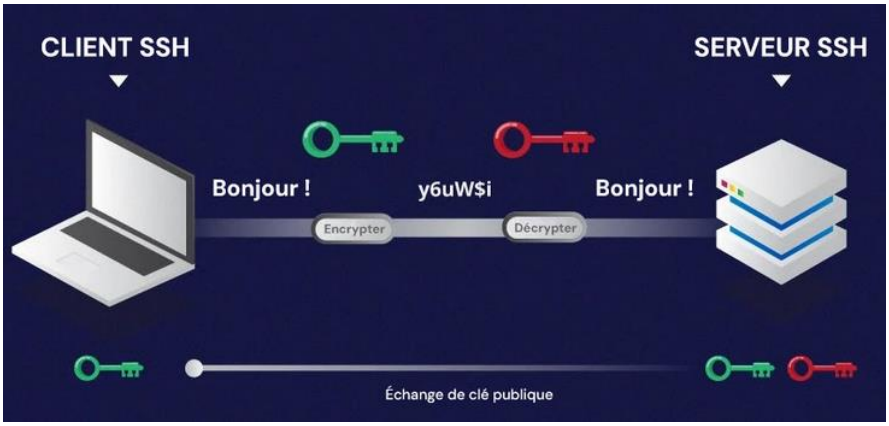
# Exercice - Le chiffrement RSA

Le **chiffrement RSA** (nommé par les initiales de ses trois inventeurs) est un algorithme de cryptographie **asymétrique**, très utilisé dans le commerce électronique, et plus généralement pour échanger des données confidentielles sur Internet. Il utilise une paire de clés (des nombres entiers) composée d'une clé publique pour chiffrer et d'une clé privée pour déchiffrer des données confidentielles.




## 1- RAPPEL DE QUELQUES NOTIONS D'ARITHMETIQUES :


Définition : Les nombres **premiers** sont des entiers naturels supérieurs ou égal à 2, qui n'ont pas d'autres diviseurs qu'eux-mêmes et 1. Dans cette famille, on a les entiers suivants : $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47$

Propriété : 2 nombres $n$ et $m$ sont **premiers entre eux** s'ils n'ont aucun diviseur en commun, autre que 1. Par exemple :

- 6 et 35 sont premiers entre eux : 6 = 2x3 et 35 = 7x5
- 6 et 27 ne sont pas premiers entre eux car ils ont 3 comme diviseur commun : 6 = 2x3 et 27 7x3

## 2- DECOUVERTE AVEC UN CAS SIMPLIFIE :

a- Génération d'une clé publique $(n, e)$ et d'une clé privée $(n, d)$ :

| Méthode | Application |
|---|---|
| 1- On choisit au hasard deux nombres premiers $p$ et $q$ | On peut choisir par exemple : $p = 11$ et $q = 17$ |
| 2- On calcule les entiers $n$ et $m$ suivants : $n = p\,q$ et $m = (p-1)(q-1)$ | On a alors : $n = 11 \times 17 = 187$ et $m = (11-1)(17-1) = 10 \times 16 = 160$ |
| 3- On choisit un entier $e$ appelé *exposant de chiffrement* qui respecte les conditions suivantes : $1 < e < m$ et $e$ **premier** avec $m$ | On peut choisir : $e = 3$ |
| 4- On choisit un entier $d$ appelé exposant de déchiffrement qui respecte la condition $(d \times e)\%m = 1$ | On peut choisir : $d = 107$ Car on a bien : $(107 \times 3)\%187 = 321\%187 = 1$ |

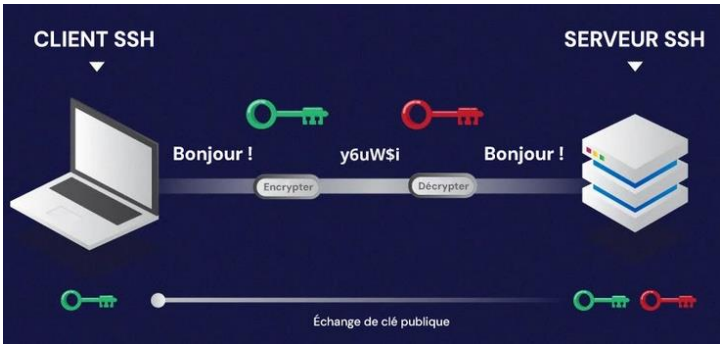
On suppose dans la suite que l'on utilise les clés publique $(n, e)$ et privé $(n, d)$ **du serveur**.


