**Partie I : Présentation du problème et de l’algorithme de Shank.**

*a) Crypto-système d’El Gamal et problème du logarithme discret.*

Soient Alice et Bob deux personnes munies des clés privées respectives et . On suppose que Bob souhaite envoyer un message noté *m* à Alice (on suppose que *m* est un entier). Pour cela les opérations réalisées par les deux individus sont les suivantes :

1 : Alice choisit un nombre premier *p*. La suite des calculs s’inscrit alors dans le groupe multiplicatif G = (, .).

*Propriété 1 :* Pour p un nombre premier, (, .) est un groupe.

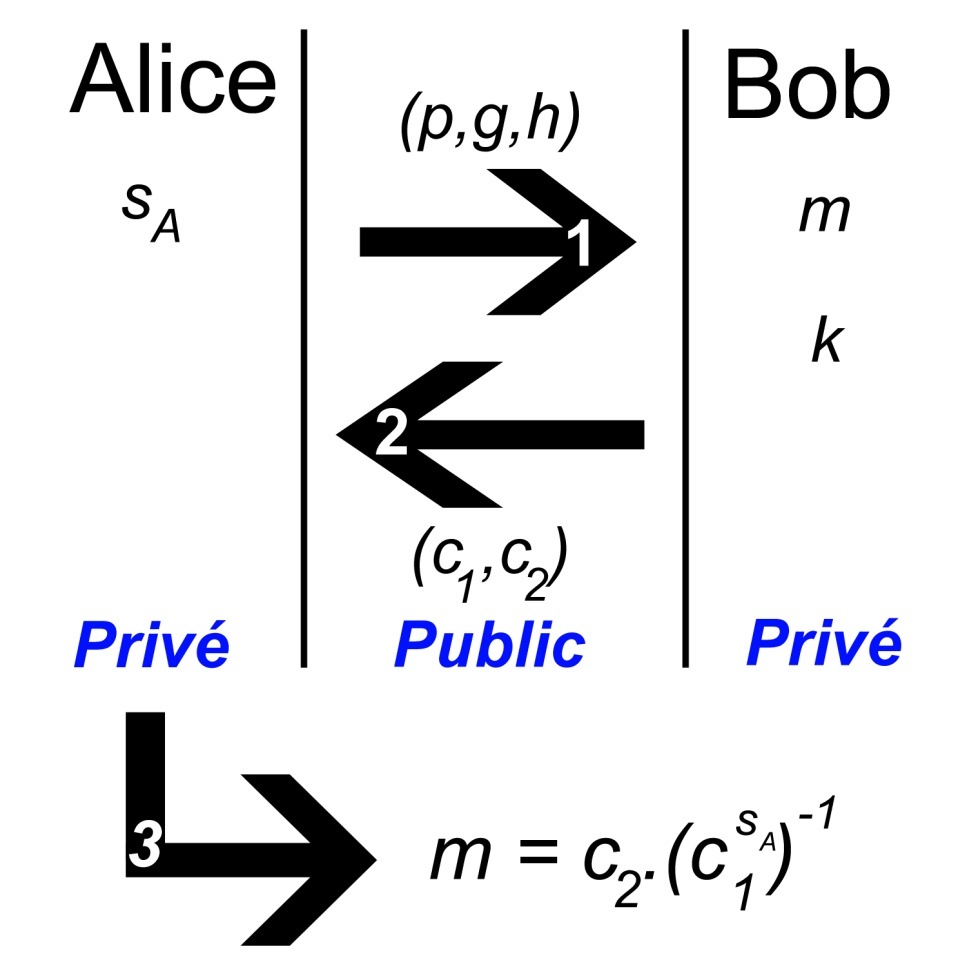
De plus, ce groupe est cyclique, c'est-à-dire qu’il existe tel que :

Alice choisit ensuite un générateur *g* de G et calcule dans G (). Alice envoi à Bob le triplet *(p,g,h)* qui constitue la clé publique.

*Note : les éléments de G ne sont pas tous générateurs pour la loi multiplicative (mais ils le sont pour la loi additive).*

2 : Bob choisit un entier *k* au hasard dans {0,…, p-1} et calcule et dans G à l’aide de la clé publique. Bob envoi le couple *( )* à Alice.

3 : En remarquant que , Alice est capable de récupérer le message à l’aide de sa clé privée en calculant : .



*p* nombre premier.

*g* générateur de G =

et

d’où,

***Note :*** *donc (choisir p assez grand).*

Supposons maintenant qu’une troisième personne tente de trouver le message *m* avec pour seules données les variables *p, g, h, et* . Cette personne n’est pas en mesure de décrypter le message de la même façon qu’Alice puisqu’elle ne connait pas sa clé privée , de plus, elle ne peut pas non plus déduire *m* directement de puisqu’elle ne connait l’entier *k*.

