Eléments de Programmation (en Python)
Recueil d'exercices – Saison 2

Sorbonne Université – Licence 1 – 2023/2024

Table des matières

Thème 9 : Exercices simples sur les ensembles et dictionnaires

---

Exercice 9.1 : Différence symétrique (corrigé)

Dans cet exercice, nous implémentons en Python l'opération ensembliste de *différence symétrique*.

Question 1

La *différence symétrique* entre deux ensembles ens_1 et ens_2 se note $ens_1 \triangle ens_2$ en mathématiques. Un élément e appartient $ens_1 \triangle ens_2$ si et seulement si :

- e appartient à ens_1
- ou e appartient à ens_2
- mais e ne doit pas appartenir *simultanément* à ens_1 et ens_2.

Sans utiliser les opérations ensemblistes prédéfinies, proposer une définition de la fonction `diff_sym` qui construit la différence symétrique entre deux ensembles `ens1` et `ens2`.

Par exemple :

```
>>> diff_sym({2, 5, 9}, {3, 5, 8})
{2, 3, 8, 9}
```

```
>>> diff_sym({2, 5, 9}, {2, 5, 8, 9})
{8}
```

```
>>> diff_sym({'a', 'b', 'c'}, {'d', 'e', 'f'})
{'a', 'b', 'c', 'd', 'e', 'f'}
```

```
>>> diff_sym({'a', 'b', 'c'}, set())
{'a', 'b', 'c'}
```

```
>>> diff_sym(set(), {'d', 'e', 'f'})
{'d', 'e', 'f'}
```

```
>>> diff_sym({'a', 'b', 'c'}, {'a', 'b', 'c'})
set()
```

Question 2

Proposer une seconde définition de la fonction `diff_sym` en exploitant directement la *propriété de différence symétrique* :

$$ens_1 \triangle ens_2 = (ens_1 \setminus ens_2) \cup (ens_2 \setminus ens_1)$$

Remarque : on utilisera bien sûr les opérateurs ensemblistes prédéfinis par Python.

Quelle définition de la fonction `diff_sym` est-elle selon-vous la plus efficace ?

Exercice 9.2 : Répétitions dans les listes (corrigé)

L'analyse des répétitions dans des séquences comme les listes représente un cas d'utilisation typique des ensembles.

Question 1

Donner une définition de la fonction `repetes` qui, étant donné une liste `l`, retourne l'ensemble des éléments répétés au moins une fois dans cette liste.

Par exemple :

```
>>> repetes([1, 2, 23, 9, 2, 23, 6, 2, 9])
{2, 9, 23}
```

```
>>> repetes([1, 2, 3, 4])
set()
```

```
>>> repetes(['bonjour', 'ça', 'ça', 'va', '?'])
{'ça'}
```

Remarque : on supposera que le type des éléments de la liste `l` est compatible avec les ensembles, c'est-à-dire immutable ou *hashable* (mais on n'écrira pas de précondition correspondante).

Question 2

Donner une définition de la fonction `sans_repetes` qui étant donné une liste `l` retourne cette même liste `l` sans les répétitions éventuelles d'éléments. Votre fonction doit cependant conserver l'ordre dans lequel la première occurrence de chaque élément apparaît dans la liste `l`.

Par exemple :

```
>>> sans_repetes([1, 2, 23, 9, 2, 23, 6, 2, 9])
[1, 2, 23, 9, 6]
```

```
>>> sans_repetes([1, 2, 3, 4])
[1, 2, 3, 4]
```

```
>>> sans_repetes([2, 1, 2, 1, 2, 1, 2])
[2, 1]
```

```
>>> sans_repetes(['bonjour', 'ça', 'ça', 'va' , '?'])
['bonjour', 'ça', 'va', '?']
```

Question 3

Donner une définition de la fonction `uniques` qui, étant donné une liste `l`, retourne l'ensemble des éléments apparaissant exactement une fois dans cette liste.

Par exemple :

```
>>> uniques([1, 2, 23, 9, 2, 23, 6, 2, 1])
{6, 9}
```

```
>>> uniques([1, 2, 1, 1])
{2}
```

```
>>> uniques([1, 2, 1, 2, 1])
set()
```

Attention : votre fonction doit uniquement utiliser des ensembles et non des dictionnaires pour enregistrer les répétitions.

Exercice 9.3 : Recettes de cuisine (corrigé)

Dans cet exercice, on s'intéresse à la définition de fonctions permettant de manipuler un livre de recettes de cuisine. Comme tout bon livre de recettes qui se respecte, chaque recette décrit notamment l'ensemble des ingrédients qui la composent. À titre d'exemple, un livre de recettes de desserts pourrait contenir les informations suivantes :

Recette	Ingrédients
Gâteau au chocolat	chocolat, oeuf, farine, sucre, beurre
Gâteau au yaourt	yaourt, oeuf, farine, sucre
Crêpes	oeuf, farine, lait
Quatre-quarts	oeuf, farine, beurre, sucre
Kouign amann	farine, beurre, sucre

Un livre de recettes est donc représenté en Python par un dictionnaire de type :

`Dict[str,Set[str]]`

- les noms des recettes, de type `str`, comme clés
- l'ensemble des ingrédients, de type `Set[str]`, comme valeurs associées.

Pour cet exercice nous allons considérer la définition globale suivante :

```
Dessert : Recette
Dessert = {
    'gateau chocolat' : {'chocolat', 'oeuf', 'farine', 'sucre', 'beurre'}
    ,'gateau yaourt' : {'yaourt', 'oeuf', 'farine', 'sucre'}
    ,'crepes' : {'oeuf', 'farine', 'lait'}
    ,'quatre-quarts' : {'oeuf', 'farine', 'beurre', 'sucre'}
    ,'kouign amann' : {'farine', 'beurre', 'sucre'}
 }
```

Question 1

Donner une définition de la fonction `nb_ingredients` qui, étant donnés un livre de recettes `des` et le nom d'une recette `r` *contenue* dans `des`, renvoie le nombre d'ingrédients nécessaires à la recette `r`.

Par exemple:

```
>>> nb_ingredients(Dessert, 'crepes')
3
```

```
>>> nb_ingredients(Dessert, 'gateau chocolat')
5
```

Question 2

Donner une définition de la fonction `recette_avec` qui, étant donnés un livre de recettes `des` et le nom d'un ingrédient `i`, renvoie l'ensemble des recettes qui utilisent cet ingrédient.

Par exemple :

```
>>> recette_avec(Dessert, 'beurre')
{'gateau chocolat', 'kouign amann', 'quatre-quarts'}
```

```
>>> recette_avec(Dessert, 'lait')
{'crepes'}
```

```
>>> recette_avec(Dessert, 'fraise')
set()
```

Question 3

Donner une définition de la fonction `tous_ingredients` qui, étant donné un livre de recettes `des`, renvoie l'ensemble de tous les ingrédients apparaissant au moins une fois dans une recette de `des`.

Par exemple :

```
>>> tous_ingredients(Dessert)
{'beurre', 'chocolat', 'farine', 'lait', 'oeuf', 'sucre', 'yaourt'}
```

Question 4

Tout livre de recettes contient une *table des ingrédients* permettant d'associer à chaque ingrédient l'ensemble des recettes qui l'utilisent. Une telle table est représentée en Python par le type `Dict[str,Set[str]]` dans lequel une clé est un ingrédient dont la valeur associée est l'ensemble des recettes qui l'utilisent.

Donner une définition de la fonction `table_ingredients` qui, étant donné un livre de recettes `des`, renvoie la table des ingrédients associée.

Par exemple :

```
>>> table_ingredients(Dessert)
{'oeuf': {'crepes', 'gateau chocolat', 'gateau yaourt', 'quatre-quarts'},
 'yaourt': {'gateau yaourt'},
 'beurre': {'gateau chocolat', 'kouign amann', 'quatre-quarts'},
 'chocolat': {'gateau chocolat'},
 'farine': {'crepes',
  'gateau chocolat',
  'gateau yaourt',
  'kouign amann',
  'quatre-quarts'},
 'sucre': {'gateau chocolat',
  'gateau yaourt',
  'kouign amann',
```

```
  'quatre-quarts'},
 'lait': {'crepes'}}
```

Question 5

Donner une définition de la fonction `ingredient_principal` qui, étant donné un livre de recettes `des`, renvoie le nom de l'ingrédient utilisé par le plus grand nombre de recettes. On supposera ici que `des` contient au moins une recette.

Par exemple :

```
>>> ingredient_principal(Dessert)
'farine'
```

Question 6

Certaines personnes sont allergiques à certains ingrédients. On aimerait donc pouvoir ne conserver d'un livre de recettes que celles qui n'utilisent pas un ingrédient donné.

Donner une définition de la fonction `recettes_sans` qui, étant donnés un livre de recettes `des` et un ingrédient `i`, renvoie un nouveau livre de recettes ne contenant que des recettes de `des` n'utilisant pas l'ingrédient `i`.

Par exemple :

```
>>> recettes_sans(Dessert, 'farine')
{}
```

```
>>> recettes_sans(Dessert, 'oeuf')
{'kouign amann': {'beurre', 'farine', 'sucre'}}
```

```
>>> recettes_sans(Dessert, 'beurre')
{'gateau yaourt': {'farine', 'oeuf', 'sucre', 'yaourt'},
 'crepes': {'farine', 'lait', 'oeuf'}}
```

Exercice 9.4 : Statistiques sur les lettres

Dans cet exercice, on effectue quelques calculs statistiques sur les fréquences de lettres dans des textes (chaînes de caractères).

Les fréquences (ou nombre d'occurrences) des lettres sont représentées sous la forme d'un dictionnaire de type `Dict[str,int]` avec :

- des lettres (caractères) comme clés
- des entiers naturels (fréquence du caractère) pour les valeurs associées

Pour cet exercice nous utiliserons les déclarations préalables suivantes :

```
from typing import Dict, Set
```

Pour séparer les lettres de la langue française des autres caractères possibles dans les chaînes, on utilise la fonction suivante :

```python
def est_lettre(c : str) -> bool:
    """Précondition : len(c) == 1    (caractère)
    Retourne True si le caractère c est une lettre, ou False sinon.
    """
    return ((c >= 'a') and (c <= 'z')) \
        or ((c >= 'A') and (c <= 'Z')) \
        or (c in {'é', 'è', 'à', 'ù', 'œ'})

# Jeu de tests
assert est_lettre('a') == True
assert est_lettre('Z') == True
assert est_lettre(',') == False
assert est_lettre('1') == False
```

Question 1

Définir la fonction `frequences_lettres` qui étant donnée un chaîne de caractère `s` retourne les fréquences des lettres de `s` sous la forme d'un dictionnaire de type `Dict[str,int]`.

Par exemple :

```
>>> frequences_lettres('alea jacta est')
{'a': 4, 'l': 1, 'e': 2, 'j': 1, 'c': 1, 't': 2, 's': 1}
```

```
>>> frequences_lettres("l'élève")
{'l': 2, 'é': 1, 'è': 1, 'v': 1, 'e': 1}
```

Question 2

Définir une fonction `lettre_freq_max` qui retourne la lettre de fréquence maximale dans un dictionnaire `freqs` de fréquences.

Par exemple :

```
>>> lettre_freq_max(frequences_lettres('alea jacta est'))
'a'
```

```
>>> lettre_freq_max(frequences_lettres("l'élève"))
'l'
```

Remarque : s'il y a plusieurs lettres de fréquence maximale, alors on n'en retourne qu'une choisie arbitrairement.

Question 3

Dans cette question, nous aimerions effectuer notre petit test statistique sur un véritable texte.

Pour cela, nous allons tout d'abord définir une fonction `chargement_texte` permettant de lire un fichier texte et de placer le résultat dans une chaîne de caractères. **Remarque** : nous n'étudions pas en détail le chargement et la sauvegarde des fichiers dans ce livre (que nous avons déjà rencontrés à l'exercice 7.3), donc on utilisera cette fonction en suivant simplement sa spécification et on se réferrera à la documentation de Python ou un autre manuel de Python concernant cette problématique.

```python
def chargement_texte(fichier : str) -> str:
    """Précondition : le fichier est présent sur le disque
    Retourne la chaîne de caractères correspondant au contenu
    du fichier.
    """
    # Contenu du fichier
    contenu : str = ''

    with open(fichier, 'r') as f:
        contenu = f.read()

    return contenu
```

On récupérera alors un fichier texte (encodage `UTF-8`) de langue française pour en étudier le contenu.

On peut par exemple récupérer un texte intégral via le *Projet Gutemberg*, à l'adresse suivante : http://www.gutenberg.org

Pour les exemples on a choisi *Quatrevingt treize* de Victor Hugo que l'on trouvera à l'adresse suivante :

https://www.gutenberg.org/ebooks/9645.txt.utf-8

Donner deux expressions Python permettant de :

1. récupérer le dictionnaire des fréquences des lettres présentes dans votre texte d'exemple.
2. trouver la lettre dont la fréquence est la plus grande.

Question 4

On souhaite maintenant connaître les lettres qui ne dépassent pas une fréquence donnée dans un texte. Donner une définition de la fonction `lettres_freq_inf` qui étant donné un dictionnaire de fréquences `freqs` et une fréquence `fseuil` retourne l'ensemble des lettres de fréquence inférieure ou égale à `fseuil`.

Par exemple :

```
>>> lettres_freq_inf(frequences_lettres('alea jacta est'), 1)
{'c', 'j', 'l', 's'}
```

```
>>> lettres_freq_inf(frequences_lettres("l'élève"), 2)
{'e', 'l', 'v', 'è', 'é'}
```

Remarque : on fera l'hypothèse que la fréquence de seuil est strictement positive. En effet, nous n'étudions pas l'absence d'un lettre dans le texte.

