Épreuve d'Informatique – Session 2017 – Filière MP/ PSI/ TSI

Les candidats sont informés que la précision des raisonnements algorithmiques ainsi que le soin apporté à la rédaction et à la présentation des copies seront des éléments pris en compte dans la notation. Il convient en particulier de rappeler avec précision les références des questions abordées. Si, au cours de l'épreuve, un candidat repère ce qui peut lui sembler être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Remarques générales :

- ✓ L'épreuve se compose de trois parties indépendantes.
- ✓ Toutes les instructions et les fonctions demandées seront écrites en Python.
- ✓ Les questions non traitées peuvent être admises pour aborder les questions ultérieures.
- ✓ Toute fonction peut être décomposée, si nécessaire, en plusieurs fonctions.

- * - * - *- * - * - * - *- * - * - *- * - * - * -

Partie I : Base de données et langage SQL

La compression des fichiers est une pratique aussi courante qu'indispensable en informatique. Elle consiste à stocker les données dans un nouveau format moins encombrant que l'original. Ce gain de place induit d'autres bénéfices dont le principal est l'accélération des transferts entre ordinateurs. Faire circuler moins de bits sur les réseaux diminue le temps de connexion, encombre moins les lignes de communication et limite les dépenses télématiques. La décompression rétablit le fichier dans son état initial.

De nombreux algorithmes de compressions existent, chacun ayant sa particularité et surtout un type de données cible. Car toutes les données ne se compressent pas de la même manière. Un algorithme de compression de texte travaillera sur les répétitions du nombre de caractères ou de parties de phrases. Un algorithme de compression d'images travaillera quant à lui sur d'autres domaines comme la différence entre un pixel et un autre. On imagine cependant mal le second algorithme en train de compresser un texte.

Néanmoins, tous les algorithmes de compression ont un point commun : leur objectif est de récupérer les données initiales partiellement, voire totalement. La décompression consiste à rétablir les données d'origine à l'aide du fichier compressé. Elle consiste souvent à appliquer l'algorithme de compression en sens inverse.

Types de compressions :

Il y a deux types majeurs de compressions : la compression *sans perte* et la compression *avec perte*.

a- La compression sans perte : *Une compression est dite sans perte si les données après décompression sont identiques aux données originelles.*

Ces compression se basent toutes sur le même principe : la répétition d'une donnée est une répétition de trop. L'objectif va être de supprimer le maximum de répétition pour obtenir une compression plus importante tout en étant capable de retrouver les répétitions retirées.

En somme, ces compressions écrivent exactement les mêmes données mais de façon plus concise.

Elles sont appliquées à tous types de données et les formats compressés sont très nombreux. Pour ne citer que les plus connus, nous retrouvons les formats : zip, rar, 7z, bz2, gz, ...

Les algorithmes, moins connu du grand public, sont aussi très nombreux. Nous retrouvons par exemple le codage de Huffman, les codages de Lempel-Ziv, …

b- La compression avec perte : *Une compression est dite avec perte si les données après décompression sont différentes des données originelles.*

Elles sont appliquées à des données perceptibles, c'est-à-dire à des images, des sons ou des vidéos. Le principe va consister à supprimer les informations là où les sens de la vue et de l'ouïe ne les perçoivent que très peu. Par exemple, l'œil humain ne distingue que très peu les zones de contraste. Aussi, nous pouvons retirer des détails à ces zones sans trop impacter sur la qualité de l'image. Le nom de format représente directement le type de compression employé : jpeg, mpeg, mp3, divx, ...

Taux de compression :

Le taux de compression est une mesure de la performance d'un algorithme de compression. Il est exprimé en pourcentage. Le taux de compression est le gain en volume rapporté au volume initial des données. Plus le taux de compression est élevé, plus la taille du fichier compressé résultant est faible. La formule correspondante s'écrit :

$$\textbf{Taux} = (1 - \frac{\textbf{taille du fichier compressé}}{\textbf{taille initiale du fichier}}) * 100$$

On dispose d'une base de données de chemin absolu : **"C:\BDcompressions.sql"**, composée de trois tables : la table **Algorithme**, la table **Fichier** et la table **Compression**, dont les structures sont les suivantes :

- **Algorithme** (**format** (texte), **type** (entier))

  La table **Algorithme** contient différents formats d'algorithmes de compressions. Le champ **format** est la clé primaire. Le champ **type** contient la valeur **1**, si l'algorithme de compression est du type avec perte de données, ou **0** sinon.