Pour déduire de *h* ou *k* de et ainsi déchiffrer le message crypté, elle se voit donc confrontée au problème du logarithme discret :

*Définition 1 :* Soit G un groupe cyclique d’ordre *p* dont la loi est notée multiplicativement, soit *g* un élément générateur de G et *y* un élément de G. On appelle logarithme discret le plus petit entier *k* tel que :

(On gardera ces notations par la suite.)

Calculer *y* à partir de *g* et *k* est une opération qui peut être réalisée relativement rapidement. En effet, avec un algorithme d’exponentiation rapide (voir annexe), la complexité temporelle du calcul de *y* est en (et ).

Cependant, l’opération inverse, c'est-à-dire calculer *k* à partir de *y* et *g*, reste encore à ce jour un véritable défi. En effet, pour des valeurs de *p* relativement élevées, on ne connait pas encore d’algorithme permettant de calculer *k* en un temps « raisonnable ».

*b) Etude algorithme naïf .*

Par exemple, considérons la méthode naïve de résolution du logarithme discret : on calcule les puissances successives de *g* jusqu’à tomber sur *y* (, , , …, *y*). La complexité d’un tel algorithme est donc en .

Plusieurs problèmes sont intervenus dans l’étude de l’algorithme naïf.

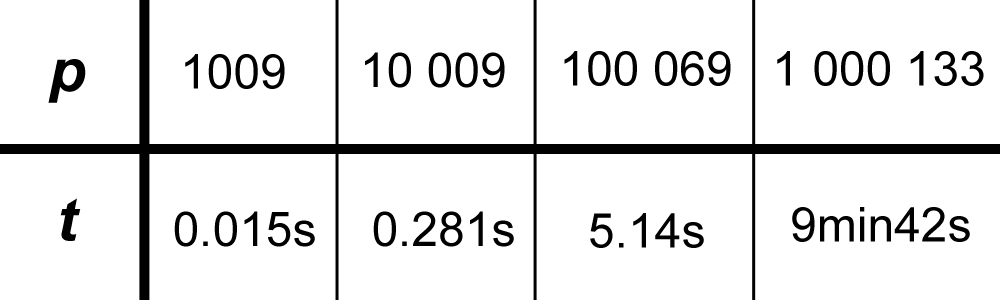
Le premier d’entre eux a été la recherche d’éléments générateurs. En effet, l’algorithme que j’avais programmé initialement pour la recherche d’éléments générateurs (*nul\_elements\_*generateurs) n’était pas assez rapide dès que *p* était de l’ordre de (ce qui est un très petit nombre en cryptographie). J’ai donc recherché des propriétés mathématiques permettant de déterminer si un élément est générateur plus rapidement et j’ai trouvé le résultat suivant :

*Théorème 1* : Soit un nombre premier tel que avec pour tout un nombre premier et un entier. Un élément α est générateur si et seulement si:

Cependant, pour exploiter ce théorème, il faut une fonction qui développe en produit de facteurs premiers. La recherche de diviseurs premiers étant un problème complexe et pas l’objet de ce TIPE, j’ai récupéré sur internet une fonction (*facteurs*) donnant la dite décomposition.

Deuxièmement, cette méthode requiert des exponentiations assez conséquentes, de sorte que est un nombre trop grand si on le calcule sans modulo. Or la formule *(\*\*(()/))%*  calcule d’abord *\*\*(()/)* avant de prendre le reste modulo ce qui génère une erreur (« OverflowError »). Il faut donc prendre les restes modulo en même temps que l’on calcule l’exponentiation : c’est l’exponentiation modulaire (*mod\_pow*).

Tout compte fait, l’algorithme final (*elements\_*generateurs) permet de trouver les éléments générateurs de nombres premiers relativement grands. Voici quelques exemples :



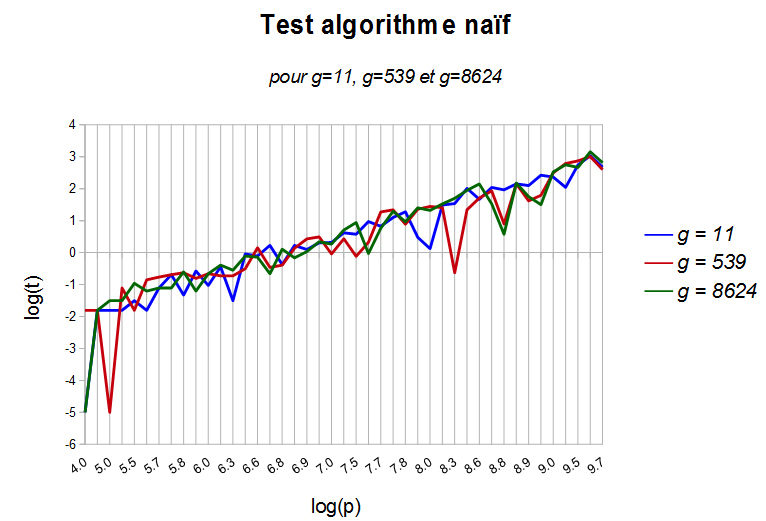
A partir d’un nombre premier de l’ordre de 10 000 000 je n’ai pas pu calculer le temps de recherche des éléments générateurs même après plusieurs heures.