Question 5

Donner une expression Python permettant d'obtenir l'ensemble des lettres utilisées moins de 100 fois dans votre fichier texte.

Exercice 9.5 : Magasin en ligne

Dans cet exercice, nous nous familiarisons avec les manipulations de dictionnaires sur une thématique de magasin en ligne.

> «*Chez **Geek and sons** tout ce qui est inutile peut s'acheter, et tout ce qui peut s'acheter est un peu trop cher.*»

La base de prix des produits de *Geek and sons* est représentée en Python par un dictionnaire de type `Dict[str, float]` avec :

— les noms de produits, de type `str`, comme clés
— les prix des produits, de type `float`, comme valeurs associées.

Question 1

Donner une expression Python pour construire la base des prix des produits correspondant à la table suivante :

Nom du produit	Prix TTC
Sabre laser	229.0
Mitendo DX	127.30
Coussin Linux	74.50
Slip Goldorak	29.90
Station Nextpresso	184.60

Question 2

Donner une définition de la fonction `prix_moyen` qui, étant donné une base de prix (contenant au moins un produit), retourne le prix moyen des produits disponibles.

Par exemple :

```
>>> prix_moyen({'Sabre Laser': 229.0,
                'Mitendo DX': 127.30,
                'Coussin Linux': 74.50,
                'Slip Goldorak': 29.90,
                'Station Nextpresso': 184.60})
129.06
```

Question 3

Donner une définition de la fonction `fourchette_prix` qui, étant donné un prix minimum `mini`, un prix maximum `maxi` et une base de `prix`, retourne l'ensemble des noms de produits disponibles dans cette fourchette de prix.

Par exemple :

```
>>> fourchette_prix(50.0, 200.0, {'Sabre Laser': 229.0,
                                  'Mitendo DX': 127.30,
                                  'Coussin Linux': 74.50,
```

```
                                    'Slip Goldorak': 29.90,
                                    'Station Nextpresso': 184.60})
{'Coussin Linux', 'Mitendo DX', 'Station Nextpresso'}
```

Question 4

Le *panier* est un concept omniprésent dans les sites marchands, *Geeks and sons* n'échappe pas à la règle. En Python, le panier du client sera représenté par un dictionnaire de type `Dict[str, int]` avec :

— les noms de produits comme clés
— une quantité d'achat comme valeurs associées.

Donner une expression Python correspondant à l'achat de 3 sabres lasers, de 2 coussins *Linux* et de 1 slip *Goldorak*.

Question 5

Donner une définition de la fonction `tous_disponibles` qui, étant donné un panier d'achat `panier` et une base de `prix`, retourne `True` si tous les produits demandés sont disponibles, ou `False` sinon.

Question 6

Donner une définition de la fonction `prix_achats` qui, étant donné un panier d'achat `Panier` et une base de `Prix`, retourne le prix total correspondant.

Par exemple :

```
>>> prix_achats({'Sabre Laser': 3, 'Coussin Linux': 2, 'Slip Goldorak': 1},
                {'Sabre Laser': 229.0,
                 'Mitendo DX': 127.30,
                 'Coussin Linux': 74.50,
                 'Slip Goldorak': 29.90,
                 'Station Nextpresso': 184.60})
865.9
```

Remarque : on supposera que tous les articles du paniers sont disponibles dans la base de produits.

Exercice 9.6 : Traduction

Comme son nom l'indique, l'une des utilités d'un dictionnaire est de s'en servir comme outil de traduction. Nous allons voir ici quelques manipulation simples d'un dictionnaire de langues. Dans la suite, on prendra en exemple les dictionnaires anglais-français et français-italien suivants :

```
Dict_Ang_Fra : Dict[str, str]
Dict_Ang_Fra = {'the': 'le', 'cat': 'chat',
                'fish': 'poisson', 'catches': 'attrape'}

Dict_Fra_Ita : Dict[str, str]
```

```
Dict_Fra_Ita = {'le': 'il', 'chat': 'gatto',
                'poisson': 'pesce', 'attrape': 'cattura'}
```

Question 1

Donner une définition de la fonction `traduction_mot_a_mot` qui, étant donné une liste `l` de mots et un dictionnaire `dico`, retourne la liste des mots de `l` traduits à partir du dictionnaire `dico`. On supposera que tous les mots apparaissant dans `l` sont une clé du dictionnaire.

Par exemple :

```
>>> traduction_mot_a_mot([],Dict_Ang_Fra)
[]
```

```
>>> traduction_mot_a_mot(['cat'],Dict_Ang_Fra)
['chat']
```

```
>>> traduction_mot_a_mot(['the', 'cat', 'catches', 'the', 'fish'],
                         Dict_Ang_Fra)
['le', 'chat', 'attrape', 'le', 'poisson']
```

```
>>> traduction_mot_a_mot(['le', 'chat', 'attrape', 'le', 'poisson'],
                         Dict_Fra_Ita)
['il', 'gatto', 'cattura', 'il', 'pesce']
```

Question 2

Donner une définition de la fonction `dictionnaire_inverse` qui étant donné un dictionnaire `dico`, renvoie le dictionnaire inverse. On supposera ici qu'une même valeur n'apparaît pas plusieurs fois dans le dictionnaire `dico`.

Par exemple :

```
>>> dictionnaire_inverse({"cat": "chat"})
{'chat': 'cat'}
```

```
>>> dictionnaire_inverse(Dict_Ang_Fra)
{'poisson': 'fish', 'le': 'the', 'chat': 'cat', 'attrape': 'catches'}
```

```
>>> dictionnaire_inverse(Dict_Fra_Ita)
{'pesce': 'poisson', 'il': 'le', 'gatto': 'chat', 'cattura': 'attrape'}
```

Question 3

Donner une définition de la fonction `composition_dictionnaires` qui étant donné deux dictionnaires `dico1` et `dico2`, renvoie le dictionnaire correspondant à la composition des traductions. On supposera que toutes les valeurs de `dico1` sont des clés de `dico2`.

Par exemple :

```
>>> composition_dictionnaires({"chat":"cat"}, {"cat":"gatto"})
{'chat': 'gatto'}
```

```
>>> composition_dictionnaires(Dict_Ang_Fra, Dict_Fra_Ita)
{'fish': 'pesce', 'catches': 'cattura', 'the': 'il', 'cat': 'gatto'}
```

Exercice 9.7 : Décomposition en facteurs premiers

Dans cet exercice, nous allons écrire une fonction qui calcule la décomposition en facteurs premiers de n'importe quel entier positif supérieur ou égal à 2.

En effet, tout entier naturel supérieur ou égal à 2 peut s'exprimer comme un produit de nombres premiers, appelé *décomposition en facteurs premiers*. Par exemple, la décomposition en facteurs premiers de 30 est $2 * 3 * 5$, tandis que celle de 56 est $2 * 2 * 2 * 7$.

Comme le montre l'exemple précédent, les facteurs premiers intervenant dans une décomposition peuvent apparaître plusieurs fois. Nous allons donc représenter une telle décomposition par un dictionnaire de type `Dict[int, int]` dans lequel les clés sont les nombres premiers et les valeurs correspondent au nombre de fois où le nombre premier intervient dans la décomposition. Ainsi, reprenant nos deux exemples, la décomposition de 30 correspond au dictionnaire

`{2:1, 3:1, 5:1}`

tandis que la décomposition de 56 est donnée par le dictionnaire :

`{2:3, 7:1}`

Question 1

Donner une définition de la fonction `valeur_decomposition` qui, étant donné un dictionnaire `decomp` correspondant à la décomposition en facteurs premiers d'un nombre, calcule la valeur de ce nombre. Par exemple :

```
>>> valeur_decomposition({2:1, 3:1, 5:1})
30
```

```
>>> valeur_decomposition({2:3, 7:1})
56
```

```
>>> valeur_decomposition({2:10})
1024
```

On peut aussi directement accéder aux valeurs :

```
def valeur_decomposition(decomp : Dict[int, int]) -> int:
    """Renvoie la valeur associée à la décomposition en facteurs premiers dico.
    """
    # valeur de la décomposition
    val : int = 1

    p : int # facteur premier
    for (p,dp) in decomp.items():
        val = val * (p ** dp)

    return val
```

Question 2

Une liste de facteurs premiers est une liste de type `List[int]` d'entiers naturels éventuellement répétés correspondant à la décomposition en facteurs premiers d'un nombre.

Par exemple :

- la liste `[2, 3, 5]` est la liste de facteurs premiers de 30
- la liste `[2, 2, 2, 7]` est la liste de facteurs premiers de 56
- la liste `[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]` est la liste de facteurs premiers de 1024 (2^{10})

Donner une définition de la fonction `decomposition` qui, étant donné une liste non vide de facteurs premiers, retourne le dictionnaire correspondant à cette décomposition.

Par exemple :

```
>>> decomposition([2, 3, 5])
{2: 1, 3: 1, 5: 1}
```

```
>>> decomposition([2, 2, 2, 7])
{2: 3, 7: 1}
```

```
>>> decomposition([2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
{2: 10}
```

Question 3 (difficile)

Soit $n \geq 2$ un entier. Soit *prems* la liste des nombres premiers inférieurs ou égaux à n. On peut déterminer la liste `l` des facteurs premiers (avec répétition) de n à l'aide de l'algorithme suivant :

- Initialement, `l = []`
- Pour chaque nombre premier `p` de *prems* pris en ordre croissant :
  - Tant que `p` divise `n` :
    - Ajouter `p` à `l`
    - Diviser `n` par `p`

Donner une définition de la fonction `liste_facteurs_premiers` qui, étant donné un entier `n` supérieur ou égal à 2, calcule la liste des facteurs premiers (avec répétition) de n implémentant l'algorithme décrit ci-dessus. Par exemple :

```
>>> liste_facteurs_premiers(30)
[2, 3, 5]
```

```
>>> liste_facteurs_premiers(56)
[2, 2, 2, 7]
```

```
>>> liste_facteurs_premiers(1024)
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

```
>>> liste_facteurs_premiers(13)
[13]
```

Remarque : on introduira pour cette question une fonction auxiliaire `liste_nombre_premiers` telle que `liste_nombres_premier(n)` renvoie la liste des nombres premiers inférieurs ou égaux à `n`. Voir par exemple l'exercice *Crible d'Eratosthène* du thème 8 pour un début de solution à ce problème.

```
# Jeu de tests
assert liste_nombres_premiers(10) == [2, 3, 5, 7]
```

```
assert liste_nombres_premiers(30) == [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
assert liste_nombres_premiers(2) == [2]
```

Question 4

À partir de la fonction précédente, donner une définition de la fonction :

`decomposition_facteurs_premiers`

qui, étant donné un entier `n` supérieur ou égal à 2, renvoie le dictionnaire correspondant à la décomposition en facteurs premiers de `n`. Par exemple :

```
>>> decomposition_facteurs_premiers(1024)
{2: 10}
```

```
>>> decomposition_facteurs_premiers(30)
{2: 1, 3: 1, 5: 1}
```

```
>>> decomposition_facteurs_premiers(56)
{2: 3, 7: 1}
```

```
>>> decomposition_facteurs_premiers(13)
{13: 1}
```

Thème 10 : Exercices avancés sur les ensembles et dictionnaires

---

Exercice 10.1 : Compréhensions en vrac (corrigé)

Cet exercice, qui doit plutôt être fait sur feuille que sur ordinateur, propose de manipuler les expressions de compréhensions de listes, d'ensembles et de dictionnaires.

Soient les variables suivantes :

```
Liste : List[int]
Liste = [ 2, 5, 12, 31, 2, 17, 31, 42, 2 ]
```

```
Dico : Dict[str,str]
Dico = {'xx':'bli', 'yzy':'blo', 'cuicui':'toutou', 'miaou':'toutou' }
```

```
Ens : Set[float]
Ens = { 2.7, 4.12, 3.31, 8.29, 1.13, 12.31 }
```

Question 1 : compréhensions de listes

Donner le résultat d'évaluation des expressions suivantes :

1) `[ (k,Dico[k]) for k in Dico ]`
2) `[ (k,v) for (k,v) in Dico.items() ]`
3) `[ Dico[k] for k in Dico ]`
4) `[ v for (k,v) in Dico.items() ]`
5) `import math`
   [math.ceil(v) for v in Ens if v $<$ 5.0]

Question 2 : compréhensions d'ensembles

Donner le résultat d'évaluation des expressions suivantes :

1) `{ c for c in 'a bac a cab' }`
2) `{ c for c in 'a bac a cab' if c != ' ' }`
3) `{ (k, Dico[k]) for k in Dico }`
4) `{ v for (k,v) in Dico.items() }`
5) `{ Dico[k] for k in Dico }`
6) `{ k + 's' for k in Dico if len(Dico[k]) > 3 }`
7) `{ n % 5 for n in Liste }`
8) `{ n % 10 for n in range(0, 20) }`

   cas 8) à comparer avec : `[ n % 10 for n in range(0, 20) ]`

Question 3 : Compréhensions de dictionnaires

Donner le résultat d'évaluation des expressions suivantes :

1) `{ k:Dico[k] for k in Dico }`
2) `{ Dico[k]:k for k in Dico }`
3) `{ (v + v):k for (k, v) in Dico.items() }`
4) `{ (Dico[k] + Dico[k]):k for k in Dico }`
5) `{ k:v for (k, v) in zip(range(1, 8), Liste)}`
6) `{ k:v for (k, v) in zip(range(1, 10), Liste) if v > 15 }`
7) `{ k:v for k in range(1, 10) for v in Liste if v > 15 }`

Exercice 10.2 : Gestion de bibliothèque (corrigé)

Dans cet exercice, nous illustrons les compréhensions sur les dictionnaires en travaillant sur une base de données de livres empruntables dans une bibliothèque.

La base de données que nous allons utiliser en exemple est la suivante :

