

Informatique

Préparation à l'oral.

Divers - Corrigés

2017

1. Centrale PC ♠

(Planche Officiel de la Taupe Gyroscope 2017 N° 158)

1. Il n'existe qu'une partition de $\llbracket 1; 1 \rrbracket = \{1\}$, c'est l'identité et... elle admet un point fixe ! ☺

On a immédiatement : $\boxed{d_1 = 0}$.

Comme $2! = 2$, il existe 2 partitions de $\llbracket 1; 2 \rrbracket$: $\begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$ (l'identité) et $\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$.

Seule la seconde n'admet pas de point fixe : $\boxed{d_2 = 1}$.

Pour $n = 3$, il existe $3! = 6$ permutations de $\llbracket 1; 3 \rrbracket$.

Notons que si l'un d'elles admet 2 points fixes alors, nécessairement, le troisième en est également un et la permutation considérée est l'identité. Ainsi, une permutation de $\llbracket 1; 3 \rrbracket$ peut admettre 3 points fixe (c'est l'identité) ou un seul, ce sont les trois permutations :

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} \text{ et } \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$$

Les deux dernières permutations sont le cycle : $\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$ et son carré : $\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$.

Ils n'admettent pas de points fixes : $\boxed{d_3 = 2}$.

Considérons maintenant l'ensemble $\llbracket 1; n \rrbracket$.

Nous savons qu'il existe $n!$ permutations de cet ensemble.

Une telle permutation admet k points fixes avec $k \in \llbracket 0; n \rrbracket$.

Il y a $\binom{n}{k}$ façons de choisir les k points fixes considérés.

Une fois les k points fixes choisis, il convient de compter le nombre de permutations sans points fixes des $n-k$ valeurs restantes. Par définition, il y en a d_{n-k} .

Ainsi, il y a $\binom{n}{k} d_{n-k}$ permutations de $\llbracket 1; n \rrbracket$ admettant exactement k points fixes.

En faisant varier k dans $\llbracket 0; n \rrbracket$, on obtient finalement le résultat cherché : il existe au total

$\sum_{k=0}^n \binom{n}{k} d_{n-k}$ permutations de $\llbracket 1; n \rrbracket$ et donc $\sum_{k=0}^n \binom{n}{k} d_{n-k} = n!$.

$$\sum_{k=0}^n \binom{n}{k} d_{n-k} = n!$$

2. Soit x un réel de l'intervalle $]-1; 1[$. On veut montrer que la série $\sum \frac{d_n}{n!} x^n$ converge.

On a, pour tout entier naturel n : $\frac{d_n}{n!} \geq 0$ et, par définition de d_n : $\frac{d_n}{n!} \leq 1$. Alors, pour tout

réel x dans $]-1; 1[$ et tout entier naturel n : $0 \leq \frac{d_n}{n!} |x|^n \leq |x|^n$.

Comme $|x| < 1$, la série $\sum |x|^n$ est une série géométrique convergente. Il en va donc de même pour la série $\sum \frac{d_n}{n!} |x|^n$. Ainsi, la série $\sum \frac{d_n}{n!} x^n$ est absolument convergente. Elle est donc convergente.

On sait que la fonction $x \mapsto \frac{1}{1-x}$ est développable en série entière sur l'intervalle $]-1; 1[$

avec : $\forall x \in]-1; 1[, \frac{1}{1-x} = \sum_{n=0}^{+\infty} x^n$.

