CPGE MPSI 3
2016-2017

Le 14.12.16

Devoir Surveillé n°2
CORRECTION

Exercice - Cryptarithme

Le programme suivant `Test` répond à la question. Il utilise la force brute de l'ordinateur : il essaye les $\frac{10!}{3!} = 604\,800$ possibilités. Il renvoie la première solution qu'il trouve.
Pour cela, il exploite les sous-fonction `Essai` et `Liste_6` dont la définition est donnée dans le code. Pour la fonction `Liste_6` (entre autre) il est tout à fait possible d'écrire un programme plus court avec `not in`...

```
def Essai(L):
    """une liste de 6 chiffres est donnée,
    ces chiffres correspondent respectivement à U,N,E,F,O,Z
    on evalue si UN+UN+NEUF=ONZE
    """
    return(L[0]*10+L[1]+L[0]*10+L[1]+L[1]*1000+L[2]*100+L[0]*10+L[3]==L[4]*1000+L
        [1]*100+L[5]*10+L[2])

def Liste_6(L):
    """une liste de 6 chiffres est donnée,
    on évalue la différence des 6 chiffres
    Le programme renvoie True si tous les chiffres sont différents
    """
    a=True
    for i in range(len(L)-1):
        for j in range(i+1,len(L)):
            if L[i]==L[j]:
                a=False
    return(a)

def Test():
    M=[]
    for i in range(10):
        for j in range(10):
            for h in range(10):
                for k in range(10):
                    for l in range(10):
                        for m in range(10):
                            L=[i,j,h,k,l,m]
                            if Liste_6(L):
                                if Essai(L):
                                    M=M+[L]
    return(M)
```

Problème - Nombres pseudo-premiers et cryptage RSA

A. Nombres pseudo-premiers

Un nombre n est donné en argument. On cherche à savoir, si ce nombre est premier.

1.

```
def crible(n):
    """
    Pour un entier n (>=2), cette fonction renvoie la liste
    des nombres premiers inferieurs ou egaux a n. Cette liste
    est construite avec la methode du crible d'Erathosthene
    """
    liste = list(range(2,n+1))
    premiers = []
    while liste != []:
        p = liste[0]
        premiers.append(p)
        for n in liste:
            if n%p == 0:
                liste.remove(n)
    return(premiers)
```

(a) En ligne 7, le programme commence par écrire une première liste, nommé `liste`, elle contient tous les nombre de 2 à n. Cette liste sera vidée au fur et à mesure : par division par `p` (ligne 14).
En ligne 11, on ajoute à la liste `premiers` le premier nombre qui se présente dans la liste `liste`. S'il se trouve là, c'est qu'il n'est le multiple d'aucun des nombres premiers, il est donc bien premier. Il sera supprimé lors de l'exécution de la ligne 14.

(b) Lorsqu'il y a une boucle `while`, la terminaison d'un programme n'est pas toujours assuré.
Il faut trouver une suite d'entiers strictement positifs, appelée variant de boucle.
On note $\texttt{a}_k$`=len(liste)` lors du k^e passage dans la boucle.
A tout moment $\texttt{a}_k \in \mathbb{N}$.
Initialement $\texttt{a}_0 = n - 2 + 1 = n - 1$.
A chaque étape, au moins le premier terme de la liste est enlevé de `liste`
donc à chaque étape : $\texttt{a}_{k+1} < \texttt{a}_k$.
Ainsi après (au moins) k passage dans la boucle, $\texttt{a}_k < n - k$
et donc après n passage, `liste=[]`, la boucle s'arrête
et le programme se termine assurément.

(c) On note $T(n)$ le temps mis par un tel programme pour afficher le crible d'Erathostène pour les nombres entiers inférieurs à n.
On constate expérimentalement que pour tout $n \in \mathbb{N}$, $T(2n) = 4T(n)$, on considère donc que pour tout x réel positif, $T(2x) = 4T(x)$.
Soit $x > 0$,

$$\frac{T(x)}{x^2} = \frac{4T(\frac{x}{2})}{x^2} = \frac{T(\frac{x}{2})}{\left(\frac{x}{2}\right)^2}$$

De même, on a pour tout $k \in \mathbb{N}$, en prenant $X = \frac{x}{2^k}$:

$$\frac{T\left(\frac{x}{2^k}\right)}{\left(\frac{x}{2^k}\right)^2} = \frac{T(X)}{X^2} = \frac{T(\frac{X}{2})}{\left(\frac{X}{2}\right)^2} = \frac{T\left(\frac{x}{2^{k+1}}\right)}{\left(\frac{x}{2^{k+1}}\right)^2}$$

On a donc, par invariance : $\forall\ x > 0,\ \dfrac{T(x)}{x^2} = \dfrac{T\left(\frac{x}{2^k}\right)}{\left(\frac{x}{2^k}\right)^2}$.
En faisant tendre k vers l'infini et en exploitant le résultat de l'énoncé :

$$\frac{T(x)}{x^2} = \lim_{k\to+\infty} \frac{T\left(\frac{x}{2^k}\right)}{\left(\frac{x}{2^k}\right)^2} = \lim_{y\to 0} \frac{T(y)}{y^2} = A$$

Donc $\forall\ x > 0,\ T(x) = A \times x^2$.

(d) Pour écrire le crible d'Erathostène, on circule dans la liste `Liste` une première fois.
Et à chaque fois, lorsqu'on en est au nombre k de la liste `Liste`, on circule à nouveau dans une liste réduite possédant environ $n - k$ termes.
Le temps de calcul est donc proportionnel à

$$\sum_{k=2}^{n}(n-k) = \sum_{h=0}^{n} h = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

en posant $h = n - k$. On trouve donc $T(n) = O(n^2)$

La méthode précédente, exhaustive, n'est pas très efficace.