```
LivresBD : Dict[str,Tuple[str,int]]
LivresBD = {'Les misérables':('Victor Hugo', 5),
            'Le dernier des Mohicans':('James F. Cooper', 0),
            'Un animal doué de raison': ('Robert Merle', 6),
            'Le grand Meaulnes':('Alain Fournier', 1),
            'Notre-dame de Paris':('Victor Hugo', 4),
            'Les comtemplations':('Victor Hugo', 0) }
```

La base permet de rechercher rapidement un livre à partir de son titre (clé du dictionnaire). On obtient alors l'auteur du livre ainsi que le nombre d'exemplaire(s) empruntable(s) en stock.

Par exemple :

```
>>> LivresBD['Notre-dame de Paris']
('Victor Hugo', 4)
```

Question 1

Proposer une définition de la fonction `auteurs` qui, étant donné une base de `livres`, retourne l'ensemble des noms d'auteurs de cette base.

Question 2

Donner une définition de la fonction `titres_empruntables` qui, étant donné une base de `livres`, retourne l'ensemble des livres empruntables.

Par exemple :

```
>>> titres_empruntables(LivresBD)
{'Le grand Meaulnes',
 'Les misérables',
 'Notre-dame de Paris',
 'Un animal doué de raison'}
```

Question 3

Donner une définition de la fonction `titres_auteur` qui, étant donné un nom d'`auteur` et une base de `livres`, retourne l'ensemble des titres de livres écrits par cet auteur.

Par exemple :

```
>>> titres_auteur('Victor Hugo', LivresBD)
{'Les comtemplations', 'Les misérables', 'Notre-dame de Paris'}
```

```
>>> titres_auteur('Robert Merle', LivresBD)
{'Un animal doué de raison'}
```

```
>>> titres_auteur('Gaston Leroux', LivresBD)
set()
```

Exercice 10.3 : Revisiter le thème 9

Dans cet exercice, nous revisitons certains exercices du thème 9 en proposant des solutions alternatives à base de compréhensions de listes, d'ensembles et de dictionnaires.

Remarque : seules les questions pouvant être résolues de façon concise avec des compréhensions sont considérées. Pour les autres questions, il est intéressant de comprendre pourquoi la solution ne peut être obtenue par compréhension.

Question 1

Reprendre l'énoncé de l'exercice 1 du Thème 9 (*Différence symétrique*).

Proposer une définition pour la fonction `diff_sym` en utilisant une compréhension d'ensemble.

Question 2

Reprendre l'énoncé de l'exercice 2 du Thème 9 (*Magasin en ligne*).

Répondre à la question 4 en utilisant une compréhension.

Question 3

Reprendre l'énoncé de l'exercice 3 du Thème 9 (*Recettes de cuisine*).

Répondre aux questions 2, 3, 4 et 6 en utilisant des compréhensions.

Question 4

Reprendre l'énoncé de l'exercice 4 du Thème 9 (*Statistiques sur les lettres*).

Répondre à la question 4 en utilisant une compréhension.

Question 5

Reprendre l'énoncé de l'exercice 6 du Thème 9 (*Traduction*).

Répondre aux questions 1, 2 et 3 en utilisant des compréhensions.

Exercice 10.4 : Multi-ensembles

En mathématiques, un *multi-ensemble* est un ensemble *avec* répétitions. Dans cet exercice, nous étudions deux possibilités de représentation de tels ensembles.

1. représentation par des listes

On peut représenter un multi-ensemble d'éléments d'un même type `T` quelconque par une liste de type `List[T]`

Par exemple, la liste :

```
['a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c']
```

représente le multi-ensemble dans lequel l'élément `'a'` est répété 2 fois, l'élément `'b'` est répété 4 fois et l'élément `'c'` est répété 3 fois.

Remarque : on ne suppose pas que les éléments sont ordonnés dans la liste. De même les répétitions ne sont pas forcément successives. Ainsi, le même multi-ensemble peut être représenté par la liste :

```
['a', 'b', 'c', 'c', 'a', 'b', 'c', 'b', 'b' ]
```

2. représentation par des multi-ensembles

La seconde représentation consiste simplement à représenter un multi-ensemble par un dictionnaire de type `Dict[T, int]`. Par exemple, le dictionnaire :

```
{'a':2, 'b':4, 'c':3}
```

représente le même multi-ensemble que précédemment.

Remarque : on suppose dans tout l'exercice que le type `T` des éléments du multi-ensemble est *immutable* donc compatible avec les clés de dictionnaire et les ensembles. La représentation basée sur les listes ne nécessite pas cette contrainte.

Les deux représentations ont chacune des avantages et des inconvénients, dont certains sont mis en lumière dans cet exercice.

Question 1

Donner une définition par compréhension de la fonction `melements_list` (resp. `melements_dict`) qui, étant donnée une liste `l` (resp. étant donné un dictionnaire `dico`) représentant un multi-ensemble, retourne l'ensemble des éléments apparaissant au moins une fois dans ce dernier.

Par exemple :

```
>>> melements_list(['a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c'])
{'a', 'b', 'c'}
```

```
>>> melements_list(['a', 'b', 'c', 'c', 'a', 'b', 'c', 'b', 'b' ])
{'a', 'b', 'c'}
```

```
>>> melements_dict({'a':2, 'b':4, 'c':3})
{'a', 'b', 'c'}
```

Question 2

Donner une définition de la fonction `mdict_de_mlist` permettant de convertir un multi-ensemble représenté par une liste `l` en ce même multi-ensemble représenté par un dictionnaire.

Par exemple :

```
>>> mdict_de_mlist(['a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c'])
{'b': 4, 'c': 3, 'a': 2}
```

```
>>> mdict_de_mlist(['a', 'b', 'c', 'c', 'a', 'b', 'c', 'b', 'b' ])
{'b': 4, 'c': 3, 'a': 2}
```

```
>>> mdict_de_mlist([])
{}
```

Peut-on définir cette fonction par compréhension ? Justifier votre réponse.

Question 3

Donner une définition de la fonction `mlist_de_mdict` qui effectue la conversion inverse : de la représentation en dictionnaire à la représentation en liste.

Remarque : on donnera d'abord une définition sans utiliser de compréhension, puis une définition avec compréhension (plus difficile).

Question 4

L'intersection de deux multi-ensembles m_1 et m_2 consiste à ne conserver que les éléments qui sont présents au moins une fois dans m_1 et m_2. Le nombre de répétitions de l'intersection est le minimum du nombre de répétitions dans chacun.

Le problème est non-trivial avec la représentation basée sur les listes, donc on ne considère dans cette question que la représentation basée sur les dictionnaires.

Donner une définition de la fonction `minter_dict` qui étant donné deux multi-ensembles `m1` et `m2` représentés par des dictionnaires retourne la représentation en dictionnaire de leur intersection.

Par exemple :

```
>>> minter_dict({'a':2, 'b':4, 'c':3},
                {'f':1, 'a':1, 'b':3})
{'b': 3, 'a': 1}
```

```
>>> minter_dict(dict(), {'f':1, 'a':1, 'b':3})
{}
```

```
>>> minter_dict({'a':2, 'b':4, 'c':3}, dict())
{}
```

Question 5

Déduire de la question précédente une définition simple de la fonction `minter_list` qui retourne l'intersection de deux multi-ensembles `l1` et `l2` représentés par des listes.

Par exemple :

```
>>> minter_list(['a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c'],
                ['f', 'a', 'b', 'b', 'b'])
['b', 'b', 'b', 'a']
```

Question 6

En revanche, l'union de deux multi-ensembles est plus simples si l'on effectue cette opération sur les listes. Il suffit en effet de sommer les répétitions des deux multi-ensembles.

Donner une définition de la fonction `munion_list` qui, étant donné deux multi-ensembles représentés par des listes `l1` et `l2`, retourne leur union également représentée par une liste.

Par exemple :

```
>>> munion_list(['a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c'],
                ['f', 'a', 'b', 'b', 'b'])
['a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c', 'f', 'a', 'b', 'b', 'b']
```

```
>>> munion_list([], ['a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c'])
['a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c']
```

```
>>> munion_list(['a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c'], [])
['a', 'a', 'b', 'b', 'b', 'b', 'c', 'c', 'c']
```

En déduire une définition simple de la fonction `munion_dict` qui effectue la même opération mais sur la représentation à base d'ensembles.

Peut-on définir la fonction `munion_dict` de façon simple par compréhension ? Justifier votre réponse.

Exercice 10.5 : Calcul de Trajets

Cet exercice utilise un dictionnaire pour représenter une carte de transport ferroviaire. Il s'agit d'un exercice avancé sur les dictionnaires et les ensembles.

On représente une *carte de transport* par un dictionnaire de type `Dict[str, Set[str]]`. Les clés sont des chaînes de caractères représentant des stations. A chaque clé-station est associée comme valeur une *correspondance* : l'ensemble des noms des stations accessibles **directement** – c'est-à-dire sans changement de train – depuis la clé-station. On suppose que toutes les stations apparaissant comme étant accessibles sont des clefs de la base.

Dans cet exercice, nous allons travailler sur un dictionnaire correspondant à une toute petite partie de la carte de transport du réseau ferré français :

```
Grandes_Lignes : Dict[str, Set[str]]
Grandes_Lignes = {'Paris': {'Strasbourg', 'Dijon', 'Toulouse',
                            'Lille', 'Lyon', 'Bordeaux'},
                  'Strasbourg':{'Paris', 'Dijon', 'München'},
                  'München': {'Strasbourg'},
                  'Dijon': {'Paris', 'Strasbourg', 'Lyon', 'Toulouse'},
                  'Lyon':{'Paris', 'Dijon', 'Toulouse'},
                  'Toulouse': {'Paris', 'Lyon', 'Dijon', 'Bordeaux'},
                  'Bordeaux': {'Nantes', 'Paris'},
                  'Nantes': {'Paris', 'Bordeaux','Quimper'},
                  'Quimper':{'Nantes'}, 'Ajaccio': {'Bastia'},
                  'Bastia': {'Ajaccio'}, 'Lille': {'Paris'}}
```

Question 1 - Trajet direct

Donner une définition du prédicat `trajet_direct(carte, st1, st2)` qui renvoie `True` quand il existe un trajet direct de la station `st1` à la station `st2` dans `carte` et `False` sinon.

Par exemple :

```
>>> trajet_direct(Grandes_Lignes, 'Paris', 'Bordeaux')
True
```

```
>>> trajet_direct(Grandes_Lignes, 'Paris', 'Ajaccio')
False
```

Question 2 - Ajouter une station

Donner une définition de la fonction `ajout_station` qui prend en paramètre une station `station`, un ensemble de stations `correspondances` et une carte de transport `carte` et qui renvoie une carte de transport correspondant à `carte` dans laquelle la station `station` est ajoutée, ainsi que ses connexions directes à toutes les stations de `correspondances`.

Remarques :

— On supposera que la connexion est symétrique (si `station1` est directement connectée à `station2`, alors `station2` est directement connectée à `station1`).
— On supposera également que toutes les stations de `correspondances` existent déjà dans la `carte`.

Par exemple :

```
>>> Nouvelles_Lignes : Dict[str, Set[str]]
>>> Nouvelles_Lignes = ajout_station('Limoges', {'Paris', 'Toulouse', 'Bordeaux'},
                                     Grandes_Lignes)
```

```
>>> trajet_direct(Nouvelles_Lignes, 'Limoges', 'Paris')
True
```

```
>>> trajet_direct(Nouvelles_Lignes, 'Bordeaux', 'Limoges')
True
```

```
>>> trajet_direct(Nouvelles_Lignes, 'Limoges', 'Dijon')
False
```

Question 3 - stations atteignables en k correspondances

Dans cette question un peu plus difficile, on souhaite récuperer l'ensemble des stations qui sont atteignables depuis une station de `depart` en exactement `k` correspondances.

Par exemple :

```
>>> stations_atteignables(Grandes_Lignes, 'Paris', 0)
{'Paris'}
```

```
>>> stations_atteignables(Grandes_Lignes, 'Paris', 1)
{'Bordeaux', 'Dijon', 'Lille', 'Lyon', 'Strasbourg', 'Toulouse'}
```

```
>>> stations_atteignables(Grandes_Lignes, 'Paris', 2)
{'Bordeaux',
```

```
 'Dijon',
 'Lyon',
 'München',
 'Nantes',
 'Paris',
 'Strasbourg',
 'Toulouse'}
```

Remarque: `Paris` est bien sûr atteignable depuis `Paris` en 2 étapes (trajet direct aller/retour)

Question 4 - Compteur de changements

Déduire de la question précédente une définition de la fonction `compteur_changements` qui retourne le nombre de correspondances à effectuer pour rejoindre une station `arrivee` depuis une station `depart` pour une `carte` donnée.

Remarque : on fera l'hypothèse qu'un trajet avec correspondance(s) existe dans la `carte` entre les deux stations.

Par exemple :