  Exemples de données de la table Algorithme :

format	type
7z	0
bz2	0
gz	0
jpeg	1
rar	0
zip	0
mp3	1
huffman	0
…	…

- **Fichier** (**numeroF** (entier), **nomF** (texte), **tailleF** (entier))

  La table **Fichier** contient des fichiers de différents types : texte, image, son, vidéo … . Le champ **numeroF** est la clé primaire. Le champ **nomF** contient le nom complet de chaque fichier. Le champ **tailleF** contient la taille originale du fichier, exprimée en Octet.

  Exemples de données de la table Fichier :

numeroF	nomF	tailleF
1	Cours Python.pdf	6 420 768
2	Nature.bmp	5 760 054
3	Langage SQL.doc	345 600
4	Chanson.wav	47 185 920
…	…	…

- **Compression** (**numeroF** (entier), **format** (texte), **tailleC** (entier))

  La table **Compression** contient les compressions de certains fichiers de la table Fichier. Les champs **numeroF** et **format** sont des clés étrangères. Le champ **tailleC** contient la taille du fichier compressé, exprimée en octet. Un fichier de la table Fichiers peut être compressé plusieurs fois, par différents algorithmes de compressions.

  Exemples de données de la table Compression :

numeroF	format	tailleC
2	jpeg	291 441
2	png	2 622 545
2	zip	2 160 937
1	cab	5 768 938
1	zip	5 818 538
4	mp3	11 364
3	zip	145 366
3	7z	133 396
…	…	…

I. 1- Écrire, en algèbre relationnelle, une requête qui donne pour résultat : Les noms des fichiers dont la taille originale est comprise entre 1 Kilo-octet et 1 Méga-octet.

I. 2- Écrire une requête SQL, qui donne pour résultat : Les noms et les tailles des fichiers textes, dont le nom se termine par : .doc ou .docx, triés dans l'ordre alphabétique des noms des fichiers.

I. 3- Écrire une requête SQL qui donne pour résultat : Les noms des fichiers compressés, les formats de compression, les types de compression, et le taux de compression de chaque fichier, dont le taux de compression dépasse 40%, triés dans l'ordre des numéros des fichiers.

I. 4- Écrire une requête SQL qui donne pour résultat : Les algorithmes sans perte de données et le compte des fichiers compressés par chacun de ces algorithmes, dont ce compte est égal à 3, 5 ou 8.

I. 5- Écrire une requête SQL qui donne pour résultat : Les 3 premiers grands taux de compressions, triés dans l'ordre croissant.

I. 6- Écrire un programme Python qui saisi un format de compression, et qui affiche **le taux moyen** des taux des compressions de ce format, si le format saisi existe dans la base de données. Si le format saisi n'existe pas, le programme doit afficher le message : « *Le format saisi n'existe pas dans la BD* ».

On suppose que le module **sqlite3** est importé. Ce module contient les fonctions et les méthodes permettant de manipuler les bases de données :

Connexion à une base de données : **`connect()`**	Exécuter une requête : **`execute()`**
Créer un curseur : **`cursor()`**	Récupérer le résultat d'une requête : **`fetchone(), fetchall()`**

Partie II : Programmation Python

La compression de HUFFMAN

La **compression de Huffman** est une méthode de compression de données sans perte proposé par « David Huffman » en 1952. C'est un processus qui permet de compresser des données informatiques afin de libérer de la place dans la mémoire d'un ordinateur. Or tout fichier informatique (qu'il s'agisse d'un fichier texte, d'une image ou d'une musique …) est formé d'une suite de caractères. Chacun de ces caractères étant lui-même code par une suite de 0 et de 1. L'idée du codage de Huffman est de repérer les caractères les plus fréquents et de leur attribuer des codes courts (c'est-a-dire nécessitant moins de 0 et de 1) alors que les caractères les moins fréquents auront des codes longs. Pour déterminer le code de chaque caractère on utilise un arbre binaire. Cet arbre est également utilisé pour le décodage.

Dans cette partie, on s'intéressera à appliquer la compression de Huffman pour compresser une liste de nombres entiers, contenant des éléments en répétition.