Lorsque le client réalise une demande au serveur (requête GET), celui-ci lui répond en lui envoyant « en clair » sa clé publique. Le client répond en utilisant cette clé pour chiffrer un premier envoi. Dans notre exemple ci-dessous, ce premier envoi sera « Bonjour ! ». Le client va le chiffrer en utilisant la clé publique $(n, e)$. Par simplicité, on ne traite ci-dessous que le chiffrement de la première lettre qui est ici « B ».

b- Chiffrement du caractère « B » en utilisant la clé publique $(n, e)$ : clé publique $(187,3)$

| Méthode | Application |
|---|---|
| 1- On convertit le caractère en nombre que l'on nommera ici $num$ en utilisant son code ASCII. En python, on peut utiliser la fonction *ord()* qui renvoie le code UNICODE du caractère : $num = ord(..)$ | `>>> ord("B")`<br>`66` |
| 2- On calcule l'entier naturel $C$ égal à $C = (num^e)\%n$ Le caractère est chiffré avec le nombre $C$ . | `>>> (66**3)%187`<br>`77` |

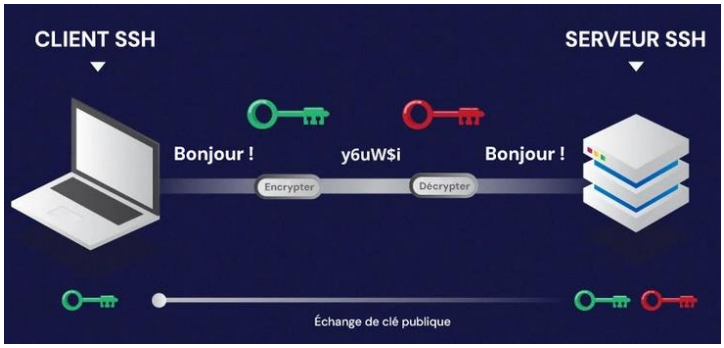
On suppose que cette valeur chiffrée $C$ est envoyée sur le réseau internet. Elle est reçue par le serveur qui va la déchiffrer en utilisant sa clé privée $(n, d)$ secrète. Seul le serveur la connait car il l'a générée lui-même.




c- Déchiffrement de la valeur chiffrée $C$ utilisant la clé privée $(n, d)$ : clé privée $(187,107)$

| Méthode | Application |
|---|---|
| 1- On calcule l'entier naturel $num$ égal à $num = (C^d)\%n$ Si la paire de clés public-privée est correcte, on retrouve le code UNICODE envoyé par le client. | `>>> (77**107)%187`<br>`66` |
| 2- On convertit ce code en caractère. En python, on utilise la fonction chr() qui renvoie le caractère en retour. | `>>> chr(66)`<br>`'B'` |

Le serveur a pu retrouver la lettre « B » avec sa clé privée qu'il est le seul à connaître. Le serveur est ainsi le seul à pourvoir déchiffrer un message qui a été chiffré avec sa clé publique.




## 3- LIMITES DE CETTE METHODE SIMPLIFIEE :

Si on utilise la même méthode sur tout le message « Bonjour ! », cela donne :

Chiffrement de « Bonjour ! » avec la clé publique $(n, e) = (187, 3)$

| | 'B' | 'o' | 'n' | 'j' | 'o' | 'u' | 'r' | ' ' | '!' |
|---|---|---|---|---|---|---|---|---|---|
| $num = ord(..)$ | 66 | 111 | 110 | 106 | 111 | 117 | 114 | 32 | 33 |
| $C = (num^e)\%n$ | 77 | 100 | 121 | 13 | 100 | 145 | 130 | 43 | 33 |

Déchiffrement de la suite de nombre chiffrés avec clé privée $(n, d) = (187, 107)$

| $C$ | 77 | 100 | 121 | 13 | 100 | 145 | 130 | 43 | 33 |
|---|---|---|---|---|---|---|---|---|---|
| $num = (C^d)\%n$ | 66 | 111 | 110 | 106 | 111 | 117 | 114 | 32 | 33 |
| caractère | 'B' | 'o' | 'n' | 'j' | 'o' | 'u' | 'r' | ' ' | '!' |

On peut mettre en avant les limites suivantes :

- Il y a plus de 100 000 codes UNICODE différents. Comme le caractère chiffré $C$ est le résultat d'un calcul modulo $n$ $(\%n)$, pour éviter que 2 caractères aient la même valeur chiffrée, il faut que $n$ soit au-moins supérieur à 100 000.

