

--

Examen 2021 – 2022-subst
Éléments de Programmation I + – LU1IN011
Durée : 1h30

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

Le sujet comporte 22 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 : 56 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 est possible, et souvent utile, d'utiliser les fonctions qui sont l'objet des questions précédentes, même si vous n'avez pas 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 : Sauf en cas d'exception (notamment dans les exercices d'analyse de code), pour les fonctions demandées, **il est nécessaire de donner une définition avec précondition(s) éventuelle(s).** En revanche, la description textuelle et les jeux de tests ne sont *pas* demandés, contrairement aux exercices sur machine.

L'examen est composé de 4 exercices indépendants :

- Listes récursives (p. 2)
- Départements (Dictionnaires) (p. 8)
- Séparation (Analyse) (p. 13)
- Pixels (Listes de n -uplets) (p. 17)

Exercice 1 : Listes récursives

Il est possible de manipuler les listes Python de façon purement récursive. Pour cela, nous introduisons les fonctions suivantes :

```
def cons(e : T, lst : List[T]) -> List[T]:
    """ """
    return [e] + lst

def first(lst : List[T]) -> T:
    """précondition: len(lst) > 0"""
    return lst[0]

def rest(lst : List[T]) -> List[T]:
    """précondition: len(lst) > 0"""
    return lst[1:]

def empty(lst : List[T]) -> bool:
    """ """
    return len(lst) == 0
```

On a par exemple :

```
>>> cons('a', [])
['a']
>>> cons(1, [2, 3, 4])
[1, 2, 3, 4]
>>> cons(1, cons(2, cons(3, cons(4, []))))
[1, 2, 3, 4]

>>> first(cons('a', []))
'a'
>>> first(cons(1, [2, 3, 4]))
1

>>> rest(cons('a', []))
[]
>>> rest(cons(1, [2, 3, 4]))
[2, 3, 4]

>>> empty([])
True
>>> empty(cons('a', []))
False
>>> empty(rest(cons('a', [])))
True
>>> empty(cons(1, [2, 3, 4]))
False
>>> empty(rest(cons(1, [2, 3, 4])))
False
```

Question 1.1 : [2/56]

Donner une description concise de chacune des fonctions définies précédemment.

Important : dans le reste de cet exercice, toutes les fonctions doivent être définies de façon **récursive**. Il est ainsi interdit d'utiliser les boucles `while ...` ou `for ... in ...`. De plus, pour manipuler les listes, seules les fonctions `cons`, `first`, `rest` et `empty` sont disponibles. Il est interdit d'utiliser la fonction `len`, la concaténation `+`, les découpages, les compréhensions ou toute autre opération de liste vue en cours. La liste vide `[]` est cependant disponible.

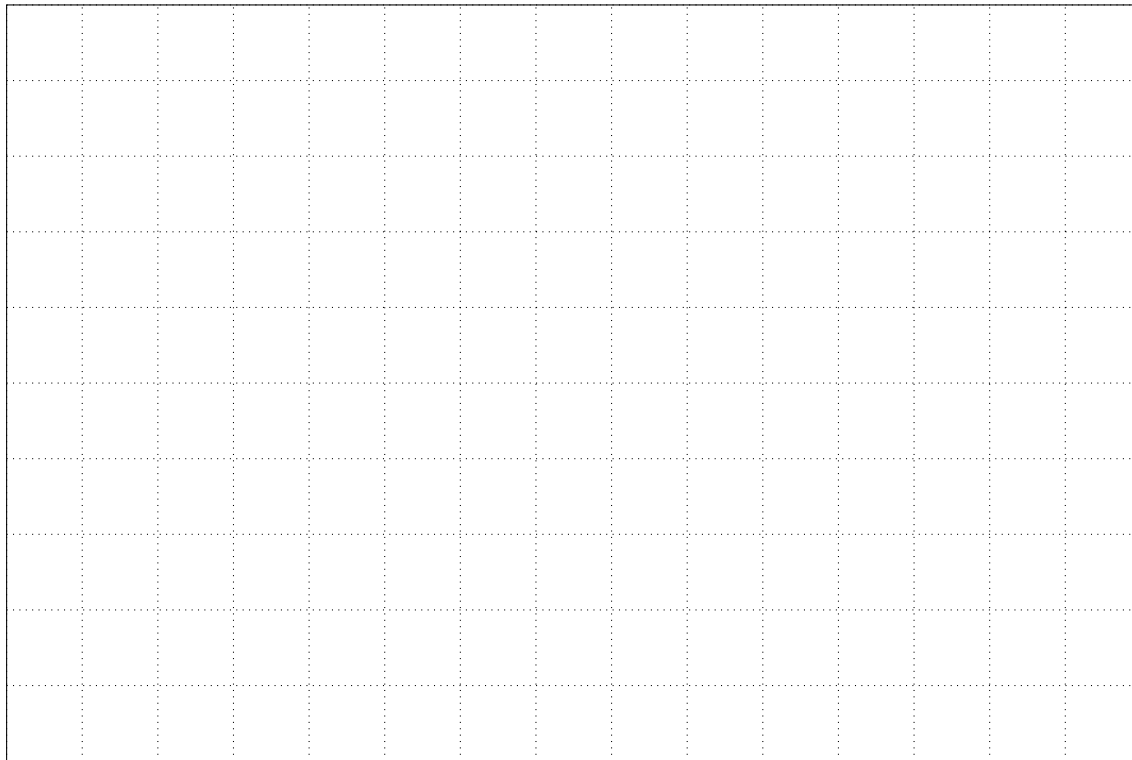
Question 1.2 : [2/56]

Donner une définition **récursive** de la fonction `length` qui, à partir d'une liste `lst` contenant des éléments d'un type arbitraire, retourne sa longueur c'est-à-dire le nombre de ses éléments.

Par exemple :

```
>>> length([])
0
>>> length(cons('a', []))
1
>>> length(cons(1, cons(2, cons(3, cons(4, [])))))
4
```

Rappel : votre définition doit être récursive et il n'est bien sûr pas possible d'utiliser la primitive `len`.



Question 1.3 : [3/56]

Donner une définition **récursive** de la fonction **last** qui, à partir d'une liste **lst** non-vide retourne son dernier élément.

Par exemple :

```
>>> last(cons('a', []))  
'a'
```

```
>>> last(cons(1, cons(2, cons(3, cons(4, [])))))  
4
```

Rappel : votre définition doit être récursive et il est interdit d'utiliser les découpages de listes.



Question 1.4 : [4/56]

Soient les deux fonctions suivantes :

```
def plusun(n : int) -> int:  
    """Retourne le successeur de n."""  
    return n + 1
```

```
def foisdeux(n : int) -> int:  
    """Retourne le double de n"""  
    return n * 2
```

Donner tout d'abord une définition de la fonction **lstplusun** qui, à partir d'une liste **lst** d'entiers, retourne la liste consistant à ajouter un à chacun des éléments de **lst**.