Exemple : `L = [12,29,55,29,31,8,12,46,29,8,12,29,31,29,8,29,8]`

***NB** : Dans toute cette partie, on suppose que la liste L contient au moins deux éléments différents.*

II. 1- L'espace mémoire occupé par un entier positif

Quel est le plus grand nombre entier positif, avec bit de signe, qu'on peut coder sur 12 bits ? Justifier votre réponse.

II. 2- Écriture binaire d'un nombre entier

II. 2- a) Écrire la fonction : **`binaire(N)`**, qui reçoit en paramètre un entiers naturel **N**, et qui retourne une chaine de caractères qui contient la représentation binaire de **N**. *Le bit du poids le plus fort se trouve à la fin de la chaine. Ne pas utiliser la fonction prédéfinie bin( ) de Python.*

Exemple : **binaire (23)** retourne la chaine : **"11101"**

II. 2- b) Déterminer la complexité de la fonction **binaire (N)**.

II. 2- c) La méthode dite de **Horner** (*William George Horner : 1786-1837*) est une méthode très pratique, utilisée pour améliorer le calcule de la valeur d'un polynôme, en réduisant le nombre de multiplications.

Soit le polynôme P à coefficients réels suivant :

$$P(x) = a_n*x^n + a_{n-1}*x^{n-1} + \dots + a_2*x^2 + a_1*x + a_0$$

Pour calculer P(k), la méthode de Horner effectue le calcul comme suit :

$$P(k) = ((\dots ((a_n*k+a_{n-1})*k+a_{n-2})*k+ \dots +a_2)*k+a_1)*k+a_0$$

En utilisant le principe de la méthode de Horner, écrire la fonction de complexité linéaire : **`entier(B)`**, qui reçoit en paramètre une chaine de caractères **B** contenant la représentation binaire d'un entier naturel, dont le premier bit est celui du poids le plus faible, et qui calcule et retourne la valeur décimale de cet entier.

Exemple : **entier ("11101")** retourne le nombre entier : **23** ($23 = 1*2^0 + 1*2^1 + 1*2^2 + 0*2^3 + 1*2^4$)

II. 3- Liste des fréquences :

La première étape de la méthode de compression de Huffman consiste à compter le nombre d'occurrences de chaque élément de la liste des entiers.

Écrire une fonction : **`frequences(L)`**, qui reçoit en paramètre une liste d'entiers **L**, et qui retourne une liste de tuples : Chaque tuple est composé d'un élément de L et de son occurrence (répétition) dans L. Les premiers éléments des tuples de la liste des fréquences doivent être tous différents (sans doublon).

Exemple : L= [12, 29, 46, 29, 31, 8, 12, 55, 29, 8, 12, 29, 31, 29, 8, 29, 8]

La fonction **frequences (L)** retourne la liste F = [(**8**, 4), (**12**, 3), (**46**, 1), (**55**, 1), (**29**, 6), (**31**, 2)]

II. 4- Tri de la liste des fréquences :

La liste F des fréquences, des différents éléments de la liste des entiers, étant créée. L'étape suivante est de trier cette liste F dans l'ordre croissant des occurrences (les 2èmes éléments des tuples).

Écrire la fonction: **`tri(F)`**, qui reçoit en paramètre la liste F des fréquences, et qui trie les éléments de la liste F dans l'ordre croissant des occurrences. *Cette fonction doit être de complexité quasi-linéaire O(nlog(n)). Ne pas utiliser ni la fonction sorted( ), ni la méthode sort( ) de Python.*

Exemple : F = [(12, 3), (29, 6), (55, 1), (31, 2), (8, 4), (46, 1)]

La fonction **tri (F)** retourne la liste : [(46, **1**), (55, **1**), (31, **2**), (12, **3**), (8, **4**), (29, **6**)]

II. 5- Création de l'arbre binaire de HUFFMAN :

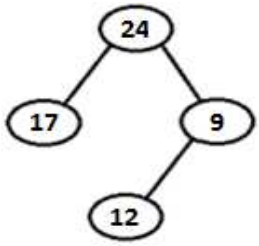
Pour représenter un arbre binaire, on va utiliser une liste composée de trois éléments :

- ✓ Chaque nœud est représenté par la liste : **[Père , [Fils gauche] , [Fils droit]]**
- ✓ Chaque feuille est représentée par la liste : **[Feuille , [ ] , [ ]]**

Exemple :

L'arbre binaire suivant, sera représenté par la liste :

`A = [24,[17,[],[]],[9,[12,[],[]],[]]]`



L'arbre binaire de Huffman est crée à partir de la liste des fréquences, selon le procédé suivant :

