Numéro d'anonymat (à remplir par un surveillant)

Examen Session 2 2017 – 2018
Éléments de Programmation I – 1I001
Durée : 2h

Documents autorisés: Aucun document ni machine électronique n'est autorisé à l'exception de la carte de référence de Python.

Le sujet comporte 19 pages. Ne pas désagrafer les feuilles.

Répondre directement sur le sujet, dans les boîtes appropriées. La taille des boîtes suggère le nombre de lignes de la réponse attendue. Le quadrillage permet d'aligner les indentations.

Le barème indiqué pour chaque question n'est donné qu'à titre indicatif. Le barème total est lui aussi donné à titre indicatif : 61 points auxquels peuvent s'ajouter des points bonus explicités dans l'énoncé des questions.

La clarté des réponses et la présentation des programmes seront appréciées. Les exercices peuvent être résolus dans un ordre quelconque. Pour répondre à une question, **il possible et souvent utile, d'utiliser les fonctions qui sont l'objet des questions précédentes, même si vous n'avez répondu à ces questions précédentes.**

Remarque: si nécessaire, on considère que la bibliothèque de fonctions mathématiques a été importée avant les fonctions à écrire. Sauf mention contraire explicite, seules les primitives Python présentes sur la carte de référence peuvent être utilisées.

Important: bien lire ce qui est demandé dans l'intitulé de la question. Sauf exception (ex.: fonction mystère), pour les fonctions demandées, **il est nécessaire de donner une définition avec signature et hypothèse(s) éventuelle(s)**, mais pas nécessaire de donner la description ou un jeu de tests.

Notation des spécifications: Une spécification (signature et hypothèses) seule ne rapporte pas de point. Une fonction correcte dont la spécification est fausse ou manquante est pénalisée.

L'examen est composé de quatre exercices indépendants.

Exercice 1 : Suites de Lehmer

En 1948, le mathématicien D.H. Lehmer a proposé un algorithme, utilisant une suite numérique, pour générer des nombres pseudo-aléatoires (comme, par exemple, ce qui est utilisé dans la librairie Random vue en TME). Cet exercice propose de programmer des fonctions Python implémentant cette suite numérique.

Une *suite de Lehmer* est définie, pour tout n entier naturel, par :

$$u_n = \begin{cases} v & \text{si } n = 0 \\ (a \cdot u_{n-1} + b) \bmod m & \text{si } n \geq 1 \end{cases}$$

avec m, un entier naturel non nul, et a, b, et v, trois entiers naturels supérieurs ou égaux à 0 et strictement inférieurs à m. On peut noter que l'utilisation du modulo (a mod b désigne le reste de la division euclidienne de a par b) fait que les valeurs prises par u_n sont forcément comprises entre 0 et $m - 1$.

Dans cet exercice, de façon générale, étant donné m, a, et b, on appelle *suite de Lehmer de v* une telle suite.

Par exemple, pour $m = 8$, $a = 1$, $b = 3$, la suite de Lehmer de $v = 4$ (ou plus simplement, suite de Lehmer de 4) est la suite 4, 7, 2, 5, 0, 3, 6, 1, 4, 7...

Pour $m = 10$, $a = 3$, et $b = 1$, la suite de Lehmer de 2 est 2, 7, 2, 7...

Pour $m = 10$, $a = 5$, et $b = 1$, la suite de Lehmer de 2 est 2, 1, 6, 1...

Remarque : dans cet exercice, on utilisera l'opérateur `%` de Python pour le modulo (mod). Ainsi, x mod y sera écrit `x % y` en Python.

Question 1.1 : [2/61]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `suivant` qui, étant donné 4 entiers naturels : m, a, b et u, le n-ième terme de la suite de Lehmer, rend la valeur du terme qui suit u dans la suite de Lehmer.

Par exemple :

```
>>> suivant(8,1,3,0)
3
>>> suivant(10,3,1,2)
7
>>> suivant(10,5,1,1)
6
```

Réponse

```
def suivant(m,a,b,u_n):
    """ int * int * int * int -> int
        Hypothese: m >0 et a,b et u_n appartiennent à [0,m[
    """
    return (a*u_n+b) % m
```

Question 1.2 : [3/61]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `terme` qui, étant donné 5 entiers naturels : m, a, b, v, et n rend la valeur du terme u_n de la suite de Lehmer

correspondante.