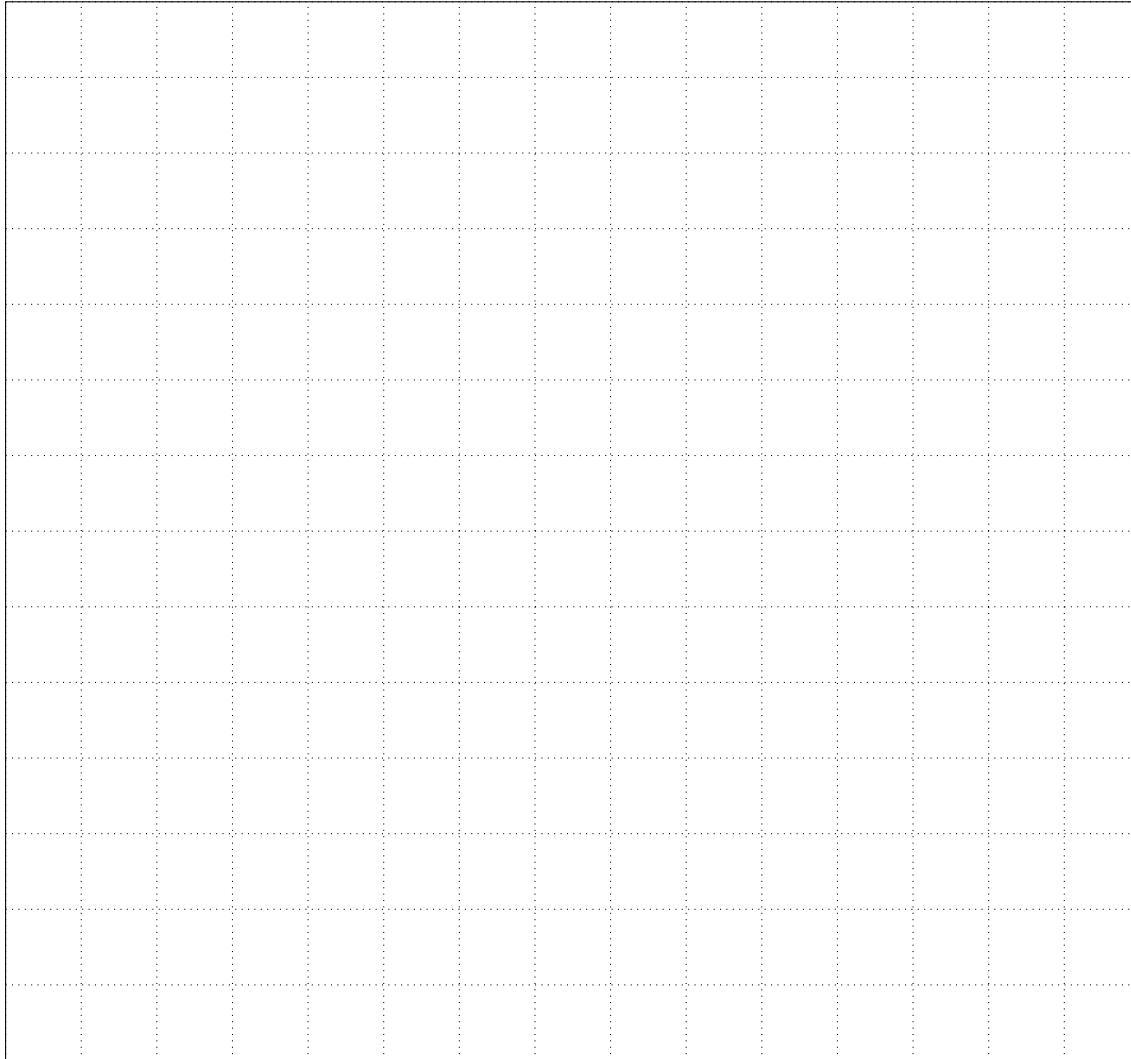
Par exemple :

```
>>> lstplusun(cons(1, cons(2, cons(3, cons(4, []))))  
[2, 3, 4, 5]  
>>> lstplusun([])  
[]
```

Remarque : votre définition devra être récursive et utiliser la fonction **plusun** ci-dessus.

De plus, expliquer ce qu'il faudrait modifier pour définir la fonction **lstfoisdeux** permettant de construire une liste des doubles comme dans les exemples suivants :

```
>>> lstfoisdeux(cons(1, cons(2, cons(3, cons(4, []))))
[2, 4, 6, 8]
>>> lstfoisdeux([])
[]
```



Question 1.5 : [5/56]

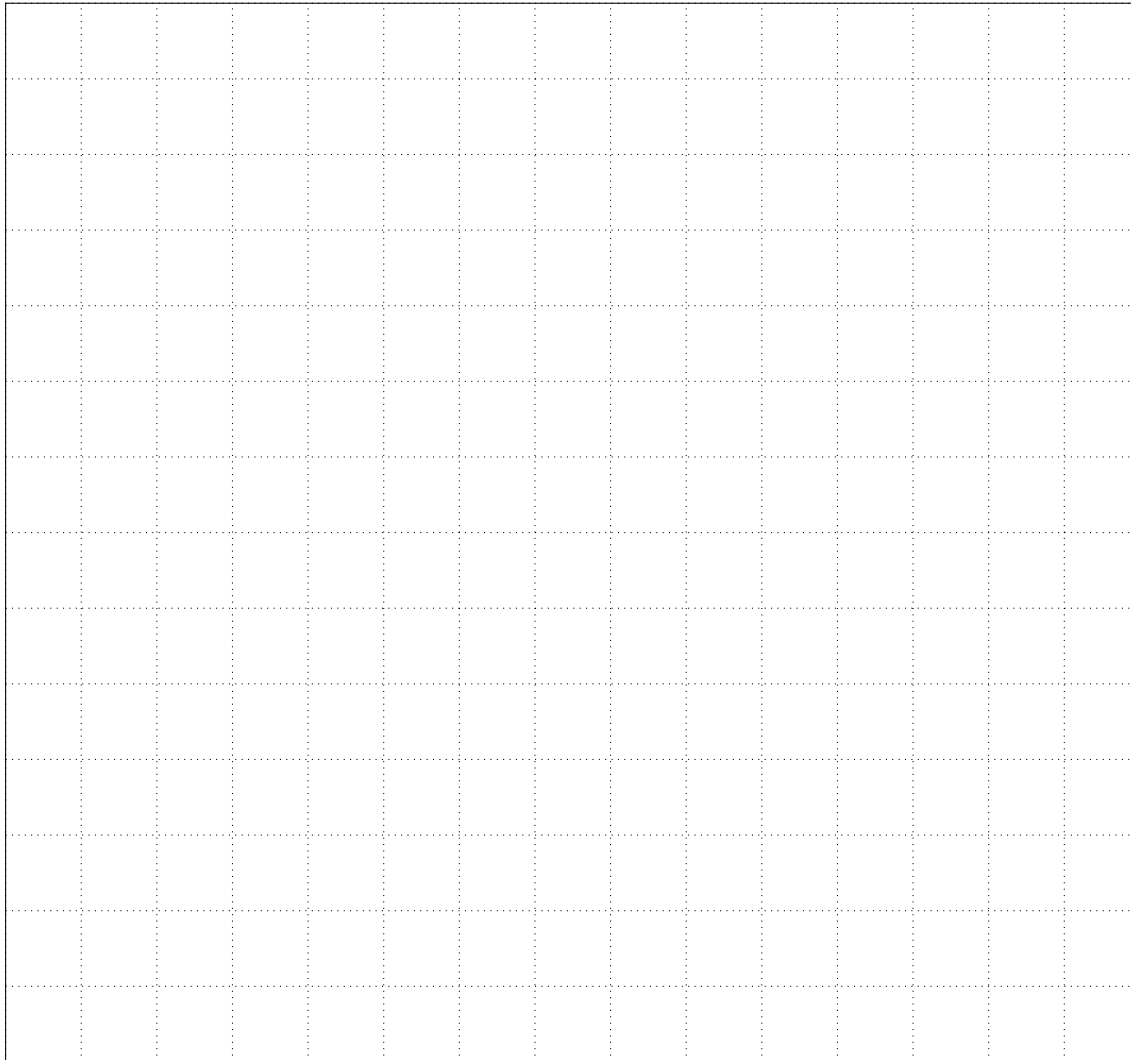
En généralisant les définitions de la question précédente, donner une définition **récursive** de la fonction `lstmap` prenant deux arguments :

- une fonction quelconque `f` prenant un unique argument de type arbitraire `T` en entrée, et avec le type de retour `U`,
- une liste `lst` de type `List[T]`

et qui retourne la liste consistant en l'application de la fonction `f` à chacun des éléments de `lst`.

