```
#-*- coding: utf8  -*-

# python3.7 nous oblige à utiliser le module typing
# pour annoter les types des paramètres de fonction
from typing import List, Tuple, Union, TypeVar

# une variable de type IntOrStr est soit un int, soit une str
IntOrStr = TypeVar('IntOrStr', int, str)
```

Deux remarques prélables.

1. Dans l'archive zip, ce notebook `sujet-knn.ipynb` est accompagné du fichier `poudlard2022.csv` utilisé à l'étape 5.
2. Les fonctions demandées sont toujours décrites partiellement : signature, doc-string et pré-conditions. Ne pas modifier ces éléments qui sont donnés pour vous aider. En revanche, vos réponse doivent remplacer les 2 dernières lignes de ces cellules :

```
# ENTRER VOTRE CODE A LA PLACE DE CES 2 LIGNES
raise NotImplementedError()
```

Bon travail.

Ce sujet propose de découvrir un algorithme d'apprentissage automatique supervisé (l'algorithme des k plus proches voisins) et de l'utiliser pour classifier de nouvelles données selon une caractéristique discrète.

Plus concrètement, nous traiterons avec cet algorithme l'exemple du *Choixpeau magique* de la série de romans "Harry Potter" de J. K. Rowling. Le *Choixpeau magique* détermine la maison d'élèves-sorciers qui accueillera les nouveaux arrivants de la fameuse école de Poudlard. Ici, la caractéristique discrète déterminée par le *Choixpeau magique* pour chaque nouvel élève est, celle des 4 maisons de sorciers (Serpentar, Griffondor, Serdaigle, Poufssouffle) qui correspond le mieux à ses qualités (courage, loyauté, sagesse, malice).

Pour arriver à un tel traitement, ce sujet se découpe en 5 étapes successives.

- Etape 1 : commençons par trier
- Etape 2 : continuons avec l'élément majoritaire de `n` éléments
- Etape 3 : mesurons des proximités un peu compliquées
- Etape 4 : classifions Poudlard maintenant
- Etape 5 : et comme le *Choixpeau magique*, affectons les nouveaux élèves dans la maison qui leur convient le mieux !

Il est bien sûr conseillé de lire le sujet en entier avant de commencer.

Les étapes 1, 2, 3 sont très largement indépendantes. Les étapes 4 et 5 dépendent des précédentes et sont liées l'une à l'autre : l'étape 5 généralise l'étape 4.

Etape 1 : commençons par trier

On suppose une relation d'ordre définie pour des valeurs à trier. Par exemple, on veut trier des entiers par ordre croissant. On suppose aussi *dans un premier temps* que les valeurs à trier sont au nombre de n et sont stockées dans un tableau 1D : une liste en python.

Le *tri par sélection* est l'un des plus simples algorithmes de tri. Il consiste à :

- parcourir les n valeurs à trier,
- *identifier la plus petite* des valeurs,
- *déplacer* cette valeur à sa position définitive dans le tableau trié, c-a-d. en première position,
- recommencer pour *identifier* et *déplacer* la seconde plus petite des valeurs,
- et ainsi de suite pour toutes les valeurs.

A l'itération i ($1 \leq i < n$), les i plus petites valeurs sont correctement placées dans un ordre trié, c-a-d. dans les i premières cases du tableau. A la fin, les n valeurs sont correctement placées, c-a-d. de façon triée par ordre croissant.

Exemple. Le tableau `t0 = [2, 4, 1, 3]` est transformé successivement en :

- `[1, 4, 2, 3]` : 1 est la plus petite valeur de `t0`, elle est donc placé à l'indice 0,
- `[1, 2, 4, 3]` : 2 est la plus petite valeur suivante, elle est donc placé à l'indice 1,
- `[1, 2, 3, 4]` : 3 est la plus petite valeur suivante, elle est donc placé à l'indice 2.
  Il reste une dernière valeur qui se retrouve correctement placée : ici la valeur 4 placée à l'indice 3.

Ce tableau `t0` sera utilisé par la suite.

Remarquons que cet algorithme *déplace* des valeurs dans le tableau initial. Il s'agit donc d'**un tri en place**.

Si on introduit un tableau 1D de taille n supplémentaire, et si le verbe *déplacer* est remplacé par *placer*, l'algorithme ainsi formulé remplit successivement ce tableau supplémentaire à partir de son indice 0. Ce tableau supplémentaire contient de façon triée les valeurs du tableau initial. Il s'agit alors d'**un tri avec copie**.

Il y a plusieurs stratégies pour rechercher et effectuer chaque déplacement/placement.

