Cours IFT 3275

DEVOIR 1 : RAPPORT

Auteurs : Sophie Langensteiner et Siyu Liao 2024-11-03

**QUESTION 1 :**

**1.1)**

Le premier problème à résoudre est lire le résultat, car on a beau essaye plusieurs clefs de déchiffrements le résultat reste toujours sous format de valeur ascii en base 256, or on veut pouvoir vérifier rapidement si la clef utilise donne un texte correct / lisible. On va donc voir comment reverse ce premier processus de chiffrement, en première partie on regarde comment à partir du nombre Int en base 256 obtenir un tableau de valeur ascii puis en deuxième partie sous format string.

Première partie :

Cherchons comment rendre un nombre comme : 280991197550 (qui est Alain encode) sous format de tableau contenant chaque lettre sous forme valeur ascii (ici [65, 108, 97, 105, 110]) pour revenir sur le String de départ (ici "Alain")

On se rappelle du cours en 1015, ou on faisait modulo la base. Ici on est en base 256 utilisons donc % 256 sur 280991197550

On obtient la dernière lettre en valeur ascii : 110

Donc essayons de répéter %256 pour obtenir toute les lettres ascii ici on a Alain donc 5 lettres on répète l'opération 5 fois :

def int\_to\_str\_list(x):  
 list\_ascii = []  
 for i in range(5):  
 list\_ascii += [x % 256]  
 return list\_ascii

Cela retourne qu'un tableau de plusieurs 110, il faut qu'on mette à jour x. On se rappelle que pour passer de base 2 a base 10 on utilise la division entière //, pour mettre à jour le nombre à convertir. On utilise pareil mais pour //256 :

def int\_to\_str\_list(x):  
 list\_ascii = []  
 for i in range(5):  
 list\_ascii += [x % 256]  
 x //= 256  
 return list\_ascii

[110, 105, 97, 108, 65]

Le problème est l'ordre, on obtient l'ordre inverse de celui voulu, trouvons donc un moyen d'ajouter en début de liste pour cela on utilise insert :

list\_ascii.insert(0, x % 256)

Le problème est maintenant de savoir comment généraliser pour une longueur quelconque pas seulement un texte chiffre de 5 caractères pour cela regardons quand

def int\_to\_str\_list(x):  
 list\_ascii = []  
 for i in range(6):  
 list\_ascii.insert(0, x % 256)  
 x //= 256  
 return list\_ascii

On obtient 0 en fin de liste, donc il faut arrêter quand x%256 == 0

Soit cela revient à qu'on peut continuer à rechercher les caractères ascii tant que x est diffèrent de 0 :

def int\_to\_str\_list(x):  
 list\_ascii = []  
 while x % 256 != 0:  
 list\_ascii.insert(0, x % 256)  
 x //= 256  
 return list\_ascii

Deuxième partie :

Maintenant qu'on a notre liste de caractères ascii on applique sur chaque élément de la liste chr () qui est l'inverse de ord(), on prend la valeur ascii et on retourne son symbole.

def int\_to\_str(x):  
 # Recupere la liste des valeurs ascii composant notre nombre x  
 list\_ascii = int\_to\_str\_list(x)  
 message = ""

# Tant qu'on a des valeurs de caractères ascii à transformer :

for value in list\_ascii :  
 message += chr(value)  
return message

On peut maintenant à chaque essaie de clefs lire son équivalent en String. De ce principe on essaie dans un premier temps la force brute, mais en appliquant un regex qui prend que les caractères de l'alphabet et les accents pour ne pas avoir à lire chaque ligne. Cependant après la 1900000 clef on voit que cela devient de plus en plus long, il faut donc s'y prendre autrement.

L'attaque par la force brute semble déraisonnable.

Essayons donc d'attaque le cryptogramme à partir des formules du cours dans RSA.

Pour trouver décoder un message on utilise d, essayons donc de trouver d.

On a la formule

On connait e, mais on ne connait ni p ni q.

Cependant on sait que , avec p, q de très grands nombres premiers.

On connait N, on va donc voir pour factoriser en

Il faut donc que l'on trouve un outil calculatoire qui permette de calculer p et q dans un temps raisonnable on trouve alors[[1]](#footnote-1) :

Cela donne :

p =10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668073457

q =13393857589828341511855313113250022632017560146319170093046879854629388139061701531164979735196198226594933411469414335314839316071153925544980721968373218504918209718530288731776343256327963927347442727691308093729477426584248459448956929932596328643213995597108177709575537289565780483547741631653719

On a plus qu'à appliquer les fonctions/ formules :

Pour la formule le code python possède la fonction modinv on utilise donc pour trouver la clef d :

d = modinv(e, (p - 1) \* (q - 1))

On récupère la valeur en base 256 en appliquant grace à la fonction :

M\_value = modular\_pow(C, d, N)

Puis on passe des valeurs ascii en base 256 bits à un String contenant les symboles ascii :

M = int\_to\_str(M\_value)

On obtient comme personnage **Umberto Eco**.

