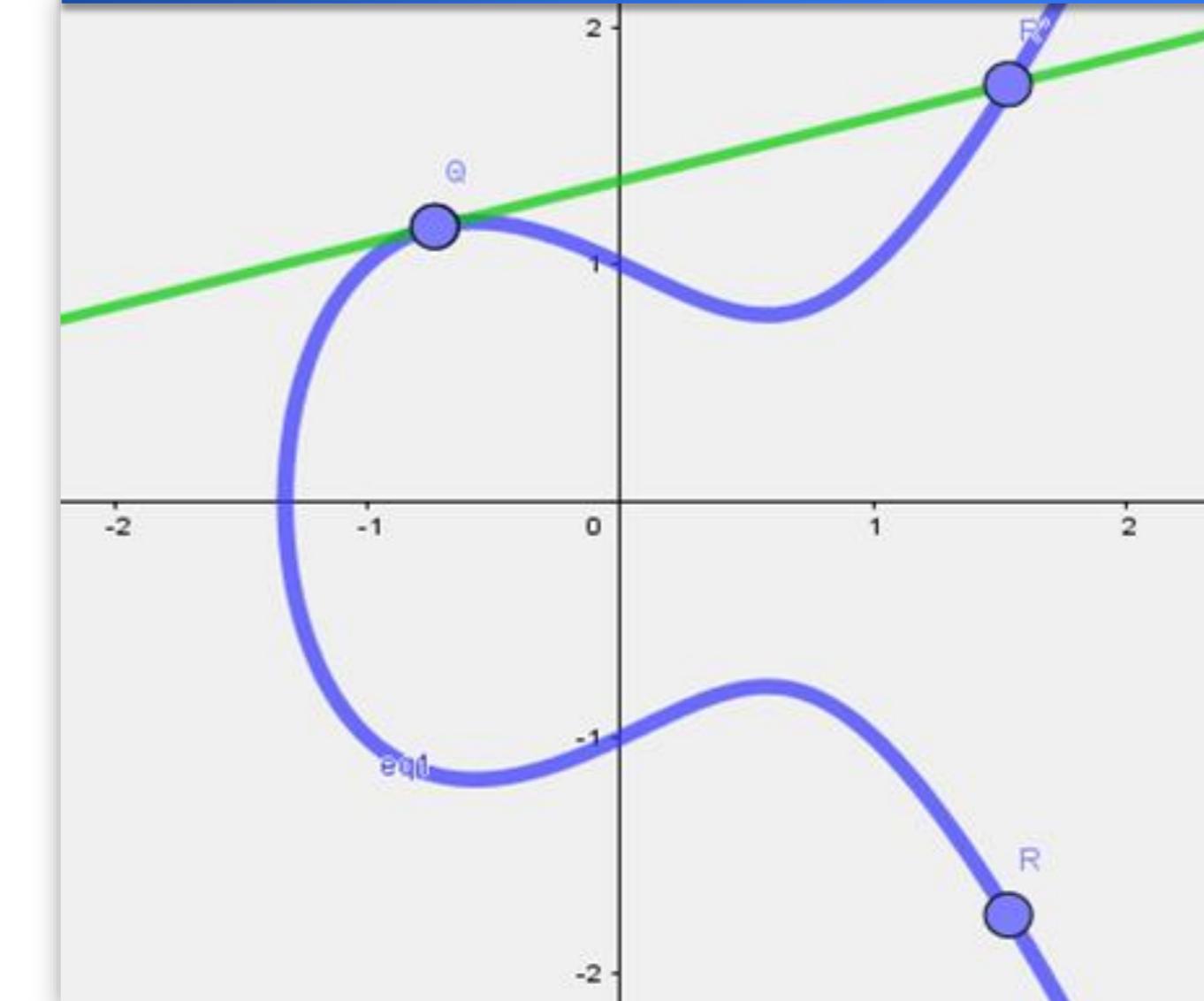
$$R_x = (\lambda^2 - P_x - Q_x)$$

Cas $P \neq Q$

$$y = \lambda x + \beta \quad \lambda = \frac{(3P_x^2 + a)}{2P_y}$$

$$R_y = (\lambda(P_x - R_x) - P_y)$$

$$R_x = \lambda^2 - 2P$$



Cas $P = Q$

Courbe elliptique

H
P

011000
101110
101010

Théorème de Hasse:

$$\text{Card}(E_{(\mathbb{Z}/p\mathbb{Z})}) \in [(p + 1 - 2\sqrt{p}), (p + 1 + 2\sqrt{p})]$$

Pour p très grand, l'ordre du groupe abélien fini n'est pas déterminable facilement

Les algorithmes développés travaillent sur cet ensemble

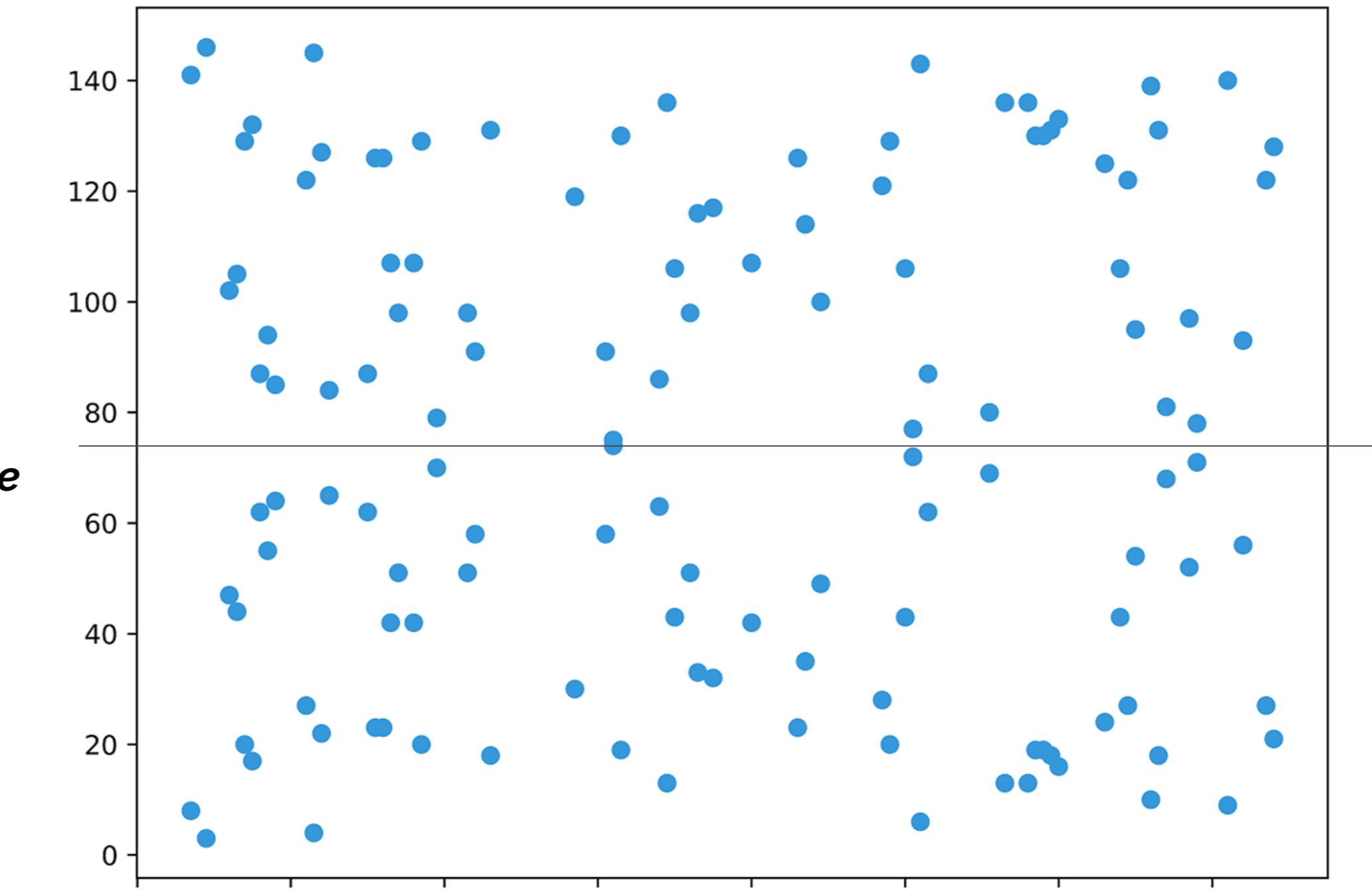
\mathbb{P} : Ensemble des nombres premiers

