Chiffrement R.S.A.

R.S.A. correspond aux initiales de ses inventeurs qui l'ont inventé en 1977 :
Ron **R**ivest, Adi **S**hamir et Leonard **A**dleman.

RSA est un algorithme de chiffrement asymétrique qui sert aussi bien à la cryptographie de documents, qu'à l'authentification (signature numérique, code de carte bancaire, ...).

Parce-qu'il est basé sur une clé publique, et qu'il soit très sûr, l'algorithme RSA est devenu un standard dans le monde.

La clé de chiffrement est publique. Elle est constituée d'un couple de nombres :
(module de chiffrement, exposant de chiffrement)

La clé de déchiffrement est privée. Elles est aussi formée d'un couple de nombres :
(module de chiffrement, exposant de déchiffrement)

Les deux clés sont associées et uniques : elles ne peuvent pas être utilisées avec d'autres clés.
Tout le monde peut chiffrer un message avec la clé publique mais seul le détenteur de la clé privée peut le déchiffrer.

Tout le principe de RSA repose sur le fait qu'il est très difficile et très long de factoriser un très grand nombre en deux facteurs premiers : il n'existe pas d'algorithme connu de la communauté scientifique pour réaliser une attaque force brute avec des ordinateurs classiques.

Il faut donc choisir deux nombres premiers suffisamment grands. La taille recommandée pour les clés RSA est 2 048 bits (617 chiffres en base 10).
Record actuel : en décembre 2019, une équipe de l'Inria à Nancy et du Laboratoire lorrain de recherche en informatique et ses applications (Loria – Inria, CNRS), associée aux universités de Limoges et de San Diego (Californie) , ont "cassé" une clé à 795 bits (240 chiffres).
35 millions d'heures de calcul sur 3 centres ont été nécessaires.

I. Fonctions préliminaires

1ère étape : Clé de chiffrement

1) Écrire une fonction '`est_premier`' qui prend en paramètre un nombre '`n`' et permet de préciser si ce nombre est premier :

```
def est_premier(n) :
    ''' ...

:Examples:
>>> premier(17)
True
>>> premier(15)
False
>>> premier(1)
False
'''
```

2) Écrire une fonction '`premier_suivant`' qui prend en paramètre un nombre entier '`rang`' et renvoie le nombre premier immédiatement supérieur à '`rang`'

```
def premier_suivant(rang) :
    ''' ...

:Examples:
>>> premier_suivant(3)
5
>>> premier_suivant(20)
23
>>> premier_suivant(32)
37
'''
```

3) Écrire une fonction '`module_chiffrement`' qui prend en paramètre deux nombres entiers '`n`' et '`p`' et renvoie leur produit.

```
def module_chiffrement(n, p) :
    ''' ...

:Examples:
>>> module_chiffrement(3, 7)
21
'''
```

4) Écrire une fonction '`valeur_indicatrice_euler`' qui prend en paramètre deux nombres entiers '`n`' et '`p`' et renvoie le produit (`n`-1) (`p`-1).

```
def valeur_indicatrice_euler(n, p) :
    ''' ...

:Examples:
>>> valeur_indicatrice_euler(3, 7)
12
>>> valeur_indicatrice_euler(13, 23)
264
'''
```

5) Écrire une fonction '`liste_nombres_premiers`' qui prend en paramètre un nombre entier '`borne`' et renvoie la liste des nombres premiers inférieurs à '`borne`'.
On pourra se limiter à une liste de moins de 100 nombres.

```
def liste_nombres_premiers(borne) :
    ''' ...

:Examples:
>>> liste_nombres_premiers(10)
[2, 3, 5, 7]
>>> liste_nombres_premiers(20)
[2, 3, 5, 7, 11, 13, 17, 19]
'''
```

2è étape : Clé de déchiffrement

6) Écrire une fonction '`solution_diophante`' qui prend en paramètre deux nombres entiers, '`phi`' et '`e`'.
La fonction renvoie la solution '`d`' de l'équation suivante :

$$(e \times d) \bmod phi = 1$$

Le reste de la division de 'ed' par 'phi' est 1
mod = modulo

```
def solution_diophante(phi, e) :
    ''' ...

:Examples:
>>> solution_diophante(220, 17)
13
>>> solution_diophante(480, 13)
37
'''
```

II. Création des clés

Méthode :

1. Choisir p et q, deux nombres premiers distincts, assez grand
2. Module de chiffrement :

   Calculer leur produit n = p q
3. Valeur de l'indicatrice d'Euler de n :

   Calculer la valeur : $\varphi(n) = (p - 1)(q - 1)$
4. Exposant de chiffrement :

   Choisir un nombre premier e strictement inférieur à φ(n) et qui ne possède de facteur commun avec φ(n)
5. Exposant de déchiffrement :

   Calculer l'entier naturel d, inférieur à φ(n) tel que d est solution de : $ed \bmod \varphi(n) = 1$

Le couple **(n, e)** est la clé **publique** du chiffrement, alors que sa clé **privée** est le couple **(n,d)**.

