Informatique
Préparation à l'oral.

Divers - Corrigés

2016

1. ENSAM (PSI) ♠♠

(Planche ODT 2016 N°240 – Exercice II)

Dans cet exercice, on doit pouvoir tester si un entier naturel est un palindrome. Un objet Python de type `int` n'étant pas indexable, nous travaillerons avec la chaîne de caractères associée.

Nous pouvons, dans un premier temps, donner une fonction qui renvoie le booléen `True` si une chaîne de caractères est un palindrome et `False` sinon. Par exemple :

```
def IsPalindrome(s):
    l = len(s)
    if l == 0 or l == 1:
        return(True)
    else:
        if s[0] == s[l-1]:
            return(IsPalindrome(s[1:-1]))
        else:
            return(False)
```

Une telle fonction est assez naturellement récursive ! Dans le cas général, les premiers et derniers caractères sont identiques et on appelle la fonction avec une nouvelle chaîne `s[1:-1]` correspondant à la chaîne `s` initialement passée en argument et à laquelle on a retiré le premier et le dernier caractère (slicing).

Ensuite, on peut utiliser une fonction qui reçoit en argument une chaîne de caractères et renvoie la chaîne inversée associée. Par exemple :

```
def StrReverse(s):
    rs = ''
    for i in range(1,len(s)+1):
        rs += s[-i]
    return(rs)
```

Ceci étant, on peut pousser la technique du slicing encore plus loin en rappelant que `s[::-1]` fournira très exactement la chaîne `s` renversée… En lieu et place de la fonction `IsPalindrome`, on pourra alors simplement considérer la valeur logique de :

```
s == s[::-1]
```

ou de sa négation…

On a alors la fonction `degree` :

```
def degree(n):
    d = 0
    s = str(n)
    while not IsPalindrome(s):
        d += 1
        n += int(StrReverse(s))
        s = str(n)
    return(d)
```

ou cette version, utilisant `s[::-1]` :

```
def degree2(n):
    d = 0
    s = str(n)
    rs = s[::-1]
    while s != rs:
        d += 1
        n += int(rs)
        s = str(n)
        rs = s[::-1]
    return(d)
```

2. ENSAM (PSI) ♠

(BEOS - 2016 – PSI – Epreuve orale N°2833)

Dans ce petit exercice on va A PRIORI avoir besoin de calculer des factorielles ! Nous choisissons d'utiliser la fonction `factorial` du module `math` (bien sûr, on peut également récrire rapidement une petite fonction renvoyant la factoriel d'un entier naturel…).

Il convient également de décomposer un entier en ces chiffres. Une façon simple de le faire consiste à transformer l'entier en une chaîne de caractères (rappelons que les objets `str` sont indexables…).

Comme l'énoncé nous donne les trois premiers entiers égaux à la somme des factorielles de leurs chiffres (1,2 et 145), nous initialisons la recherche à 146. Nous utilisons une boucle `while` qui s'exécute tant que le 4ème entier n'a pas été trouvé. On peut donc envisager le code suivant :

```
from math import factorial

i = 146
s = 0

while i != s:
     i += 1
     s = 0
     c = str(i)
     for e in c:
          s += factorial(int(e))

print("Le 4ème entier naturel égal à la somme des
factorielles de ses chiffres est :",i)
```

Pour éviter l'utilisation de la fonction `factorial`, on peut placer les factorielles des entiers de 0 à 9 (leur calcul ne pose pas de problème… même sans calculatrice ! ☺) dans un tuple (inutile d'utiliser une liste puisque nous n'aurons pas besoin de modifier cet objet !). On obtient alors une autre version du code :

```
F = (1,1,2,6,24,120,720,5040,40320,362880)

while i != s:
     i += 1
     s = 0
     c = str(i)
     for e in c:
          s += F[int(e)]

print("Le 4ème entier naturel égal à la somme des
factorielles de ses chiffres est :",i)
```

3. ENSAM (PSI) ♠

(Planche ODT 2016 N°242 – Exercice II)

On peut construire la fonction `Binaire` à l'aide de la fonction Python `divmod` qui, rappelons-le, prend comme argument deux entiers *a* et *b* et renvoie un tuple de deux entiers correspondant respectivement au quotient et au reste de la division euclidienne de *a* par *b*. En effectuant la division euclidienne de l'entier n par 2 et en itérant avec les quotients successivement obtenus, on obtient l'écriture en base 2 de l'entier initial en considérant les restes obtenus. Par exemple avec 23, on obtient :

$$23 = 2\times 11 + \boxed{1}$$
$$11 = 2\times 5 + \boxed{1}$$
$$5 = 2\times 2 + \boxed{1}$$
$$2 = 2\times 1 + \boxed{0}$$
$$1 = 1\times 0 + \boxed{1}$$

et on a : $23_{10} = 10111_2$.