- Le caractère 'o' de Bonjour se retrouve 2 fois dans le mot 'Bonjour !'. En réalisant une analyse fréquentielle sur la liste de nombre chiffrés, il serait possible de casser ce chiffrement. Pour palier à ce problème, on peut fixer un nombre de chiffres alloué à chaque caractère et ensuite regrouper les caractères. Par exemple si on chiffre des groupes de 2 caractères, chacun représenté par un UNICODE complété à 3 chiffres, cela donnerait pour le mot 'Bonjour !' :

- Chiffrement de « Bonjour ! » avec la clé publique $(n, e) = (1592537, 16349)$

| | 'B' | 'o' | 'n' | 'j' | 'o' | 'u' | 'r' | ' ' | '!' |
|---|---|---|---|---|---|---|---|---|---|
| $num = ord(..)$ | 66 | 111 | 110 | 106 | 111 | 117 | 114 | 32 | 33 |
| | 66111 | | 110106 | | 111117 | | 114032 | | 33 |
| $C = (num^e)\%n$ | >>> (66111**16349)%1592537<br>546825<br>546825 | | >>> (110106**16349)%1592537<br>1025144<br>1025144 | | >>> (111117**16349)%1592537<br>68388<br>68388 | | >>> (114032**16349)%1592537<br>1459330<br>1459330 | | 33 |

Déchiffrement de la suite de nombre chiffrés avec clé privée $(n, d) = (1592537, 389)$

| $C$ | 546825 | | 1025144 | | 68388 | | 1459330 | | 33 |
|---|---|---|---|---|---|---|---|---|---|
| $num = (C^d)\%n$ | >>> (546825**389)%1592537<br>66111<br>66111 | | >>> (1025144**389)%1592537<br>110106<br>110106 | | >>> (68388**389)%1592537<br>111117<br>111117 | | >>> (1459330**389)%1592537<br>114032<br>114032 | | 33 |
| $num$ | 66 | 111 | 110 | 106 | 111 | 117 | 114 | 32 | 33 |
| caractère | 'B' | 'o' | 'n' | 'j' | 'o' | 'u' | 'r' | ' ' | '!' |

## 3- ROBUSTESSE ET FAILLES POSSIBLES :

Pour « casser » RSA il est nécessaire de pouvoir factoriser le nombre $n$ pour retrouver le produit initial des nombres *p* et *q*. Avec les algorithmes classiques, le temps que prend cette factorisation croît exponentiellement avec la longueur de la clé. Ce qui fait le succès du RSA est qu'il n'existe pas d'algorithme connu de la communauté scientifique pour réaliser une attaque force brute avec des ordinateurs classiques. On peut trouver la factorisation d'une clé de taille inférieure à 256 bits en quelques minutes sur un ordinateur individuel, en utilisant des logiciels librement disponibles. Pour une taille allant jusqu'à 512 bits, il faut faire travailler conjointement plusieurs centaines d'ordinateurs. Par sûreté, il est couramment recommandé que la taille des clés RSA soit au moins de 2 048 bits .

## 4- AUTRE EXERCICE DE CHIFFREMENT RSA SIMPLIFIE ET TRAITE A LA MAIN :

**Alice** décide de choisir $(n\,;\,e) = (21\,;\,5)$ comme clé publique.

a) Peut-elle faire ce choix ? Expliquer pourquoi ce choix de clé publique ne garantit pas la sécurité du message.

La valeur de $n$ est trop faible car on peut facilement factoriser pour retrouver $p = 3$ et $q = 7$

b) Rappeler les valeurs de $p, q, m.$ : $p = 3$ , $q = 7$ donc $m = 2 \times 6 = 12$

c) Parmi les clés suivantes déterminer celle qu'Alice peut choisir comme clé privée :

$(21\,;6)$ ; $(21\,;11)$ ; $(21\,;13)$ ; $(21\,;15)$ ; $(21\,;17)$ ; $(21\,;19)$.