Par exemple :

```
>>> lstmap(plusun , [1, 2, 3, 4])
[2, 3, 4, 5]
>>> lstmap(foisdeux , [1, 2, 3, 4])
[2, 4, 6, 8]
>>> lstmap(plusun , [])
[]
>>> lstmap(foisdeux , [])
[]
```



Exercice 2 : Départements

On représente les effectifs étudiants d'un département universitaire par un *dictionnaire* dont les clefs sont les noms d'UEs - des chaînes de caractères - proposées par ce département, et la valeur associée à une UE est l'*ensemble* des prénoms - une chaîne de caractères - des étudiants inscrits dans cette UE.

On appelle *départements* de tels dictionnaire et on utilise l'alias de type suivant :

```
Dep = Dict[str, Set[str]]
```

Voici un exemple de département :

```
LicenceInfo : Dep = { "BDD" : { "Alice", "Bob", "Carole"},
                      "Lambda" : { "Alice", "Bob", "Carole", "David", "Elise"},
                      "POO" : { "Bob", "Elise"},
                      "IA" : set (),
                      "Compil" : { "Alice", "Bob", "David"}}
```

Dans lequel on comprend, entre autres,

- qu'Alice est inscrite aux UEs "BDD", "Lambda" et "Compil"
- qu'Alice n'est pas inscrite en "POO",
- que personne n'est inscrit en "IA".

Question 2.1 : [2/56]

Donner une définition de la fonction `effectifs_UE` qui prend en entrée un département `d`, un nom d'UE `ue` et qui renvoie le nombre d'étudiants inscrits à l'UE `ue` dans `d`. Si `ue` n'apparaît pas dans `d`, on renvoie 0.

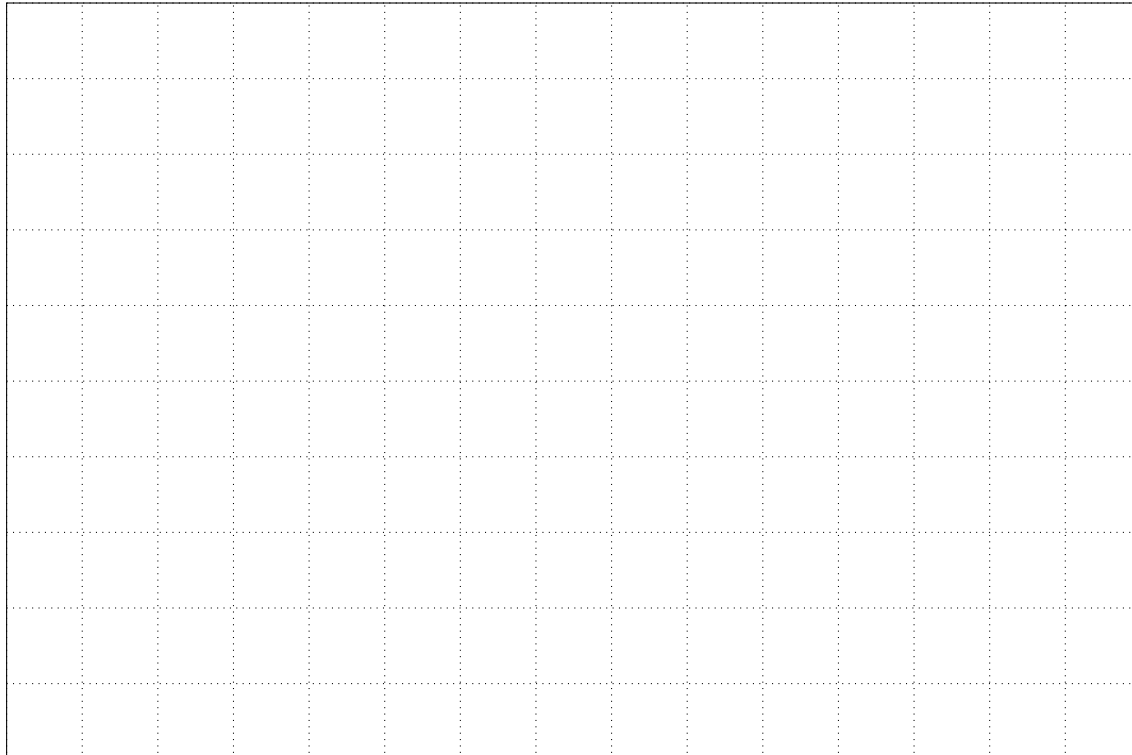
```
>>> effectifs_UE(LicenceInfo, "Lambda")
5
>>> effectifs_UE(LicenceInfo, "IA")
0
>>> effectifs_UE(LicenceInfo, "Microbio")
0
>>> effectifs_UE(dict(), "Lambda")
0
```

A blank sheet of graph paper featuring a uniform grid of small squares. The grid consists of 10 columns and 6 rows, creating a total of 60 square units. The lines are thin and light gray, set against a white background. There are no margins or additional markings on the page.

Question 2.2 : [3/56]

Donner la définition d'une fonction `etudiants` qui prend en entrée un département `d` et renvoie l'ensemble de tous les étudiants inscrits à au moins une UE dans ce département.

```
>>> étudiants(LicenceInfo)
{'Carole', 'Elise', 'Alice', 'Bob', 'David'}
>>> étudiants(dict())
set()
```



Question 2.3 : [2/56]

Donner une définition de la fonction `inscriptions_etu` qui prend en entrée un département `d` et un prénom d'étudiants `etu` et qui renvoie l'ensemble des UEs de `d` auxquelles `etu` est inscrit.