Par exemple :

```
>>> terme(8,1,3,4,0)
4
>>> terme(8,1,3,4,5)
3
>>> terme(10,3,1,2,4)
2
```

Réponse

```
def terme(m,a,b,x_0,n):
    """ int **5-> int
        Hypothese: m >0 et a,b et u_n appartiennent à [0,m[
    """
    # valeur : int
    valeur = x_0
    for i in range(0,n):
        valeur = suivant(m,a,b,valeur)
    return valeur
```

Question 1.3 : [3/61]

Une suite $(u_n)_{n\in\mathbb{N}}$ est *périodique* quand il existe $k > 0$, tel que pour tout n, $u_{n+k} = u_n$. Dans ce cas, le plus petit entier k vérifiant cette propriété est appelé *période de la suite.*

Dans le cas particulier des suites de Lehmer, on peut montrer qu'une suite de Lehmer de v est périodique si et seulement si l'un des m termes qui suivent u_0 est égal à v. Dans ce cas, la période de la suite est le nombre de termes existants entre u_0 et le premier terme suivant qui prend la valeur v, en le comptant.

Ainsi, la suite de Lehmer de v n'est pas périodique si aucun des m termes qui suivent u_0 n'est égal à v.

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `est_periodique` qui, étant donné 4 entiers naturels m, a, b, et v rend le booléen vrai si la suite ainsi définie est périodique, ou le booléen faux sinon.

Par exemple :

```
>>> est_periodique(8,1,3,4)
True
>>> est_periodique(10,3,1,2)
True
>>> est_periodique(10,5,1,2)
False
```

Réponse

```
def est_periodique(m,a,b,v):
    """ int * int * int * int -> bool
        Hypothese: m >0 et a,b et v appartiennent à [0,m[
    """
    # valeur : int
    valeur = v
    # i : int
    for i in range(0,m):
        valeur = suivant(m, a, b, valeur)
        if valeur == v:
            return True
    return False
```

Question 1.4 : [4/61]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `periode` qui, étant donné 4 entiers naturels m, a, b, et v qui définissent une suite de Lehmer de v qui est périodique, rend la valeur de la période de cette suite.

Par exemple :

```
>>> periode(8,1,3,4)
8
>>> periode(10,3,1,2)
2
```

Réponse

```
def periode(m, a, b, v):
    """ int * int * int * int -> int
        Hypothese: m >0 et a,b et v appartiennent à [0,m[
        Hypothese: la suite est periodique
    """
    # longueur : int
    longueur = 0
    # valeur : int
    valeur = suivant(m, a, b, v)
    while (valeur != v) and (longueur < m):
        valeur = suivant(m,a,b,valeur)
        longueur = longueur +1
    return longueur+1
```

Question 1.5 : [3/61]

Une suite de Lehmer est dite *pleine* si elle prend m valeurs différentes, ce qui équivaut à dire qu'elle est de période m.

En utilisant la fonction `periode`, donner une définition avec signature et hypothèse(s) éventuelle(s) du prédicat `est_pleine` qui, étant donné 4 entiers naturels m, a, b, et v rend le booléen vrai si la suite ainsi définie est pleine, ou le booléen faux dans le cas contraire.

Par exemple :

```
>>> est_pleine(11,2,1,5)
False
>>> est_pleine(10,3,1,7)
False
>>> est_pleine(10,5,1,2)
False
>>> est_pleine(256,25,15,12)
True
```

Réponse

Remarque : compter la moitié des points seulement s'il y a usage d'un `if`

```
def est_pleine(m,a,b,x0):
    """ int * int * int * int -> bool
        Hypothese: m >0 et a,b et u_n appartiennent à [0,m[
        rend True si la suite est de pàriode maximale
    """
    return periode(m,a,b,x0) == m
```

Exercice 2 : Fonction mystère

Question 2.1 : [2/61]