Soit $p \in \mathbb{P}$, $(a, b) \in \mathbb{N}^2$ tel que $\Delta \neq 0$

Les points naturels vérifiant l'équation de la courbe modulo p sont :

$$E_{(\mathbb{Z}/p\mathbb{Z})} = \left\{ (x, y) \in (\mathbb{Z}/p\mathbb{Z})^2 \mid y^2 = x^3 + ax + b \pmod{p} \right\}$$

Axe de symétrie



$p=149 ; a=3 ; b=-2$

Algorithme

de la définition de l'addition & multiplication

Paramètres :

- p : couple coordonnées d'un point P de la courbe
- Q : couple coordonnées d'un point Q de la courbe
- a : coefficient de la courbe
- b : coefficient de la courbe
- n : nombre premier

Retourne :

- Le point R = P+ Q

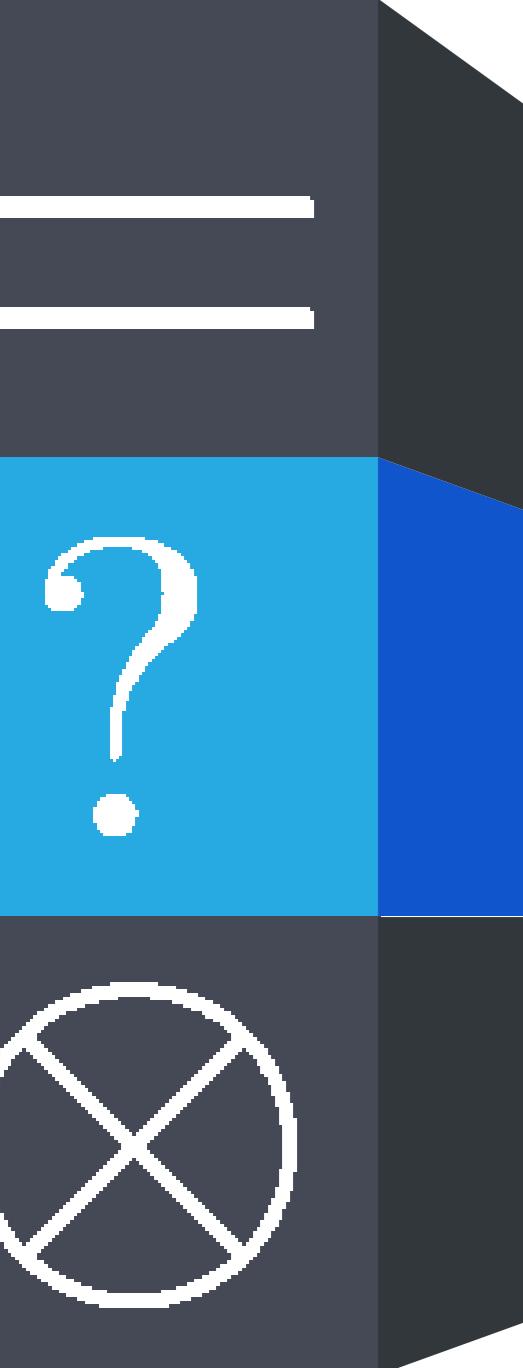
AlgoECC.py

```

12def AdditionPoints(P, Q, a, b, n):
13    # Addition de points sur La courbe
14
15    if [P[0],P[1]] == [0,0]: # R = Q + 0
16        return Q
17    elif [Q[0],Q[1]] == [0,0]: # R = P + 0
18        return P
19    elif P == PointInverse(Q,n): #Return Le point à L'infini si Q = -P => P + -P =0
20        return [0,0]
21    else:
22        if P == Q: #Dans Le cas où P = P => R = 2P, Doubling Add
23            l = (3 * P[0] * P[0] + a) * InverseModule(2 * P[1], n)%n
24        else: #Dans Le cas où P != Q => R = P + Q, Simple Add
25            l = (Q[1] - P[1]) * InverseModule(Q[0] - P[0], n)%n
26
27        #Formules Wikipédia
28        x = (l * l - P[0] - Q[0]) % n
29        y = (l * (P[0] - x) - P[1]) % n
30        return [x, y] #Return Le point R
31
32def MultiplicationPoints(p, f, a, b, n):
33    # Produit de points sur La courbe
34    #Décomposition binaire de La multiplication appelée Double Add
35    r = [0,0]
36    q = p
37    while f > 0: #On boucle L'addition pour obtenir La multiplication des points.
38        if (f & 1) == 1:
39            r = AdditionPoints(r, q, a, b, n)
40            f, q = f >> 1, AdditionPoints(q, q, a, b, n)
41    return r