De plus, en s’intéressant juste a une liste de 3 éléments générateurs [11,539,8624] on réaliste 3 tests pour chaque nombre premiers au lieu de . J’ai pu alors déterminer une liste relativement exhaustive de nombres premiers entre 10 mille et 10 millards qui ont la particularité d’admettre 11, 539 et 8624 comme éléments générateurs : [10009,50069,100069,200003,300007,400069,500029,600011,700027,800117,900091,1000133,2000003,3000029,4000037,5000201,6000047,7000061,8000009,9000011,10000103,20000003,30000023,40000123,50000063,60000011,70000037,80000069,90000049,100000049,200000033,300000119,400000043,500000057,600000019,700000001,800000063,900000011,1000000097,2000000011,3000000077,4000000007,5000000497,6000000089,7000000013,8000000081,9000000307,10000000019].

C’est sur cette même liste que je testerai tous mes algorithmes pour les 3 éléments générateurs distincts.

Du aux écarts considérables entre différentes valeurs, j’ai utilisé des échelles logarithmiques pour le nombre premier testé et pour le temps d’exécution.

Voici les résultats obtenus pour l’algorithme naïf :



c) Une méthode plus efficace : algorithme de Shank.

Heureusement pour nous, il existe un algorithme, appelé algorithme de Shank ou encore « Baby-steps Giant-steps », permettant d’assurer une complexité temporelle moindre au coût d’une complexité spatiale accrue. Le principe de cet algorithme est le suivant :

1 : On choisit un entier *n*.

2 : Cette phase, nommée « Baby-steps », consiste à calculer les puissances successives de *g* jusqu’à tout en les gardant en mémoire. On stocke , , , …, en mémoire.

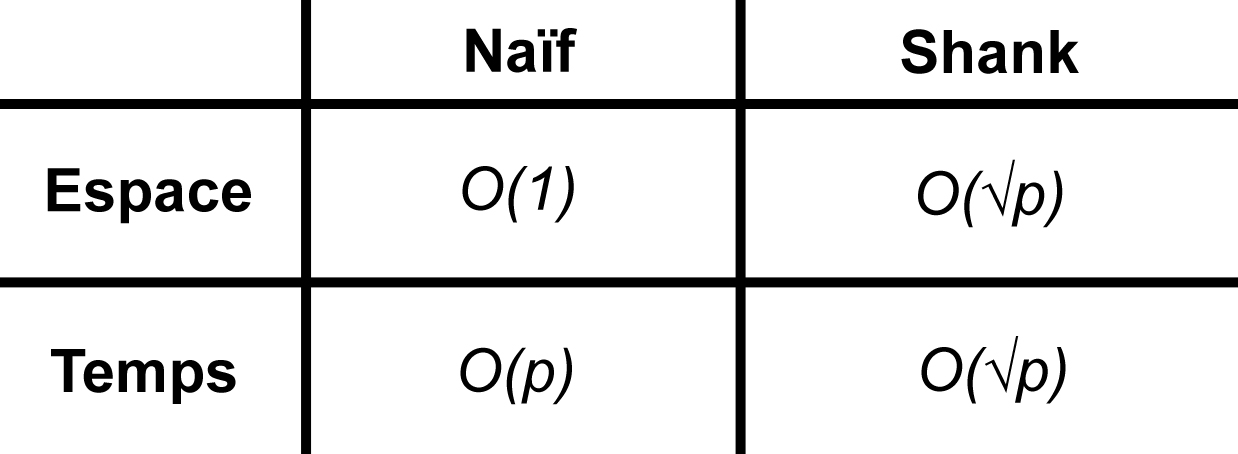
3 : A l’aide de l’algorithme d’Euclide étendu, on calcule la valeur .

4 : Cette phase, nommée « Giant-steps », consiste à calculer les valeurs successives , , , … tout en vérifiant à chaque étape si la valeur calculée correspond à une des valeurs stockées.

5 : Une fois que l’on trouve *q* tel que dans G, on en déduit que et donc .

D’après la division euclidienne de *k* par *n*, on sait qu’il existe un unique couple *(q,r)* tel que avec  ; ce qui justifie la terminaison de l’algorithme.

On peut donc passer d’une complexité temporelle en et spatiale en à une complexité temporelle en et spatiale en **à condition d’assurer le stockage et la recherche des valeurs en** . L’algorithme de Shank effectuerai un partage équitable des complexités temporelles et spatiales et, par conséquent, des ressources informatiques (processeur et mémoire vive).

Bien sur, l’algorithme est optimal lorsque l’on choisit . 

*Exemple :* Prenons , et . 