On peut alors proposer la fonction suivante :

```
def Binaire(n):
    L = []
    while n > 0:
        (n,r) = divmod(n,2)
        L.insert(0,r)
    return(L)
```

La fonction `Binaire` renvoie une liste d'entiers. Pour obtenir le nombre de 1, on peut écrire une fonction comme celle-ci :

```
def NombreDeUn(n):
    L = Binaire(n)
    UN = 0
    for e in L:
        if e:
            UN += 1
    return(UN)
```

ou utiliser plus simplement, puisqu'il n'y a que des 0 et des 1 dans la liste renvoyée par la fonction `Binaire`, la fonction `sum`. On a alors la fonction suivante :

```
def NombreDeUn2(n):
    return(sum(Binaire(n)))
```

Pour plus de détails, on pourra se reporter à l'exercice 1 ci-dessus : on peut proposer la fonction récursive suivante :

```
def Pal(L):
    l = len(L)
    if l == 0 or l == 1:
        return(True)
    else:
        if L[0] != L[-1]:
            return(False)
        else:
            return(Pal(L[1:-1]))
```

puis la fonction `Palindrome` suivante :

```
def Palindrome(n):
    L = Binaire(n)
    return(Pal(L))
```

ou bien la fonction `Palindrome2` suivante en lieu et place des deux précédentes :

```
def Palindrome2(n)
    L = Binaire(n)
    return(L == L[::-1])
```

Pour afficher tous les entiers 2-palindromes inférieurs à 100, on peut considérer les instructions suivantes :

```
for i in range(101):
    if Palindrome2(i):
        print(i,Binaire(i))
```

2015

1. D'après ENSAM (PSI) ♠

1) L'instruction `list(str(n))` renvoie une liste contenant les chiffres de l'entier *n* en tant que chaîne de caractères. Par exemple `list(str(3972005))` renverra la liste :

```
['3','9','7','2','0','0','5']
```

2) On utilise la fonction précédente :

```
def somme(n):
    S = 0
    L = list(str(n))
    for e in L:
        S += int(e)
    return(S)
```

3) Il suffit de tester si le reste de la division euclidienne de `somme(n)` par 10 est nul ou pas. Ce reste étant obtenu grâce à `somme(n)%10`, on a, par exemple :

```
def test(n):
    return(somme(n)%10 == 0)
```

4) Soit *un* le chiffre des unités de *n* et *usn* celui de `somme(n)`. Pour obtenir un nombre adéquat *n'*, on a deux possibilités :
   - Ajouter $10 - usn$ à n.
   - Soustraire *usn* à *n*.

   Evidemment, on choisira l'une ou l'autre de ces deux possibilités selon que *un* est supérieur ou égal à *usn* (dans ce cas, on choisit la deuxième possibilité) ou non (dans ce cas, on choisit la première possibilité). On obtient ainsi le code :

```
def modification(n):
    if test(n):
        return(n)
    else:
        un = n%10
        usn = somme(n)%10
        if un >= usn:
            return(n - usn)
        else:
            return(n + 10 - usn)
```

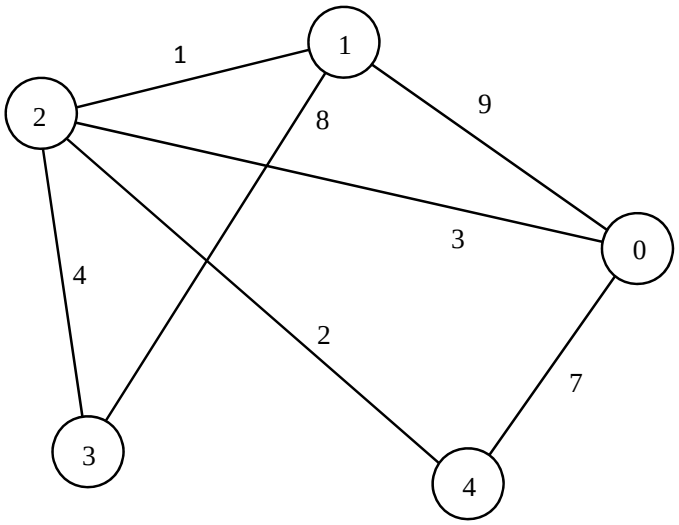
5) On a simplement la boucle :

```
for i in range(10):
    n = randint(10000,100000)
    n2 = modification(n)
    print("Nombre aléatoire :",n,"(somme des
    chiffres :",somme(n),")","/ nombre modifié
    :",n2,"(somme des chiffres
    :",somme(n2),")")
```