```
>>> compteur_changements(Grandes_Lignes, 'Paris', 'Paris')
0
```

```
>>> compteur_changements(Grandes_Lignes, 'Paris', 'Dijon')
1
```

```
>>> compteur_changements(Grandes_Lignes, 'Paris', 'Quimper')
3
```

Question 5 - Existence d'un trajet

On souhaite maintenant vérifier si dans une `carte` donnée il exsite un trajet entre une station de `depart` et une station `arrivee`.

La fonction à définir se nommme `existence_trajet`, voici quelques exemples d'utilisation :

```
>>> existence_trajet(Grandes_Lignes, 'Paris', 'München')
True
```

```
>>> existence_trajet(Grandes_Lignes, 'Ajaccio', 'Bordeaux')
False
```

Remarque: la fonction est proche de `stations_atteignables` (cf. Question 3) mais il faudra penser à *ne pas visiter deux fois la même station*. Que se passe-t-il d'après vous s'il on ne garantit pas cette propriété ?

Thème 11 : Exercices sur la récursion et les fonctionnelles

Exercice 11.1 : La suite de Fibonacci (corrigé)

La suite de Fibonacci (OEIS A000045) est définie, récursivement, de la façon suivante :

- $fibo(0) = 0$
- $fibo(1) = 1$
- $\forall n > 1, fibo(n) = fibo(n-2) + fibo(n-1)$

Question 1

Définir en Python la fonction `fibo` qui, à partir d'une entier naturel `n`, calcule le `n`-ième terme de la suite de Fibonacci.

On a, par exemple :

```
>>> fibo(3)
2
```

```
>>> fibo(5)
5
```

```
>>> fibo(8)
21
```

Remarque: on essaiera de traduire la définition récursive mathématique le plus littéralement possible.

Question 2

En utilisant le principe d'évaluation par réécriture, donner les étapes de réécriture pour l'évaluation de l'expression suivante : `fibo(6)`

- Combien d'appels récursifs à `fibo` sont nécessaires pour pouvoir effectuer le calcul ?
- D'après-vous, dans quel ordre de grandeur le nombre d'appels récursifs nécessaires croît lorsque `n` croît ? Cet ordre de grandeur est-t-il logarithmique, linéaire, polynomial ou exponentiel ?

Question 3

En utilisant le principe des *accumulateurs*, on peut obtenir une définition bien plus efficace des éléments de la suite de Fibonacci :

- $fibofast(0, a, b) = a$
- $\forall n > 0,\ fibofast(n, a, b) = fibofast(n-1, b, a+b)$

(avec a, b des entiers naturels)

vérifiant la propriété : $fibofast(n, 0, 1) = fibo(n)$

Proposer une définition de la fonction `fibofast` réalisant ce calcul.

Remarque : on exploitera la propriété énoncée dans le jeu de tests.

Question 4

Donner les étapes de réécriture pour l'évaluation de l'expression suivante : `fibofast(6, 0, 1)`

— Combien d'appels récursifs à `fibofast` sont nécessaires pour pouvoir effectuer le calcul ?
— D'après-vous, dans quel ordre de grandeur le nombre d'appels récursifs nécessaires croît lorsque `n` croît ? Cet ordre de grandeur est-t-il logarithmique, linéaire, polynomial ou exponentiel ?

Question 5

La définition proposée en Question 3 est dites *récursive terminale*, c'est-à-dire qu'aucun calcul (par exemple, une addition) ne reste à effectuer après un appel récursif. Il est possible de transformer une telle définition récursive terminale en une boucle `while`, en introduisant des variables pour les accumulateurs (donc `a` et `b` dans la définition de `fibofast`) ainsi qu'un compteur explicite (pour parcourir l'intervalle $[1; n]$).

Définir la fonction `fibit` qui, à partir d'un entier naturel `n` calcule le `n`-ième terme de la suite de Fibonacci en utilisant une boucle `while`.

Exercice 11.2 : La puissance revisitée (corrigé)

Nous avons vu en cours deux définitions de la fonction qui calcule x^n pour x un nombre et n un entier naturel, en utilisant des boucles *while*. Dans cet exercice, l'objectif est de proposer des définitions récursives, plus proches des définitions mathématiques originales.

Question 1

On rappelle la définition récursive simple du calcul de la puissance :

$$x^n = \begin{cases} 1 \text{ si } n = 0 \\ x \times x^{n-1} \text{ sinon} \end{cases}$$

Proposer une définition récursive de la fonction `puissance` réalisant ce calcul.

Question 2

Donner les étapes de réécriture de `puissance(2, 5)` Combien faut-il d'appels récursifs pour effectuer ce calcul ? Et plus généralement pour `puissance(x, n)` ?

Question 3

Montrer, par induction (récurrence), que pour tout nombre x et n entier naturel la propriété suivante :

$P(n)$: `puissance(x, n)` calcule bien x^n

Question 5

On considère maintenant la définition récursive permettant un calcul beaucoup plus rapide des puissances entières :

$$x^n = \begin{cases} 1 \text{ si } n = 0 \\ x^{\frac{n}{2}} \times x^{\frac{n}{2}} \text{ si } n \text{ est pair} \\ x^{\lfloor \frac{n}{2} \rfloor} \times x^{\lfloor \frac{n}{2} \rfloor} \times x \text{ sinon} \end{cases}$$

Définir la fonction `puissance_rapide` qui calcule x^n de façon récursive selon le schéma décrit ci-dessus.

Remarque : la notation $\lfloor \frac{a}{b} \rfloor$ correspond à la division entière de a par b.

Attention : on prendra soin de ne pas effectuer deux fois le même calcul.

Question 6

Combien faut-il d'appels récursifs pour effectuer le calcul `puissance_rapide(2,10)` ? Plus généralement, pour `puissance_rapide(x, n)` pensez vous que le nombre d'appels récursifs nécessaires soit d'un ordre logarithmique, linéaire, polynomial ou exponentiel ?

Exercice 11.3 : Propriétés récursives

Dans cet exercice, l'objectif est de définir *récursivement* des prédicats qui vérifient des propriétés sur les listes.

(cet exercice est une adaptation d'un exerice de l'épreuve finale de 2023)

Question 1

Donner une définition récursive de la fonction `egaux_depuis` qui, étant donnés une liste, un élément `elem` et un entier naturel `i`, indique si tous les éléments de la liste sont égaux à `elem` *à partir de* l'indice `i`

Par exemple :

```
>>> egaux_depuis([], 1, 0)
True
```

```
>>> egaux_depuis(['a'], 'a', 0)
True
```

```
>>> egaux_depuis(['a','a','a'], 'a', 0)
True
```

```
>>> egaux_depuis(['b', 'a', 'a', 'a'], 'a', 0)
False
```

```
>>> egaux_depuis(['b', 'a', 'a', 'a'], 'a', 1)
True
```

```
>>> egaux_depuis(['a', 'a', 'a', 'a'], 'b', 4)
True
```

Question 2

En utilisant `egaux_depuis`, donner une définition de la fonction `pareils` qui vérifie que tous les éléments d'une liste sont identiques.

Par exemple :

```
>>> pareils([])
True
```

```
>>> pareils(['a'])
True
```

```
>>> pareils(['a', 'a', 'a', 'a'])
True
```

```
>>> pareils(['a', 'a', 'b', 'a'])
False
```

Remarque : pour toutes les questions qui suivent, il s'agira de répondre de façon similaire avec : - une fonction auxiliaire *récursive* (comme `egaux_depuis`) qui propose une solution récursive au problème en se basant sur les indices des éléments de la liste à analyser - une fonction principale (comme `tous_egaux`) qui correspond au prédicat à définir.

On fera également attention aux *préconditions* éventuelles concernant les indices des éléments de listes.

Question 3

Donner une définition du prédicat `croissante` qui, étant donnée une liste d'entiers, indique si les éléments de cette liste est sont rangés en ordre croissant (au sens large).

Par exemple :

```
>>> croissante([])
True
```

```
>>> croissante([1])
True
```

```
>>> croissante([1, 2])
True
```

```
>>> croissante([1, 2, 2])
True
```

```
>>> croissante([1, 2, 2, 3])
True
```

```
>>> croissante([1, 2, 3, 2])
False
```

Remarque : on pourra nommer le prédicat récursif associé `croissante_depuis`

Question 4

Donner une définition du prédicat `palindrome` qui, étant donnée une liste, indique si ses éléments forment un palindrome.

Par exemple :

```
>>> palindrome([])
True
```

```
>>> palindrome([1])
True
```

```
>>> palindrome([1,1])
True
```

```
>>> palindrome([1,2])
False
```

```
>>> palindrome([1,2,1])
True
```

```
>>> palindrome([4,1,2,1,4])
True
```

```
>>> palindrome([4,1,2,1,5])
False
```

```
>>> palindrome([c for c in "amanaplanacanalpanama"])
True
```

Remarque : le problème devra être résolu de façon récursive, en s'inspirant des explications ci-dessous.

Pour déterminer récursivement si la liste `[1, 2, 3, 2, 1]` est un palindrome, on vérifie que :

— le premier élément (ici `1`) et le dernier élément de la liste sont égaux
— la sous-liste centrale (ici `[2, 3, 2]`) est un palindrome

On pourra délimiter la sous-liste centrale par deux indices `i` et `j`

Question 5

Donner une définition de la *fonctionnelle* `verification` qui, étant donnés une liste `ll` de listes d'éléments et un prédicat `pred` prenant en argument une liste et retournant un booléen, retourne `True` si et seulement si toutes les listes de `ll` vérifient le prédicat `pred`.

Par exemple :

```
>>> verification([[], [1], [1, 2, 3], [4, 5, 6, 6]], croissante)
True
```

```
>>> verification([[], [1], [1, 2, 3], [4, 5, 7, 6]], croissante)
False
```

```
>>> verification([[], [1], [1, 1, 1], [1, 1, 1, 1]], pareils)
True
```

```
>>> verification([[], [1], [1, 1, 1], [1, 2, 1, 1]], pareils)
False
```

```
>>> verification([['a','b','a'], ['c', 'b','a','b','c']], palindrome)
True
```

```
>>> verification([['a','b','a'], ['c', 'x','a','b','c']], palindrome)
False
```

Remarque : il n'est pas demandé de solution récursive pour cette fonction.

Exercice 11.4 : Devine un nombre

Dans cet exercice, nous cherchons à résoudre le jeu du *Devine un Nombre*.

— une personne tierce choisi un nombre n (un entier naturel) entre deux bornes a et b
— le joueur tente de deviner ce nombre, et indique le nombre d'étape nécessaires à cette divination.

Question 1

Définir la fonction `divination` dont la spécification est la suivante :

```
def divination(oracle : Callable[[int], str], a : int, b : int) -> int:
    """Précondition : l'oracle retourne la chaîne 'plus petit',
                      'plus grand' ou 'égal'
    Précondition : b > a >= 0
    Retourne le nombre d'étapes permettant de deviner le nombre caché,
    selon les réponses données par l'oracle.
    Ce nombre est compris entre les bornes a et b (incluses)
    """
```

Pour cette première version, l'algorithme choisi consiste à effectuer toutes les tentatives possibles entre `a` et `b`, par ordre croissant.

Dans les tests, on choisira `a=1`, `b=100` et pour l'oracle la fonction suivante :

```
def oracle42(k : int) -> str:
    """Indique si k est 'plus petit', 'plus grand' ou 'égal' au nombre secret.
    """
    if k == 42:
        return 'égal'
    elif k < 42:
        return 'plus petit'
    else:
        return 'plus grand'
```

Question 2

On cherche une méthode plus rapide pour trouver les nombres secrets. L'idée, est d'utiliser le principe récursif suivant :

— pour les bornes `a` et `b` on prend comme candidat le *milieu* `m` entre ces bornes.
  — si l'oracle répond `'plus petit'` on recommence la divination entre `a` et `m - 1`
  — si l'oracle répond `'plus grand'` on recommence la divination entre `m + 1` et `b`
  — sinon on a trouvé et on retombe le nombre d'étapes effectuées.

On appelle cette méthode une recherche *par dichotomie*.

Définir la fonction `divination_rec` qui implémente ce principe récursif.

Remarque : on pensera à calculer le nombre d'étape explicitement lors des appels récursifs.

On a ainsi :

```
>>> divination_rec(oracle42, 1, 100, 0) # 0 = nombre d'étapes (au début)
7
```

```
>>> divination(oracle42, -1000, 1000)
1043
```

```
>>> divination_rec(oracle42, -1000, 1000,0)
9
```

Donner les étapes de réécriture de l'exemple ci-dessus.

Exercice 11.5 : Méthode de Newton (en TME)

Dans cet exercice, nous implémentons en Python la méthode de *Newton* (aussi appelée méthodes des tangentes) permettant d'approcher numériquement le 0 d'une fonction réelle continue et dérivable.

Le principe de la méthode est le suivant :

- soit f une fonction de $\mathbb{R}$ (ou $\mathbb{R}^+$) dans $\mathbb{R}$ continue et dérivable, et soit f' sa dérivée
- et soit un point x_0 dans le domain de f, alors la suite $x_0, x_1, x_2, \ldots$ avec $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ converge (conditionnellement) vers une solution de l'équation $f(x) = 0$.

Remarque : la méthode de Newton est incomplète et peut, dans certains cas, ne pas converger. C'est le cas, par exemple, si l'on choisit un point initial x_0 trop distant du «vrai» zéro cherché. Cependant, si l'on se contente de calculer le `n`-ième terme de la suite alors on garantit la terminaison de l'algorithme.

Question 1

Pour commencer, nous allons effectuer un calcul approché de la racine carrée $\sqrt{a}$ d'un nombre $a \in \mathbb{R}^+$. Pour cela, on considère la fonction $f(x) = x^2 - a$. On a bien $x = \sqrt{a}$ quand $f(x) = 0$. On souhaite approcher la valeur de $\sqrt{2}$ donc on prendra $a = 2$ pour la suite.

La dérivée de f est $f'(x) = 2x$.

Donner les définitions de f et de f' en Python (sous le nom `f_fun` et `f_deriv`)

Question 2

On souhaite définir *récursivement* la fonction `approx_newton` qui, à partir d'une fonction `f`, sa dérivée `df`, un point `x0` de `f` et d'un rang `n` (entier naturel), calcule le `n`-ième terme de la suite $x_0, x_1, x_2, \ldots$ telle que définie dans l'énoncé.

La signature de cette fonction est la suivante :

```
def approx_newton(f : Callable[[float], float], df : Callable[[float], float]
                  , x0 : float, n : int) -> float:
    """..."""
```

Par exemple :

```
>>> approx_newton(f_fun, f_deriv, 1.0, 2)
1.4166666666666667
```

```
>>> approx_newton(f_fun, f_deriv, 1.0, 5)
1.4142135623730951
```

On compare cette valeur approchée avec celle obtenue directement en Python :

```
import math
>>> math.sqrt(2)
1.4142135623730951
```

Il s'agit donc exactement de la même approximation (en seulement 5 itérations). Mais notre algorithme peut faire «encore mieux».

