Informatique
Préparation à l'oral.

Algèbre et arithmétique - Corrigés

2016

1. D'après Centrale (PSI) ♠♠

A venir…

2. Centrale (PSI) ♠♠

A venir…

3. Centrale (PSI) ♠♠

(Planche RMS 2016-2017 N°1014)

a) On peut utiliser la bibliothèque `numpy` et construire la matrice M_n via une fonction `MatM_Create` que l'on appelle ensuite dans une boucle `for` : $(2 \leq n \leq 10)$:

```
import numpy as np

def MatM_Create(n):
     M = np.ones((n,n))
     for i in range(1,n):
          for j in range(1,n):
               M[i,j] = min(i,j) + 1
     return(M)
```

```
for n in range(2,11):
     print("\nn = ",n)
     print(MatM_Create(n))
```

b) On peut utiliser la fonction `det` du module `linalg` de la bibliothèque `numpy` pour obtenir le déterminant cherché. On peut alors remplacer la boucle `for` précédente par :

```
for n in range(2,11):
     print("\nn = ",n)
     M = MatM_Create(n)
     print(M)
     print("det(M) = ",np.linalg.det(M))
```

L'exécution du script nous conduit alors à conjecturer :

$$\forall n \in \mathbb{N} \setminus \{0;1\}, \det(M_n) = 1$$

c) On peut cette fois proposer la fonction `MatT_Create` suivante :

```
def MatT_Create(n):
     M = np.ones((n,n))
     for i in range(n):
          for j in range(i):
               T[i,j] = 0
     return(T)
```

d) On peut cette fois calculer le produit ${}^tT_nT_n$ à l'aide des fonctions dot et transpose de `numpy` :

```
for n in range(2,11):
     print("\nn = ",n)
     T = MatT_Create(n)
     print(np.dot(np.transpose(T),T))
```

L'exécution du script nous conduit alors à conjecturer :

$$\forall n \in \mathbb{N} \setminus \{0;1\}, {}^tT_nT_n = M_n$$

e) Pour tout entier naturel n non nul, on peut considérer le déterminant $\det(M_{n+1})$. En retranchant la première colonne à chacune des autres colonnes, il vient :

$$\det(M_{n+1}) = \begin{vmatrix} 1 & 1 & \cdots & \cdots & 1 \\ 1 & 2 & \cdots & \cdots & 2 \\ \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & & n & n \\ 1 & 2 & \cdots & n & n+1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & \cdots & \cdots & 0 \\ 1 & 1 & 1 & \cdots & \cdots & 1 \\ 1 & 1 & 2 & \cdots & \cdots & 2 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & n-1 & n-1 \\ 1 & 1 & 2 & \cdots & n-1 & n \end{vmatrix}$$

En développant alors suivant la première ligne, il vient immédiatement :

$$\det\left(M_{n+1}\right)=\begin{vmatrix}1&0&0&\cdots&\cdots&0\\1&1&1&\cdots&\cdots&1\\1&1&2&\cdots&\cdots&2\\\vdots&\vdots&\vdots&\ddots&&\vdots\\\vdots&\vdots&\vdots&&n-1&n-1\\1&1&2&\cdots&n-1&n\end{vmatrix}=\begin{vmatrix}1&1&\cdots&\cdots&1\\1&2&\cdots&\cdots&2\\\vdots&\vdots&\ddots&&\vdots\\\vdots&\vdots&&n-1&n-1\\1&2&\cdots&n-1&n\end{vmatrix}=\det\left(M_n\right)$$

On en déduit : $\forall n\in\mathbb{N}^*, \det\left(M_n\right)=cte$. Comme $M_1=\left(1\right)$ et $\det\left(M_1\right)$, il vient finalement :

$$\forall n\in\mathbb{N}^*, \det\left(M_n\right)=1$$

La première conjecture est ainsi démontrée.

Pour tout *n* entier naturel non nul, posons $T_n=\left(t_{ij}\right)$ avec $\begin{cases}t_{ij}=1 & \text{si } j\geq i\\ t_{ij}=0 & \text{si } i<j\end{cases}$ et ${}^tT_n=\left(t'_{ij}\right)=\left(t_{ji}\right)$.

Il vient alors : ${}^tT_nT_n=A=\left(a_{ij}\right)$ avec $a_{ij}=\sum_{k=1}^{n}t'_{ik}\,t_{kj}=\sum_{k=1}^{n}t_{ki}t_{kj}$.

On a alors : $t_{ki}t_{kj}=1\Leftrightarrow t_{ki}=t_{kj}=1\Leftrightarrow\begin{cases}k\leq i\\k\leq j\end{cases}\Leftrightarrow k\leq\min\left(i,j\right)$.

D'où : $a_{ij}=\sum_{k=1}^{n}t_{ki}t_{kj}=\sum_{k=1}^{\min(i,j)}1=\min\left(i,j\right)=m_{ij}$.