2. E3A – 2015 - Exercice type N°2 (PSI) ♠♠

Dans cet exercice, on utilise la bibliothèque `numpy` et les objets `array`. Ainsi, l'indexation des sommets du graphe (de 1 à 4) correspond à celle des éléments des tableaux utilisés. Cette bibliothèque sera classiquement importée dans le script via l'instruction :

```
import numpy as np
```



1. On obtient naturellement une matrice symétrique (le graphe n'est pas un graphe orienté : une arête est indifféremment parcouru dans un sens ou dans l'autre).

   On obtient immédiatement (cette ligne apparaîtra dans le script) :

   ```
   M = np.array([[0,9,3,-1,7],[9,0,1,8,-1],[3,1,0,4,2],[-1,8,4,0,-1],[7,-1,2,-1,0]])
   ```

2. Pour la fonction **voisins**, on peut indifféremment balayer la colonne ou la ligne d'indice *i*. Dans un premier temps, on récupère la dimension de la matrice M grâce à la méthode `shape`. Ensuite, on construit la liste des indices *j* tels que $M_{ij} \neq 0$ (un voisin d'un sommet n'est pas le sommet lui-même) et $M_{ij} \neq -1$ (un voisin d'un sommet lui est... relié !).

D'où le code possible :

```
def voisins(i):
    global M
    n = M.shape[0]
    L = []
    for j in range(n):
       if M[i,j] != -1 and M[i,j] != 0:
          L.append(j)
```

3. Le nombre de voisins d'un sommet donné n'est rien d'autre que la longueur de la liste renvoyée par la fonction **voisins**. On pourrait donc proposer le code très compact suivant :

```
def degre(i):
    global M
    return(len(voisins(i))
```

L'inconvénient de ce code est qu'il construit une liste, éventuellement très longue si on manipule un graphe comportant un grand nombre de sommets, dont nous n'avons absolument pas besoin ! On lui préfèrera donc le code suivant, correspondant au code de la fonction **voisins** légèrement modifié :

```
def degre(i):
    global M
    n = M.shape[0]
    N = 0
    for j in range(n):
        if M[i,j] != -1 and M[i,j] != 0:
            N += 1
    return(N)
```

4. En toute rigueur, on devra vérifier que la liste `L` fournie en argument de la fonction **longueur** n'est pas vide. Nous nous plaçons dans la situation générale où L contient au moins un élément. Pour tout indice *i* variant de 0 à `len(L)-2`, on s'intéresse aux sommets d'indices `L[i]` et `L[i+1]`. Deux situations sont possibles : il existe, ou pas, une arête entre ces deux sommets. S'il n'existe pas d'arête, on interrompt l'exécution de la boucle puisque la liste L ne correspond pas à la liste des sommets d'une chaîne du graphe G.

Ainsi, on travaillera plutôt avec une boucle `while` comme dans le code suivant :

```
def longueur(L):
    global M
    longueur = 0
    chaine = True
    i = 0
    while chaine == True and i <= len(L)-2:
        p = M[L[i],L[i+1]]
        if p != -1:
            longueur += p
        else:
            chaine = False
        i += 1
    if chaine:
        return(longueur)
    else:
        return(-1)
```

3. D'après ENSAM 2015 (PT) ♠

1. Il y a une imprécision dans l'énoncé : dans quel sens la liste doit-elle être lue ? Nous prenons ici le parti suivant qui simplifie la manipulation des listes : l'indice d'un élément correspondra à l'exposant de la puissance de 2 correspondante. Ainsi, [0,0,1] sera le codage binaire de $4 = 0\times 2^0 + 0\times 2^1 + 1\times 2^2$ (qui est traditionnellement écrit 100).
   On procède à des divisions euclidiennes successives de *n* jusqu'à obtenir un quotient nul. Comme nous avons besoin, à chaque itération, du reste (qui est ajouté à la liste) et du quotient (sur lequel on itère), on a intérêt à utiliser la fonction `divmod` :

```
def Binaire(n):
    if n == 0:
        return[0]
    else:
        L = []
        q = n
        while q != 0:
            (q,r) = divmod(q,2)
            L.append(r)
        return(L)
```