Compléter la définition de `mystere` ci-dessous.

```
def mystere(x, y):
    """ str *                                  ->
        Hypothese :
    """
    # z :
    z = ''
    # t :
    t = ''
    # u :
    u = True
    # i :
    i = 0
    while i < len(x):
        u = u and (x[i] != y)
        if u:
            z = z + x[i]
        else:
            t = t + x[i]
        i = i+1
    return (z, t[1:])
```

Réponse

cette fonction se sert de la première occurrence de y pour couper la chaîne x en 2 parties : la partie du mot avant y et la partie du mot après y. La lettre y est perdue dans le processus.

```
def mystere(x, y):
    """ str * str-> tuple(str, str)
        Hypothese : len(y) == 1
        rend le tuple (lettres avant y dans x, lettres apres y dans x)
        en considerant la premiere occurrence de y
    """
    # z : str
    z = ''
    # t : str
    t = ''
    # u : bool
    u = True
    # i : int
    i = 0
    while i < len(x):
        u = u and (x[i] != y)
        if u:
            z = z + x[i]
        else:
            t = t + x[i]
        i = i+1
    return (z, t[1:])
```

Une version un peu plus commentée :

```
def mystere_mieux(x, y):
    """ str * str-> tuple(str, str)
        Hypothese : len(lettre) == 1
        rend le tuple (lettres avant y dans x, lettres apres y dans x)
        en considerant la premiere occurrence de y
    """
    # x : un mot et y : une lettre
    # z : str
    z = ''          # caracteres avant y dans x
    # t : str
    t = ''          # caracteres apres y dans x
    # u : bool
    u = True        # vrai tant qu'on n'a pas vu y
    # i : int
    i = 0           # position a regarder dans x
    while i < len(x):
        u = u and (x[i] != y)
        if u:       # u est vrai: on n'a pas encore vu y
            z = z + x[i] # le caractere x[i] est avant y
        else:
            t = t + x[i] # le caractere x[i] est apres y
        i = i+1
    return (z, t[1:])    # on enleve le 1er caractere qui est y
```

Question 2.2 : [3/61]

Compléter la table de simulation ci-dessous pour l'application :

`mystere('python','t')`

et donner le résultat rendu par la fonction.

Remarque : la table comporte de lignes que nécessaire.

Tour	Variable `__z___`	Variable `__t___`	Variable `__u___`	Variable `__i____`
entrée				
1er				

Tour	Variable `__z___`	Variable `__t___`	Variable `__u___`	Variable `__i____`	Variable `_`
entrée	`''`	`''`	`True`	0	
1er	`'p'`	`''`	`True`	1	
2e	`'py'`	`''`	`True`	2	
3e	`'py'`	`'t'`	`False`	3	
4e	`'py'`	`'th'`	`False`	4	
5e	`'py'`	`'tho'`	`False`	5	
6e (sortie)	`'py'`	`'thon'`	`False`	6	

Réponse

Barème : 2pts pour la table et 1pt pour le résultat (le `'t'` doit disparaître...)

```
>>> mystere('python','t')
('py', 'hon')
```

Question 2.3 : [4/61]

On cherche un invariant de boucle pour cette fonction. On liste les propositions suivantes :

1. "la longueur de `z` est strictement plus petite que celle de `x`, la longueur de `t` est strictement plus petite que celle de `x` et la somme des longueurs de `z` et `t` est la longueur de `x`."
2. "`z` contient la sous-chaîne des caractères de `x` qui se trouvent avant la position `i` et avant la première occurrence de `y` et `t` contient la sous-chaîne des caractères de `x` qui se trouvent après la première occurrence de `y` et jusqu'à la position `i`"
3. "le i-ème caractère de `x` n'est pas `y` et la somme de `z` et de `t` donne la chaîne `x`"
4. "`u`"

Une seule de ces propositions est un invariant de boucle **pertinent**.

Trouver cette proposition, vérifier qu'elle est un invariant sur la simulation précédente (c'est à dire, calculer l'invariant pour chaque ligne du tableau), et expliquer comment, quand on sort de la boucle, on peut déduire ce que calcule `mystere` à partir de cet invariant.