```

Logarithme discret



$$g^l = y[n]$$

$$l = \log_g(y)[n]$$

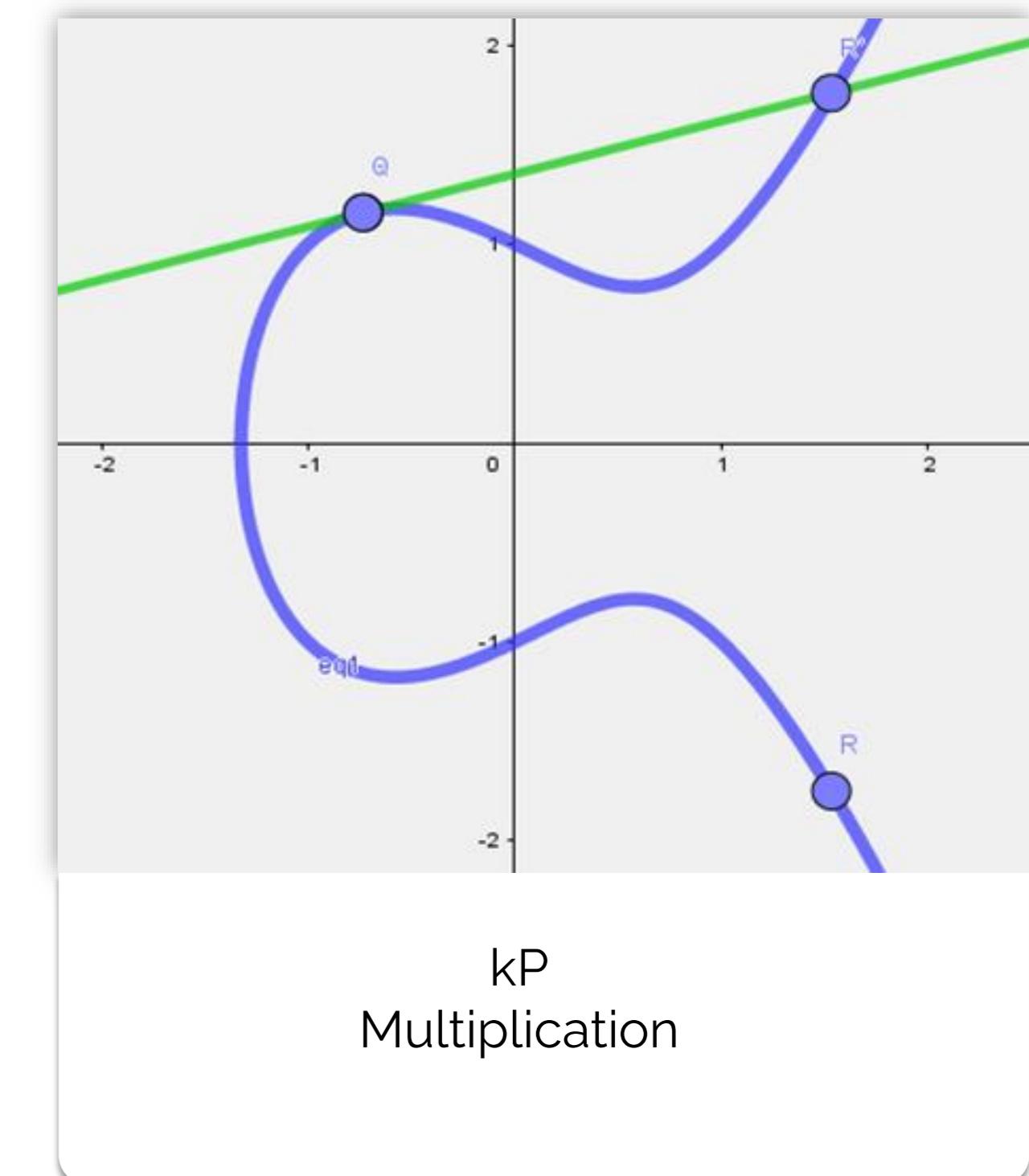
Multiplication scalaire utilisant la loi d'addition pour la courbe elliptique

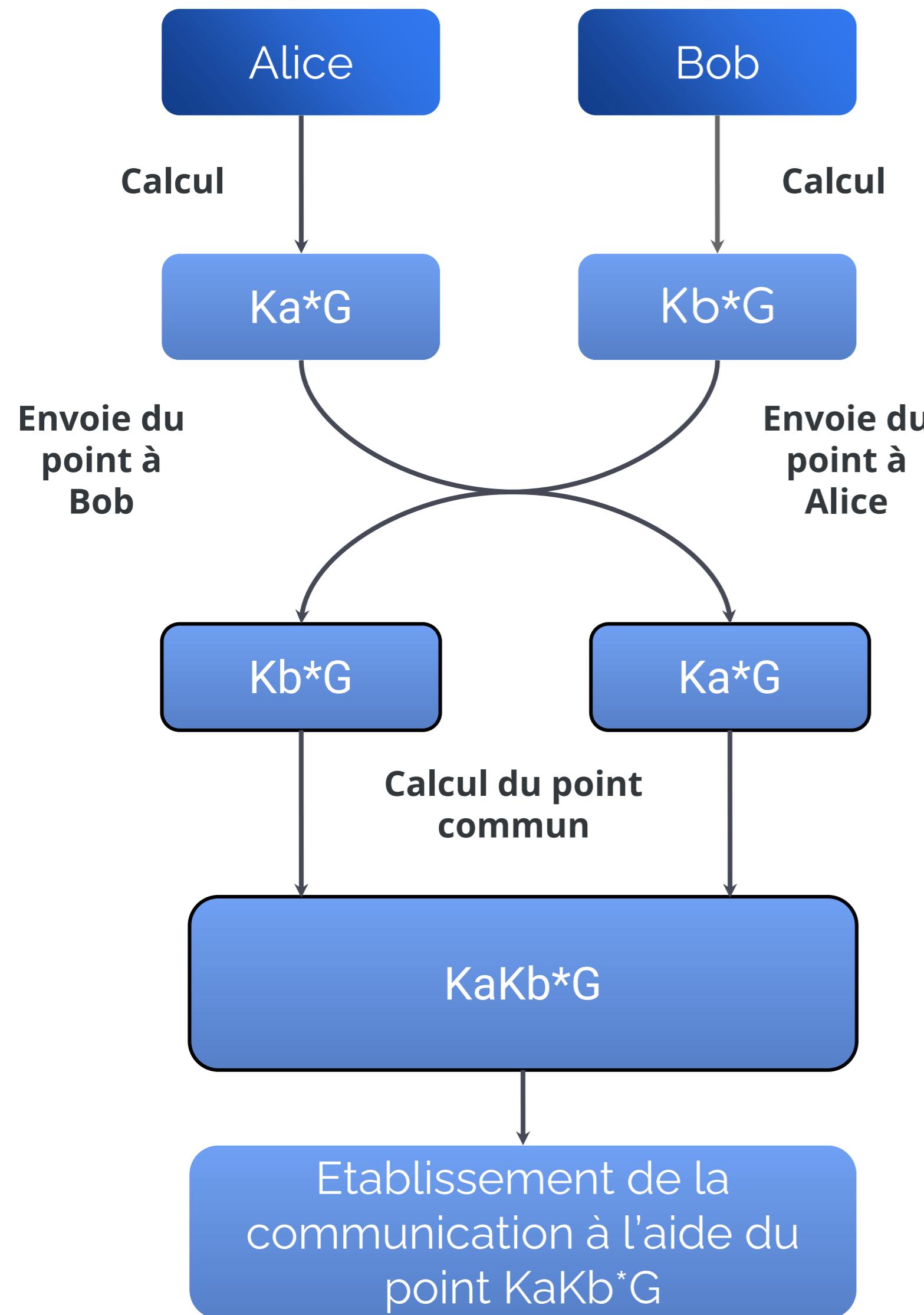


Une solution pour les courbes elliptiques: l'algorithme baby-step giant-step

$$G = \mathbb{Z}/n\mathbb{Z}$$

$$E(\mathbb{Z}/p\mathbb{Z})$$





Procédé de Diffie-Hellman

Génération des clefs des courbes elliptiques

Alice et Bob souhaite communiquer

- Alice choisit un entier Ka
- Bob choisit un entier Kb
- L'un d'entre eux choisit un point quelconque de la courbe appelée G pour point générateur
- Chacun calcul de leur côté KaG et KbG
- Alice envoie le couple coordonnées du point KaG à Bob
- Bob envoie le couple coordonnées du point KbG à Alice
- Alice récupère KbG et calcul $KaKbG$
- Bob récupère KaG et calcul $KbKaG$
- On a l'égalité $KaKbG = KbKaG$
- Ainsi, le point $KaKbG$ est le nouveau point générateur de la conversation et il est sécurisé car on ne peut pas récupérer les valeurs de $KaKb$ depuis $KaKbG$