Par ailleurs, d'après le résultat précédent, pour tout réel x de l'intervalle $]-1; 1[$, on peut

effectuer le produit de Cauchy des séries entières $e^x = \sum_{n=0}^{+\infty} \frac{x^n}{n!}$ et $f(x) = \sum_{n=0}^{+\infty} \frac{d_n}{n!} x^n$ et on

obtient :

$$\begin{aligned} e^x f(x) &= \sum_{n=0}^{+\infty} \left(\sum_{k=0}^n \frac{1}{k!} \times \frac{d_{n-k}}{(n-k)!} \right) x^n = \sum_{n=0}^{+\infty} \left(\frac{1}{n!} \sum_{k=0}^n \frac{n!}{k! \times (n-k)!} \times d_{n-k} \right) x^n \\ &= \sum_{n=0}^{+\infty} \left(\frac{1}{n!} \sum_{k=0}^n \frac{n!}{k! \times (n-k)!} \times d_{n-k} \right) x^n = \sum_{n=0}^{+\infty} \left(\frac{1}{n!} \underbrace{\sum_{k=0}^n \binom{n}{k} d_{n-k}}_{=n!} \right) x^n \\ &= \sum_{n=0}^{+\infty} x^n \end{aligned}$$

On a ainsi : $\forall x \in]-1; 1[, e^x f(x) = \frac{1}{1-x}$.

3. D'après la question précédente : $\forall x \in]-1; 1[, f(x) = \frac{1}{1-x} \times e^{-x}$.

On effectue encore un produit de Cauchy avec $\forall x \in]-1; 1[, \frac{1}{1-x} = \sum_{n=0}^{+\infty} x^n$ et

$$e^{-x} = \sum_{n=0}^{+\infty} \frac{(-x)^n}{n!} = \sum_{n=0}^{+\infty} \frac{(-1)^n}{n!} x^n.$$

Il vient immédiatement : $\frac{1}{1-x} \times e^{-x} = \sum_{n=0}^{+\infty} \left(\sum_{k=0}^n \frac{(-1)^k}{k!} \times 1 \right) x^n = \sum_{n=0}^{+\infty} \left(\sum_{k=0}^n \frac{(-1)^k}{k!} \right) x^n$.

Or $\forall x \in]-1; 1[, f(x) = \sum_{n=0}^{+\infty} \frac{d_n}{n!} x^n$. On en tire : $\forall n \in \mathbb{N}, \sum_{k=0}^n \frac{(-1)^k}{k!} = \frac{d_n}{n!}$, soit, finalement :

$$\forall n \in \mathbb{N}, d_n = n! \sum_{k=0}^n \frac{(-1)^k}{k!}$$

4. Dans cette question, on va devoir écrire une fonction permettant de construire la matrice D_n sous forme d'un tableau numpy. Pour ce faire, nous allons, dans un premier temps, écrire une fonction `dn` qui recevra en argument un entier `n` (nous n'incluons pas ici de test sur l'argument mais il conviendrait de le faire dans un programme complètement « opérationnel » puisque cet entier doit être positif) et renverra un entier correspondant à d_n . Sur ce point, on doit être prudent. Le codage direct du résultat obtenu à la question précédente (une factorielle que multiplie une somme) ne donnera pas un entier mais un flottant et des phénomènes d'arrondis surviendront qui seront tels que la valeur entière ou le transtypage du résultat ne donnera pas forcément la valeur exacte de d_n . Il convient donc d'utiliser la deuxième somme ci-dessous :

$$\forall n \in \mathbb{N}, d_n = \sum_{k=0}^n \frac{n!}{k!} \times (-1)^k = (-1)^n + \sum_{k=0}^{n-1} n \times (n-1) \times \dots \times (k+1) \times (-1)^k$$

En effet, la première somme, si elle améliore la situation, requiert nombre de calculs inutiles...

Notons enfin que l'on a intérêt à faire varier `k` de `n-1` à 0 pour, une fois encore, limiter le nombre de multiplications :

```
def dn(n):
    d = 0 # Variable correspondant au résultat du calcul
    r = 1 # Variable correspondant au produit n x (n-1) x ... x (k+1)
    for k in range(n, -1, -1):
        d += (-1)**k * r
        r *= k
    return(d)
```

Nous allons maintenant donner une fonction D_n qui recevra en argument un entier n et renverra un tableau numpy correspondant à la matrice D_n . Cette fonction fera naturellement appel à la fonction dn .