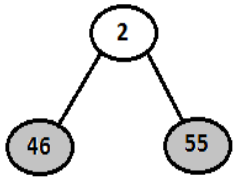
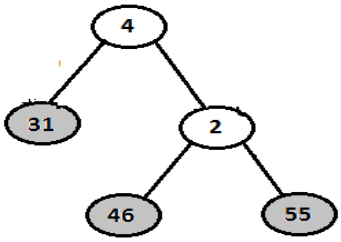
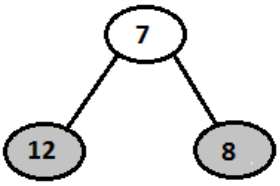
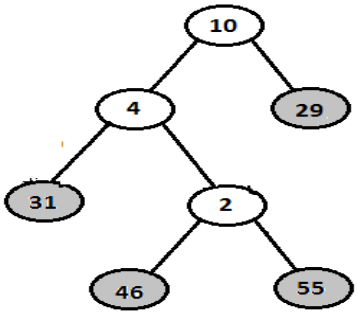
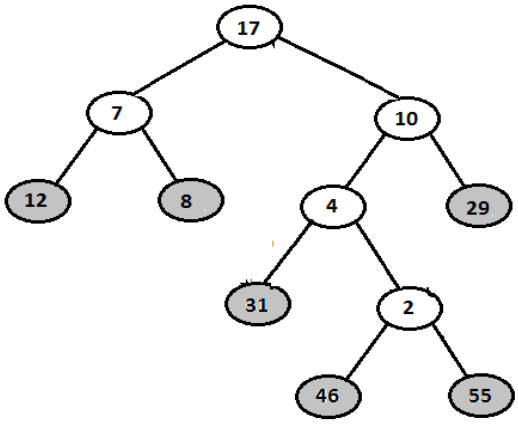
- *Trier la liste des fréquences dans l'ordre croissant des occurrences ;*
- *Tant que la liste des fréquences contient au moins deux éléments, on répète les opérations :*
  - *Retirer les deux premiers éléments dont les poids (2ème élément de chaque tuple) sont les plus faibles, et créer un tuple dont le poids est la somme des poids de ces deux éléments ;*
  - *Insérer le tuple obtenu dans la liste des fréquences, de façon à ce qu'elle reste croissante.*

Les feuilles de l'arbre binaire obtenu sont les différents éléments de la liste à compresser.

Exemple :

La liste F des fréquences suivante, est triée dans l'ordre croissant des occurrences.

F = [(46, 1), (55, 1), (31, 2), (12, 3), (8, 4), (29, 6)]

1ère étape F = [**(46, 1), (55, 1),** (31, 2), (12, 3), (8, 4), (29, 6)] Retirer : **(46, 1)** et **(55, 1)** Insérer le tuple **(A , 2)** dans la liste F La liste F triée devient : F = [(31, 2), **(A , 2)**, (12, 3), (8, 4), (29, 6)]	**A** 
2ème étape F = [**(31 , 2), (A , 2),** (12, 3), (8, 4), (29, 6)] Retirer : **(31 , 2)** et **(A , 2)** Insérer le tuple **(A , 4)** dans la liste F La liste F triée devient : F = [(12, 3), (8, 4), **(A , 4)**, (29, 6)]	**A** 
3ème étape F = [**(12, 3), (8, 4)**, (A , 4), (29, 6)] Retirer : **(12 , 3)** et **(8 , 4)** Insérer le tuple **(B , 7)** dans la liste F La liste Ftriée devient : F = [(A, 4), (29, 6), **(B , 7)**]	**B** 
4ème étape F = [**(A, 4), (29, 6)**, (B , 7)] Retirer : **(A , 4)** et **(29 , 6)** Insérer le tuple **(A , 10)** dans la liste F La liste F triée devient : F = [(B , 7), **(A , 10)**]	**A** 
5ème étape F = [**(B , 7)**, **(A , 10)**] Retirer : **(B , 7)** et **(A , 10)** Insérer le tuple **(A , 17)** dans la liste F La liste F triée devient : F = **[(A , 17)]** ***La liste F ne compte plus qu'un seul élément, le procédé est arrêté.***	**A** 

II. 5- a) Écrire la fonction de complexité linéaire : **`insere(F,T)`**, qui reçoit en paramètre la liste **F** des fréquences triée dans l'ordre croissant des occurrences, et un tuple **T**, de même nature que les éléments de **F**. La fonction insère le tuple **T** dans **F** de façon à ce que la liste **F** reste triée dans l'ordre croissant des occurrences.