On vérifie si la relation $(d \times e)\%m = 1$ est respectée. Si on teste pour chacune des valeurs de $d$ données, on constate que seule la valeur $d = 17$ fonctionne : $(17 \times 5)\%12 = 85\%12 = 1$

Dans la suite on admet qu'Alice choisit la seule clé privée possible parmi les propositions précédentes.

**Bob** veut envoyer à Alice le message (*M*) : PAL. Il va utiliser la clé publique (21 ; 5).

b) Associer à chaque lettre du message, sa **place** dans l'alphabet. Par exemple la lettre C est codée par 3. Obtenir ainsi trois entiers naturels $M_1, M_2, M_3$.

| Pour 'P' , on a $M_1 = 16$ | Pour 'A' , on a $M_2 = 1$ | Pour 'L' , on a $M_3 = 12$ |
|---|---|---|

c) Déterminer les 3 nombres $C_1, C_2, C_3$ qui chiffrent les nombres $M_1, M_2, M_3$.

| $C_1 = (16^5)\%21 = 4$ | $C_2 = (1)\%21 = 1$ | $C_3 = (12^5)\%21 = 3$ |
|---|---|---|

d) En déduire le message chiffré (*C*) que Bob envoie à Alice :

message envoyé constitué des nombres : 4 1 3

Alice reçoit le message ($C$) : 4 1 3. Elle veut déchiffrer ce message avec sa clé privé $(21\ ;17)$

e) Déterminer $M_1, M_2, M_3$ en déchiffrant les 3 nombres

| | | |
|---|---|---|
| $C_1 = (4^{17})\%21 = 16$ | $C_2 = (1^{17})\%21 = 1$ | $C_3 = (3^{17})\%21 = 12$ |

f) En déduire le message en clair que Bob a envoyé à Alice.
On retrouve les lettres initiales : 16ième lettre : 'P' , 1ère lettre : 'A' , 12ième lettre : 'L'

## 5- GENERATION DE CLES RSA SUR PYTHON :

La bibliothèque *PyCryptodome* est une bibliothèque Python qui fournit des fonctions et des algorithmes cryptographiques. Il s'agit d'un package Python autonome offrant une large gamme d'opérations cryptographiques, notamment le chiffrement, le déchiffrement, le hachage et la vérification de signature.

⇨ Ouvrir un fichier nommé *rsa.py* qui sera à uploader sur *nsibranly.fr*.

⇨ Installer la bibliothèque sur votre poste, en exécutant les commandes suivantes dans la console de pyzo :

- Mettre à jour pip car les ordinateurs du lycée Branly ne sont pas à jour. Pour cela exécuter :
```
>>> pip install --user --upgrade pip
```
- Installer la bibliothèque :
```
>>> pip install pycryptodome
```
- **Fermer** Pyzo et le **réouvrir** afin que l'installation soit prise en compte

⇨ Dans le fichier *rsa.py* , copier-coller le code ci-contre. Il permet de générer une paire de clés publique / privé qui sont écrites dans des fichier au format .pem.

```
from Crypto.PublicKey import RSA
# Génération de la clé
cle_rsa = RSA.generate(2048)
# Export des clés publique et privée
cle_privee = cle_rsa.exportKey()
cle_publique = cle_rsa.publickey().exportKey()
# Écriture des clés dans des fichiers
fd = open("cle_privee.pem", "wb")
fd.write(cle_privee)
fd.close()
fd = open("cle_publique.pem", "wb")
fd.write(cle_publique)
fd.close()
```

Exécuter ce script et répondre aux questions suivantes :

a) Exécuter dans la console la commande *type(cle_rsa)*
On constate que cette variable *cle_rsa* est une instance de la classe RSA qui contient une paire de clés de *2048* bits de longueur.

```
>>> type(cle_rsa)
<class 'Crypto.PublicKey.RSA.RsaKey'>
```

b) Exécuter dans la console la commande *type(cle_privee)* . On constate que cette variable est une instance de la class bytes. Les *bytes* sont des séquences immuables d'octets.

```
>>> type(cle_privee)
<class 'bytes'>
```

c) Exécuter dans la console `>>> cle_privee.decode("utf-8")`

puis
```
>>> cle_privee
```
puis
```
>>> cle_privee.hex()
```

Ces 3 exécutions permettent d'afficher les 2048 bits de la clé privée en format utf-8 ou hexadécimal.