Réponse

La proposition 2. est un invariant de boucle pertinent.

1. pas pertinent ce n'est vrai qu'en sortie de boucle. C'est "presque"pertinent mais il aurait fallu dire : "la longueur de `z` est strictement plus petite que celle de `x`, la longueur de `t` est strictement plus petite que celle de `x` et la somme des longueurs de `z` et `t` est `i`."
2. pertinent : reste vrai tout le temps et permet de valider le résultat
3. pas toujours, la deuxième partie est vraie en sortie de boucle
4. pas un invariant car `u` peut devenir faux en cours de boucle

Question 2.4 : [3/61]

On s'intéresse à la terminaison de `mystere`. Trouver un variant qui garantit que la fonction termine.

Réponse

un bon variant : `len(x)-i >0` qui décroît jusqu'à la sortie de la boucle.

Question 2.5 : [4/61]

La fonction `mystere` n'est pas optimale car elle réalise, dans tous les cas, un parcours complet de la chaîne `x`, alors que dans beaucoup de cas, cela n'est pas nécessaire : il est inutile de poursuivre l'examen de `x` une fois que l'on a rencontré `y`.

Donner une définition de la fonction `mystere_bis`, de même signature que la fonction `mystere`, qui fournit le même résultat que la fonction `mystere` mais en optimisant le traitement réalisé.

Par exemple :

```
>>> assert mystere_bis('python','t') == mystere('python','t')
>>> assert mystere_bis('hello','z') == mystere('hello','z')
>>> assert mystere_bis('examen','m') == mystere('examen','m')
```

Réponse

Il faut utiliser un `while` pour sortir de la boucle dès que l'on a trouvé `y`, ou alors un `for` avec une sortie anticipée.

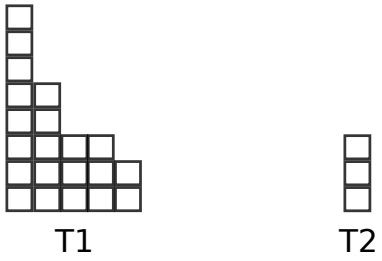
```
def mystere_bis(x, y):
    """ str * str-> tuple(str, str)
        Hypothese : len(y) == 1
        rend le tuple (lettres avant y dans x, lettres apres y dans x)
        en considerant la premiere occurrence de y
    """
    # i : int
    i = 0
    while (i < len(x)) and x[i] != y:
        i = i+1
    if i == len(x):
        return (x, '')
    return (x[0:i], x[i+1:])
```

ou

```
def mystere_for(x, y):
    """ str * str-> tuple(str, str)
        Hypothese : len(y) == 1
        rend le tuple (lettres avant y dans x, lettres apres y dans x)
        en considerant la premiere occurrence de y
    """
    # i : int
    for i in range(0, len(x)):
        if y == x[i]:
            return(x[0:i], x[i+1:])
    return (x, '')
```

Exercice 3 : Tas de sable

Le but de cet exercice est de définir une fonction permettant de calculer le résultat de l'éboulement d'un tas de sable. On considère ici que les tas de sable sont constitués de plusieurs colonnes de grains de sable côte à côte. Chaque colonne est elle-même constituée d'un certain nombre de grains de sable.
On choisit donc de représenter un tas de sable par une liste d'entiers naturels, chaque entier représentant le nombre de grains dans une colonne.
Soit les deux exemples de tas de sable `T1` et `T2` suivant :

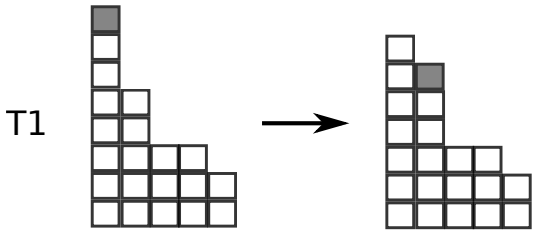


Le tas `T1` sera donc représenté par la liste `(8 5 3 3 2)` et `T2` par la liste `(3)`.

Dans la suite, on supposera toujours que **les colonnes sont rangées par hauteur décroissante**. Autrement dit la liste d'entiers qui représente le tas est décroissante (au sens large). Il ne sera pas nécessaire de rappeler cette condition en hypothèse des fonctions de l'exercice.