Exemple : F = [(46, **1**), (55, **1**), (31, **2**), (12, **3**), (8, **4**), (29, **6**)] et T = **(17, 5).**

Après l'appel de la fonction : **insere (F, T)**, la liste F devient :

F = [(46, **1**), (55, **1**), (31, **2**), (12, **3**), (8, **4**), (17, **5**), (29, **6**)]

II. 5- b) Écrire la fonction : **`arbre_Huffman(F)`**, qui reçoit en paramètre la liste des fréquences **F**, composée des tuples formés des différents éléments de la liste à compresser, et de leurs occurrences. La fonction retourne un arbre binaire de Huffman, crée selon le principe décrit ci-dessus.

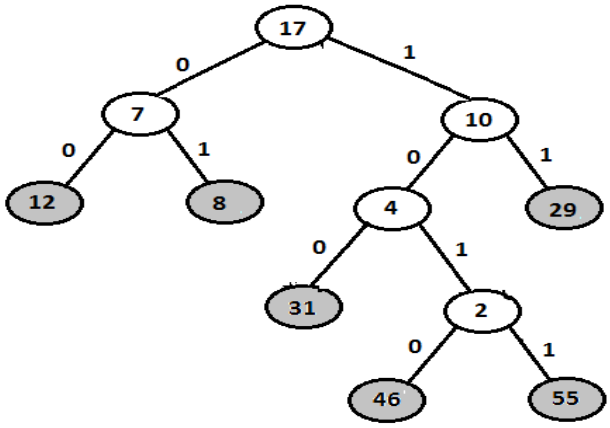
II. 6- Construction du dictionnaire des codes de HUFFMAN :

On suppose que l'arbre binaire de Huffman est crée selon le principe de la question précédente. L'étape suivante est de construire un dictionnaire où les clés sont les différents éléments de la liste à compresser (les feuilles de l'arbre de Huffman), et les valeurs sont les codes binaires de Huffman correspondants.

Parcours de l'arbre binaire de HUFFMAN :

On associe le code **0** à chaque branche de gauche et le code **1** à chaque branche de droite.

Pour obtenir le code binaire de chaque feuille, on parcourt l'arbre à partir de la racine jusqu'aux feuilles en rajoutant à chaque fois au code **0** ou **1** selon la branche suivie :



Ainsi, en parcourant l'arbre, on obtient les codes suivants :

Feuilles	*Codes*
12	**"00"**
8	**"01"**
31	**"100"**
46	**"1010"**
55	**"1011"**
29	**"11"**

Écrire la fonction : **`codes_Huffman(A,code,codesH)`** , qui reçoit trois paramètres :

- **A** : est un arbre binaire de Huffman, crée selon le principe de la question précédente.
- **code** : est une chaine de caractère initialisée par la chaine vide.
- **codesH** : est un dictionnaire initialisé par le dictionnaire vide

La fonction effectue un parcours de l'arbre, en ajoutant dans le dictionnaire **codesH**, les feuilles de l'arbre comme clés et les codes binaires correspondants comme valeurs des clés.

Exemple :

codesH = { } *# codesH est un dictionnaire vide.*

code = " " *# code est une chaine de caractères vide.*

Après l'appel de la fonction : **codes_Huffman (Arbre, code, codesH)**, Le contenu du dictionnaire codesH est :

{ 55 : '1011' , 8 : '01' , 12 : '00' , 29 : '11' , 46 : '1010' , 31 : '100' }

II. 7- Compression de la liste des entiers :

Le dictionnaire des codes binaires contient les différents éléments de la liste à compresser, et leurs codes Huffman binaires correspondants. On peut maintenant procéder à la compression de la liste des entiers.

Écrire la fonction : **`compresse(L,codesH)`** , qui reçoit en paramètres la liste **L** des entiers, et le dictionnaire **codesH** des codes binaires Huffman. La fonction retourne une chaine de caractères contenant la concaténation des codes binaires Huffman correspondants à chaque élément de L.