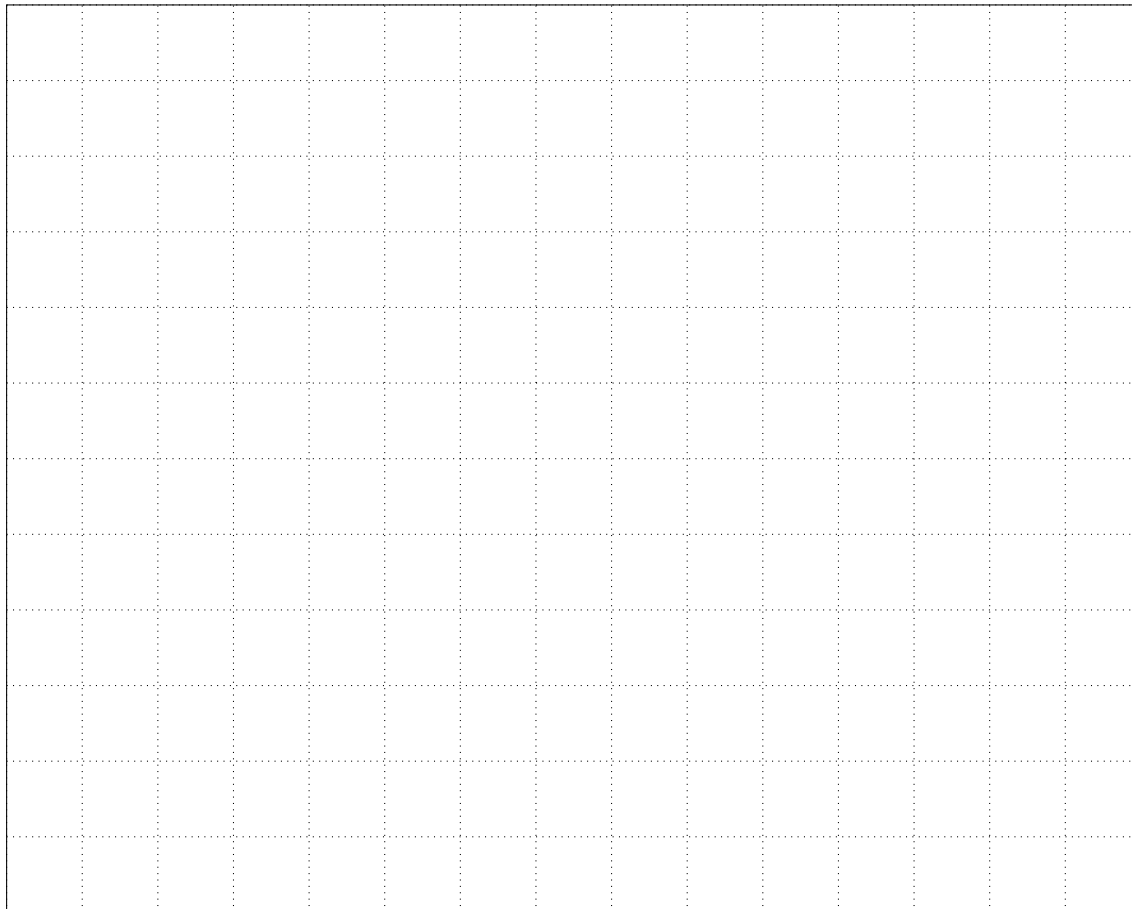
Barème : Cette fonction sera notée sur 2 points supplémentaires si elle utilise, de manière pertinente, une *compréhension*.

```
>>> inscriptions_etu(LicenceInfo, "Bob")
{"BDD", "Lambda", "POO", "Compil"}
>>> inscriptions_etu(LicenceInfo, "Elise")
{"POO", "Lambda"}
>>> inscriptions_etu(LicenceInfo, "Fadia")
set()
```

Question 2.4 : [4/56]

Donner une définition de la fonction `inscriptions_tous` qui prend en entrée un département `d` et renvoie un dictionnaire dont les clefs sont les prénoms des étudiants inscrits aux UE de `d` et la valeur associée à l'étudiant `etu` est l'ensemble des UEs de `d` auxquelles est inscrit `etu`.

```
>>> inscriptions_tous(LicenceInfo)
{'Alice': {'Compil', 'BDD', 'Lambda'},
 'Bob': {'Compil', 'BDD', 'POO', 'Lambda'},
 'Carole': {'BDD', 'Lambda'},
 'David': {'Compil', 'Lambda'},
 'Elise': {'POO', 'Lambda'}}
```



Question 2.5 : [5/56]

On manipule maintenant des *listes de départements*. Par exemple, la liste `Faculte2Science` définie ainsi

```
LicenceBio : Dep = {"Microbio" : {"Fadia", "David"},  
                  "Genetique" : {"Fadia", "David"},  
                  "Animale" : {"Fadia"}}
```

```
LicenceMath : Dep = {"Topologie" : {"Elise", "Gwenael"},  
                   "AlgLin" : {"Elise", "Gwenael"}}
```

```
Faculte2Science : List[Dep] = [LicenceBio, LicenceInfo, LicenceMath]
```

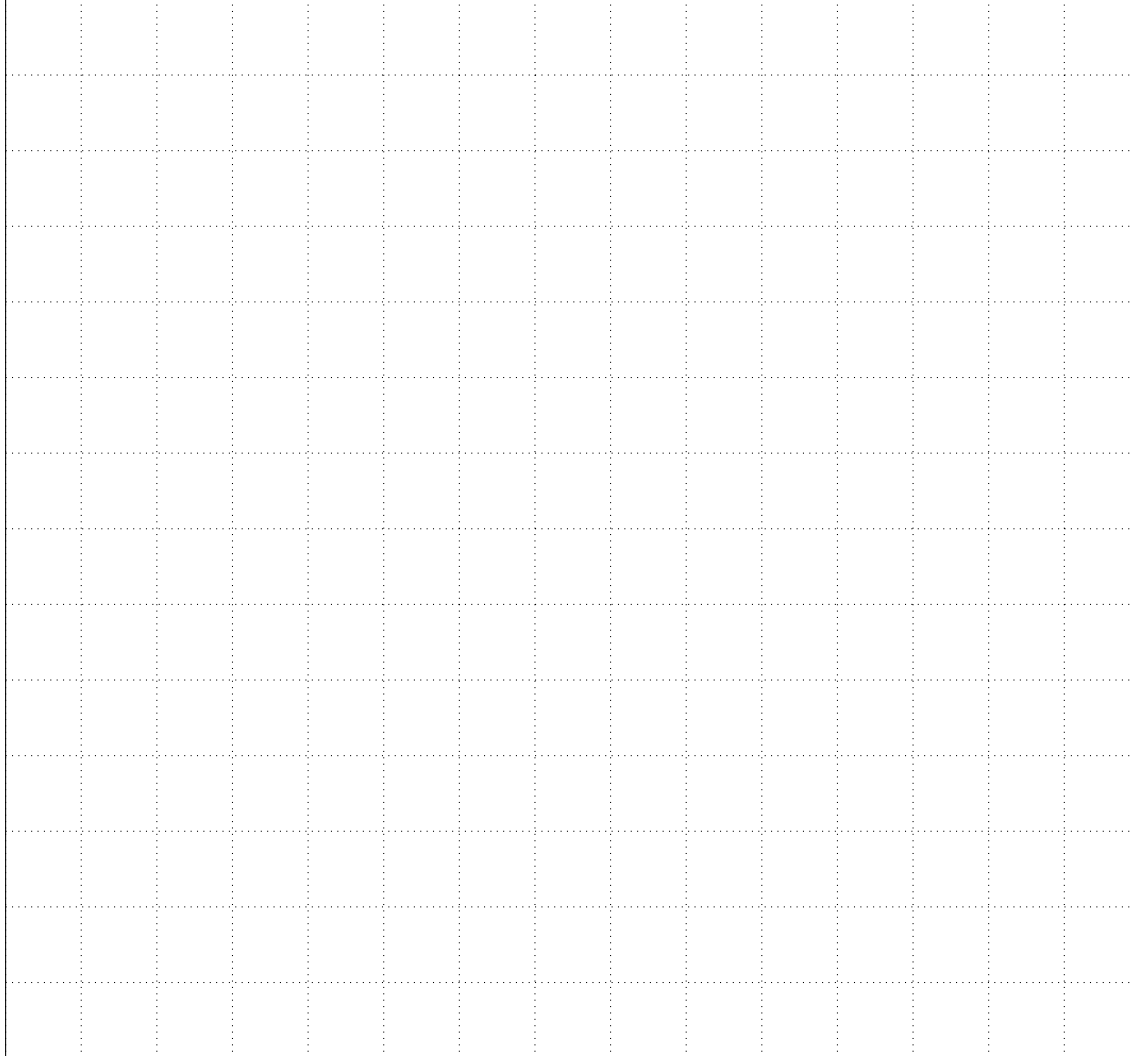
Donner une définition de la fonction `doubles_licences` qui prend en entrée une liste de départements et renvoie l'ensemble des prénoms des étudiants qui sont inscrits dans (au moins) deux UEs appartenant à des départements différents.

```
>>> doubles_licences(Faculte2Science)  
{"David", "Elise"}
```