```
>>> approx_newton(f_fun, f_deriv, 1.0, 6)
1.414213562373095
```

```
>>> approx_newton(f_fun, f_deriv, 1.0, 10)
1.414213562373095
```

Cette approximation est la «meilleure» possible de $\sqrt{2}$ avec les flottants, et elle est obtenue en 6 itérations seulement, ce qui montre l'efficacité importante de la méthode de *Newton* (lorsqu'elle s'applique).

Question 3

Proposer une définition de la fonction `approx_newton_it` qui implémente la méthode de *Newton* avec une boucle et donc sans récursion.

```
def approx_newton_it(f : Callable[[float], float], df : Callable[[float], float]
                     , x0 : float, n : int) -> float:
    """ cf. approx_newton """
    y : float = x0
    i : int
    for i in range(n):
        y = y - f(y)/df(y)
    return y
```

Question 4

En utilisant la méthode de *Newton* et `approx_newton`, définir la fonction `resolution` qui donne le `n`-ième terme de l'approximation de *Newton* pour l'équation $\cos(x) = x^3$, en considérant la fonction $g(x) = \cos(x) - x^3$.

Remarque : on prendra le point de départ $x0 = 0,5$ et on se rappelle que $g'(x) = -\sin(x) - 3x^2$.

Remarque : en raison des approxmations effectuées par les calculs flottants, la méthode de newton ne converge pas ici vers la valeur théorique correcte. On assiste plutôt à une oscillation autour du zéro théorique. On voit donc ici la relative fragilité de la méthode.

Exercice 11.6 : Réductions (corrigé)

Dans cet exercice, nous étudions la fonction `reduce` vue en cours, et dont nous répétons la définition ci-dessous :

```
def reduce(op : Callable[[T, U], T], init : T, lst : List[U]) -> T:
    """Retourne la réduction de la liste lst par l'opérateur op
    , initialisée par init.
```

```
    """
    res : T = init
    for x in lst:
        res = op(res, x)
    return res
```

Question 1

En utilisant `reduce`, proposer une définition de la fonction `min_liste` qui retourne l'élément minimal d'une liste `lst` *non-vide* d'entiers.

Par exemple :

```
>>> min_liste([3, 9, 11, 5])
3
```

```
>>> min_liste([11, 9, 3, 5])
3
```

```
>>> min_liste([42])
42
```

Question 2

On souhaite définir une variante de `reduce`, appelée `reductions`, qui au lieu de retourner la valeur de la réduction retourne toutes les étapes intermédiaires.

Ainsi, pour `reduction(op, init, [e1, e2, ..., eN]`

on retournera la liste :

`[init, op(init,e1), op(op(init, e1), e2), ..., op(op(...op(init,e1), e2), ..., eN)]`

Par exemple :

```
>>> reductions(monmin, 20, [5, 9, 3, 11])
[20, 5, 5, 3, 3]
```

```
def plus(a : int, b : int) -> int:
    """Retourne a+b"""
    return a + b
```

```
>>> reduce(plus, 0, [5, 9, 3, 11])
28
```

```
>>> reductions(plus, 0, [5, 9, 3, 11])
[0, 5, 14, 17, 28]
```

Exercice 11.7 : Map et Filter

Dans cet exercice, nous définissons deux fonctionnelles opérant sur les listes :

— la fonctionnelle `fmap` effectuant une *transformation de liste*
— la fonctionnelle `filter` effectuant un *filtrage de liste*

Question 1

On rappelle que la *transformation* d'une liste `[e1, e2, ..., eN]` par une fonction unaire `f` consiste à construire la liste `[f(e1), f(e2), ..., f(eN)]`.

La spécification de la fonctionnelle `fmap` réalisant ce schéma est la suivante :

```
def fmap(f : Callable[[T], U], lst : List[T]) -> List[U]:
    """Retourne la liste des applications de f aux éléments de lst
    (schéma de transformation)
    """
```

Par exemple :

```
def oppose(n : int) -> int:
    """Retourne l'opposé de n"""
    return -n
```

```
>>> fmap(oppose, [1, 2, 3, 4, 5])
[-1, -2, -3, -4, -5]
```

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

```
>>> fmap(double, [1, 2, 3, 4, 5])
[2, 4, 6, 8, 10]
```

```
>>> fmap(len, ["un", "deux", "trois", "quatre", "cinq"])
[2, 4, 5, 6, 4]
```

Remarque : pour cette question, on proposera une définition de la fonction *sans* utiliser de compréhension.

Question 2

Proposer une définition alternative du schéma de transformation, nommée `fmap2`, et utilisant cette fois ci une compréhension.

Question 3

Le schéma de filtrage, que nous appellerons `ffilter`, prend en paramètre un prédicat `pred` et une liste `lst`, et construit la liste des seuls éléments de `lst` qui vérifient `pred`.

Par exemple :

```
def even(n : int) -> bool:
    """Retourne True ssi n est pair"""
    return n % 2 == 0

def odd(n : int) -> bool:
    """Retourne True ssi n est impair"""
    return not(even(n))
```

```
>>> ffilter(even, [1, 2, 3, 4, 5, 6, 7])
[2, 4, 6]
```

```
>>> ffilter(odd, [1, 2, 3, 4, 5, 6, 7])
[1, 3, 5, 7]
```

Remarque : pour cette question, on proposera une définition sans utiliser de compréhension.

Question 4

Proposer une définition alternative du schéma de filtrage, nommée `ffilter2`, et utilisant cette fois ci une compréhension.

Exercice 11.8 : Zip, unzip, etc.

Dans cet exercice, nous étudions des *fonctionnelles* permettant de gérer des listes de couples.

Question 1

La *fonctionnelle* `fzip` permet de créer une liste de couples à partir de deux listes distinctes. Sa spécification est la suivante :

```
def fzip(lst1 : List[T], lst2 : List[U]) -> List[Tuple[T, U]]:
    """Retourne la liste des couples formés par les éléments de lst1
    (premier élément du couple) et les éléments de lst2 (second élément).
    """
```

On a par exemple :

```
>>> fzip([1, 3, 5], [2, 4, 6])
[(1, 2), (3, 4), (5, 6)]
```

```
>>> fzip(["un", "deux", "trois"], [1, 2, 3, 4])
[('un', 1), ('deux', 2), ('trois', 3)]
```

Remarque : on voit sur cet exemple que la construction s'arrête à la plus courte des deux listes

Question 2

On souhaite maintenant programmer la fonction `funzip` qui est, en quelque sorte, le complémentaire de `fzip`.

La spécification de la fonction est la suivante :

```
def funzip(lst : List[Tuple[T, U]]) -> Tuple[List[T], List[U]]:
    """Retourne un couple formé de la
    - la liste des premiers éléments de la liste lst,
    - la liste des seconds éléments de cette même liste
    """
```

Par exemple :

```
>>> funzip([(1, 2), (3, 4), (5, 6)])
([1, 3, 5], [2, 4, 6])
```

```
>>> funzip([("un", 1), ("deux", 2), ("trois", 3)])
(['un', 'deux', 'trois'], [1, 2, 3])
```

Question 3

On souhaite maintenant implémenter une variante de `fzip`, appelée `fzip_with` qui prend, en plus des deux listes `lst1` et `lst2`, une fonction `f` de deux arguments. La fonction combine les éléments des listes de départ deux-à-deux pour produire une liste de valeurs résultats.

Ainsi :

```
fzip_with(f, [a1, a2, ..., aN], [b1, b2, ..., bN])
--> [f(a1,b1), f(a2, b2), ... f(aN, bN)]
```

Remarque : on ne considérera comme pour `fzip` que les éléments correspondants à la plus courte des deux listes.

On aura par exemple :

```
def plus(a : int, b : int) -> int:
    """Retourne a + b"""
    return a + b

def inf(a : int, b : int) -> bool:
    """Indique si a < b"""
    return a < b
```

```
>>> fzip_with(plus, [1, 2, 3], [9, 8, 7])
[10, 10, 10]
```

```
>>> fzip_with(inf, [1, 2, 3], [3, 2, 1, 4])
[True, False, False]
```

Thème 12 : Exercices sur les procédures et tableaux

Exercice 12.1 : Procédures modifiant des listes en place (corrigé)

Dans cet exercice, l'objectif est de définir des procédures diverses qui modifient des listes *en place*. Nous commençons par les déclarations suivantes :

Question 1

Définir une procédure `censurer` qui prend en entrée une liste de mots `l` (un texte à censurer) et un ensemble de `mots` interdits, et qui remplace dans la liste `l` tous les mots interdits par la chaîne `***CENSURE***`.

Par exemple :

```
>>> t0 : List[str] = ["le", "loup", "est", "un",
                      "loup", "pour", "l'", "homme"]
>>> censurer(t0, {"loup"})
```

```
>>> t0
['le', '***CENSURE***', 'est', 'un',
 '***CENSURE***', 'pour', "l'", 'homme']
```

```
>>> t1 : List[str] = ["le", "loup", "est", "un",
                      "loup", "pour", "l'", "homme"]
>>> censurer(t1, {"loup", "homme"})
```

```
>>> t1
['le',
 '***CENSURE***',
 'est',
 'un',
 '***CENSURE***',
 'pour',
 "l'",
 '***CENSURE***']
```

Question 2

Définir une procédure `decaler` qui opère le décalage d'une liste *en place*.

Par exemple :

```
>>> l1 : List[int] = [1, 2, 3, 4, 5, 6]
>>> decaler(l1, 2)
>>> l1
[5, 6, 1, 2, 3, 4]
```

```
>>> l2 : List[int] = [1, 2, 3, 4, 5, 6]
>>> decaler(l2, -2)
>>> l2
[3, 4, 5, 6, 1, 2]
```

```
>>> l3 : List[int] = [1, 2, 3, 4, 5, 6]
>>> decaler(l3, 50)
>>> l3
[5, 6, 1, 2, 3, 4]
```

```
>>> l4 : List[int] = [1, 2, 3, 4, 5, 6]
>>> decaler(l4, 0)
>>> l4
[1, 2, 3, 4, 5, 6]
```

```
>>> l5 : List[int] = [1, 2, 3, 4, 5, 6]
>>> decaler(l5, 5)
>>> l5
[2, 3, 4, 5, 6, 1]
```

Exercice 12.2 : Rotation de 180° d'une matrice

On a vu, dans le corps du chapitre, la procédure `renverser` qui inverse les éléments d'une liste, par modification *en place*. On donne ci-dessous une définition avec la procédure interne d'échange intégrée, et que l'on renomme pour l'occasion `inverser`.