Un éboulement est possible dans un tas si les hauteurs de deux colonnes adjacentes diffèrent de 2 grains de sable ou plus. Un éboulement n'est pas possible à partir de la dernière colonne.

Si un éboulement est possible entre deux colonnes de sable, celui-ci a pour effet de transférer un grain de sable de la colonne la plus haute (celle de gauche) vers la colonne la plus basse (celle de droite). En reprenant l'exemple du tas `T1`, le premier éboulement possible est le suivant :



Le grain en haut de la première colonne (en gris)
tombe sur la deuxième colonne

Question 3.1 : [3/61]

Donner une définition avec signature et hypothèse(s) éventuelle(s) des fonctions suivantes :

- `eboult_possible` qui, étant donné un tas contenant au moins une colonne de sable, teste si un éboulement est possible.
- `indice_eboult` qui, étant donné un tas contenant au moins un éboulement possible, renvoie l'indice de la première colonne permettant un éboulement.

Par exemple :

```
>>> eboult_possible([3])
False
>>> eboult_possible([8, 7, 7, 6, 5])
False
>>> eboult_possible([8, 5, 3, 3, 2])
True
>>> indice_eboult([8, 5, 3, 3, 2])
0
>>> eboult_possible([8, 7, 6, 3, 2])
True
>>> indice_eboult([8, 7, 6, 3, 2])
2
```

Remarque : la totalité des points ne sera allouée qu'aux fonctions qui contiennent des sorties anticipées.

Réponse

La question n'est notée que sur 2.5 pt s'il n'y a pas de sortie anticipée (de boucle ou de fonction).

```
def eboult_possible(L):
    """ list[int] -> bool
    hyp: len(L) >= 1"""

    #i: int
    i=0

    while i < len(L)-1:
        if L[i] - L[i+1] >=2:
            return True
        i = i + 1

    return False

def indice_eboult(L):
    """ list[int] -> int
    hyp: len(L) >= 1
    hyp: un eboulement est possible """

    #i: int
    i=0

    while i < len(L)-1:
        if L[i] - L[i+1] >=2:
            return i
        i = i + 1

    return i
```

Question 3.2 : [3/61]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `eboult_col` qui, étant donné un tas `T` et un indice `i` correspondant à un éboulement valide, réalise l'éboulement de la colonne `i` dans `T`.

Par exemple :

```
>>> eboult_col([8, 5, 3, 3, 2], 0)
[7, 6, 3, 3, 2]
>>> eboult_col([8, 7, 1], 1)
[8, 6, 2]
```

Réponse

```
def eboult_col(L, i):
    """ list[int] -> list[int]
    hyp: len(L) >= i+2 and L[i]-L[i+1] >=2"""

    #LR : list[int]
    LR = L[0:i] + [L[i] - 1, L[i+1] + 1] + L[i+2:]

    return LR
```

Question 3.3 : [3/61]

Un tas est dit *stable* s'il n'y a plus d'éboulement possible. En vous servant des fonctions précédentes, donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `stabilisation` qui, étant donné un tas contenant au moins une colonne de sable, réalise les éboulements successifs dans le tas jusqu'à ce que le tas soit stabilisé.

Par exemple :

```
>>> stabilisation([3])
[3]
>>> stabilisation([8, 5, 3, 3, 2])
[6, 5, 4, 3, 3]
>>> stabilisation([8, 7, 1])
[6, 5, 5]
```

Réponse

```
def stabilisation(L):
    """ list[int] -> list[int]
    hyp: len(L) >= 1"""

    #LR : list[int]
    LR = L

    #i: int
    i = 0

    while eboult_possible(LR):
        i = indice_eboult(LR)
        LR = eboult_col(LR, i)

    return LR
```

Exercice 4 : Expositions

Dans cet exercice, on s'intéresse à la définition de fonctions permettant de gérer les artistes exposés dans des musées. Dans une telle base de gestion, chaque musée est associé à l'ensemble des artistes exposés. À titre d'exemple, on considère la base d'expositions suivante décrivant des artistes exposés dans des musées parisiens :