Exemple :

L= [12 , 29 , 46 , 29 , 31 , 8 , 12 , 55 , 29 , 8, 12, 29, 31, 29, 8, 29, 8]

codesH = { 55: '1011' , 8: '01' , 12: '00' , 29: '11' , 46: '1010' , 31: '100' }

La fonction **compresse (L, codesH)** retourne la chaine :

"00111010111000100101111010011100110111101"

II. 8- Décompression de la chaine de caractères binaires :

L'opération de décompression consiste à retrouver la liste initiale des nombres entiers, à partir de la chaine binaire et du dictionnaire des codes.

Écrire la fonction : **`decompresse(codesH, B)`** , qui reçoit en paramètres le dictionnaire **codesH** des codes de Huffman, et la chaine de caractères binaires **B**. La fonction construit et retourne la liste initiale.

Exemple :

B = "00111010111000100101111010011100110111101"

codesH = { 55: '1011', 8: '01', 12: '00', 29: '11', 46: '1010', 31: '100' }

La fonction **decompresse (codesH , B)** retourne la liste initiale :

[12, 29, 46, 29, 31, 8, 12, 55, 29 , 8, 12, 29, 31, 29, 8, 29, 8]

Partie III : Calcul scientifique

Calcul de la matrice inverse d'une matrice carrée inversible

Méthode : Élimination de 'Gauss-Jordan'

En mathématiques, l'élimination de « Gauss-Jordan », aussi appelée pivot de Gauss, nommée en hommage à **Carl Friedrich Gauss** et **Wilhelm Jordan**, est un algorithme de l'algèbre linéaire pour déterminer les solutions d'un système d'équations linéaires, pour calculer l'inverse d'une matrice carrée inversible ou pour déterminer le rang d'une matrice…

Pour calculer la matrice inverse d'une matrice carrée M inversible, la méthode de l'élimination de « Gauss-Jordan » consiste à effectuer des transformations, simultanément sur les deux matrices : la matrice M, et la matrice identité E de même dimension que M.

Le but de la méthode de l'élimination de « Gauss-Jordan » est de transformer la matrice M en matrice identité, tout en effectuant simultanément les mêmes opérations (effectuer les mêmes échanges, utiliser les mêmes valeurs des pivots …), sur les deux matrices M et E. La matrice E contient, alors, la matrice inverse de la matrice M initiale.

Exemple :

Matrice carrée inversible M :

$$\begin{bmatrix} 1. & -1. & 2. & -2. \\ 3. & 2. & 1. & -1. \\ 2. & -3. & -2. & 1. \\ -1. & -2. & 3. & -3. \end{bmatrix}$$

Matrice identité E, de même dimension que M :

$$\begin{bmatrix} 1. & 0. & 0. & 0. \\ 0. & 1. & 0. & 0. \\ 0. & 0. & 1. & 0. \\ 0. & 0. & 0. & 1. \end{bmatrix}$$

Après les transformations, les deux matrices M et E deviennent ainsi :

Matrice M transformée en identité:

$$\begin{bmatrix} 1. & 0. & 0. & 0. \\ 0. & 1. & 0. & 0. \\ 0. & 0. & 1. & 0. \\ 0. & 0. & 0. & 1. \end{bmatrix}$$

E contient l'inverse de la matrice M initiale :

$$\begin{bmatrix} 0.8 & -0.1 & 0. & -0.5 \\ -1. & 0.5 & 0. & 0.5 \\ 5. & -2. & -1. & -3. \\ 5.4 & -2.3 & -1. & -3.5 \end{bmatrix}$$

On suppose que le module **numpy** est importé :

```
import numpy as np
```

Le module **numpy** contient les méthodes et les fonctions permettant de manipuler les matrices.

*On rappelle que les indices des listes et tableaux en Python commencent à **0**.*

Exemple :

M = np.array([[1, -1, 2, -2] , [3, 2, 1, -1] , [2, -3, -2, 1] , [-1, -2, 3, -3]] , float)

Cette instruction permet de créer la matrice **M** suivante :

$$\begin{bmatrix} 1. & -1. & 2. & -2. \\ 3. & 2. & 1. & -1. \\ 2. & -3. & -2. & 1. \\ -1. & -2. & 3. & -3. \end{bmatrix}$$

III. 1- Matrice inversible :

Une matrice carrée est inversible, si son déterminant est différent de 0.

Le module **numpy** contient un sous-module appelé : **linalg**. Ce dernier contient la méthode : **det ()** qui reçoit en paramètre une matrice, et qui retourne la valeur du déterminant de cette matrice.