2. La contraposée de cet énoncé est :
Si il existe $a < n$ tel que $a^{n-1} \not\equiv 1[n]$, alors n n'est pas un nombre premier
On appelle témoins de Fermat de n les nombres $a \in [\![2, n]\!]$ tel que $a^{n-1} \not\equiv 1[n]$.
Avantage : il existe fréquemment de très petits témoins de Fermat.

3.
```
def Temoin_Fermat(a,n):
    """
    On teste si a^(n-1)=1[n].
    On peut faire une exponentiation rapide,
    mais on préfère faire les calculs modulo n
    """
    t=1
    for i in range(n-1):
        t=(t*a)%n  # à l'étape i, on calcul t^(i-1)[n]
    return(t!=1)
```

4. On dit qu'un nombre n est pseudo-premier, s'il n'admet aucun témoin de Fermat.

(a)
```
def Pseudo_premier(n):
    for a in range(2,4):
        if Temoin_Fermat(a,n):
            return(False)
    return(True)
```

(b) Pour `a`=2 et `a`=3, on effectue n calculs dans `Temoin_Fermat(a,n)` (une boucle de taille n).
On a donc $S(n) = O(n)$.

(c) Il est beaucoup plus rapide de trouver si un nombre n est pseudo-premier que de faire le crible jusqu'à n. (D'une complexité linéaire à quadratique.)

Malheureusement, il existe des nombres non premiers p et sans témoin de Fermat, c'est-à-dire qu'ils sont également pseudo-premiers. On appelle ces nombres, les nombres de Carmichaël.
Ce plus petit nombre est $561 = 3 \times 11 \times 17$.
On sait maintenant qu'il existe une infinité « rare » de tel nombre

B. RSA

RSA
En 1978, R. Rivest, A. Shamir et L. Adleman proposent un mécanisme adapté, exploitant le calcul des congruences.
Le principe est le suivant : l'interlocuteur i considère deux nombres premiers p_i et q_i et annonce publiquement le nombre $n_i = p_i \times q_i$, ainsi qu'un nombre r_i, premier avec le nombre $n_i' = (p_i - 1)(q_i - 1)$ (gardé secret).
L'application de chiffrement est alors $c_i : m \mapsto m^{r_i}[n_i]$.
Pour déchiffrer le message, l'interlocuteur i exploite le nombre s_i tel que $r_i \times s_i \equiv 1[n_i']$.
Plus précisément, $d_i : m' \mapsto (m')^{s_i}[n_i]$.
A l'heure actuelle, il n'existe pas d'algorithme « rapide et simple » pour trouver s_i connaissant r_i et n_i (pour p_i et q_i de grands nombres premiers).

1. Nous avons vu en cours que tout message informationnel se code numériquement : une image, un texte (codage ASCII)...
   Les premières difficultés pour les nombres réelles infiniment grands ou infiniment petits.

2.
```
def codage(m,r,n):
    """
    calcul de m^r[n]
    """
    t=1
    for i in range(r):
        t=(t*m)%n
    return(t)
```

3. Le programme complété devient :

```
def Bezout(a,b):
    """
    Renvoie sous forme de liste (triplet),
    1. le pgcd de a et b
    2. un nombre u
    3. un nombre v
    tels que pgcd(a,b)=au+bv
    """
    (...)

from random import randint

def choix(p,q):
    """
    Choisit r aléatoirement, premier avec n'=(p-1)(q-1)
    Il est renvoyé avec s tel que rs=1[n']
    """
```

```
18      n2=(p-1)*(q-1)
19      r=randint(2,n2-1)
20      L=Bezout(r,n2)
21      while L[0]!=1:
22          r=randint(2,n2-1)
23          L=Bezout(r,n2)
24      s=L[1]%n2
25      return(r,s)
```

Les lignes sont toutes les deux complétées par `L=Bezout(r,n2)`, la première fois (ligne 17), il s'agit de initialisation de la boucle retrouvée en ligne 23. Il s'agit de récupérer le couple (u, v) de Bézout à partir $(r, n2)$ avec $ur + vn2 = r \wedge n2$.

4.

```
1       def cle(n):
2       """
3       On trouve deux nombres pseudo-premiers d'au plus n chiffres
4       """
5       a=randint(2,10**n)
6       b=randint(2,10**n)
7       while Pseudo_premier(a)==False :
8           a=randint(2,10**n)
9       while Pseudo_premier(b)==False :
10          b=randint(2,10**n)
11      return(a,b)
```

Dans la pratique, on ne choisit pas p_i et q_i premiers, mais seulement pseudo-premiers (voir remarque plus loin).
Ecrire un algorithme qui choisit aléatoirement deux nombres pseudo-premiers de 4 chiffres.
On ne cherchera pas à montrer la terminaison de l'algorithme.
Pour 4 chiffres, Python effectue les calculs en 1 minute environ.

5. Chaque élève tire au sort des nombres p_i et q_i et partage, avec le reste de la classe, son nombre $n_i = p_i \times q_i$ et r_i obtenu grâce à la fonction `choix(`p_i`,`q_i`)` (il garde secrêtement auprès de lui s_i).
Lorsqu'un élève i veut communiquer un message à j, pour le coder (ou le décoder dans l'autre sens), il emploie alors la fonction `codage(`m, r_j, n_j (resp. `codage(`m', s_i, n_i`)`)