Par sécurité, les autres nombres nécessaires à l'établissement de ces clés sont alors à effacer et oublier : les deux nombres premiers n et p, à la base de l'algorithme.

7) Utiliser les fonctions préliminaires pour créer une dernière fonction 'cles_RSA'.
Cette fonction prend en paramètre deux nombres entiers 'borne_inf' et 'borne_sup'.
Elle retourne deux clés (sous forme de deux tuples) établies à partir de deux nombres premiers choisis aléatoirement entre 'borne_inf' et 'borne_sup'

On ne doit pas connaître les 2 nombres premiers utilisés.

III. Chiffrement d'un message

Le chiffrement RSA permet de transformer un nombre '`x`' strictement plus petit que '`n`' en un autre nombre '`y`' , lui aussi strictement plus petit que '`n`' , en utilisant la clé de chiffrement (n, e).

8) Écrire une fonction '`fonction_RSA`' qui prend en paramètre un nombre entier '`x`' et une clé de chiffrement RSA de type tuple (n,e). Cette fonction renvoie la valeur $x^e \mod n$.

```
def fonction_RSA(x, cle) :
    ''' ...

:Examples:
>>> fonction_RSA(82, (253, 17))
190
>>> fonction_RSA(65, (8907,13))
2102
'''
```

Pour crypter un texte, on applique le chiffrement sur l'encodage UTF-8 des caractères utilisés. Si on prend un caractère à la fois, cela revient tout simplement à une substitution classique (donc facilement déchiffrable). L'idée est donc de grouper les valeurs d'encodages en formant des blocs de même longueur.

Exemple de texte : '1 million €

Caractères	code UTF-8 *(sur 4 chiffres, en base 10)*	Caractères	code UTF-8 *(sur 4 chiffres, en base 10)*
1	0049	i	0105
<espace>	0032	o	0111
m	0109	n	0110
i	0105	<espace>	0032
l	0108	€	8364
l	0108		

Codes UTF-8 assemblé puis redivisé par blocs de 3, en partant de la fin :
490 032 010 901 050 108 010 801 050 111 011 000 328 364

Après chiffrement avec la clé (2631,29), les valeurs deviennent :

490	$490^{29} \mod 2631 = 1705$	050	1628	050	1628	328	2092
032	$32^{29} \mod 2631 = 1613$	108	1791	111	1287	364	2377
010	$10^{29} \mod 2631 = 2590$	010	2590	011	869		
901	$901^{29} \mod 2631 = 1186$	801	795	000	0		

Texte chiffré : 1705 1613 2590 1186 1628 1791 2590 795 1628 1287 869 0 2092 2377

9) Écrire une fonction '`chiffrement_RSA`' qui prend en paramètre un '`texte`' et une clé de chiffrement RSA de type tuple.

```
def chiffrement_RSA(texte, cle) :
    ''' ...

:Examples:
>>> chiffrement_RSA('1 million €', (2631, 29))
'1705 1613 2590 1186 1628 1791 2590 795 1628 1287 869 0 2092 2377'
>>> chiffrement_RSA('NSI 2020', (316307,44503))
'73703 165375 147077 279034 283619 20211 244719 10149 297610 119422'
'''
```

IV. Déchiffrement du message

A l'inverse, le déchiffrement permet de transformer un nombre `y` strictement plus petit que `n` en un autre nombre `x` , lui aussi strictement plus petit que `n`, en utilisant la clé de déchiffrement (n, d).

$$y \xrightarrow[RSA]{} x = y^d \bmod n$$

Bien que le chiffrement RSA soit asymétrique (clés différentes), la fonction 'fonction_RSA' permet également le déchiffrement.

10) Écrire une fonction '`dechiffrement_RSA`' qui prend en paramètre un '`texte_chiffre`' et une clé de chiffrement RSA de type tuple.

```
def chiffrement_RSA(texte_chiffre, cle) :
    ''' ...

:Examples:
>>> dechiffrement_RSA('73703 165375 147077 279034 283619 20211 244719 10149 297610
119422', (316307, 7))
'NSI 2020'
>>> dechiffrement_RSA('457 1 8159 3533', (8907, 3653))
'Fin'
'''
```