## 6- UTILISATION CLES RSA SUR PYTHON :

⇨ Dans le fichier *rsa.py* , copier-coller à la suite le code ci-contre. Il permet d'utiliser les clés qui ont été générées pour chiffrer et déchiffrer un message.

Exécuter ce script et répondre aux questions suivantes :

Copier dans un fichier nommé *rsa.doc* qui sera à uploader sur nsibranly.fr en fin d'activité :

a) une copie d'écran du message en clair en hexadécimal (exécuter message.hex())

```
>>> message.hex()
'436563692065737420756e206d6573736167652064652074657374'
```

b) une copie d'écran du message chiffré en hexadécimal

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

#Lecture de la clé public
clePB = RSA.import_key(open('cle_publique.pem').read())

#Chiffrement d'un message avec la clé publique
message = b'Ceci est un message de test'
cipher = PKCS1_OAEP.new(clePB)
ciphertext = cipher.encrypt(message)

#Dechiffrement d'un message avec la clé privé
clePR = RSA.import_key(open('cle_privee.pem').read())
cipher = PKCS1_OAEP.new(clePR)
message_dechiffre = cipher.decrypt(ciphertext)
```

```
>>> ciphertext.hex()
'0472f877b7e0c864199f10f3a15e268827df6e68f662c0d1cdf2f391ea916
cb65eb051fc86219ceb23399ca21095e378b3a94e8d205b00a1d36a1b944f6
5e5ab6f844c1c33d422458baeea266565744abbc9695346f459975aab48649
23b0cab1c35fa782d6bcaf1a7cf8a0eb88144bb39d3e616696ee8ff467a3c2
9c55483aafa1f134ee54e378cbe528a14ec412049c2827775e0a7952021002
d1869790290458e001f453458525c0e1dd11d921605bab10e6050e58e14645
8782e211a5e4beedcc6cc6ec54b0e9c72c11616d07261cf2e77d145cc872d6
4baea28109d697383007ae8878f8ba0d35e09b415bc9d7d24c1928b943b1df
ff6c0f3b787934dc7'
```

c) une copie d'écran du message déchiffré en utf-8 (exécuter message.decode('utf-8'))

```
>>> message_dechiffre
b'Ceci est un message de test'
```

## 7- CLE PUBLIC DU SITE NSIBRANLY.FR :

Les clés publiques des sites sont consultables. Il suffit de cliquer sur le cadenas de la barre d'URL de la page du site. Par exemple pour *nsibranly.fr*, le click renvoie la fenêtre ci-contre. Continuer sur Connexion sécurisée – Plus d'informations – Afficher le certificat.
On retrouve sur la page, dans la rubrique « Informations sur la clé publique » la valeur de l'exposant $e$ et celle du module $n$.


a) Copier dans le fichier *rsa.doc* une copie d'écran de $e$ et de $n$

| | |
|---|---|
| Exposant | 65537 |
| Module | E2:14:52:61:01:52:21:F8:33:98:1D:2D:25:9A:8A:ED:07:73:B6:8A:38:00:B9:D3:0E:C5:BB:90:A4:A2:A2:BB:34:ED:3F:B6:B8:8D:7C:11:5A:8E:1D:6F:39:91:DC:E5:42:73:30:B0:E2:E0:95:71:DA:D2:77:61:00:FD:5C:10:19:49:F6:AC:FB:0C:17:36:F7:1B:1D:EF:47:50:DB:B0:5D:56:08:7A:7B:0F:B0:F7:E8:49:2E:36:98:14:E0:F3:EF:39:6F:12:1B:B6:BA:4E:2E:8A:CC:C8:81:21:34:15:47:17:5C:0A:75:CE:40:69:B0:D0:D7:1D:D3:1D:CD:60:77:7A:3E:6B:50:05:55:7E:34:27:04:17:5B:0E:87:50:61:4F:7D:E4:2C:02:30:51:61:27:08:ED:A3:5A:29:71:8B:2F:D9:30:AE:E9:0F:50:C9:B3:CB:54:7C:F2:43:D0:80:41:55:05:76:B3:DD:14:1C:E6:40:95:5E:C7:C8:61:5E:7C:09:D6:0F:A7:1E:F1:34:0E:E6:FB:3E:8F:C9:A2:0F:F6:BD:29:CB:DB:4E:A5:CC:D4:AB:4F:6E:77:96:73:13:F0:71:9B:65:D1:54:18:75:3C:A5:EF:2A:A0:F1:9A:C6:F6:31:9B:AB:0D:D9:73:46:77:E0:63:DD:65:47:53 |

b) Le nombre $n$ est donné en hexadécimal. Compter le nombre d'octets de $n$ . Cette clé a-t-elle vraiment une longueur de 2048 ?

Le nombre est défini avec 256 octets, ce qui donne 2048 bits précidément.

## 8- HACHAGE DES MOTS DE PASS :

Le hachage des mots de passe consiste à convertir les mots de passe en une chaîne alphanumérique à l'aide d'algorithmes spécialisés.

⇨ Dans le fichier *rsa.py* , copier-coller à la suite le code ci-contre.

Exécuter ce script et répondre dans le fichier nommé *rsa.doc* aux questions suivantes :