Écrire la fonction : **`inversible(M)`**, qui reçoit en paramètre une matrice carrée **M**, et qui retourne ***True*** si la matrice M est inversible, sinon, elle retourne ***False***.

III. 2- Matrice identité :

Écrire la fonction : **`identite(n)`**, qui reçoit en paramètre un entier strictement positif n, et qui crée et retourne la matrice identité d'ordre n (n lignes et n colonnes), de nombres réels.

Exemple :

La fonction **identite (4)** retourne la matrice identité E d'ordre 4 suivante :

$$\begin{vmatrix} 1. & 0. & 0. & 0. \\ 0. & 1. & 0. & 0. \\ 0. & 0. & 1. & 0. \\ 0. & 0. & 0. & 1. \end{vmatrix}$$

III. 3- Opération de transvection :

Écrire une fonction : **`transvection(M,p,q,r)`**, qui reçoit en paramètres une matrice **M**, deux entiers **p** et **q** représentant les indices de deux lignes dans la matrice **M**, et un nombre réel **r**. Dans la matrice M, la fonction remplace la ligne **p** par la ligne résultat de la combinaison linéaire des deux lignes **p** et **q,** suivante : **`Mp+r*Mq`.**

Exemple :

Matrice M :

$$\begin{bmatrix} 1. & -1. & 2. & -2. \\ 3. & 2. & 1. & -1. \\ 2. & -3. & -2. & 1. \\ -1. & -2. & 3. & -3. \end{bmatrix}$$

Après l'appel : **transvection (M, 2, 0, -3.)**, la matrice M devient :

$$\begin{bmatrix} 1. & -1. & 2. & -2. \\ 3. & 2. & 1. & -1. \\ -1. & 0. & -8. & 7. \\ -1. & -2. & 3. & -3. \end{bmatrix}$$

III. 4- Échanger deux lignes dans une matrice :

Écrire la fonction : **`echange_lignes(M,p,q)`**, qui reçoit en paramètres une matrice **M**, et deux entiers **p** et **q** représentant respectivement deux indices de deux lignes dans la matrice **M**. la fonction échange les lignes **p** et **q** dans la matrice **M**.

Exemple :

Matrice M :

$$\begin{bmatrix} 1. & -1. & 2. & -2. \\ 3. & 2. & 1. & -1. \\ 2. & -3. & -2. & 1. \\ -1. & -2. & 3. & -3. \end{bmatrix}$$

Après l'appel : **echanger _lignes (M, 1, 3)**, les lignes 1 et 3 de la matrice M sont échangées :

$$\begin{bmatrix} 1. & -1. & 2. & -2. \\ -1. & -2. & 3. & -3. \\ 2. & -3. & -2. & 1. \\ 3. & 2. & 1. & -1. \end{bmatrix}$$

III. 5- Recherche de la ligne du pivot dans une colonne de la matrice :

Écrire la fonction : **`ligne_pivot(M,c)`**, qui reçoit en paramètres une matrice **M**, et un entier **c** représentant une colonne dans la matrice **M**.

La fonction retourne l'indice **p** tel que : **| M[p, c] | = max { | M[i, c] | / 0 ≤ i ≤ c }**

Exemple :

Si la matrice M est la suivante :

$$\begin{vmatrix} -1. & 1. & 5. & 2. \\ 10. & 1. & -6. & -4. \\ 8. & 6. & -3. & 7. \\ -9. & 5. & 7. & -2. \end{vmatrix}$$

La fonction **ligne_pivot (M, 2)** retourne l'indice : **1**, car, dans la colonne 2, l'élément -6.0 est le plus grand en valeur absolue, parmi les éléments des lignes 0, 1 et 2.

III. 6- Transformer la matrice M en matrice triangulaire inférieure :

Écrire la fonction : **`triangulaire_inf(M,E)`**, qui reçoit en paramètre une matrice carrée inversible **M**, et une matrice identité **E** de même dimension que **M**. La fonction transforme la matrice **M** en matrice triangulaire inférieure. Les transformations effectuées sur la matrice **M**, doivent être effectuées simultanément sur la matrice **E**.