En effet :

- David est inscrit en *Lambda* du département `LicenceInfo` et en *MidcroBio* du département `LicenceBio`.

- Elise est inscrite en *POO* du département **LicenceInfo** et en *Topologie* du département **LicenceMath**,
- chaque autre étudiant est inscrit à des UEs d'un même département.



Exercice 3 : Séparation équitable

Une *séparation* d'une liste `li` d'entiers strictement positifs est un couple de listes $(l1, l2)$ tel que `li` et `l1 + l2` possèdent les mêmes éléments (avec le même nombre d'occurrences, pas forcément dans le même ordre).

Par exemple $([3, 4], [4, 5])$ est une séparation de la liste `[3, 5, 4, 4]`.

Dans cet exercice, on s'intéresse au problème suivant :

Soit une liste d'entiers strictement positifs `li`, est-il possible de séparer les éléments de `li` en deux listes `l1` et `l2` tels que la somme des éléments de `l1` soit égale à la somme des éléments de `l2` ?

Dans la suite, on appelle *séparation équitable* un tel couple $(l1, l2)$.

Par exemple :

- Une séparation équitable de la liste `[1, 2, 1]` est $([1, 1], [2])$. La somme de chacune des deux listes vaut 2.
- Une séparation équitable de la liste `[3, 5, 4, 4]` est $([3, 5], [4, 4])$. La somme de chacune des deux listes vaut 8
- Il n'existe pas de séparation équitable de la liste `[1, 2, 1, 1]` (c'est facile à voir, car la somme des éléments de cette liste est impaire).
- Il n'existe pas de séparation équitable de la liste `[3, 3, 6, 2]` (c'est plus difficile à voir, on peut essayer les différentes séparations possibles pour s'en rendre compte).

On propose la fonction suivante pour résoudre le problème :

```
def sep_eq_possible(li : ) ->  :  
    """ decide s'il existe une separation  
    equitable de li """  
  
    s1 :  = 0  
    s2 :  = 0  
    i :  = 0  
    while i < len(li):  
        if s1 <= s2 :  
            s1 = s1 + li[i]  
        else :  
            s2 = s2 + li[i]  
        i = i + 1  
    return s1 == s2
```

Question 3.1 : [1/56]

Remplir les annotations de types de `sep_eq_possible`.

Question 3.2 : [3/56]

Remplir le tableau ci-dessous pour la simulation de l'appel :

```
sep_eq_possible([2, 5, 2, 3, 2])
```

Dans la colonne `li[i]` on donnera la valeur de l'expression `li[i]` quand elle existe, et - sinon.

[illegible]

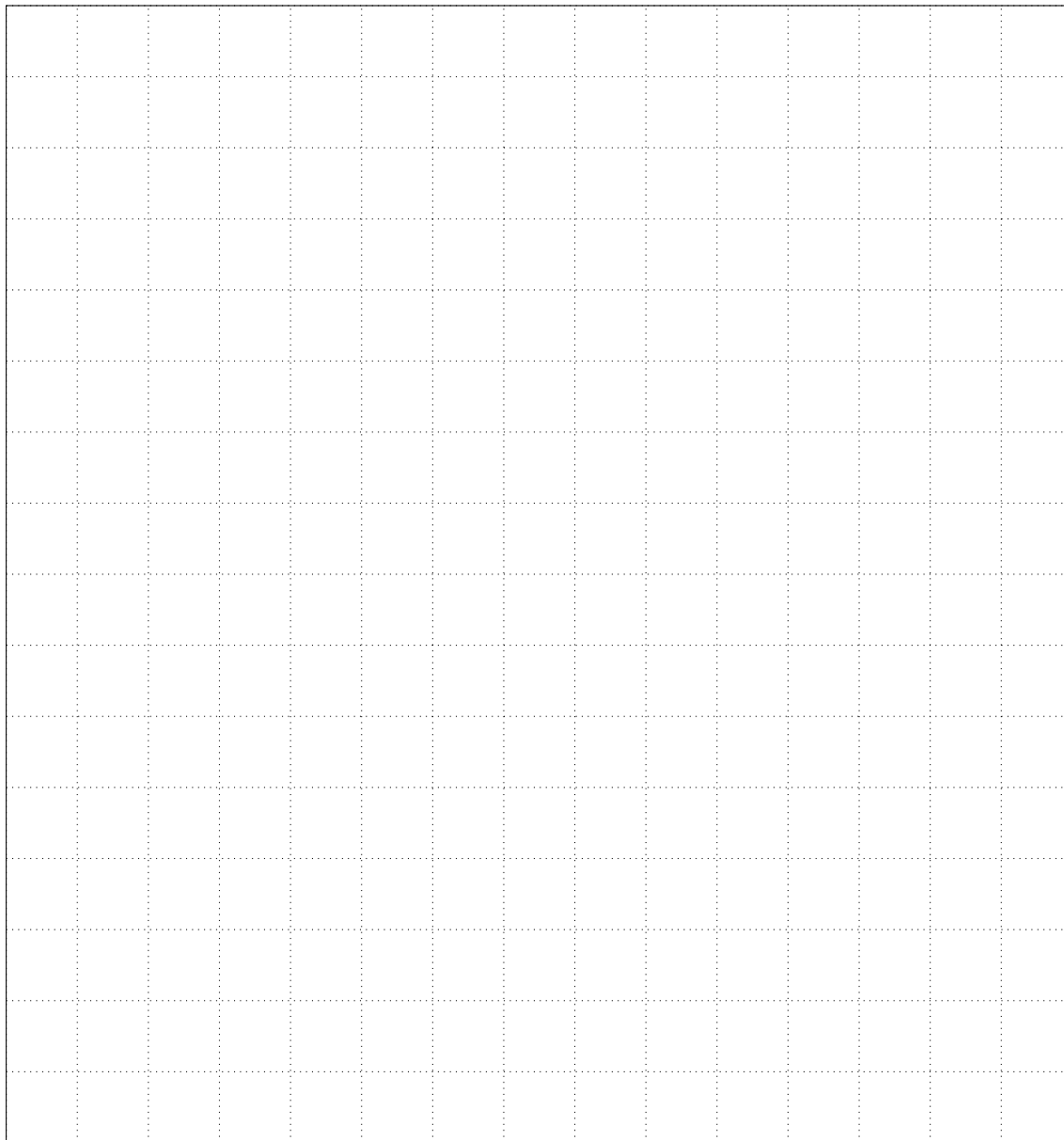
Valeur de `sep_eq_possible([2, 5, 2, 3, 2])` :

Question 3.3 : [3/56]

La fonction `sep_eq_possible` renvoie uniquement un booléen indiquant si elle a trouvé une séparation équitable.