Le procédé de Diffie-Hellman fonctionne car $KaKb \cdot G = KbKa \cdot G$
Ceci provient du fait que la loi posée est celle d'un groupe abélien

ECDH est soumis au problème du logarithme discret. La difficulté repose sur la recherche de Kb ou de Ka à partir de $Kb \cdot G$ ou $Ka \cdot G$

Algorithme

du procédé Diffie-Hellman

Paramètres :

- a : coefficient de la courbe
- b : coefficient de la courbe
- n : nombre premier
- G : couple coordonnées d'un point générateur de la courbe

Retourne :

- Le couple coordonnées du point $KaKb^*G$

Le choix des entiers Ka et Kb est réalisé de manière aléatoire

AlgoECC.py

```

59 def Procede_DiffieHellman(a,b,n):
60     # Alice et Bob choisissent chacun un nombre
61     Ka = random.randint(1,maxKa)
62     Kb = random.randint(1,maxKa)
63     Px = random.randint(0,n-1)
64     P = [Px, CourbeElliptique(Px,a,b,n)]
65
66     #Calcul de  $Ka^*G$ , et  $Kb^*G$ 
67     # Calcul de la clef commune d'échange  $KaKb^*G$ , on a donc  $Ea=Eb$ 
68     Ea = MultiplicationPoints(MultiplicationPoints(P, Kb, a, b, n), Ka, a, b, n)
69
70     #Verification du procédé de Diffie Hellman
71     """
72     Eb = MultiplicationPoints(MultiplicationPoints(P, Ka, a, b, n), Kb, a, b, n)
73
74     if Eb!=Ea:
75         print('Procédé Diffie-Hellman échoué')
76     """
77     #Retourne Le couple coordonnées du point  $KaKb^*G$ 
78     return(Ea)