```
from Cryptodome.Hash import SHA256
#Le mot de pass est haché
mdpHach = SHA256.new(data=b'MonMotDePass')
mdpHach = mdpHach.hexdigest()
print("hach du mot de passe\n",mdpHach)

#Un utilisateur saisi un mot de passe
saisie = "MonMotDePass"
saisie = saisie.encode("utf-8")
hash_saisie = SHA256.new(data=saisie)
hash_saisie = hash_saisie.hexdigest()

if hash_saisie == mdpHach :
print("Le mot de pass saisi est le bon")
```

a) Donner une copie d'écran du mot de passe haché en hexadécimal

b) Quelle est l'utilité de ce code ?

Ce code permet de hacher un mot de passe et de conserver uniquement cette empreinte numérique. Pour vérifier la correspondance avec un autre mot de passe saisi, on comparera les 2 empreintes hachées.

c) Rechercher sur le net la réponse à la question suivante : « quelle différence y a-t-il entre le hachage et le chiffrement » ?

Le chiffrement se fait dans les 2 sens. On chiffre et on déchiffre. Le hachage ne se fait que dans un sens. Impossible à partir de l'empreinte hachée, de revenir au mot de départ.

```
>>> (executing file "hachage.py")
hach du mot de passe
 08b6335efd7d01fbc35789efb840b3ed63a6ed423128dc1c76fc95e9789f031b
```

## 9- IMPLEMENTATION D'UNE FONCTION SIMPLIFIEE RSA:

⇨ Télécharger sur *nsibranly.fr* le fichier *creerRsa.py* qui contient entre autres :

- La fonction *premiers(n)* qui retourne la liste de tous les nombres premiers inférieurs à n.
- La fonction cles($p, q$) qui retourne 2 tuples $(n, e)$ et $(n, d)$ qui correspondent à une paire de clé publique / privée calculée à partir des nombres premiers $p$ et $q$ qui sont en paramètre.

Les fonctions suivantes permettent de chiffrer ou déchiffrer un message. Elles sont incomplètes.

```
def chiffrement(n,e,message) :
    l = []
    for c in message :



    return l
```

```
def deChiffrement(n,d,l) :
    sortie = ""



    return sortie
```

En exécutant le programme principal :

```
# Main
(n,e),(n,d) = cles(21,37)
print(f"publique : {(n,e)}\nprivée : {(n,d)}\n")
message = "Tout va bien madame la marquise"
l = chiffrement(n,e,message)
print(l)
texteSortie = deChiffrement(n,d,l)
print(texteSortie)
```

On obtient dans la console :

```
>>> (executing file "outilsRsa.py")
publique : (777, 7)
privée : (777, 103)

[84, 111, 327, 536, 389, 34, 643, 389, 560, 672, 101, 110, 389, 760, 643
, 100, 643, 760, 101, 389, 255, 643, 389, 760, 643, 744, 239, 327, 672,
178, 101]
Tout va bien madame la marquise
```

⇨ Compléter les scripts des fonctions chiffrement() et dechiffrement().

```
def chiffrement(n,e,texte) :
    l = []
    for c in texte :
        num = ord(c)
        numChiffre = (num**e)%n
        l.append(numChiffre)
    return l
```

```
def deChiffrement(n,d,l) :
    sortie = ""
    for numChiffre in l :
        num = (numChiffre**d)%n
        c = chr(num)
        sortie = sortie + c
    return sortie
```

⇨ Uploader le fichier sur *nsibranly.fr* .