On a bien :

$$\forall n\in\mathbb{N}^*,\ {}^tT_nT_n=M_n$$

La deuxième conjecture est ainsi démontrée.

f) D'après le deuxième résultat démontré à la question précédente, la matrice M_n est le produit d'une matrice réelle et de sa transposée. Elle est donc réelle symétrique. On en déduit immédiatement qu'elle est diagonalisable dans $\mathbb{R}$.

Soit alors λ une valeur propre de M_n. Il existe une matrice non nulle *X* de $\mathcal{M}_{n,1}\left(\mathbb{R}\right)$ telle que $M_nX=\lambda X$, c'est-à-dire : ${}^tT_nT_nX=\lambda X$. D'où : ${}^tX\left({}^tT_nT_nX\right)={}^tX\left(\lambda X\right)$, soit ${}^t\left(T_nX\right)\left(T_nX\right)=\lambda\left({}^tXX\right)$.

Comme les éléments diagonaux de la matrice T_n sont égaux à 1, donc non nuls, elle est inversible. Et comme *X* n'est pas nulle, on en déduit immédiatement que la matrice colonne T_nX n'est pas nulle. On a alors : ${}^t\left(T_nX\right)\left(T_nX\right)>0$ (en confondant classiquement $\mathcal{M}_1\left(\mathbb{R}\right)$ et $\mathbb{R}$). Enfin, comme ${}^tXX>0$, il vient enfin $\lambda>0$.

Finalement :

$$\mathrm{Sp}\left(M_n\right) \subset \mathbb{R}_+^*$$

g) Notons S_n la somme des valeurs propres de la matrice M_n. On a :

$S_n = \mathrm{tr}\left(M_n\right) = 1+2+3+\ldots+n = \dfrac{n(n+1)}{2}$. Mais on a également, la matrice M_n comportant *n* valeurs propres (en tenant compte de leur multiplicité) : $S_n \leq n \times \lambda_n$.

On en tire : $\dfrac{n(n+1)}{2} \leq n \times \lambda_n$, c'est-à-dire $\lambda_n = \max\left(\mathrm{Sp}\left(M_n\right)\right) \geq \dfrac{n+1}{2}$.

Le résultat est établi.

h) On a : $M_n^{-1} = \left({}^tT_nT_n\right)^{-1} = T_n^{-1}\left({}^tT_n\right)^{-1} = T_n^{-1}\,{}^t\left(T_n^{-1}\right)$.

On peut donc chercher à déterminer T_n^{-1}.

Pour ce faire, on peut s'intéresser au système :

$$T_nX = Y \Leftrightarrow \begin{pmatrix} 1 & & \\ & \ddots & (1) \\ & (0) & \ddots \\ & & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

On a :

$$\begin{pmatrix} 1 & & \\ & \ddots & (1) \\ & (0) & \ddots \\ & & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \Leftrightarrow \begin{cases} x_1 + x_2 + \ldots + x_n = y_1 \\ \quad x_2 + \ldots + x_n = y_2 \\ \qquad\qquad \ldots \\ \quad x_{n-1} + x_n = y_{n-1} \\ \qquad x_n = y_n \end{cases}$$

En effectuant une « remontée », on obtient facilement :

$$\begin{cases} x_1 + x_2 + \ldots + x_n = y_1 \\ \quad x_2 + \ldots + x_n = y_2 \\ \qquad\qquad \ldots \\ \quad x_{n-1} + x_n = y_{n-1} \\ \qquad x_n = y_n \end{cases} \Leftrightarrow \begin{cases} x_1 = y_1 - y_2 \\ \qquad \ldots \\ x_{n-2} = y_{n-2} - y_{n-1} \\ x_{n-1} = y_{n-1} - y_n \\ x_n = y_n \end{cases}$$

D'où :

$$T_n^{-1} = \begin{pmatrix} 1 & -1 & & & \\ & 1 & -1 & (0) & \\ & & 1 & \ddots & \\ & (0) & & \ddots & -1 \\ & & & & 1 \end{pmatrix}$$

C'est-à-dire :

$$T_n^{-1} = \left(u_{ij}\right) \text{ avec } u_{ij} = \begin{cases} 1 & \text{si } j = i \\ -1 & \text{si } j = i+1 \\ 0 & \text{sinon} \end{cases}$$

Posons alors : ${}^t\left(T_n^{-1}\right) = \left(u'_{ij}\right)$ avec $u'_{ij} = \begin{cases} 1 & \text{si } j = i \\ -1 & \text{si } j = i-1 \\ 0 & \text{sinon} \end{cases}$.

On a : $M_n^{-1} = T_n^{-1}\,{}^t\left(T_n^{-1}\right) = \begin{pmatrix} 1 & -1 & & & \\ & 1 & -1 & (0) & \\ & & 1 & \ddots & \\ & (0) & & \ddots & -1 \\ & & & & 1 \end{pmatrix}\begin{pmatrix} 1 & & & & \\ -1 & 1 & & (0) & \\ & -1 & 1 & & \\ & (0) & \ddots & \ddots & \\ & & & -1 & 1 \end{pmatrix} = \left(b_{ij}\right)$.