```

Résultats des algorithmes

Pour le chiffrement - déchiffrement d'image

Temps total
RSA

4.2197 sec

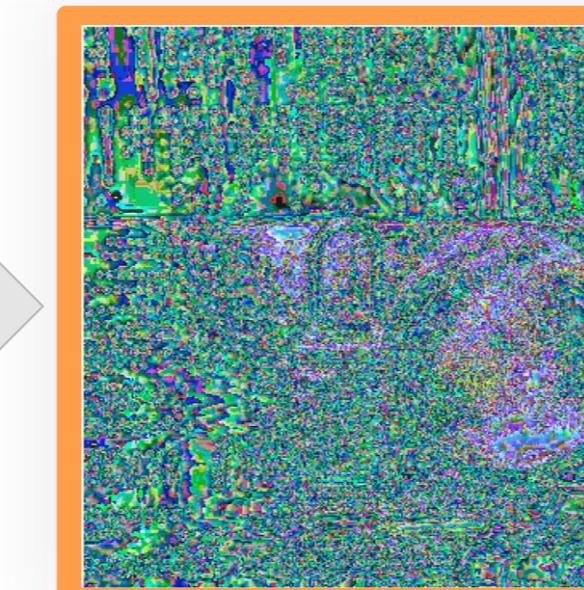
Temps total
ECDH

2.1822 sec

Procédé Diffie--
Hellman



Chiffrement ECDH



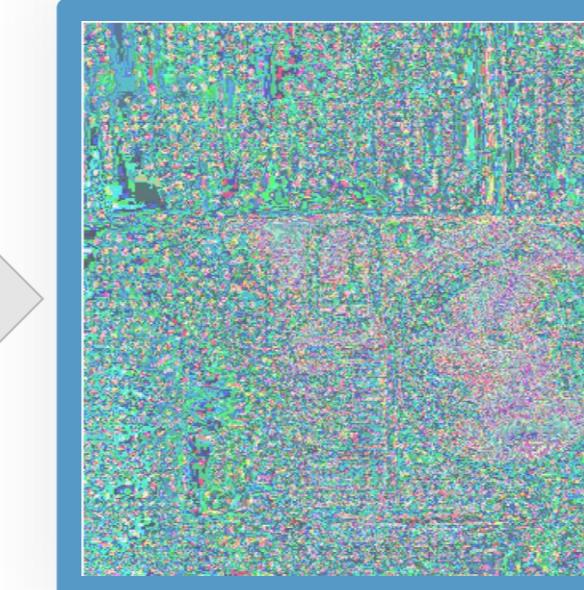
Déchiffrement ECDH



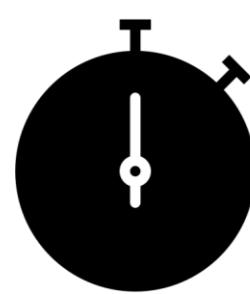
Génération des
clefs RSA



Chiffrement RSA



Déchiffrement RSA



Comparaison temporelle des deux méthodes de chiffrement

Résultats des algorithmes

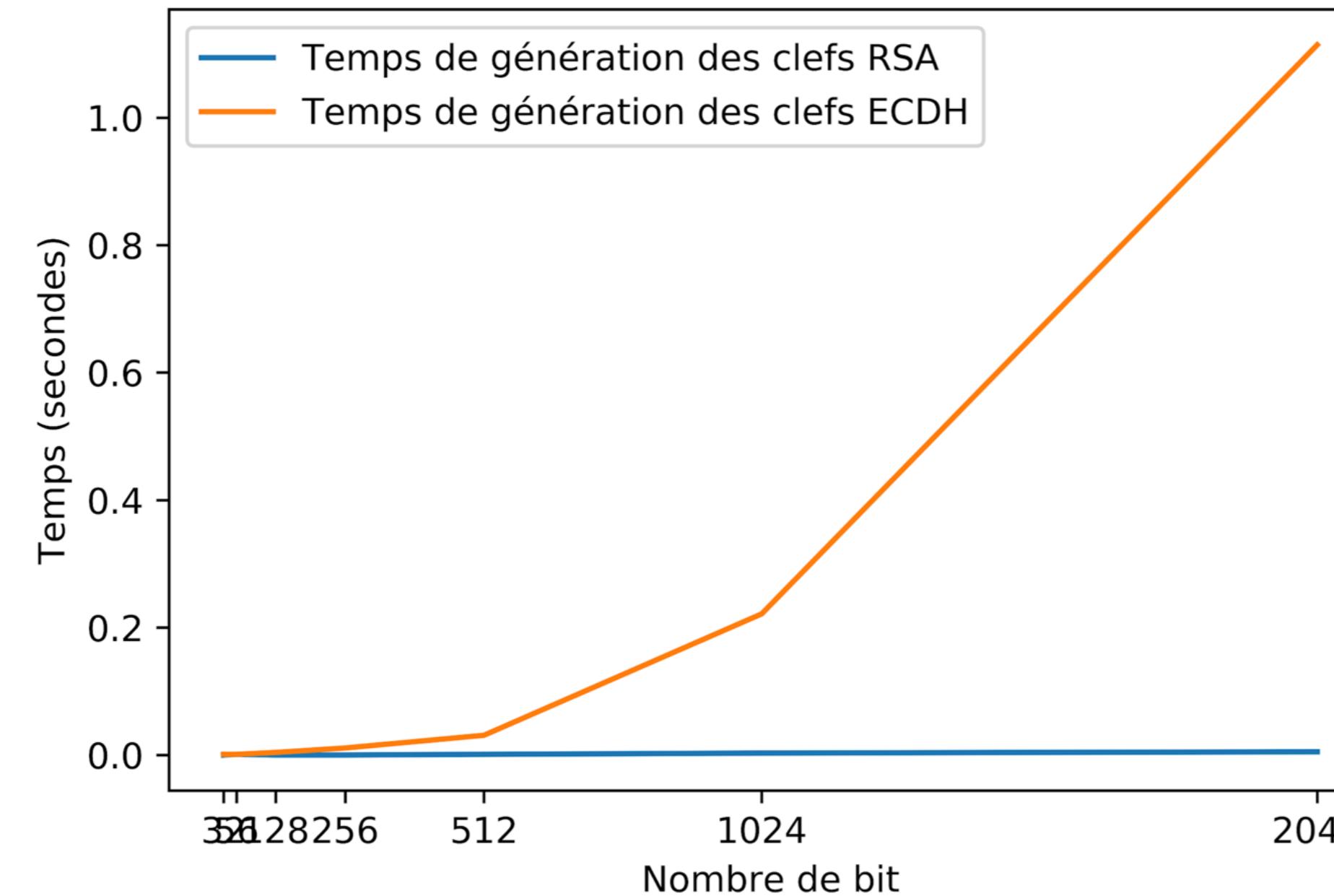
Pour la génération des clés

Temps Moyen
RSA

0.001425 sec

Temps Moyen
Procédé Diffie-Hellman

0.197614 sec



Nombres premiers utilisés en abscisse pour le test

Pour une clef de 2048 bits

RSA

$$P = \text{randomPrime}(2^{1024}, 2^{1025} - 1)$$

$$Q = \text{randomPrime}(2^{1024}, 2^{1025} - 1)$$

$$N = P \times Q$$

Valeur en abscise : N

N s'écrit sur 2048 bits par produit de nombre de 1024 bits

ECDH

$$N = \text{randomPrime}(2^{2048}, 2^{2049} - 1)$$

Valeur en abscise : N

Résultats des algorithmes

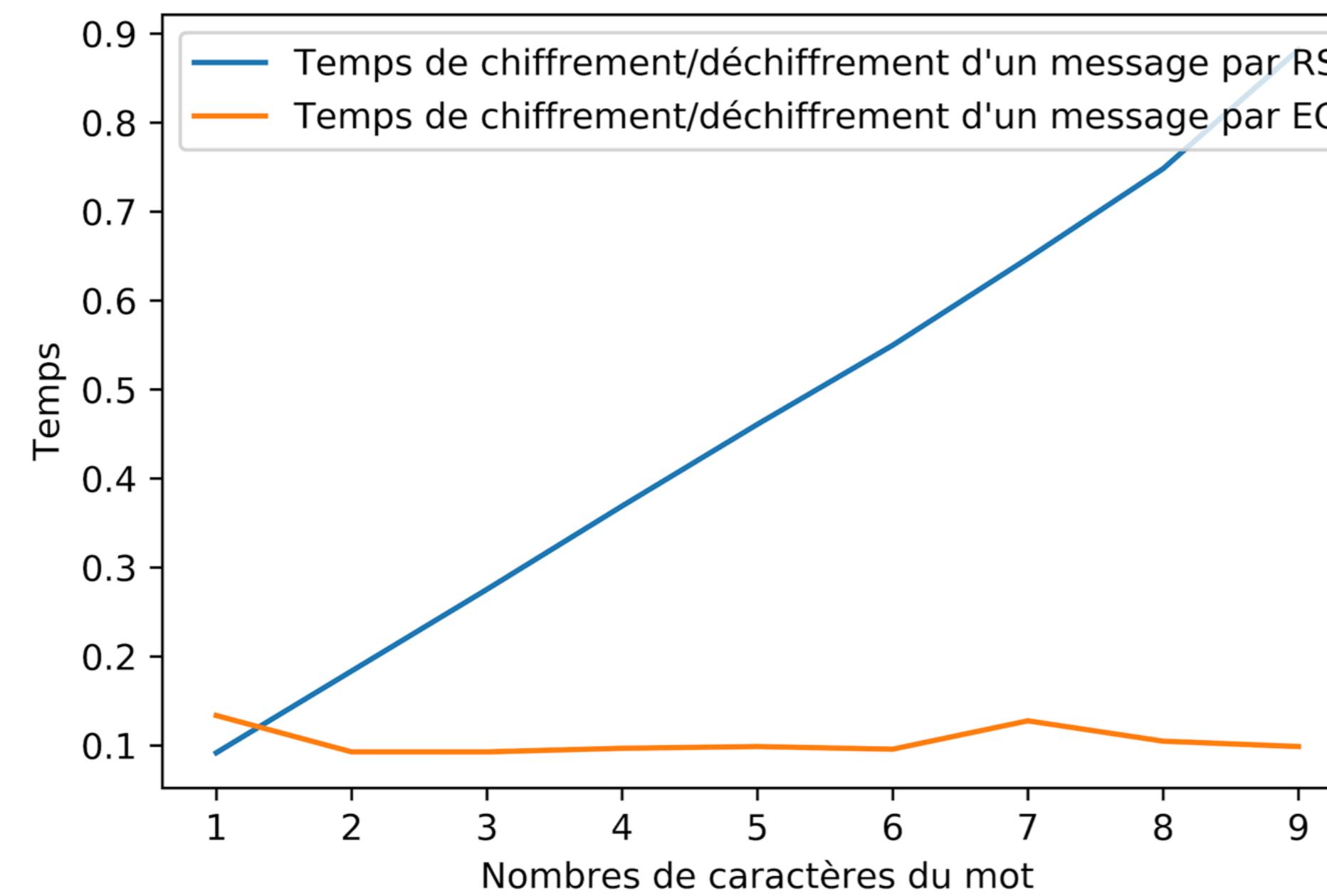
Pour le chiffrement - déchiffrement

Temps Moyen
RSA

0.467416 sec

Temps Moyen
Courbe Elliptique

0.104612 sec



Taille des clefs utilisées
pour le test :

2048 Bits

Exemple:

$n =$

44423430070614350414463964008777448600225
53638805124011759849621405419745893050601
70384844876506599261090701943877921777390
06273780694667790614005376632691955804346
84046600775822246657212814178273815993592
39706009036397971706945741409378577559921
35783722400919186422805096003482589434682
59172952577880763227207549748024939685716
86740522584208349123241405380042781438175
58477248636291356162040871116626092492617
64009223007113697546458377768278844974373
49149471221640045093382681093927421502608
59952830461694451443871978686720052453101
10708316246537649561890116441968590683002
90930313179117079786248957019013725950332