2. Avec la méthode `count`, la fonction s'écrit très simplement :

```
def NombreDeUn(n):
    return(Binaire(n).count(1))
```

   Sinon, on peut procéder plus classiquement en balayant les éléments de la liste correspondant à l'écriture binaire de l'entier fourni en argument :

```
def NombreDeUn(n):
    L = Binaire(n)
    N = 0
    for e in L:
```

```
            if e == 1:
                N += 1
    return(N)
```

3. Avec la méthode `reverse`, la fonction s'écrit, ici encore, très simplement :

```
def Palindrome(n):
    L = Binaire(n)
    L2 = list(L)
    L2.reverse()
    return(L == L2)
```

   Ce codage requiert de créer une copie de la liste L car la méthode `reverse` effectue une inversion sur place de la liste.

   Sinon, on procéder plus classiquement en balayant les éléments de la liste simultanément de la gauche vers la droite (indice `k` allant de 0 à…) et de la droite vers la gauche (indice `-1-k` allant de -1 à…).

4. Déterminer tous les 2-palindromes inférieurs à un entier N passé en argument (à chaque fois, on affichera le nombre et son écriture en base 2 sous forme d'une liste).

4. E3A – 2015 - Exercice type N°7 (PSI) ♠

Soit n un entier naturel $n \leq 26$. On souhaite écrire un programme qui code un mot en décalant chaque lettre de l'alphabet de n lettres.
Par exemple pour $n = 3$, le décalage sera le suivant :

Avant décalage	*a*	*b*	*c*	*d*	…	…	*x*	*y*	*z*
Après décalage	*d*	*e*	*f*	*g*	…	…	*a*	*b*	*c*

Le mot `oralensam` devient `rudohqvdp`.

1. En appelant ALPHA la chaîne demandée, on a :

```
ALPHA = 'abcdefghijklmnopqrstuvwxyz'
```

2. On construit la chaîne de caractères demandée de la façon suivante :

   - D'abord on considère la sous-chaîne de ALPHA de premier caractère le caractère d'indice n. Cette sous-chaîne est obtenue en considérant `ALPHA[n:]`.
   - Ensuite, on considère la chaîne constituée des n premiers caractères de la chaîne ALPHA et on l'ajoute à la fin de la sous-chaîne précédente. Cette autre sous-chaîne est obtenue en considérant `ALPHA[:n]`.

   Ces deux opérations conduisent à la fonction simple suivante :

```
def decalage(n):
    global ALPHA
    return(ALPHA[n:]+ALPHA[:n])
```

3. En utilisant un compteur `i` variant de `0` à `len(phrase)-1`, il vient :

```
def indices(x,phrase):
    L = []
    for i in range(len(phrase)):
        if phrase[i] == x:
            L.append(i)
    return(L)
```

4. On a le code possible suivant :

```
def codage(n,phrase):
   global ALPHA
   # on construit ALPHA2 correspondant à ALPHA avec un décalage de n.
   ALPHA2 = decalage(n)
   # on convertit phrase en une liste
   Lphrase = list(phrase)
   # on balaie ALPHA.
   for i in range(len(ALPHA)):
      # Pour chaque lettre de l'alphabet, on construit la liste des
      # (éventuels) indices de cette lettre dans phrase.
      L = indices(ALPHA[i],phrase)
      # Si cette liste d'indices est non vide...
      if L != []:
            # Pour chaque on remplace chaque élément de Lphrase par
            sa lettre
            # codée correspondante
      for e in L:
            Lphrase[e] = ALPHA2[i]
   # On construit la chaîne codée à partir de Lphrase qui est ici
   vidée pour
   # optimiser l'occupation mémoire
   phrase2 = ""
   while Lphrase != []:
      phrase2 += Lphrase.pop(0)
   # renvoi du résultat
   return(phrase2)
```

5. On remarque que l'on a la propriété élémentaire suivante :

$$\begin{aligned}\texttt{codage(n,codage(m,phrase)} &= \texttt{codage(m,codage(n,phrase))}\\ &= \texttt{codage(n+m,phrase)}\end{aligned}$$

Par ailleurs, un décalage d'un multiple de 26 va laisser la chaîne de caractères `phrase` invariante. Ainsi, si on a codé phrase avec un décalage égal à n ($0 \leq n \leq 26$), on pourra décoder le texte codé en le codant à l'aide d'un décalage égal à $26 - n$.

5. D'après ENSAM 2015 (PSI) ♠♠

Corrigé à venir…