```
def inverser(l : List[T]) -> None:
    """***Procédure***
    Renverse la liste l en place.
    """
    for k in range(0, len(l) // 2):
        temp : T = l[k]
        l[k] =  l[len(l) - k - 1]
        l[len(l) - k - 1] = temp
```

```
# Jeux de tests
l0 : List[int] = [1, 2, 3]
assert inverser(l0) == None
assert l0 == [3, 2, 1]
```

Question 1

Définir une procédure `inverser_lignes` qui inverse l'ordre des lignes d'une matrice d'entiers. On rappelle l'alias de type pour les matrices entières :

```
Matrice = List[List[int]]
```

Par exemple :

```
>>> m1 : Matrice = [[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 9]]
>>> inverser_lignes(m1)
```

```
>>> m1
[[7, 8, 9],
```

```
 [4, 5, 6],
 [1, 2, 3]]
```

Question 2

Définir une procédure `inverser_colonnes`qui inverse l'ordre des colonnes d'une matrice.

Par exemple :

```
>>> m2 : Matrice = [[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 9]]
>>> inverser_colonnes(m2)
```

```
>>> m2
[[3, 2, 1],
 [6, 5, 4],
 [9, 8, 7]]
```

Question 3

En déduire une procédure `demitour` qui fait tourner une matrice de 180° (le coefficient en haut à gauche devient le coefficient en bas à droite, ...).

Par exemple :

```
>>> m3 : Matrice = [[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 9]]
>>> demitour(m3)
```

```
>>> m3
[[9, 8, 7],
 [6, 5, 4],
 [3, 2, 1]]
```

Exercice 12.3 : Rotation de 90° en place

Pour effectuer la rotation à 90° (un quart de tour) d'une matrice carrée entière de côté une puissance de 2, on propose l'algorithme (récursif) suivant:

- diviser la matrice en 4 sous matrices égales,
- faire tourner de 90°, récursivement, chaque sous-matrice,
- opérer **une rotation par quadrant** de la matrice: (la sous-matrice en haut à gauche devient celle en haut à droite, celle en bas à gauche devient celle en haut à gauche, ...)

Dans cet exercice, nous allons traduire cet algorithme en style procédural, avec des modifications *en place*. Nous commençons par la définition de l'alias de type suivant :

```
Matrice = List[List[int]]
```

Nous ne considérons que des matrices carrées et de dimension au moins 2 dans la suite.

Question 1

Donner une procédure `tourner_sous_matrice` qui prend en entrée un matrice `m`, des coordonnées `x`, `y` de la matrice et un entier `k` pair et qui modifie `m` en faisant échangeant circulairement les quatres quadrants de la sous-matrice dont le coin haut-gauche est en (x, y) et le côté est k.

Par exemple si (x, y) est le coin haut gauche de la sous-matrice $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$ et `k` est son côté des matrices, la modification opérée est donnée par:

$$\begin{pmatrix} & & \\ & A & B \\ & C & D \\ & & \end{pmatrix} \rightarrow \begin{pmatrix} & & \\ & C & A \\ & D & B \\ & & \end{pmatrix}$$

Par exemple :

```
>>> m0 : Matrice = [[0,0,0,0],
                    [0,1,2,0],
                    [0,3,4,0],
                    [0,0,0,0]]
>>> tourner_sous_matrice(m0, 1, 1, 2)
```

```
>>> m0
[[0, 0, 0, 0],
 [0, 3, 1, 0],
 [0, 4, 2, 0],
 [0, 0, 0, 0]]
```

```
>>> m1 : Matrice = [[ 1,  2,  3,  4, 0, 0, 0, 0],
                    [ 5,  6,  7,  8, 0, 0, 0, 0],
                    [ 9, 10, 11, 12, 0, 0, 0, 0],
                    [13, 14, 15, 16, 0, 0, 0, 0],
                    [ 0,  0,  0,  0, 0, 0, 0, 0],
                    [ 0,  0,  0,  0, 0, 0, 0, 0],
                    [ 0,  0,  0,  0, 0, 0, 0, 0],
                    [ 0,  0,  0,  0, 0, 0, 0, 0]]
>>> tourner_sous_matrice(m1, 0, 0, 4)
```

```
>>> m1
[[ 9, 10, 1, 2, 0, 0, 0, 0],
 [13, 14, 5, 6, 0, 0, 0, 0],
 [11, 12, 3, 4, 0, 0, 0, 0],
 [15, 16, 7, 8, 0, 0, 0, 0],
 [ 0,  0, 0, 0, 0, 0, 0, 0],
 [ 0,  0, 0, 0, 0, 0, 0, 0],
 [ 0,  0, 0, 0, 0, 0, 0, 0],
 [ 0,  0, 0, 0, 0, 0, 0, 0]]
```

Question 2

Donner une procédure `tourner_matrice` qui prend en entrée une matrice carrée `m`, un entier `k` qui divise le côté de `m` (donc sa dimension), et qui opère une rotation des quadrants pour **toutes**

les sous-matrices de côté k formant une paritition de `m`:

Par exemple si `k` est la moité du côté de la matrice, la modification opérée est donnée par:

$$\begin{pmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{pmatrix} \rightarrow \begin{pmatrix} E & A & G & C \\ F & B & H & D \\ M & I & O & K \\ N & J & P & L \end{pmatrix}$$

Les sous-matrices de côté `k` $\begin{pmatrix} A & B \\ E & F \end{pmatrix}$, $\begin{pmatrix} C & D \\ G & H \end{pmatrix}$, $\begin{pmatrix} I & J \\ M & N \end{pmatrix}$ et $\begin{pmatrix} K & L \\ O & P \end{pmatrix}$ ont chacune subi une rotation par quadrant.

Par exemple :

```
>>> m0 : Matrice = [[0,0,0,0],
                    [0,1,2,0],
                    [0,3,4,0],
                    [0,0,0,0]]
>>> tourner_matrice(m0, 2)
```

```
>>> m0
[[0, 0, 2, 0],
 [1, 0, 0, 0],
 [0, 0, 0, 4],
 [0, 3, 0, 0]]
```

```
>>> m1 : Matrice = [[0,0,0,0],
                    [0,1,2,0],
                    [0,3,4,0],
                    [0,0,0,0]]
>>> tourner_matrice(m1, 1)
```

```
>>> m1
[[0, 0, 0, 0],
 [0, 1, 2, 0],
 [0, 3, 4, 0],
 [0, 0, 0, 0]]
```

```
>>> m2 : Matrice = [[0,0,0,0],
                    [0,1,2,0],
                    [0,3,4,0],
                    [0,0,0,0]]
>>> tourner_matrice(m2, 4)
```

```
>>> m2
[[0, 3, 0, 0],
 [0, 0, 0, 1],
 [4, 0, 0, 0],
 [0, 0, 2, 0]]
```

Question 3

Donner une procédure `tourner` qui réalise, de manière impérative et **en place** l'algorithme de rotation à 90° pour une matrice carrée de côté 2^n.

```
>>> m1 : Matrice = [[1, 2],
                    [3, 4]]
>>> tourner(m1)
```

```
>>> m1
[[3, 1],
 [4, 2]]
```

```
>>> m2 : Matrice = [[l for l in range(2 ** 3 * k, 2 ** 3 * (k + 1))]
                    for k in range(2 ** 3)]
>>> m2
[[0, 1, 2, 3, 4, 5, 6, 7],
 [8, 9, 10, 11, 12, 13, 14, 15],
 [16, 17, 18, 19, 20, 21, 22, 23],
 [24, 25, 26, 27, 28, 29, 30, 31],
 [32, 33, 34, 35, 36, 37, 38, 39],
 [40, 41, 42, 43, 44, 45, 46, 47],
 [48, 49, 50, 51, 52, 53, 54, 55],
 [56, 57, 58, 59, 60, 61, 62, 63]]
```

```
>>> tourner(m2)
>>> m2
[[56, 48, 40, 32, 24, 16, 8, 0],
 [57, 49, 41, 33, 25, 17, 9, 1],
 [58, 50, 42, 34, 26, 18, 10, 2],
 [59, 51, 43, 35, 27, 19, 11, 3],
 [60, 52, 44, 36, 28, 20, 12, 4],
 [61, 53, 45, 37, 29, 21, 13, 5],
 [62, 54, 46, 38, 30, 22, 14, 6],
 [63, 55, 47, 39, 31, 23, 15, 7]]
```

Exercice 12.4 : Tri à bulles

Le *tri à bulles*, ou tri *par propagation*, est un des plus simples algorithmes de tri, son principe est le suivant.

Pour chaque indice `i` allant de 0 (inclus) à la longueur de la liste triée (exclue) et pour chaque indice `j` allant de 0 (inclus) à la longueur de la liste *moins* `i + 1` (exclu), comparer les éléments aux indices `j` et `j+1` et les placer "dans l'ordre" (c'est-à-dire place le plus petit des deux à l'indice `j` et le plus grand à l'indice `j+1`).

Lorsque `i` vaut 0 on compare toutes les paires d'éléments successifs dans l'ordre, ce qui garantit que le maximum de la liste va «remonter» vers la fin de la liste (comme une bulle de champagne dans une coupe, ce qui donne son nom à l'algorithme).

Lorsque `i` vaut 1 on recommence, mais en s'arrêtant un indice plus tôt, ce qui garantit que le deuxième plus grand élément se trouve à l'avant dernier indice.

Remarque : dans cet exercice, on considérera le tri de listes dont les éléments sont d'un type `T` arbitraire. Les opérateurs de comparaison du langage Python (`<`, `<=`, `>`, `>=` et `==`) sont *génériques*.

Question 1.

Ecrire une procédure qui trie une liste d'entier selon la méthode du tri à bulles. Voici quelques exemples :

```
>>> l0 : List[int] = [6, 1, 3, 2, 4, 5]
>>> trier_bulles(l0)
>>> l0
[1, 2, 3, 4, 5, 6]
```

```
>>> l1 : List[int] = [1, 0, 0, 1, 0, 1, 1]
>>> trier_bulles(l1)
>>> l1
[0, 0, 0, 1, 1, 1, 1]
```

Question 2.

On donne ci-dessous la simulation de `trie_bulles(l)` avec `l` valant `[6, 1, 3, 2, 4, 5]`.

tour externe	tour interne	i	j	l
entrée	-	-	-	[6, 1, 3, 2, 4, 5]
1	entrée	0	-	[6, 1, 3, 2, 4, 5]
1	1	0	0	[1, 6, 3, 2, 4, 5]
1	2	0	1	[1, 3, 6, 2, 4, 5]
1	3	0	2	[1, 3, 2, 6, 4, 5]
1	4	0	3	[1, 3, 2, 4, 6, 5]
1	5	0	4	[1, 3, 2, 4, 5, 6]
2	entrée	0	-	[1, 3, 2, 4, 5, 6]
2	1	0	0	[1, 3, 2, 4, 5, 6]
2	2	0	1	[1, 2, 3, 4, 5, 6]
2	2	0	2	[1, 2, 3, 4, 5, 6]
2	2	0	3	[1, 2, 3, 4, 5, 6]
3	entrée	0	-	[1, 2, 3, 4, 5, 6]
3	1	0	0	[1, 2, 3, 4, 5, 6]
3	1	0	1	[1, 2, 3, 4, 5, 6]
3	1	0	2	[1, 2, 3, 4, 5, 6]
4	entrée	0	-	[1, 2, 3, 4, 5, 6]
4	1	0	0	[1, 2, 3, 4, 5, 6]
4	1	0	1	[1, 2, 3, 4, 5, 6]

Dans cette simulation, on se rend compte que si la liste est triée assez tôt dans l'algorithme (par exemple quand `i` vaut `2`), les comparaisons qui suivent ne sont plus nécessaires.

Une manière peut coûteuse de vérifier si la liste est déjà triée est de vérifier si, pour une valeur de `i` donnée, aucun échange n'a été fait. Utiliser cette condition pour définir une procédure optimisée `trier_bulles_optim` qui exploite cette propriété.

Exercice 12.5 : Tri rapide

Etudions un autre algorithme de tri pour illustrer la notion d'opération en place.

Le *tri rapide* (*quicksort*), imaginé par *Sir Tony Hoare* (célèbre informaticien), est un algorithme de tri très courant, grâce notamment à son implémentation *en place* et ses très bonnes performances en pratique.

Son principe est le suivant : on choisit un élément de la liste à trier (de façon arbitraire, par exemple, le premier élément) qu'on appelle *le pivot*. On *partitionne* ensuite la liste à trier en comparant ses éléments avec le pivot. Les éléments qui sont plus petits se retrouvent «à gauche» et les éléments plus grands «à droite» Les deux sous-partiens sont ensuite triées récursivement.

Question 1

Ecrire une procédure `partitionner` qui prend une liste `l`, choisit son premier élément comme pivot, déplace les éléments de `l` de telle sorte qu'à la fin: - tous les éléments à gauche du (d'indice inférieur à celui du) pivot soient strictement plus petits, - tous les éléments à droite du (d'indice supérieur à celui du) pivot soient plus grands, au sens large.

Après cette opération, nous aurons besoin de l'indice du pivot une fois le partitionnement effectué, il sera donc retourné par la procédure.

Par exemple :

```
>>> l1 : List[float] = [2, 1, 4, 0, 3]
>>> partitionner(l1)
2
```

```
>>> l1
[2, 1, 4, 0, 3]
```

La partie gauche est donc `[1, 0` et la partie droite `2, 4, 3]`.

Question 2

Définir une procédure `partitionner_sl` qui prend en entrée une liste `l`, deux indices `i` et `j` et effectue une la même opération que `partitionner` sur la sous-liste de `l` allant des indices `i` (inclus) à `j` (exclu).

Par exemple :

```
>>> l1 : List[int] = [8, 5, 2, 1, 3, 0, 4, 6, 7]
>>> partitionner_sl(l1, 2, 7)
4
```

```
>>> l1
[8, 5, 1, 0, 2, 3, 4, 6, 7]
```

```
>>> l2 : List[int] = [2, 1, 3, 0, 4]
>>> partitionner_sl(l2, 0, 5)
2
```

```
>>> l2
[1, 0, 2, 3, 4]
```

Question 3

On donne ci-dessous une définition de la procédure `trier_rapide_sl` qui réalise l'algorithme de tri rapide sur une sous-liste à partir du partitionnement obtenu à la question précédente.

```
def trier_rapide_sl(l: List[T], i: int, j: int) -> None:
    """***Procédure***
    Précondition : 0 <= i <= j < len(l)
    Trie la sous-liste de l allant de i  (inclus) à j (exclu)
    """
    ip : int # indice du pivot
    if i != j:
        ip = partitionner_sl(l, i, j)
        trier_rapide_sl(l, i, ip)
        trier_rapide_sl(l, ip + 1, j)
```

Remarque : Le tri rapide est réalisé indépendamment sur les deux sous-listes résultant du partitionnement, ce qui illustre l'utilisation de la récursion (cf. chapitre 4). D'un point de vue algorithmique, il s'agit de mettre en œuvre un principe algorithmique dit de «diviser pour régner» [1].