23

Etude Probabiliste

Pour un brute force naïf

Courbes Elliptiques

Le théorème de Hasse nous indique:

$$\text{Card}(E(\mathbb{Z}/p\mathbb{Z})) \in [(p + 1 - 2\sqrt{p}), (p + 1 + 2\sqrt{p})]$$

Nombres de points pour un premier de 2048bits:

$$2^{2048} \pm 2^{1025} \approx 2^{2048}$$

Il y a donc $\approx 2^{2048}$ possibilités pour une clé de 2048 bits de trouver $KaKb^*G$ du premier coup

Le rapport des possibilités $\left(\frac{\text{ECDH}}{\text{RSA}}\right)$ à

Théorème
des nombres premiers

$$\pi(n) \sim \frac{n}{\ln n}$$

RSA

Quantité de nombres premiers de 1024 bits

$$\pi(P) \approx \frac{2^{1024}}{\ln 2^{1024}} \approx 2^{1014} \approx \pi(Q)$$

Nombre de possibilités $N = P^*Q$

$$\frac{2^{2014} \times 2^{2014}}{2} \approx 2^{2027}$$

Il y a donc $\approx 2^{2027}$ possibilités pour une clé 2048 bits pour trouver P et Q

est donc approximatif $\frac{2^{2048}}{2^{2027}} = 2^{21} = 2\,097\,152$

Résoudre différemment les problèmes des chiffrement

Résoudre ECDH, c'est résoudre le logarithme discret

- Baby-step giant-step adapté pour les courbes elliptiques
- Algorithme du crible du corps de nombres généralisé (**GNFS**)
 - Très remarquablement, avec quelques modifications, cet algorithme peut aussi gérer la résolution du logarithme discret et semble être le plus efficace connu aujourd'hui

Résoudre RSA, c'est résoudre le problème de la décomposition de facteurs premiers

- Algorithme du crible quadratique $O(e^{\sqrt{\ln p \times \ln \ln p}})$
 - Algorithme en deux phases: collecte puis exploitation des données
 - Le record (2001) de la décomposition d'un nombre de 135 chiffres en produit de deux facteurs premiers, l'un de 66 chiffres et l'autre de 69 par l'algorithme du crible.
- Algorithme du crible du corps de nombres généralisé (**GNFS**)
 - Algorithme probabiliste. Le plus efficace connu aujourd'hui mais particulièrement complexe
- Compétition de décomposition RSA
 - Record actuel : RSA-768Bits - Thorsten Kleinjung (2009), 232 chiffres
- Décomposition par courbes elliptiques $O(e^{\sqrt{2 \times \ln p \times \ln \ln p}})$
 - La factorisation de Lenstra: utilise le théorème de Hasse et génère des courbes elliptiques pour factoriser des nombres. Dès que cela ne marche pas, l'algorithme génère une nouvelle courbe. Algorithme probabiliste

1. ECDH

LES PLUS

- + **Plus rapide**

De meilleures performances à taille de clé équivalente

-
- + **Plus sûr**

Pour une attaque de brute force naïf

LES MOINS

- + **Plus récent**

Ce qui peut poser des problèmes à l'avenir

Conclusion

en réponse à la problématique

2. RSA

LES PLUS

- + **Génération des clés rapides**
- + **Plus de connaissance sur la décomposition RSA**

LES MOINS

- **Moins rapide globalement**
- **Moins de possibilités**

Listing des algorithmes

AlgoMaths.py

01

Fonctions:

InverseModule, InverseModuleRSA (Algorithm Etendu D'Euclide), pgcd (itératif) , RacineCarreeEntiere (Méthode Newton)

AlgoECC.py

02

Fonctions:

CourbeElliptique, AdditionPoints, MultiplicationsPoints (Exponentiation Rapide), PointInverse, Procede_DiffieHellman, PointValide, CrypterECDH, DecrypterECDH

AlgoRSA.py

03

Fonctions:

RSAGenerationDesCles, RSACrypte, RSAEncrypte

Chiffrement Image.py

04

Descriptif:

Réalise le test de chiffrement-déchiffrement d'une image de 350x350px

PerformanceTeste.p
v

05

Descriptif:

Réalise le test de la génération des clés pour des clefs variant de 32 bits à 2048bits

RapiditeTeste.py

06

Descriptif:

Réalise le test de rapidité de chiffrement-déchiffrement d'un message généré aléatoirement de taille variant avec la longueur de la clé fixé

```
6 def InverseModule(x,p):
7     if x % p == 0:
8         raise ZeroDivisionError("Inverse Module Fail : Les deux valeurs ne sont pas premières entre elle")
9     return pow(x, p-2, p)
10
11 def InverseModuloRSA(a, m):
12     # Retourne la valeur x tq a*x % m = 1
13     if pgcd(a, m) != 1:
14         return 'Inverse Module RSA Bug : Les deux valeurs ne sont pas première entre elle'
15     # Ceci est l'Algortihme Etendu D'Euclide
16     u1, u2, u3 = 1, 0, a
17     v1, v2, v3 = 0, 1, m
18     while v3 != 0:
19         q = u3 // v3
20         v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2), (u3 - q * v3), v1, v2, v3
21     return(u1 % m)
22
23 def pgcd(a,b):
24     #Calcul itératif du pgcd pour éviter les problèmes de stack récursif
25     while b!=0:
26         a,b=b,a%b
27     return a
28
29 def RacineCarreeEntiere(n): #Newton Méthode
30     x = n
31     y = (x + 1) // 2
32     while y < x:
33         x = y
34         y = (x + n // x) // 2
35     return x
36
37 def PuissanceModuloRapide(x, n, y):
38     #Fonction qui calcul  $x^{n \% 2}$  en interne en calcul exponentiation rapide
39     if n==1:
40         return(x%y)
41     elif n%2==0:
42         return(PuissanceModuloRapide(x**2,n/2,y)%y)
43     else:
44         return(PuissanceModuloRapide(x**2,(n-1)/2,y)*x%y)
```

```

def PointValide(P,a,b,n): #Retourne True si le point est bien sur la courbe
    return((P[0]**3 + a * P[1] + b)%n == (P[1]**2)%n)

def PointInverse(P,n): #Renvoie le point Inverse Q tq P + Q = O = P + (-P)
    if P == [0,0]:
        return P
    return [P[0], -P[1]%n]

def CourbeElliptique(x, a, b, n):
    return RacineCarreeEntiere(x ** 3 + a * x + b, n)

def CrypterECDH(msg, Ea,n):
    #Le message est sous la forme [[a1,b1],..,[an,bn]] où [an,bn] point de la courbe étant l'unité
    if type(msg) == list:
        msgcrypt=[]
        for i in range(0, len(msg)):
            msgcrypt.append([(msg[i][0]*Ea[0])%n,(msg[i][1]*Ea[1])%n])
        return(msgcrypt)
    elif isinstance(msg,np.ndarray):
        N = (msg.astype(int))
        NCrypt=np.zeros((len(N), len(N[0])), int)
        for i in range(len(N)):
            for j in range(len(N[0])):
                for k in range(len(N[0][0])):
                    NCrypt[i][j][k] = (N[i][j][k]*EaInv[0]) % n
        return NCrypt