En supposant que le fichier contienne l'importation classique :

```
import numpy as np
```

on peut considérer (attention aux indices qui démarrent à 0...) la fonction suivante :

```
def Dn(n):
    D = np.zeros((n+1,n+1))
    for i in range(n+1):
        for j in range(n+1):
            D[i,j] = dn(i+j)
    return(D)
```

Pour « établir » le résultat demandé (nous ne pouvons en fait que comparer la valeur de l'expression fournie dans l'énoncé avec la valeur du déterminant calculé par numpy que pour quelques valeurs de l'entier $n...$), nous allons utiliser la fonction `det` du module `numpy.linalg` (notre fichier comportera : `from numpy.linalg import det`).

Ici encore, il serait très malvenu de calculer $\left(\prod_{k=0}^n k!\right)^2$ pour différentes valeurs de n en utilisant de nombreuses fois la fonction `factorial` du module `math`. Il vaut mieux calculer les factorielles au fur et à mesure puisque $(k+1)! = (k+1) \times k!$

On peut donc fournir le premier script :

```
for n in range(8):
    # Calcul du produit des factorielles au carré (Variable PFC)
    PFC = 1
    fact = 1
    for k in range(n+1):
        PFC *= fact
        fact *= k+1
    PFC = PFC**2
    # Calcul du déterminant de la matrice Dn
    # ATTENTION ! La fonction det de numpy.linalg renvoie un flottant !
    d = det(Dn(n))
    print("n = ",n," , énoncé : ",PFC," , déterminant : ",int(d+0.5))
```

Ce premier script ne tire cependant pas parti de la relation de récurrence simple :

$$H_{n+1} = \left(\prod_{k=0}^{n+1} k!\right)^2 = \left((n+1)! \times \left(\prod_{k=0}^n k!\right)\right)^2 = ((n+1)!)^2 \times H_n$$

On met en œuvre cette relation dans le second script :

```
PFC = 1
fact2 = 1
for n in range(8):
    # Calcul du déterminant de la matrice Dn
    # ATTENTION ! La fonction det de numpy.linalg renvoie un flottant !
    d = det(Dn(n))
    print("n = ",n," , énoncé :",PFC," , déterminant :",int(d+0.5))
    # Mise à jour de la variable fact2
    fact2 *= (n+1)**2
    # Mise à jour de la variable PFC
    PFC *= fact2
```

2. D'après ENSAM PSI ♠

(Planche Revue de Mathématiques Spéciales 2017 N°1129)

- Pour n entier, l'instruction `list(str(n))` renvoie la liste des caractères correspondant aux chiffres de l'entier.
Par exemple, `list(str(281))` renvoie la liste `['2', '8', '1']`.
- Par exemple :

```
def facto(n):
    F = 1
    for i in range(1,n+1):
        F *= i
    return(F)
```

- Pour tout entier naturel n non nul, $\sigma(n)$ est la somme des factorielles des chiffres de n et $\Sigma = \{n \in \mathbb{N} / \sigma(n) = n\}$.

- L'entier n comportant p chiffres, la somme des factorielles de ses chiffres est majorée par $\underbrace{9!+9!+\dots+9!}_{p \text{ termes égaux à } 9!}$, c'est-à-dire $p \times 9!$. On a bien $\boxed{\sigma(n) \leq p \times 9!}$.

Par ailleurs, le plus petit entier naturel comportant p chiffres est $\underbrace{1000\dots0}_{p-1 \text{ chiffres "0"}} = 10^{p-1}$

et on a immédiatement : $\boxed{n \geq 10^{p-1}}$.

- Pour tout entier naturel non nul p , on a :

$$\frac{u(p+1)}{u(p)} = \frac{10^p}{(p+1) \times 9!} = \frac{10^p}{(p+1) \times 9!} \times \frac{p \times 9!}{10^{p-1}} = \frac{10p}{p+1}$$

Or, le rapport $\frac{p}{p+1}$ est croissant et vaut $\frac{1}{2}$ pour $p = 1$. On en déduit :

$$\forall n \in \mathbb{N}^*, \frac{u(p+1)}{u(p)} \geq 5 > 1$$

La suite u est strictement croissante.