Exemple :

Après l'appel de la fonction **triangulaire_inf (M, E)**, les matrices M et E deviennent ainsi :

Matrice M triangulaire inférieure

$$\begin{bmatrix} 1.25 & 0. & 0. & 0. \\ 3.33 & 2.67 & 0. & 0. \\ 1.67 & -3.67 & -1. & 0. \\ -1. & -2. & 3. & -3. \end{bmatrix}$$

Matrice E

$$\begin{bmatrix} 1. & -0.12 & 0. & -0.62 \\ 0. & 1. & 0. & -0.33 \\ 0. & 0. & 1. & 0.33 \\ 0. & 0. & 0. & 1. \end{bmatrix}$$

III. 7- Transformer la matrice triangulaire inférieure M en matrice diagonale :

Les deux matrices **M** et **E** sont les résultats de la fonction triangulaire_inf (M, E). La matrice **M** est devenue triangulaire inférieure. Écrire la fonction : **`diagonale(M,E)`**, qui transforme la matrice **M** en matrice diagonale. Les transformations effectuées sur la matrice **M**, doivent être aussi effectuées simultanément sur la matrice **E**.

Exemple :

Après l'appel de la fonction **diagonale (M, E)**, les matrices M et E deviennent ainsi :

Matrice M diagonale

$$\begin{bmatrix} 1.25 & 0. & 0. & 0. \\ 0. & 2.67 & 0. & 0. \\ 0. & 0. & -1. & 0. \\ 0. & 0. & 0. & -3. \end{bmatrix}$$

Matrice E

$$\begin{bmatrix} 1. & -0.12 & 0. & -0.62 \\ -2.67 & 1.33 & 0. & 1.33 \\ -5. & 2. & 1. & 3. \\ -16.2 & 6.9 & 3. & 10.5 \end{bmatrix}$$

III. 8- Calcul de la matrice inverse par élimination de « Gauss-Jordan » :

Écrire la fonction : **`inverse(M)`**, qui reçoit en paramètre une matrice carrée inversible **M**, et qui calcule et renvoie la matrice inverse de **M**. *(Ne pas utiliser la méthode prédéfinie inv( ) de Python)*

Exemple :

M matrice carrée inversible :

$$\begin{bmatrix} 1. & -1. & 2. & -2. \\ 3. & 2. & 1. & -1. \\ 2. & -3. & -2. & 1. \\ -1. & -2. & 3. & -3. \end{bmatrix}$$

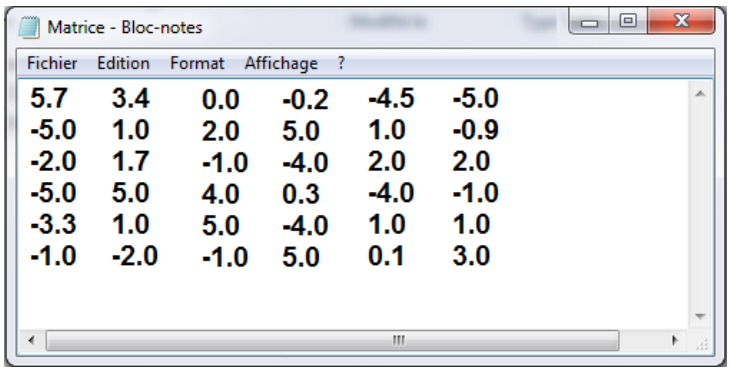
La fonction **inverse (M)** renvoie la matrice inverse de M :

$$\begin{bmatrix} 0.8 & -0.1 & 0. & -0.5 \\ -1. & 0.5 & 0. & 0.5 \\ 5. & -2. & -1. & -3. \\ 5.4 & -2.3 & -1. & -3.5 \end{bmatrix}$$

III. 9- Matrice stockée dans un fichier texte :

On suppose avoir créé un fichier texte contenant une matrice carrée. Chaque ligne du fichier contient une ligne de la matrice, et les éléments de chaque ligne du fichier sont séparés par le caractère espace.

Exemple :



Matrice carrée inversible d'ordre 6, stockée dans un fichier texte

Écrire la fonction : **`matrice_fichier(ch)`**, qui reçoit en paramètre une chaîne de caractère **ch**, qui contient le chemin absolu du fichier texte, contenant une matrice carrée :

- Si cette matrice est inversible, la fonction renvoie sa matrice inverse ;
- Si cette matrice n'est pas inversible, la fonction affiche le message « Matrice non inversible » ;
- Si le fichier texte ne se trouve pas à l'emplacement spécifié dans la chaine ch, l'exception **FileNotFoundError** sera levée. La fonction affiche le message : « *Fichier texte inexistant* »

------- **FIN DE l'ÉPREUVE** ------