```
# Jeu de tests
l1 : List[int] = [8, 5, 2, 1, 3, 0, 4, 6, 7]
assert trier_rapide_sl(l1, 2, 7) == None
assert l1 == [8, 5, 0, 1, 2, 3, 4, 6, 7]

l2 : List[int] = [8, 5, 2, 1, 3, 0, 4, 6, 7]
assert trier_rapide_sl(l2, 0, 3) == None
assert l2 == [2, 5, 8, 1, 3, 0, 4, 6, 7]

l3 : List[int] = [8, 5, 2, 1, 3, 0, 4, 6, 7]
assert trier_rapide_sl(l3, 0, 9) == None
assert l3 == [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

En déduire une procédure `trier_rapide` qui trie la liste passée en entrée selon le principe du tri rapide.

Par exemple :

```
>>> l1 : List[int]= [4, 2, 1, 5, 3, 6]
>>> trier_rapide(l1)
>>> l1
[1, 2, 3, 4, 5, 6]
```

```
>>> l2 : List[int] = [0, 1, 1, 1, 0, 0, 1, 0]
>>> trier_rapide(l2)
>>> l2
[0, 0, 0, 0, 1, 1, 1, 1]
```

1. cf. https://fr.wikipedia.org/wiki/Diviser_pour_r%C3%A9gner_(informatique)

Solutions des exercices corrigés

Solution de l'exercice 9.1

Solution de la question 1

```python
def diff_sym(ens1 : Set[T], ens2 : Set[T]) -> Set[T]:
    """Retourne la différence symétrique entre ens1 et ens2.
    """
    # Ensemble résultat
    ens : Set[T] = set()

    e1 : T
    for e1 in ens1:
        if e1 not in ens2:
            ens.add(e1)

    e2 : T
    for e2 in ens2:
        if e2 not in ens1:
            ens.add(e2)

    return ens
```

```python
# Jeu de tests
assert diff_sym({2, 5, 9}, {3, 5, 8}) == {2, 3, 8, 9}
assert diff_sym({2, 5, 9}, {2, 5, 8, 9}) == {8}
assert diff_sym({'a', 'b', 'c'}, {'d', 'e', 'f'}) \
       == {'a', 'b', 'c', 'd', 'e', 'f'}
assert diff_sym({'a', 'b', 'c'}, set()) == {'a', 'b', 'c'}
assert diff_sym(set(), {'d', 'e', 'f'}) == {'d', 'e', 'f'}
assert diff_sym({'a', 'b', 'c'}, {'a', 'b', 'c'}) == set()
```

Solution de la question 2

```python
def diff_sym(ens1 : Set[T], ens2 : Set[T]) -> Set[T]:
    """Retourne la différence symétrique entre ens1 et ens2.
    """
    return (ens1 - ens2) | (ens2 - ens1)
```

Remarque : concernant l'efficacité, ici on a besoin de parcourir plusieurs fois chaque ensemble, alors qu'un seul parcours suffit dans la première définition, qui est donc plus efficace «théoriquement». Cependant, les opérateurs ensemblistes de Python sont implémentés en langage C et compilés en code optimisé. Il est donc possible que la seconde version soit donc plus performante en pratique. Dans tous les cas, à moins de considérer de très gros ensembles (plusieurs centaines de milliers d'éléments au minimum), on n'observera pas de différence importante.

Solution de l'exercice 9.2

Solution de la question 1

```
def repetes(l : List[T]) -> Set[T]:
    """Retourne l'ensemble des éléments répétés dans L.
    """
    # Eléments déjà vus
    vus : Set[T] = set()  # éléments déjà vus

    # Ensemble résultat
    ens : Set[T] = set()

    e : T
    for e in l:
        if e in vus:  # déjà vu ?
            ens.add(e) # répétition
        else:
            vus.add(e) # c'est tout vu !

    return ens
```

```
# Jeu de tests
assert repetes([1, 2, 23, 9, 2, 23, 6, 2, 9]) == {2, 9, 23}
assert repetes([1, 2, 3, 4]) == set()
assert repetes(['bonjour', 'ça', 'ça', 'va', '?']) == {'ça'}
```

Solution de la question 2

```
def sans_repetes(l : List[T]) -> List[T]:
    """Retourne la liste des éléments de l sans
    leur(s) répétition(s)."""
    # Eléments déjà vus
    vus : Set[T] = set()

    # Liste résultat
    lr : List[T] = []

    e : T
    for e in l:
        if e not in vus:
            lr.append(e)
            vus.add(e)

    return lr
```

```
# Jeu de tests
assert sans_repetes([1, 2, 23, 9, 2, 23, 6, 2, 9]) == [1, 2, 23, 9, 6]
assert sans_repetes([1, 2, 3, 4]) == [1, 2, 3, 4]
assert sans_repetes([2, 1, 2, 1, 2, 1, 2]) == [2, 1]
assert sans_repetes(['bonjour', 'ça', 'ça', 'va' , '?']) \
       == ['bonjour', 'ça', 'va', '?']
```

Solution de la question 3

```python
def uniques(l : List[T]) -> Set[T]:
    """Retourne l'ensemble des éléments apparaissant
    une seule fois dans l."""

    # Eléments vus au moins une fois
    unefois : Set[T] = set()

    # Eléments vus plus d'une fois
    trop : Set[T] = set()

    e : T
    for e in l:
        if e in unefois:
            # vu au moins 2 fois
            trop.add(e)
        else:
            # vu pour la première fois
            unefois.add(e)

    return unefois - trop
```

```python
# Jeu de tests
assert uniques([1, 2, 23, 9, 2, 23, 6, 2, 1]) == {6, 9}
assert uniques([1, 2, 1, 1]) == {2}
assert uniques([1, 2, 1, 2, 1]) == set()
```

Solution de l'exercice 9.3

Solution de la question 1

```python
def nb_ingredients(des : Recette, r : str) -> int:
    """Précondition : r in des
    Renvoie le nombre d'ingredients necessaires à la recette r
    dans le dictionnaire des recettes des
    """
    return len(des[r])
```

```python
# Jeu de tests
assert nb_ingredients(Dessert, 'crepes') == 3
assert nb_ingredients(Dessert, 'gateau chocolat') == 5
```

Solution de la question 2

```python
def recette_avec(des : Recette, i : str) -> Set[str]:
    """Renvoie les recettes de des qui utilisent l'ingrédient i.
    """
    # Ensemble résultat
    ens : Set[str] = set()
```

```
    r : str
    for r in des:
        if i in des[r]:
            ens.add(r)

    return ens
```

```
# Jeu de tests
assert recette_avec(Dessert, 'beurre') \
  == {'kouign amann', 'quatre-quarts', 'gateau chocolat'}
assert recette_avec(Dessert, 'lait') == {'crepes'}
assert recette_avec(Dessert, 'fraise') == set()
```

Solution de la question 3

```
def tous_ingredients(des : Recette) -> Set[str]:
    """Renvoie les ingrédients apparaissant dans une recette de D.
    """
    # Ensemble résultat
    ens : Set[str] = set()

    r : str
    for r in des:
        i : str
        for i in des[r]:
            ens.add(i)

    return ens
```

```
# Jeu de tests
assert tous_ingredients(Dessert) \
    == {'farine', 'lait', 'yaourt', 'beurre'
        ,'chocolat', 'sucre', 'oeuf'}
assert tous_ingredients(dict()) == set()
```

Variante utilisant l'union :

```
def tous_ingredients(des : Recette) -> Set[str]:
    """Renvoie les ingrédients apparaissant dans une recette de des.
    """
    ens : Set[str]
    ens = set()

    r : str
    for r in des:
        ens = ens | des[r]

    return ens
```

Solution de la question 4

```python
def table_ingredients(des : Recette) -> Dict[str,Set[str]]:
    """Renvoie les recettes associées à chaque ingrédient de des.
    """
    # Dictionnaire résultat
    table : Dict[str, Set[str]] = dict()

    ingredients : Set[str] = tous_ingredients(des)

    i : str
    for i in ingredients:
        table[i] = recette_avec(des, i)

    return table
```

```python
# Jeu de tests
assert table_ingredients(Dessert) \
    == {'lait': {'crepes'}
        ,'beurre': {'gateau chocolat', 'quatre-quarts', 'kouign amann'}
        ,'oeuf': {'gateau chocolat', 'quatre-quarts',
                  'crepes', 'gateau yaourt'}
        ,'yaourt': {'gateau yaourt'}
        ,'sucre': {'kouign amann', 'gateau chocolat'
                   ,'quatre-quarts', 'gateau yaourt'}
        ,'farine': {'kouign amann', 'gateau chocolat'
                    ,'quatre-quarts', 'crepes', 'gateau yaourt'}
        ,'chocolat': {'gateau chocolat'}}

assert table_ingredients(dict()) == dict()
```

Solution de la question 5

```python
def ingredient_principal(des : Recette) -> str:
    """Précondition : len(des) > 0
    Renvoie le nom de l'ingredient le plus utilisé.
    """
    table : Dict[str,Set[str]] = table_ingredients(des)

    # Ingredient le plus utilisé
    ires : str = ''

    # Nombre de recettes avec l'ingrédient le plus utilisé
    n : int = 0

    i : str
    for i in table:
        if len(table[i]) > n:
            ires = i
            n = len(table[i])

    return ires
```

```python
# Jeu de tests
assert ingredient_principal(Dessert) == 'farine'
assert ingredient_principal({'A' : {'a', 'b'},
                             'B' : {'a', 'c'}}) == 'a'
assert ingredient_principal({'A' : {'a', 'b'},
                             'B' : {'a', 'b'}}) in {'a', 'b'}
```

Solution de la question 6

```python
def recettes_sans(des : Recette, i : str) -> Recette:
    """Renvoie un livre de recettes n'utilisant pas l'ingrédent i.
    """
    # Dictionnaire résultat
    dres : Recette = dict()

    r : str
    for r in des:
        if i not in des[r]:
            dres[r] = des[r]

    return dres
```

```python
# Jeu de tests
assert recettes_sans(Dessert, 'beurre') \
    == {'gateau yaourt' : {'yaourt', 'oeuf','farine', 'sucre'}
        ,'crepes' : {'oeuf', 'farine', 'lait'}}
assert recettes_sans(Dessert, 'oeuf') \
    == {'kouign amann': {'farine', 'beurre', 'sucre'}}
assert recettes_sans(Dessert, 'farine') == {}
```

Solution de l'exercice 10.1

Solution de la question 1 : compréhensions de listes

```
>>> [ (k,Dico[k]) for k in Dico ]
[('xx', 'bli'), ('yzy', 'blo'), ('cuicui', 'toutou'), ('miaou', 'toutou')]
```

```
>>> [ (k,v) for (k,v) in Dico.items() ]
[('xx', 'bli'), ('yzy', 'blo'), ('cuicui', 'toutou'), ('miaou', 'toutou')]
```

```
>>> [ Dico[k] for k in Dico ]
['bli', 'blo', 'toutou', 'toutou']
```

```
>>> [ v for (k,v) in Dico.items() ]
['bli', 'blo', 'toutou', 'toutou']
```

```
>>> import math
>>> [ math.ceil(v) for v in Ens if v < 5.0 ]
[2, 3, 4, 5]
```

Solution de la question 2 : compréhensions d'ensembles

```
>>> { c for c in 'a bac a cab' }
{' ', 'a', 'b', 'c'}
```

```
>>> { c for c in 'a bac a cab' if c != ' ' }
{'a', 'b', 'c'}
```

```
>>> { (k, Dico[k]) for k in Dico }
{('cuicui', 'toutou'), ('miaou', 'toutou'), ('xx', 'bli'), ('yzy', 'blo')}
```

```
>>> { v for (k,v) in Dico.items() }
{'bli', 'blo', 'toutou'}
```

```
>>> { Dico[k] for k in Dico }
{'bli', 'blo', 'toutou'}
```

```
>>> { k + 's' for k in Dico if len(Dico[k]) > 3 }
{'cuicuis', 'miaous'}
```

```
>>> { n % 5 for n in Liste }
{0, 1, 2}
```

```
>>> { n % 10 for n in range(0, 20) }
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
>>> [ n % 10 for n in range(0, 20) ]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Solution de la question 3 : Compréhensions de dictionnaires

```
{ k:Dico[k] for k in Dico }
{'cuicui': 'toutou', 'miaou': 'toutou', 'xx': 'bli', 'yzy': 'blo'}
```

```
{ Dico[k]:k for k in Dico }
{'bli': 'xx', 'blo': 'yzy', 'toutou': 'miaou'}
```

```
{ (v + v):k for (k, v) in Dico.items() }
{'blibli': 'xx', 'bloblo': 'yzy', 'toutoutoutou': 'miaou'}
```

```
{ (Dico[k] + Dico[k]):k for k in Dico }
{'blibli': 'xx', 'bloblo': 'yzy', 'toutoutoutou': 'miaou'}
```

```
{ k:v for (k, v) in zip(range(1, 8), Liste)}
{1: 2, 2: 5, 3: 12, 4: 31, 5: 2, 6: 17, 7: 31}
```

```
{ k:v for (k, v) in zip(range(1, 10), Liste) if v > 15 }
{4: 31, 6: 17, 7: 31, 8: 42}
```

```
{ k:v for k in range(1, 10) for v in Liste if v > 15 }
{1: 42, 2: 42, 3: 42, 4: 42, 5: 42, 6: 42, 7: 42, 8: 42, 9: 42}
```

Solution de l'exercice 10.2

Solution de la question 1