```

Ici le point Ea est issu du procédé de Diffie-Hellman et très difficile à trouver. C'est le point d'intersection de deux droites.

```

def DecrypterECDH(msg, Ea,n):
    #Calcul de l'inverse du point  $KaKbG$ 
    EaInv = [InverseModule(Ea[0],n), InverseModule(Ea[1],n)]
    if type(msg) == list:
        decryptmsg=[]
        for i in range(0,len(msg)):

```

AlgoECC.py

1 / 2

02

Algo

```

103    elif isinstance(msg,np.ndarray):
104        N = (msg.astype(int))
105        NDecrypt=np.zeros((len(N), len(N[0])), len(N[0][0])), dtype=int
106        for i in range(len(N)):
107            for j in range(len(N[0])):
108                for k in range(len(N[0][0])):
109                    NDecrypt[i][j][k] = (N[i][j][k]*EaInv[0]) % n
110        return NDecrypt
111
112
113
114 ***** FIN FONCTION ECDH *****

```

```

9 def RSAGenerationDesCles(p,q):
10    n = p*q
11    phiden = (p-1)*(q-1)
12    e = random.randrange(p, phiden)
13    d=e
14    #On fait en sorte que la clé publique ne soit jamais égale à la clé privée
15    while e == d:
16        #On cherche e tel que e soit premier avec (p-1)(q-1) (phiden)
17        PGCD1 = 0
18        while PGCD1 != 1 :
19            e = random.randrange(p, phiden)
20            PGCD1 = pgcd(e,phiden)
21        #On cherche d tel que d soit l'inverse du module
22        d = InverseModuloRSA(e, phiden)
23    return(e, d, n)
24
25 def RSADecrypte(MOT, e, n):
26    if type(MOT) == list:
27        L=[]
28        for i in range(0, len(MOT)):
29            L.append(PuissanceModuloRapide(MOT[i],e,n))
30    return(L)
31    elif type(MOT) == int:
32        return(PuissanceModuloRapide(MOT,e,n))
33    elif isinstance(MOT,np.ndarray):
34        N=copy.copy(MOT)
35        N2 = (N.astype(int))
36        NDecrypt=np.zeros((len(N), len(N[0])), len(N[0][0])), dtype=int)
37        for i in range(len(N)):
38            for j in range(len(N[0])):
39                for k in range(len(N[0][0])):
40                    s = PuissanceModuloRapide(int(N2[i][j][k]), e, n)
41                    NDecrypt[i][j][k] = s
42    return NDecrypt
43
44 def RSACrypte(mot,d,n):
45    if type(mot) == list:
46        crypt=[]
47        for i in range(len(mot)):
48            crypt.append(PuissanceModuloRapide(mot[i],d,n))

```

```

49    return(crypt)
50    elif type(mot) == int:
51        return(PuissanceModuloRapide(mot,d,n))
52    elif isinstance(mot,np.ndarray):
53        N=copy.copy(mot)
54        N2 = (N.astype(int))
55        Ncrypt=np.zeros((len(N), len(N[0])), len(N[0][0])), dtype=int)
56        for i in range(len(N)):
57            for j in range(len(N[0])):
58                for k in range(len(N[0][0])):
59                    s = PuissanceModuloRapide(int(N2[i][j][k]), d, n)
60                    Ncrypt[i][j][k] = s
61    return Ncrypt

```

ChiffrementImage.py

ChiffrementImage.py
1/2

```

15 imgpil = Image.open("C:/Users/le2fi/Desktop/PYTHON/tipe/Finale/crypto.png")
16 img = np.array(imgpil) # Transformation de l'image en tableau numpy
17 imgcrypt = (img.astype(int))
18
19 #RSA
20 RSAPremier1 = sympy.randprime(20,2**6)
21 RSAPremier2 = sympy.randprime(20,2**6)
22 #ECDH
23 ECDHPrime = sympy.randprime(0,2**16)
24 a=3
25 b=-2
26
27 """ DEBUT DU TIMER RSA """
28 start_time = time.time()
29 #Génération des clés
30 RSAKeys = RSAGenerationDesCles(RSAPremier1,RSAPremier2)
31
32 #Chiffrement de l'image initiale
33 imgcrypt2 = RSACrypte(img.astype(int), RSAKeys[0], RSAKeys[2])
34 N1 = imgcrypt2.astype('uint8')
35 print('\n\nRSA - Image Cryptée : ')
36 plt.imshow(N1)
37 plt.axis('off')
38 plt.show()
39
40 #Déchiffrement de l'image initiale
41 N2 = RSADecrypte(imgcrypt2,RSAKeys[1],RSAKeys[2])
42 N3 = N2.astype('uint8')
43 print('\n\nRSA - Image Décryptée : ')
44 plt.imshow(N3)
45 plt.axis('off')
46 plt.show()
47 """ FIN TIMER RSA """
48 print(round((time.time() - start_time),10))
49
50
51 """ DEBUT DU TIMER ECDH """
52 start_time = time.time()
53 #Génération des clés
54 ClefEchange = Procede_DiffieHellman(a,b,ECDHPrime)

```

```

55 print("Clefs ECDH", ClefEchange)
56
57 #Chiffrement de l'image initiale
58 N4 = CrypterECDH(img.astype(int), ClefEchange,ECDHPrime)
59 N5 = N4.astype('uint8')
60 print('\n\nECDH - Image Cryptée : ')
61 plt.imshow(N5)
62 plt.axis('off')
63 plt.show()
64
65 #Déchiffrement de l'image cryptée
66 N6 = DecrypterECDH(N4, ClefEchange, ECDHPrime)
67 N7 = N6.astype('uint8')
68 print('\n\nECDH - Image Décryptée : ')
69 plt.imshow(N7)
70 plt.axis('off')
71 plt.show()
72 """ FIN TIMER ECDH """
73 print(round((time.time() - start_time),10))