**1.2)**

Vu que l'on fait la même chose pour la question 1.1 et 1.2 on crée une fonction pour éviter la répétition de code.

On applique le même procède pour la question 1.2 on obtient p et q grâce à l’outil calculatoire[[2]](#footnote-2)

p = 10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069673

q = 16072629107794009814226375735900027158421072175583004111656255825555265766874041837397975682235437871913920093763297202377807179285384710653976866362047862205901851662236346478131611907593556712816931273229569712475372911901098151338748315919115594371856794716529813251490644747478936580257043048672231

En appliquant decode(C,N,e,p,q) on obtient comme auteur célèbre **Marcel Proust**.

**QUESTION 2 :**

On fait 2 checks un des fréquences et un autre lors de l'association on vérifie que l'on ne met pas deux caractères ayant un espace à côté ensemble, car cela est très rare en français de laisser 2 espaces côte à côte donc par exemple "e\_ \_a" n'est pas accepté dans notre boucle. On s'attaque sur l'analyse des espaces car ce sont les caractères le plus utilises dans les textes en français ils donnent donc beaucoup d'informations en l'analysant.

On vérifie aussi quand on veut placer un saut de ligne que si l'élément d'avant forme un mot seule de forme espace caractère comme " a", si oui alors il faut vérifier si ce mot seule est correct, s’il est alors on peut placer notre symbole qui contient le saut de ligne

if "\n" in symbole:  
 for i in range(1, len(list\_modif)):  
 if list\_modif[i].binary == motif\_bin:  
 # Valeur précédente  
 pred = list\_modif[i - 1].symbol  
 if not (is\_alone\_letter\_correct(pred)):  
 return False

Correction après décryptage : On crée une fonction def correction afin de corriger des erreurs de décryptage de caractère : Notamment le cas lorsque on a placer deux symboles cote a cote séparément comme 's’et 'e' pour 'se' alors que cela n'est pas possible. Cela s'explique du fait que si 'se' aurait été vu il aurait été remplacé par un bicaractères 'se' et non code séparément. Cependant avant de déclarer que cela est vraiment une erreur il faut voir si le symbole précède forme un bicaractères avec celui actuel, car si oui alors on aurait juste encrypté les 2 caractères en 1 symbole et donc savoir que le prochain caractère forme aussi un bicaractères serait pas une erreur exemple 'ide' forme le bicaractères 'ss' donc e s'ajoute naturellement. Mais si le caractère précèdent était juste un caractère quelconque précédant celui actuel alors dans ce cas le fait que celui actuel ne soit pas associe au caractère suivant un bicaractères alors qu'il est reconnu comme tel, est une erreur et c'est ce qu'on va chercher à corriger exemple de cas à corriger 'pse' ‘ps’ ne forme pas un bicaractères car rarement utilise, cependant 'se' souvent utilise en langue français est un bicaractères.

Partir sur les bonnes associations dans le dictionnaire dès le départ, peut réduire les erreurs d'associations plus tard, notamment quand on vérifie la syntaxe avec les espaces, c'est pour cela que l'on rajoute une substitution se concentrant uniquement sur les premiers 9 symboles les plus fréquents, avant d'itérer sur tous les caractères. On utilise pour cela la librairie itérons, qui permet de simuler toutes les substitutions possibles pour ses 9 symboles, et l'on s'arrête lorsque l'on remarque qu'il n'y a pas d'erreurs grammaticales générales entre 4 symboles fréquents d'être sur les 9 premiers soit : "e ", "s ", "\r\n", et ", ". Une fois faits, nous pouvons continuer le décryptage pour les autres symboles.

**Reference :**

*“Prime Factors Decomposition.” Online Factorization Calculator. Accessed November 4, 2024. https://www.dcode.fr/prime-factors-decomposition.*

1. “Prime Factors Decomposition,” Online Factorization Calculator, accessed November 4, 2024, https://www.dcode.fr/prime-factors-decomposition. [↑](#footnote-ref-1)
2. Ibid [↑](#footnote-ref-2)