Modifier la fonction `sep_eq_possible` - on appellera `sep_eq` cette nouvelle fonction - pour qu'elle renvoie une séparation équitable quand elle en trouve une; c'est-à-dire quand `sep_eq_possible` vaut `True`, `sep_eq` renvoie un couple de listes correspondant à la séparation équitable, quand `sep_eq_possible` vaut `False`, `sep_eq` renvoie `None`.

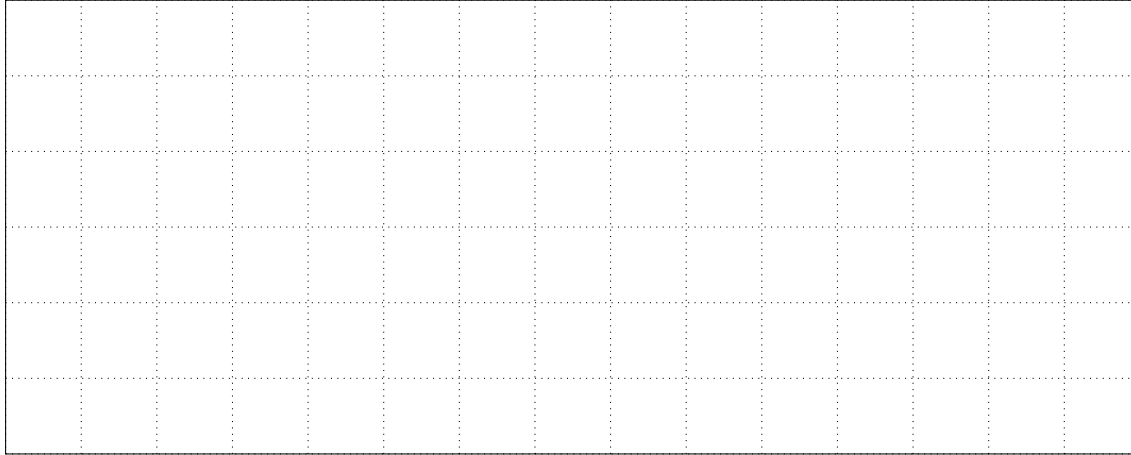
```
>>> sep_eq([2, 4, 2])
([2, 2], [4])
>>> sep_eq([1, 1, 1])
None
```



Question 3.4 : [2/56]

Expliquer, sans détailler la simulation (qu'on pourra faire au brouillon), ce que vaut `sep_eq_possible([3, 5, 4, 4])`.

`sep_eq_possible([3, 5, 4, 4])`

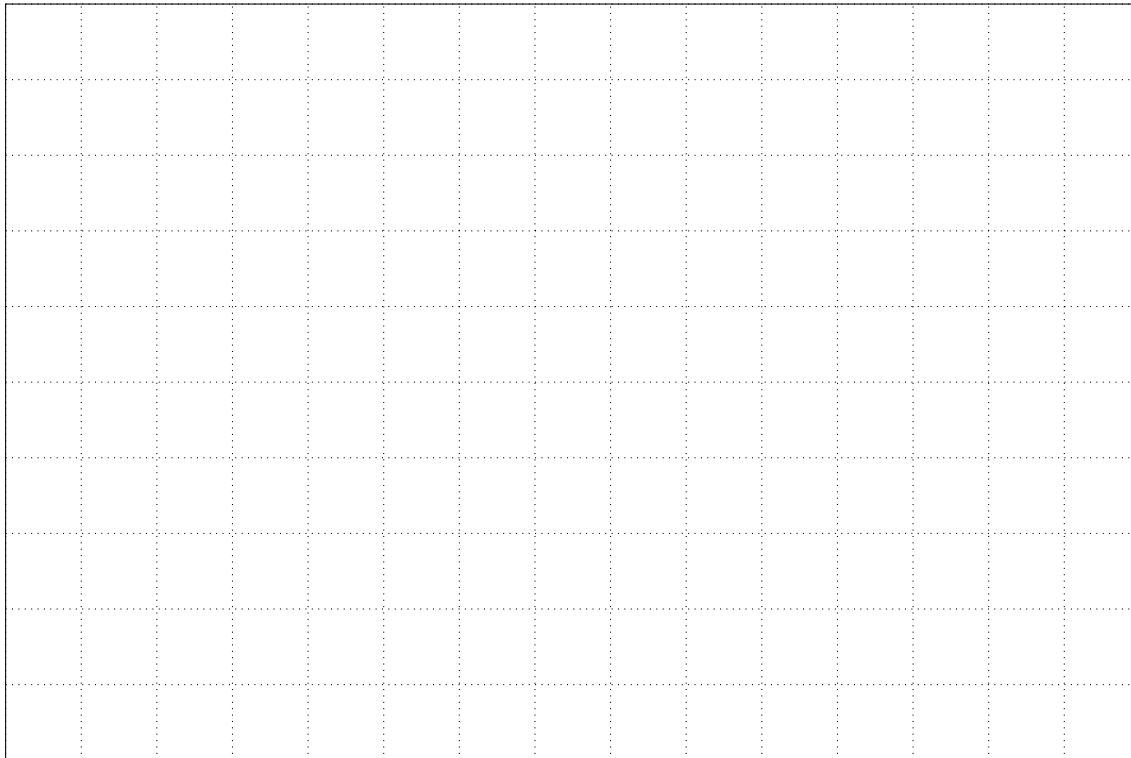


Question 3.5 : [2/56]

La correction de `sep_eq_possible` s'exprime par :

`sep_eq_possible(li)` vaut `True` si, et seulement si, il existe une séparation équitable de `li`.

La fonction est-elle correcte ? Si oui, donner un invariant de boucle pour cette fonction et montrer comment en déduire la correction ; si non, donner un contre-exemple.