Musée	Artistes
Centre Pompidou	Chagall, Delaunay, Matisse, Miro, Munch, Picasso
Musée d'Art Moderne	Delaunay, Picabia, Picasso, Warhol
Musée de l'Orangerie	Matisse, Picasso
Musée d'Orsay	Munch
Musée Picasso	Picasso, Braque, Miro

Une base de données d'expositions est représentée en Python par un dictionnaire ayant :

— comme clés : le nom des musées
— comme valeurs associées : l'ensemble des artistes exposés dans le musée

Ainsi, l'exemple précédent donne le dictionnaire `Paris` suivant :

```
>>> Paris = {
'Centre Pompidou' : {'Chagall', 'Delaunay',  'Matisse', 'Miro', 'Munch',
  'Picasso'},
"Musee d'Art Moderne" : {'Delaunay', 'Picabia', 'Picasso', 'Warhol'},
"Musee de l'Orangerie" : {'Matisse', 'Picasso'},
"Musee d'Orsay" : {'Munch'},
'Musee Picasso' : {'Picasso', 'Braque', 'Miro'}}
```

Question 4.1 : [2/61]

Donner le type en python de la base de données d'expositions.

Réponse

```
dict[str : set[str]]
```

Dans la suite de cet exercice, on pourra utiliser l'alias de type `Exposition` pour le type d'une base de données d'expositions.

Question 4.2 : [3/61]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `tous_artistes` qui, étant donné une base de données d'expositions, renvoie l'ensemble des artistes exposés dans l'un des musées.

Par exemple :

```
>>> tous_artistes(Paris)
{'Chagall', 'Delaunay', 'Munch', 'Picasso', 'Warhol', 'Picabia', 'Braque',
'Matisse', 'Miro'}
```

Réponse

```
def tous_artistes(Expo):
    """ Exposition -> set[str] """

    #ER : set[str]
    ER = set()

    #m: str
    for m in Expo:
        ER = ER | Expo[m]

    return ER
```

Variante avec compréhension :

```
def tous_artistes_comp(Expo):
    """ Exposition -> set[str] """

    return {a for m in Expo for a in Expo[m]}
```

Question 4.3 : [3/61]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `expose_artiste` qui, étant donné une base de données d'exposition E et le nom d'un artiste a, renvoie l'ensemble des musées qui exposent a.

Par exemple :

```
>>> expose_artiste(Paris, 'Chagall')
{'Centre Pompidou'}
>>> expose_artiste(Paris, 'Picasso')
{'Centre Pompidou', "Musee d'Art Moderne", 'Musee Picasso', "Musee de l'Orangerie"}
>>> expose_artiste(Paris, 'Rubens')
{}
```

Réponse

```
def expose_artiste(Expo, artiste):
    """ Exposition * srt -> set[str] """

    #ER : set[str]
    ER = set()

    #m: str
    for m in Expo:
        if artiste in Expo[m]:
            ER.add(m)

    return ER
```

Variante avec compréhension :

```
def expose_artiste_comp(Expo, artiste):
    """ Exposition * srt -> set[str] """

    return {m for m in Expo if artiste in Expo[m]}
```

Question 4.4 : [4/61]

On aimerait disposer d'une seconde base de données permettant de rechercher des informations à partir du nom d'un artiste. Une telle base de données, appelée *base de données d'artistes*, est représentée en Python par le type `dict[str:set[str]]` dans lequel une clé est un nom d'artiste dont la valeur associée est l'ensemble des musées dans lesquels il (ou elle) est exposé.

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `conversion` qui, étant donné une base de données d'exposition, renvoie la base de données d'artistes associée.