Pour calculer des termes de la suite u , on peut utiliser la fonction :

```
def u(p):
    return(10**(p-1)/(p*facto(9)))
```

On peut alors considérer le script suivant :

```
p = 1
up = u(1)
while up <= 1:
    print("u( ", p, " ) = ", up)
    p += 1
    up = u(p)
```

La dernière valeur affichée est celle de $u(7)$ et on en déduit que :

Le plus petit entier p tel que $u(p) > 1$ est 8.

c. D'après la question précédente, on a $p \geq 8 \Rightarrow u(p) > 1$.

D'après la question 3.a, on a : $\forall n \in \mathbb{N}^*, \sigma(n) \leq p \times 9!$ et $n \geq 10^{p-1}$.

On en déduit $\frac{n}{\sigma(n)} \geq \frac{10^{p-1}}{p \times 9!} = u(p)$.

Ainsi, si $n \geq 10\,000\,000$ (c'est-à-dire $p \geq 8$), on aura $u(p) > 1$ et donc n ne sera pas dans Σ .

Pour déterminer Σ , on peut donc tester l'égalité $\sigma(n) = n$ pour tous les entiers compris entre 1 et 9 999 999 à l'aide de la fonction `PetitSigma` (qui calcule $\sigma(n)$) et du script suivants :

```
def PetitSigma(n):
    L = list(str(n))
    Res = 0
    for e in L:
        Res += facto(int(e))
    return(Res)

for n in range(1,10000000):
    if n == PetitSigma(n):
        print(n)
```

On obtient finalement :

$$\Sigma = \{1; 2; 145; 40\,585\}$$

3. E3A PSI ♠

1. Rappelons d’abord que `chr` correspond au type d’objet « caractère ». Ainsi, `chr(i)` renvoie le caractère ayant pour code (Unicode) l’entier `i` passé en argument.

En choisissant des codes compris entre 65 et 90 (`nfin - 1`), on va afficher toutes les lettres majuscules de l’alphabet (instruction : `print(i, "<->", chr(i))`).

Dans un second temps, le script construit une chaîne de 30 caractères choisis aléatoirement parmi ces 26 lettres majuscules et affiche la chaîne obtenue.

Le script affichera par exemple dans la console :

```
65 <-> A
66 <-> B
67 <-> C
68 <-> D
...
87 <-> W
88 <-> X
89 <-> Y
90 <-> Z
EODYOBXUDJWIHDSVCCQOHRTOVDYDXE
```

Remarque : nous n’avons pas fait apparaître l’affichage correspondant aux lettres E à V.

2. La fonction `tirer` doit s’inspirer du script fourni dans l’énoncé...
Puisque la chaîne générée ne doit comporter que des caractères parmi les `k` premières lettres majuscules de l’alphabet, on va remplacer `nfin = 65 + 26` par `nfin = 65 + k`.
Ensuite, la longueur de la chaîne n’est pas égale à 30 mais à `n`.
On peut ainsi considérer le code suivant :

```
def tirer(n,k):
    nfin = 65 + k
    chaine = ""
    for i in range(n):
        chaine += chr(randint(65,nfin-1))
    return(chaine)
```

3. Notons, dans un premier temps, qu’il est inutile d’effectuer quoi que ce soit si la longueur de la chaîne `m` est supérieure à celle de `c`.

Ensuite, un mot de longueur k correspond à une sous-chaîne de c de la forme $c[i:i+Lm]$, Lm désignant la longueur de la chaîne m (attention au deuxième indice apparaissant dans cette syntaxe ! Il convient bien de s'arrêter au caractère d'indice $i+Lm$). On fait donc varier l'indice i de 0 à $Lc-Lm$, Lc désignant la longueur de la chaîne c .