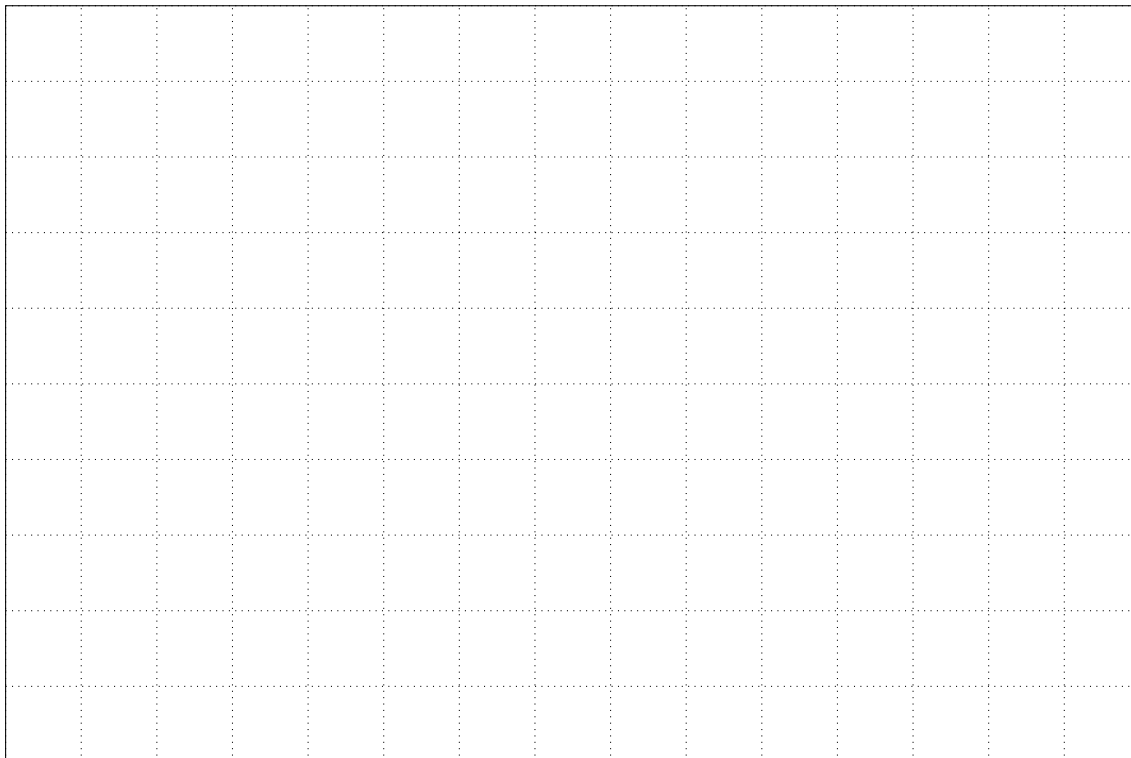


Exercice 4 : Pixels

Question 4.1 : [2/56]

Donner une définition de la fonction `moyenne` qui prend en entrée une liste d'entiers positifs `li` **non vide** et renvoie la partie entière de la moyenne des entiers de `li`.

```
>>> moyenne([0, 0, 0, 0])
0
>>> moyenne([255, 255, 0, 0])
127
>>> moyenne([100])
100
```



Une manière de coder en machine des *pixels* (composants atomiques d'une image) est de les représenter par trois valeurs entières comprises entre 0 et 255 (inclus), correspondant aux composantes rouge, verte et bleue de la couleur du pixel.

Ainsi un pixel blanc est représenté par (255, 255, 255), un pixel noir par (0, 0, 0), un pixel violet sombre par (148, 0, 211), un pixel vert foncé par (0, 128, 0), ...

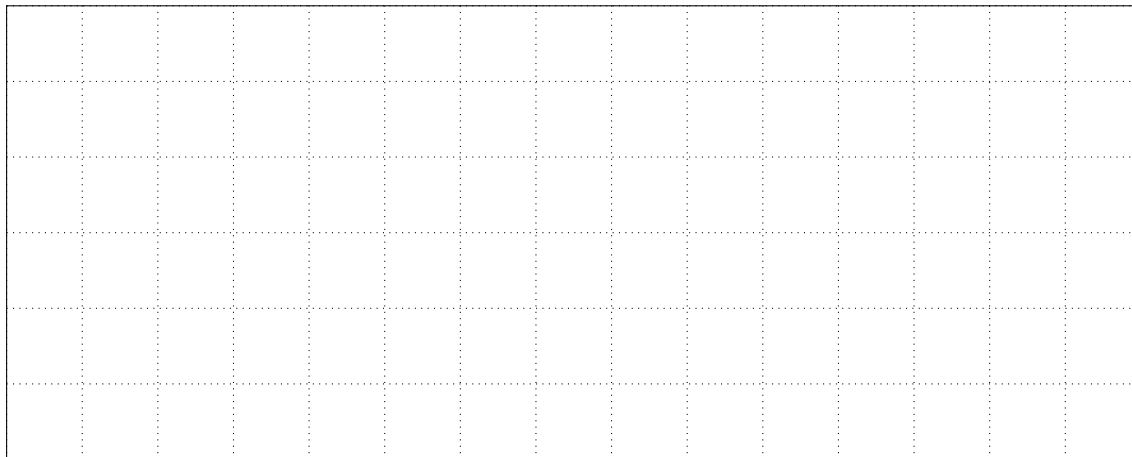
On utilisera l'alias de type suivant :

```
Pixel = Tuple[int, int, int]
# les entiers d'un Pixel sont compris entre 0 et 255
```

Question 4.2 : [1/56]

Donner une définition de la fonction `est_noir` qui prend en entrée un pixel `p` et décide si les trois composantes de `p` sont nulles (c'est-à-dire renvoie `True` si c'est le cas, et `False` sinon).

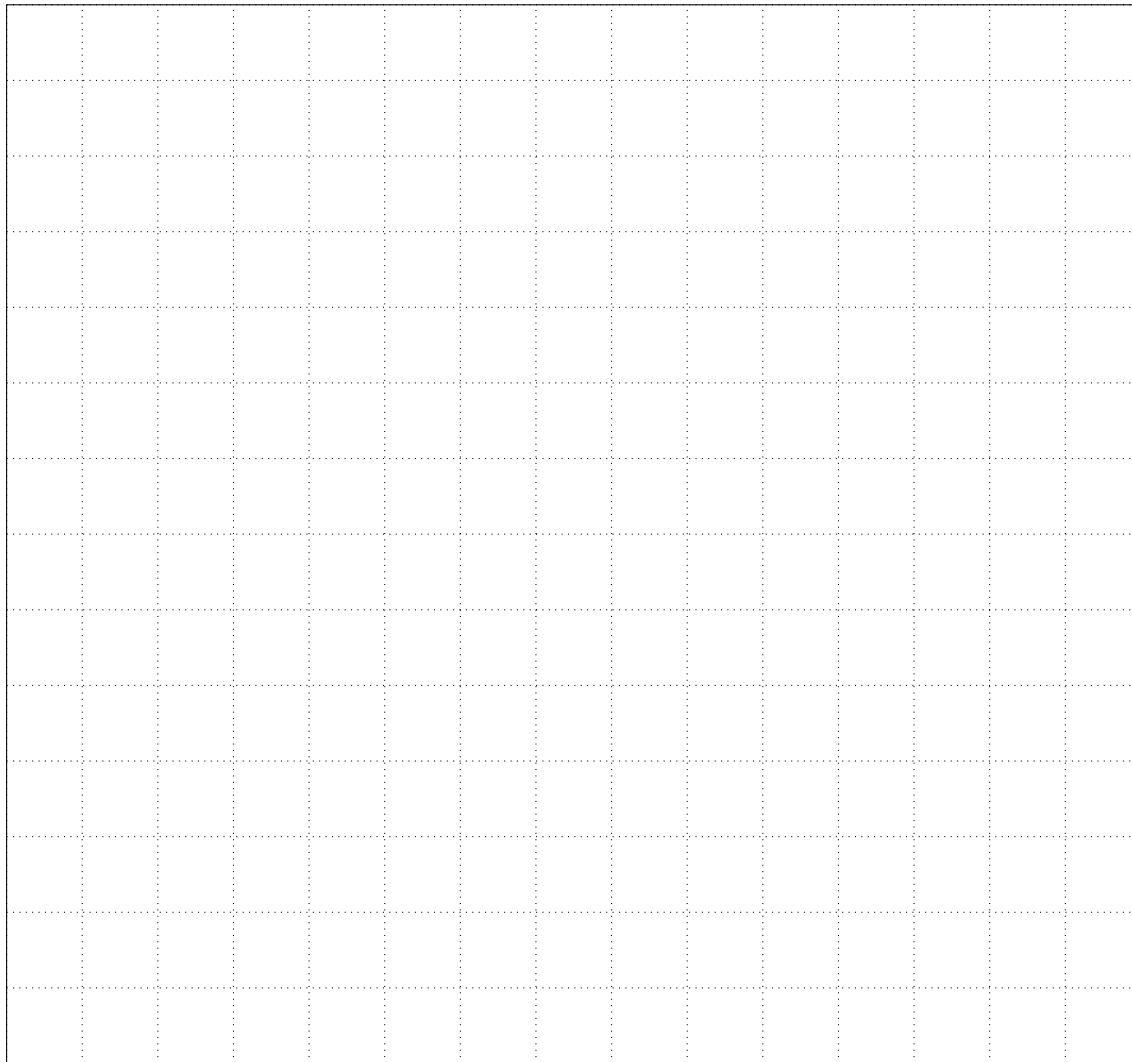
```
>>> est_noir( (0, 0, 0) )
True
>>> est_noir( (127, 127, 0) )
False
```