```
def auteurs(livres : Dict[str,Tuple[str,int]]) -> Set[str]:
    """Retourne l'ensemble des noms d'auteurs de la base de Livres.
    """
    return { auteur for (_, (auteur, _)) in livres.items() }
```

```
# Jeu de tests
assert auteurs(LivresBD) == { 'Victor Hugo', 'James F. Cooper',
                              'Robert Merle', 'Alain Fournier'}
```

Solution de la question 2

```
def titres_empruntables(livres : Dict[str,Tuple[str,int]]) -> Set[str]:
    """Retourne l'ensemble des titres empruntables dans la base de Livres.
    """
    return { titre for (titre, (_, stock)) in livres.items()
                       if stock > 0 }
```

```
# Jeu de tests
assert titres_empruntables(LivresBD) == {'Le grand Meaulnes',
                                         'Les misérables',
                                         'Notre-dame de Paris',
                                         'Un animal doué de raison'}
```

Solution de la question 3

```
def titres_auteur(auteur : str, livres : Dict[str,Tuple[str,int]]) \
        -> Set[str]:
    """Retourne l'ensemble des titres de l'auteur spécifié
    dans la base des livres.
    """
    return { titre for (titre,(auteur_livre,_)) in livres.items()
             if auteur == auteur_livre }
```

```
# Jeu de tests
assert titres_auteur('Victor Hugo', LivresBD) \
    == {'Les comtemplations','Les misérables','Notre-dame de Paris'}

assert titres_auteur('Robert Merle', LivresBD) \
    == {'Un animal doué de raison'}

assert titres_auteur('Gaston Leroux', LivresBD) == set()
```

Solution de l'exercice 11.1

Solution de la question 1

```python
def fibo(n : int) -> int:
    """Précondition : n > 0
    Calcule le n-ième terme de la suite de Fibonacci.
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibo(n-2) + fibo(n-1)
```

```python
# Jeu de tests
assert fibo(0) == 0
assert fibo(1) == 1
assert fibo(3) == 2
assert fibo(5) == 5
assert fibo(8) == 21
```

Solution de la question 2

```
fibo(6) --> fibo(4) + fibo(5)
--> fibo(2) + fibo(3) + fibo(3) + fibo(4)
--> fibo(0) + fibo(1) + fibo(1) + fibo(2) + fibo(1) + fibo(2)
    + fibo(2) + fibo(3)
--> 0 + 1 + 1 + fibo(0) + fibo(1) + 1 + fibo(0) + fibo(1)
    + fibo(0) + fibo(1) + fibo(1) + fibo(2)
--> 1 + 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 + 1 + fibo(0) + fibo(1)
--> 1 + 1 + 1 + 1 + 1 + 1 + 1 + 0 + 1
--> 8
```

Il faut 24 appels récursifs

En expérimentant un peu, on trouve :

- pour `n=0`, `nb_calls=0`
- pour `n=1`, `nb_calls=0`
- pour `n=2`, `nb_calls=2`
- pour `n=3`, `nb_calls=4`
- pour `n=4`, `nb_calls=8`
- pour `n=5`, `nb_calls=14`
- pour `n=6`, `nb_calls=24`
- pour `n=7`, `nb_calls=40`
- pour `n=8`, `nb_calls=66`
- pour `n=9`, `nb_calls=108`

Cette croissance du nombre d'appels récursifs est clairement de nature *exponentielle*. Cette suite du nombre d'appels récursifs est référencée OEIS A019274.

Solution de la question 3

```python
def fibofast(n : int, a : int, b : int) -> int:
    """Précondition : n >= 0 and a >= 0 and b >= 0.
    Calcule le n-ième terme de la suite de Fibonacci pour a=0 et b=1
```

```
    """
    if n == 0:
        return a
    else:
        return fibofast(n-1, b, a + b)
```

```
# Jeu de tests
assert fibofast(0, 0, 1) == fibo(0)
assert fibofast(1, 0, 1) ==fibo(1)
assert fibofast(3, 0, 1) ==fibo(3)
assert fibofast(5, 0, 1) ==fibo(5)
assert fibofast(8, 0, 1) ==fibo(8)
```

Solution de la question 4

```
fibofast(6, 0, 1) --> fibofast(5, 1, 1)
--> fibofast(4, 1, 2)
--> fibofast(3, 2, 3)
--> fibofast(2, 3, 5)
--> fibofast(1, 5, 8)
--> fibofast(0, 8, 13)
--> 8
```

Il faut `6` appels récursifs

et, plus généralement, pour calculer `fibofast(n, a, b)` il faut `n` appels récursifs, donc la croissance du nombre d'appels récursifs est *linéaire*.

Solution de la question 5

```
def fibit(n : int) -> int:
    """Précondition : n >= 0
    Calcule le n-ième terme de la suite de Fibonacci.
    """
    a : int = 0
    b : int = 1
    i : int = 1

    while i <= n:
        temp : int = b
        b = a + b
        a = temp
        i = i + 1
    return a
```

```
# Jeu de tests
assert fibit(0) == fibo(0)
assert fibit(1) ==fibo(1)
assert fibit(3) ==fibo(3)
assert fibit(5) ==fibo(5)
assert fibit(8) ==fibo(8)
```

Solution de l'exercice 11.2

Solution de la question 1

```python
def puissance(x : float, n : int) -> float:
    """Précondition : n >= 0
    Retourne x élevé à la puissance n.
    """
    if n == 0:
        return 1
    else:
        return x * puissance(x, n - 1)
```

```python
# Jeu de tests
assert puissance(4.2, 0) == 1
assert puissance(4.2, 1) == 4.2
assert puissance(2, 5) == 32
assert puissance(2, 6) == 64
assert puissance(2, 10) == 1024
```

Solution de la question 2

```
puissance(2, 5) --> 2 * puissance(2, 4)
--> 2 * 2 * puissance(2, 3)
--> 2 * 2 * 2 * puissance(2, 2)
--> 2 * 2 * 2 * 2 * puissance(2, 1)
--> 2 * 2 * 2 * 2 * 2 * puissance(2, 0)
--> 2 * 2 * 2 * 2 * 2 * 1
--> 32
```

Il faut `5` appels récursifs, plus généralement pour calculer `puissance(x, n)` il faut effectuer `n` appels récursifs.

Solution de la question 3

Remarque : on ne posera pas ce type de question en épreuve écrite (interro, devoir sur table, contrôle final)

— **cas de base** : $n = 0$

On a :

`puissance(x, 0) --> 1`

(par réécriture)

et $x^0 = 1$ par définition donc $P(0)$ est vraie

— **cas récursif** :

On suppose $P(k)$ vraie pour un entier $k \geq 0$. L'hypothèse d'induction est donc : `puissance(x, k)` calcule bien x^k. On doit montrer que $P(k+1)$ est vraie.

Puisque $k + 1 > 0$, on a :

`puissance(x, k+1) --> x * puissance(x, k)`

(par réécriture)

Et par hypothèse d'induction on a égalément `puissance(x, k) -->` x^k

Donc `puissance(x, k+1) --> x *` $x^k = x \times x^k = x^{k+1}$

Donc $P(k+1)$ est vraie

— **Conclusion** : pour tout $n \geq 0$, $P(n)$ est vraie

(CQFD)

Solution de la question 5

```python
def puissance_rapide(x : float, n : int) -> float:
    """Précondition : n >= 0
    Retourne x élevé à la puissance n.
    """
    if n == 0:
        return 1
    else:
        y : float = puissance_rapide(x, n // 2)
        if n % 2 == 0:
            return y * y
        else:
            return y * y * x
```

```python
# Jeu de tests
assert puissance_rapide(4.2, 0) == puissance(4.2, 0)
assert puissance_rapide(4.2, 1) == puissance(4.2, 1)
assert puissance_rapide(2, 5) == puissance(2, 5)
assert puissance_rapide(2, 6) == puissance(2, 6)
assert puissance_rapide(2, 10) == puissance(2, 10)
```

Solution de la question 6

On ne peut pas directement appliquer le principe d'évaluation par réécriture vu en cours en raison du stockage des calculs intermédiaires dans la variable `y`, mais voici une variation qui permet de compter précisémment les appels récursifs :

```
puissance_rapide(2, 10) --> <y1>puissance_rapide(2, 5) * y1
--> <y1>[y2<puissance_rapide(2, 2)> * y2 * 2] * y1
--> <y1>[<y2>[<y3>puissance_rapide(2, 1) * y3] * y2 * 2] * y1
--> <y1>[<y2>[<y3>[<y4>puissance_rapide(2, 0) * y4 * 2] * y3] * y2 * 2] * y1
--> <y1>[<y2>[<y3>[<y4>1 * y4 * 2] * y3] * y2 * 2] * y1
--> <y1>[<y2>[<y3>[1 * 1 * 2] * y3] * y2 * 2] * y1
--> <y1>[<y2>[2 * 2] * y2 * 2] * y1
--> <y1>[4 * 4 * 2] * y1
--> 32 * 32
--> 1024
```

Au delà des détails de calcul, on a besoin de 4 appels récursifs. Et plus généralement, l'ordre de grandeur est de $\log_2(n)$ appels récursifs pour calculer x^n.

Solution de l'exercice 11.6

Solution de la question 1

Pour se rapprocher du cours, on peut programmer une fonction spécifique pour le calcul du minimum :

```
def monmin(a : int, b : int) -> int:
    """Retourne le minimum de a et b.
    """
    if a <= b:
        return a
    else:
        return b
```

Mais on peut aussi bien sûr utilier la fonction prédéfinie `min`.

```
def min_liste(lst : List[int]) -> int:
    """Précondition : len(lst) > 0
    Retourne l'élément minimal de la liste lst
    """
    return reduce(monmin, lst[0], lst)   # ou min
```

```
# Jeu de test
assert min_liste([3, 9, 11, 5]) == 3
assert min_liste([11, 9, 3, 5]) == 3
assert min_liste([42]) == 42
```

Solution de la question 2

```
def reductions(op : Callable[[T, U], T], init : T, lst : List[U]) -> List[T]:
    """Retourne la liste des réductions de lst par l'opérateur op
    , initialisée par init.
    """
    # accumulateur
    acc : T = init
    # liste résultat
    lres : List[T] = [acc]

    for x in lst:
        acc = op(acc, x)
        lres.append(acc)
    return lres
```

```
# Jeu de tests
assert reductions(monmin, 20, [5, 9, 3, 11]) == [20, 5, 5, 3, 3]
assert reductions(plus, 0, [5, 9, 3, 11]) == [0, 5, 14, 17, 28]
```

Solution de l'exercice 12.1

Solution de la question 1

```
def censurer(l : List[str], mots : Set[str]) -> None:
    """***Procédure***
    Censure la liste à partir des mots interdits.
    """
    j : int
    for j in range(len(l)):
        if l[j] in mots:
            l[j] = "***CENSURE***"
```

```
# Jeu de Test
t0 : List[str] = ["le", "loup", "est", "un",
                  "loup", "pour", "l'", "homme"]
assert censurer(t0, {"loup"}) == None
assert t0 == ['le', '***CENSURE***', 'est', 'un',
              '***CENSURE***', 'pour', "l'", 'homme']

t1 : List[str] = ["le", "loup", "est",
                  "un", "loup", "pour", "l'", "homme"]
assert censurer(t1, {"loup", "homme"}) == None
assert t1 == ['le', '***CENSURE***', 'est', 'un',
              '***CENSURE***', 'pour', "l'", '***CENSURE***']
```

Solution de la question 2

Une méthode simple consiste à faire `k % mod(len(l))` décalages successifs de 1 indice. Cette méthode en place effectue un nombre de remplacements égal à `k *len(l)`.

```
def decaler(l : List[T], k : int) -> None:
    """***Procédure***
    Decale l de k indices vers la droite.
    """
    p : int
    for p in range(k % len(l)):
        temp : T = l[-1]
        i : int
        for i in range(len(l)-1):
            l[-i-1] = l[-i-2]
        l[0] = temp
```

```
# Jeu de Test
l1 : List[int] = [1, 2, 3, 4, 5, 6]
assert decaler(l1, 2) == None
assert l1 == [5, 6, 1, 2, 3, 4]

l2 : List[int] = [1, 2, 3, 4, 5, 6]
assert decaler(l2, -2) == None
assert l2 == [3, 4, 5, 6, 1, 2]

l3 : List[int] = [1, 2, 3, 4, 5, 6]
```

```
assert decaler(l3, 50) == None
assert l3 == [5, 6, 1, 2, 3, 4]

l4 : List[int] = [1, 2, 3, 4, 5, 6]
assert decaler(l4, 0) == None
assert l4 == [1, 2, 3, 4, 5, 6]
```

Remarque : il existe une méthode un peu plus subtile mais beaucoup plus efficace. L'idée est calulerd'abord le PGCD de `k` et `len(l)`. Si c'est `1` ils sont premiers entre eux et l'ensemble des $\{n * k \mid n \in [0; \texttt{len(l)} - 1]\}$ parcourt bien l'ensemble des indices de `l`, c'est-à-dire qu'en commençant en 0 et en décalant l'indice de k vers à la droite à chaque étape, on parcourt bien toute la liste.

Si le pgcd vaut $p > 1$, quand on démarre de l'indice 0 et qu'on décale de k à chaque étape, on revient en 0 au bout de `len(l)//k` étapes, il faut ensuite recommencer en démarrant de l'indice 1, puis 2, ... jusque $p - 1$.

Dans les deux cas, on fait uniquement `len(l)` affectations. Cette solution plus complexe ne sera pas donnée en détail, mais il s'agit d'un bon complément à cet exercice.