Or : $b_{ij} = \sum_{k=1}^{n} u_{ik} u'_{kj}$ avec $u_{ik} \neq 0 \Leftrightarrow k = i$ ou $k = i+1$ et $u'_{kj} \neq 0 \Leftrightarrow k = j$ ou $k = j+1$.

Ainsi, ces contraintes sur k entraînent, pour obtenir des produits $u_{ik}u'_{kj}$ non nuls, des contraintes sur i et j : si $k = i$ alors $j = i$ ou $j = i-1$ et si $k = i+1$ alors $j = i+1$ ou $j = i$.
En d'autres termes, seules la diagonale principale ($i = j$) et les deux diagonales la bordant ($j = i+1$ et $j = i-1$) comporteront des coefficients non nuls.
Etudions ces trois situations :

- Pour $j = i$, on a :

  Pour $i \in [\![1; n-1]\!]$, $b_{ii} = \sum_{k=1}^{n} u_{ik}u'_{ki} = u_{ii}u'_{ii} + u_{i,i+1}u'_{i+1,i} = 1\times 1 + (-1)\times(-1) = 2$.

  Pour $i = n$, $b_{ii} = b_{nn} = \sum_{k=1}^{n} u_{nk}u'_{kn} = u_{nn}u'_{nn} = 1\times 1 = 1$

- Pour $j = i+1$ ($i \in [\![1; n-1]\!]$), on a :

  $b_{ij} = b_{i,i+1} = \sum_{k=1}^{n} u_{ik}u'_{k,i+1} = u_{i,i+1}u'_{i+1,i+1} = -1\times 1 = -1$

- Pour $j = i-1$ ($i \in [\![2; n]\!]$), on obtient encore -1.

En définitive, on a :

$$M_n^{-1} = \begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & (0) & \\ & -1 & 2 & -1 & & \\ & & -1 & \ddots & \ddots & \\ & (0) & & \ddots & 2 & -1 \\ & & & & -1 & 1 \end{pmatrix}$$

i) Pour obtenir M_n^{-1}, on utilise la fonction `inv` du module `linalg`.
On peut alors considérer le script suivant :

```
for n in range(2,11):
    print("\nn = ",n)
    print(np.linalg.inv(MatM_Create(n)))
```

On vérifie la correction du résultat obtenu à la question précédente.

2015

1. D'après ENSAM 2015 (PSI) ♠

1. On peut utiliser la syntaxe efficace suivante :

```
L30 = [True for i in range(31)]
```

2. On a, par exemple :

```
def modifier(L,p):
    if p == 0:
        L[0] = False
    elif p == 1:
        L[1] = False
    else:
        k = 2*p
        while k < len(L):
            L[k] = False
            k += p
```

3. On réutilise bien sûr la fonction `modifier` afin d'éliminer les multiples de p (à partir de $2p$) :

```
def premiers(n):
    # Génération d'une liste contenant n+1 True
    L = [True for i in range(n+1)]
    # Mise à jour de la liste (crible d'Erathostène)
    for p in range(n+1):
        modifier(L,p)
    # Génération de la liste des entiers premiers <= n
    LP = []
    for i in range(len(L)):
        if L[i] == True:
            LP.append(i)
    return(LP)
```

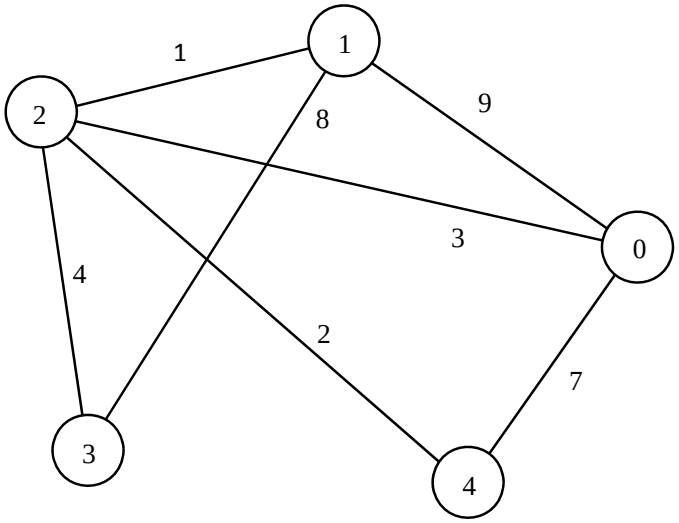
4. Il suffit ici d'appeler la fonction `premiers` avec 823 417 comme argument et d'afficher le dernier terme de la liste renvoyée. Ce dernier terme étant 823 399, on en déduit que 823 417 n'est pas premier (d'ailleurs : $823\,417 = 7 \times 117\,631$.

5. Il s'agit du crible d'Erathostène.

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) ♠

A venir…

5. D'après ENSAM 2015 (PSI) ♠♠

A venir…