On peut donc considérer le code suivant :

```
def mot(m,c):
    Lm, Lc = len(m), len(c)
    if Lm > Lc:
        return(0)
    else:
        N = 0
        for i in range(Lc-Lm+1):
            if c[i:i+Lm] == m:
                N += 1
        return(N)
```

4. Dans la fonction demandée, nous allons gérer deux listes qui vont être modifiées au fur et à mesure de l'exécution de la fonction : une liste M contenant les mots de longueur k détectés dans c ; une liste O contenant les nombres d'occurrence des mots de la liste M . Bien sûr, l'élément $O[i]$ correspond au nombre d'occurrences du mot $M[i]$.

Dans la boucle principale, on balaie la chaîne c de la gauche vers la droite en considérant les sous-chaînes de longueur k : $c[i:i+k]$ avec un indice i variant de 0 à $n-k$. Pour une telle sous-chaîne, on a deux possibilités : soit elle n'est pas présente dans la liste M et on comptabilise son nombre d'occurrence dans la sous-liste $c[i:]$ (inutile de s'intéresser à la sous-liste $c[:i]$ puisque l'on sait que la sous-chaîne courante n'avait pas encore été détectée) grâce à la fonction `mot` ; soit elle est déjà présente dans la liste M et son nombre d'occurrence a déjà été déterminé : il n'y a rien à faire.

Pour construire la liste de doublets demandée, il suffit alors de balayer les listes M et O simultanément et de « poper » (même si ce n'est pas obligatoire) le premier élément.

D'où le code :

```
def compositions(c,k):
    n = len(c)
    M, O = [], []
    for i in range(n-k+1):
        mot_courant = c[i:i+k]
        if mot_courant not in M:
            M.append(mot_courant)
            O.append(mot(mot_courant,c[i:]))
    R = []
    for _ in range(len(M)):
        R.append((M.pop(0),O.pop(0)))
    return(R)
```

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 `degre` :

```
def degre(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 degre2(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 4^{ème} 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 :

$$\begin{aligned} 23 &= 2 \times 11 + \boxed{1} \\ 11 &= 2 \times 5 + \boxed{1} \\ 5 &= 2 \times 2 + \boxed{1} \\ 2 &= 2 \times 1 + \boxed{0} \\ 1 &= 1 \times 0 + \boxed{1} \end{aligned}$$

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)
```

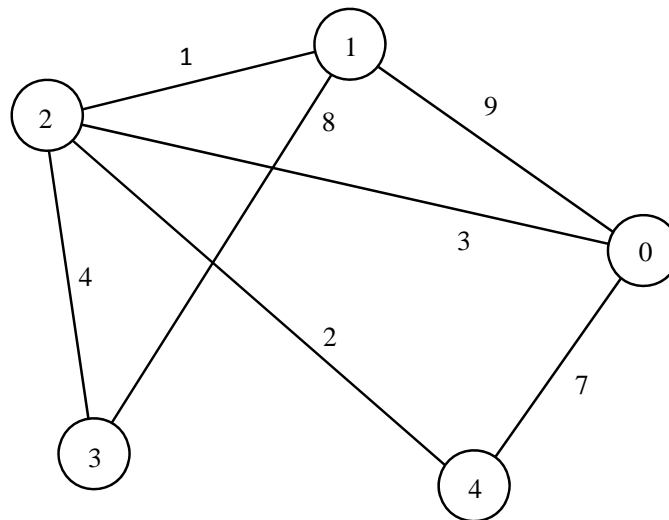
5) On a simplement la boucle :

```
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 préférera 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 $\text{len}(L) - 2$, 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 procède 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	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>x</i>	<i>y</i>	<i>z</i>
Après décalage	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>a</i>	<i>b</i>	<i>c</i>

Le mot `oralensam` devient `rudohqvdp`.

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

```
ALPHA = 'abcdefghijklmnopqrstuvwxyzz'
```

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 $\text{len}(\text{phrase}) - 1$, 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 :

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

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...