Question 4.3 : [3/56]

Donner une définition de la fonction `listes_par_couleur` qui prend en entrée une liste de pixels `lp` et renvoie le triplet des listes des composantes (rouge, vert, bleu) des pixels de `lp`.

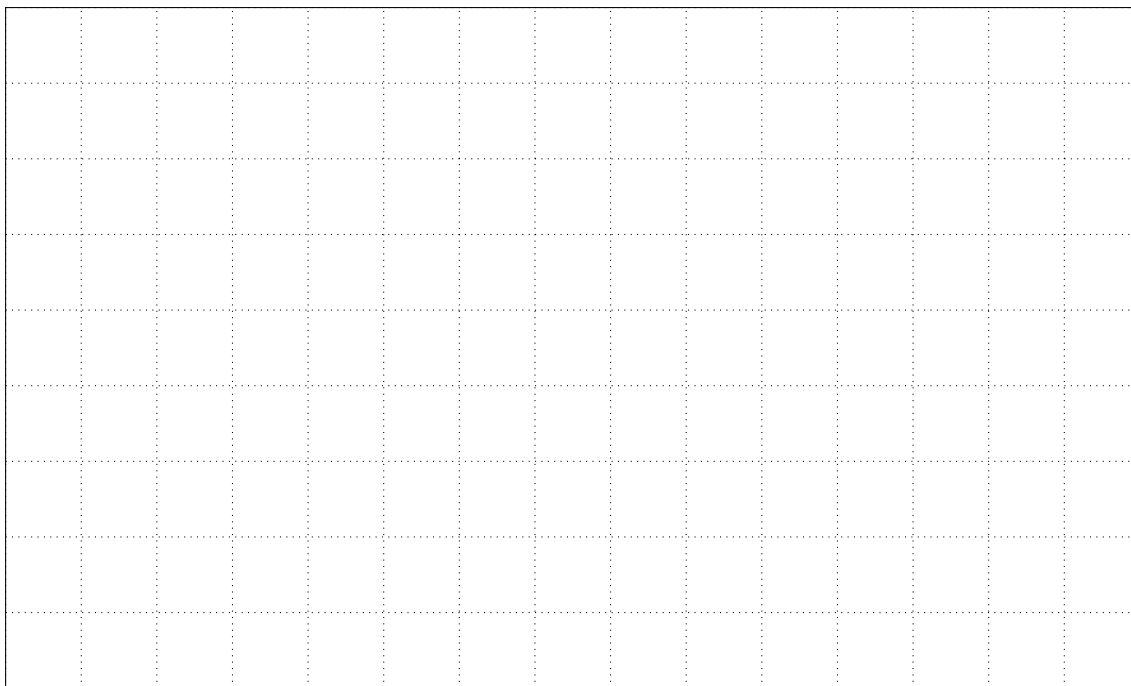
```
>>> listes_par_couleur([(0, 0, 0), (255, 255, 255), (0, 127, 255), (255, 127, 0)])  
([0, 255, 0, 255], [0, 255, 127, 127], [0, 255, 255, 0])  
>>> listes_par_couleur([(0, 127, 50)])  
([0], [127], [50])  
>>> listes_par_couleur([])  
([], [], [])
```



Question 4.4 : [2/56]

Donner une définition de la fonction `pixel_moyen` qui prend en entrée une liste **non vide** de pixels `lp` et renvoie le pixel moyen de `l` : un pixel dont chaque composante vaut la moyenne de la composante correspondante des pixels de `lp` (sa composante rouge est la moyenne des composantes rouge de `lp`, ...).

```
>>> pixel_moyen([(0, 0, 0), (255, 255, 255), (0, 127, 255), (255, 127, 0)])
(127, 127, 127)
>>> pixel_moyen([(0, 100, 100), (0, 200, 0)])
(0, 150, 50)
>>> pixel_moyen([(0, 127, 50)])
(0, 127, 50)
```



Une image peut être représentée par un rectangle de pixels, et implémentée par une liste de listes de pixels, toutes de même longueur.

Chaque élément d'une image `im` représente une ligne, et les éléments de `im[i]` sont, dans l'ordre, les pixels de la ligne i de l'image, en partant du haut, et de la gauche. La couleur du pixel de la ligne i (en partant du haut) et de la colonne j (en partant de la gauche) de l'image `im` se trouve donc en `im[i][j]`.

On utilisera l'alias de type suivant :

```
Im = List[ List[ Pixel ] ]
# tous les elements d'une Im ont la meme longueur
```

Par exemple,

```
[ [(0, 0, 0), (255, 0, 0)],
  [(148, 0, 101), (0, 128, 0)]]
```

est une image de deux lignes, de chacune deux pixels. Le pixel en haut à gauche de l'image est noir, celui en bas à droite est vert foncé.

Question 4.5 : [5/56]

Pour compresser une image, et obtenir une image qui prend 4 fois moins de place en mémoire, on remplace **chaque carré** de 4 pixels ($2 * 2$) de l'image initiale par un unique pixel correspondant à leur moyenne.

Par exemple on remplace le carré :

```
[[ (0, 0, 0), (255, 0, 0) ],
 [ (148, 0, 101), (0, 128, 0) ]]
```

par :

```
[[ (100, 32, 25) ]]
```

Donner une définition de la fonction `compression` qui prend en entrée une image `im` ayant un nombre pair de lignes, et dont les lignes sont de longueur paire, et qui renvoie une image obtenue par compression de `im`.

```
>>>compression([(0, 0, 0), (255, 0, 0)],[(148, 0, 101), (0, 128, 0)])
[[ (100, 32, 25) ]]
```

```
# moyenne([0, 255, 148, 0]) vaut 100
# moyenne([0, 0, 0, 128]) vaut 32
# moyenne([0, 0, 101, 0]) vaut 25
```

```
# im0 : Im
```

```
im0 = [[(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 0, 0)],
        [(0, 255, 0), (0, 0, 255), (255, 0, 0), (0, 255, 0)],
        [(0, 0, 255), (255, 0, 0), (0, 255, 0), (0, 0, 255)],
        [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 0, 0)]]
```

```
>>> compression(im0)
[[ (63, 127, 63), (127, 63, 63) ],
 [ (127, 63, 63), (63, 63, 127) ]]
```