Par exemple :

```
>>> conversion(Paris)
{'Miro': {'Centre Pompidou', 'Musee Picasso'},
'Matisse' : {"Musee de l'Orangerie", 'Centre Pompidou'},
'Munch': {'Centre Pompidou', "Musee d'Orsay"},
'Braque': {'Musee Picasso'},
'Picabia': {"Musee d'Art Moderne"},
'Warhol': {"Musee d'Art Moderne"},
'Picasso': {'Centre Pompidou', "Musee de l'Orangerie", 'Musee Picasso',
  "Musee d'Art Moderne"},
'Chagall': {'Centre Pompidou'},
'Delaunay': {'Centre Pompidou', "Musee d'Art Moderne"},
```

Réponse

```
def conversion(Expo):
    """ Exposition -> dict[set:set[str]]"""

    # DR : dict[str : set[str]}
    DR = dict()

    # artistes : set[str]
    artistes = tous_artistes(Expo)

    # a : str
    for a in artistes:
        DR[a] = expose_artiste(Expo,a)

    return DR
```

Variante avec compréhension :

```
def conversion_comp(Expo):
    """ Exposition -> dict[set:set[str]]"""

    return { a:expose_artiste(Expo, a) for a in tous_artistes(Expo) }
```

Question 4.5 : [4/61]

Donner une définition avec signature et hypothèse(s) éventuelle(s) de la fonction `exposition_unique` qui, étant donné une base de données d'expositions E, renvoie une nouvelle base de données d'expositions dont les clés sont toujours les musées mais la valeur associée est l'ensemble des artistes n'étant exposés **que** dans ce musée.

Par exemple :

```
>>> exposition_unique(Paris)
{"Musee d'Art Moderne": {'Warhol', 'Picabia'},
  'Centre Pompidou': {'Chagall'},
  'Musee Picasso': {'Braque'},
  "Musee de l'Orangerie": set(),
  "Musee d'Orsay": set()}
```

Remarque : vous pourrez notament vous aider de la fonction précédente.

Réponse

```
def exposition_unique(Expo):
    """ Exposition -> dict[set:set[str]]"""

    # DR : dict[str : set[str]}
    DR = dict()

    # artiste: dict[str : set[str]]
    artiste = conversion(Expo)

    #m: str
    for m in Expo:
        DR[m] = {a for a in Expo[m] if len(artiste[a]) == 1}

    return DR
```

Genres artistiques

On suppose à partir de maintenant que l'on dispose d'une seconde base de données qui décrit les catégories artistiques auxquelles appartiennent les artistes. Par exemple :

Genre	Artistes
Pop art	Warhol
Cubisme	Picasso, Braque, Delaunay
Expressionnisme	Munch
Fauvisme	Braque, Matisse
Réalisme	Munch, Picabia
Surréalisme	Chagall, Miro, Picasso

Cette base de données de genres est obtenue en python avec la définition suivante :

```
>>> Genre = {
  'Pop art' : {'Warhol'},
  'Cubisme' : {'Braque', 'Picasso', 'Delaunay'},
  'Expressionnisme' : {'Munch'},
  'Fauvisme' : {'Braque', 'Matisse'},
  'Realisme' : {'Munch', 'Picabia'},
  'Surrealisme' : {'Chagall', 'Miro', 'Picasso'}}
```

Question 4.6 : [5/61]

Donner une définition avec signature et hypothèse éventuelle(s) de la fonction `expose_genre` qui, étant donné une base de données d'expositions, une base de données de genres G et un genre artistique appartenant à G, renvoie l'ensemble des musées qui exposent un artiste de ce genre artistique.

Par exemple :

```
>>> expose_genre(Paris, Genre, 'Pop art')
{"Musee d'Art Moderne"}
>>> expose_genre(Paris, Genre, 'Surrealisme')
{"Musee d'Art Moderne", 'Centre Pompidou', 'Musee Picasso', "Musee de l'Orangerie"}
```

Réponse

```
def expose_genre(Expo, Genre, g):
    """ Exposition * dict[str:set[str]] * str  -> set[str]
        hyp: g in Genre    """

    #ER : set[str]
    ER = set()

    #m : str
    for m in Expo:
        #a : str
        for a in Expo[m]:
            if a in Genre[g]:
                ER.add(m)

    return ER
```

Variante avec compréhension :

```
def expose_genre_comp(Expo, Genre, g):
    """ Exposition * dict[str:set[str]] * str  -> set[str]
        hyp: g in Genre    """

    return {m for m in Expo for a in Expo[m] if a in Genre[g]}
```