```

ChiffrementImage.py
2/2

PerformanceTeste.py

1/2

```

29 # Paramètres des clés
30 TailleClef=[32,56,128,256,512,1024,2048]
31
32 #Génération des nombres premiers de différente taille
33 RSATime=[]
34 RSAPrime=[]
35 [sympy.randprime(2**16, 2**17-1),sympy.randprime(2**16, 2**17-1)],
36 [sympy.randprime(2**28, 2**29-1),sympy.randprime(2**28, 2**29-1)],
37 [sympy.randprime(2**64, 2**65-1),sympy.randprime(2**64, 2**65-1)],
38 [sympy.randprime(2**128, 2**129-1),sympy.randprime(2**128, 2**129-1)],
39 [sympy.randprime(2**256, 2**257-1),sympy.randprime(2**256, 2**257-1)],
40 [sympy.randprime(2**512, 2**513-1),sympy.randprime(2**512, 2**513-1)],
41 [sympy.randprime(2**1024, 2**1025-1),sympy.randprime(2**1024, 2**1025)],
42 ]
43
44 ECDHTime=[]
45 ECDHPrime=[]
46 sympy.randprime(2**28, 2**29-1),
47 sympy.randprime(2**64, 2**65-1),
48 sympy.randprime(2**128, 2**129-1),
49 sympy.randprime(2**256, 2**257-1),
50 sympy.randprime(2**512, 2**513-1),
51 sympy.randprime(2**1024, 2**1025-1),
52 sympy.randprime(2**2048, 2**2049-1)
53 ]
54
55
56 #Debut du test pour le RSA
57 for i in range(0,len(RSAPrime)):
58     start_time = time.time()
59     print(RSGénérationDesClés(RSAPrime[i][0],RSAPrime[i][1]))
60     RSATime.append(round((time.time() - start_time),10))
61 plt.plot(TailleClef,RSATime,label="Temps de génération des clefs RSA")
62
63 print('\n\n')
64
65 #Debut du test pour le ECDH
66 for i in range (0, len(ECDHPrime)):
67     start_time = time.time()
68     print(Procédé_DiffieHellman(a,b,ECDHPrime[i]))

```

```

69     ECDHTime.append(round((time.time() - start_time),10))
70 plt.plot(TailleClef,ECDHTime,label="Temps de génération des clefs ECDH")
71
72 #Définition de la courbe avec matplotlib
73 plt.legend(loc='upper left')
74 plt.legend(loc='upper left')
75 plt.ylabel('Temps (secondes)')
76 plt.xlabel('Nombre de bit')
77 plt.xticks(TailleClef)
78 plt.savefig('C:/Users/le2fi/Desktop/PYTHON/tipe/Finale/PerformanceTest.png', dpi=800)
79 plt.show()
80
81 #Calculs temps moyens
82 print('Temps moyen pour le RSA :', round(sum(RSATime)/len(RSATime),6),"s")
83 print('Temps moyen pour le ECDH :', round(sum(ECDHTime)/len(ECDHTime),6),"s")

```

PerformanceTeste.py

2/2

```

17 #Paramètres du programme
18 BitTotal= 2048
19 #RSA
20 from GenerateurGrandsNombresPremiers import *
21 RSAPremier1 = GenererUnNombrePremierDeTaille(int(BitTotal/2))
22 RSAPremier2 = GenererUnNombrePremierDeTaille(int(BitTotal/2))
23
24 #ECDH
25 ECDHPremier = GenererUnNombrePremierDeTaille(BitTotal)
26
27
28 #Régistre des mots à chiffrer/déchiffrer
29 NombreDeMots = 10
30 TailleMotMax = 10 #Nb de caractères maximal du dernier mot
31
32 #Génération de la liste des nombres de mots
33 LongueurMot = []
34 for i in range(1, NombreDeMots):
35     LongueurMot.append(int(i*TailleMotMax/NombreDeMots))
36
37
38 #Génération de la liste des mots pour le chiffrement / déchiffrement RSA
39 MSGRSA=[]
40 for i in range(0, len(LongueurMot)):
41     MSGRSA.append([])
42     for j in range(0, LongueurMot[i]):
43         MSGRSA[i].append(random.randint(1,RSAPremier1*RSAPremier2 -1))
44
45
46 #Début du test de rapidité,
47 #établissement de la liste des valeurs T en fonction du nombre de caractères du mot
48 RSATemps=[]
49 RSAMessageFinale = []
50 RSAKeys = RSAGenerationDesCles(RSAPremier1,RSAPremier2)
51 print(RSAKeys)
52 print(" RSA - Clefs :\n","Clé Privée :",RSAKeys[1],"Clé Publique :",RSAKeys[0],"nP*Q :",RSAKeys[2])
53 for i in range (0, len(LongueurMot)):
54     #Début du timer
55     start_time = time.time()
56     #Ligne qui permet le déchiffrement ainsi que le chiffrement

```

RapiditeTeste.py

1/3

```

57     RSAMessageFinale.append(RSADecrypte(RSACrypte(MSGRSA[i], RSAKeys[0], RSAKeys[1], RSAKeys[2])), RSAKeys[1], RSAKeys[2]))
58     RSATemps.append(time.time() - start_time)
59     #Fin du timer
60 #Vérification que la liste issues du chiffrement et déchiffrement est bien égale à la liste de départ, si cas ci
61 if RSAMessageFinale == MSGRSA :
62     print('\nRSA: Chiffrement / Déchiffrement correctement déroulé')
63 else:
64     print('\nRSA : Problème de cohérence de la liste MSGRSA et RSAMessageFinale')
65
66
67
68
69 #--ECDH--
70 # Définition de la courbe elliptique  $y^{**2} = x^{**3} + ax + b \text{ mod } n$ 
71 a = random.randint(20,150)
72 b = random.randint(20,150)
73 n = ECDHPremier
74 # n doit être un nombre premier
75
76 #Calcul de la clé d'échange grâce au procédé de DiffieHellman
77 ClefEchange = Procede_DiffieHellman(a,b,n)
78 ECDHTemps=[]
79 #Début du test de rapidité, établissement de la liste des valeurs T en fonction du nombre de caractères du mot
80 for i in range(0, len(LongueurMot)):
81     MessageACrypter = []
82     for j in range(0,LongueurMot[i]):
83         rdm= random.randint(0,n)
84         MessageACrypter.append([rdm, CourbeElliptique(rdm, a, b, n)])
85     #Début du timer
86     start_time2 = time.time()
87     MessageCrypte = CrypterECDH(MessageACrypter, ClefEchange,n)
88     MessageDecrypte = DecrypterECDH(MessageCrypte, ClefEchange, n)
89     ECDHTemps.append(time.time() - start_time2)
90     #Fin du timer
91
92 #Vérification que la liste issues du chiffrement et déchiffrement est bien égale à la liste de départ, si cas ci
93 if MessageACrypter == MessageDecrypte :
94     print('\nECDH: Chiffrement / Déchiffrement correctement déroulé')
95 else:
96     print('\nECDH : Problème de cohérence de la liste MessageACrypter et MessageDecrypte')

```

RapiditeTeste.py

2/3

```
97  
98  
99  
100 #Définition de la courbe par matplotlib  
101 plt.plot(LongueurMot,RSATemps,label="Temps de chiffrement/déchiffrement d'un message par RSA")  
102 plt.plot(LongueurMot,ECDHTemps,label="Temps de chiffrement/déchiffrement d'un message par ECDH")  
103 plt.xlabel("Nombres de caractères du mot")  
104 plt.ylabel("Temps")  
105 plt.legend(loc='upper left')  
106  
107 print('\nTests réalisés avec un processeur Intel Core i3-6006U 2.0Ghz, 4GB DDR4')  
108 plt.savefig('C:/Users/le2fi/Desktop/PYTHON/tipe/Finale/RapiditeTest.png', dpi=800)  
109 plt.show()  
110  
111 #Calculs des temps moyens  
112 print('Temps moyen pour le RSA :', round(sum(RSATemps)/len(RSATemps),6),"s")  
113 print('Temps moyen pour l\'ECDH :', round(sum(ECDHTemps)/len(ECDHTemps),6),"s")  
114
```

RapiditeTeste.py

3 / 3