Dans ces deux cas, la spécification de ce tri, pour un tableau de n entiers, à la forme classique suivante.

```
def tri_selection(t: List[int], n: int) -> List[int]:
    '''signature d'un tri par selection de n entiers'''
    pass
```

Tri sélection en place d'un tableau 1D d'entiers

Ecrire `tri_selection_1()` une version "en place" du tri par sélection. Une variable entière temporaire pourra servir pour les déplacements successifs.

```
def tri_selection_1(t: List[int], n: int) -> List[int]:
    '''tri par selection de n entiers : version en place'''
    assert n == len(t)
    ### BEGIN CORR
    for ind_min_trie in range(0, n-1):
        ind_min = ind_min_trie
        for i in range(ind_min_trie, n):
            if t[i] < t[ind_min]:
                ind_min = i
        # permutation t[ind_min] et t[ind_min_trie]
        tmp = t[ind_min_trie]
        t[ind_min_trie] = t[ind_min]
        t[ind_min] = tmp
        #print(t)
    return t
    ### END CORR
```

Auto-validation

La validation suivante doit s'exécuter sans erreur.

```
# Exécuter cette cellule sans la modifier, ni écrire dedans
# Son exécution ne doit pas déclencher d'erreur
t0 = [2, 4, 1, 3]
tt0 = [1, 2, 3, 4]
assert tt0 == tri_selection_1(t0, len(t0))
```

Votre validation

Définir les tableaux `t10_up` et `t10_down` qui contiennent les entiers 1, 2, ..., 10 respectivement rangés par ordre croissant (up) et décroissant (down). Puis vérifier la correction de votre algorithme avec `t10_up` et `t10_down`.

```
### BEGIN CORR
t10_up = [i for i in range(1, 11)]
t10_down = [i for i in range(10, 0, -1)]
assert t10_up == tri_selection_1(t10_down, 10)
### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
t10_up = [i for i in range(1, 11)]
t10_down = [i for i in range(10, 0, -1)]
```

```
assert t10_up == tri_selection_1(t10_down, 10)
### END TEST CACHE
```

Tri sélection avec copie d'un tableau 1D

Ecrire `tri_selection_2()` une version du tri par sélection avec copie, c-a-d. avec un tableau supplémentaire.

On rappelle par exemple que la fonction-méthode `l.pop(i)` enlève et retourne la valeur d'indice `i` de la liste python `l`. En revanche, on **interdit d'utiliser** la fonction-méthode `l.append()`.

```
def tri_selection_2(t: List[int], n: int) -> List[int]:
    '''tri par selection de n entiers : version avec copie'''
    assert n == len(t)
    ### BEGIN CORR
    t_out = [0 for i in range(n)]
    ind_min_trie = 0
    while len(t) > 0:
        ind_min = 0
        for i in range(len(t)): # parcours complet de t
            if t[i] < t[ind_min]:
                ind_min = i
        # suppresssion de t et placement dans t_out
        t_out[ind_min_trie] = t.pop(ind_min)
        #print(t, t_out)
        ind_min_trie = ind_min_trie + 1
    return t_out
    ### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
t0 = [2, 4, 1, 3]
tt0 = [1, 2, 3, 4]
assert tt0 == tri_selection_2(t0, len(t0))
#
t10_down = [i for i in range(10, 0, -1)]
assert t10_up == tri_selection_2(t10_down, 10)
### END TEST CACHE
```

Votre validation

Ecrire les tests similaires à ceux de la version en place.

```
### BEGIN CORR
t0 = [2, 4, 1, 3]
tt0 = [1, 2, 3, 4]
assert tt0 == tri_selection_2(t0, len(t0))
```

```
#
t10_down = [i for i in range(10, 0, -1)]
assert t10_up == tri_selection_2(t10_down, 10)
### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
t0 = [2, 4, 1, 3]
tt0 = [1, 2, 3, 4]
assert tt0 == tri_selection_2(t0, len(t0))
#
t10_down = [i for i in range(10, 0, -1)]
assert t10_up == tri_selection_2(t10_down, 10)
### END TEST CACHE
```

(⋆) Complexité en temps

Effectuer une analyse de la complexité en temps de ces 2 versions. Ne pas oublier de commencer par préciser le paramètre et la mesure de cette complexité. Justifier votre analyse. Expliciter les meilleurs et pire cas. En déduire la complexité asymptotique et l'écrire avec les notations de Landau.

BEGIN CORR

Réponse

- paramètre : n
- mesure : nbre de comparaisons dans le `if`
- versions 1 et 2 (itératives) :
  - on effectue n-1 parcours de t, chacun avec n-1, n-2, ..., n-(n-1)=1 comparaisons,
  - soit au totale n(n-1)/2 comparaisons
  - complexité quadratique
  - meilleur cas = pire cas
  - donc complexité asymptotique exactement quadratique
  - c-a-d $\theta(n^2)$ -version 3 : récursive
  - il y a exactement n appel récursifs sur des tableaux de longueur n, n-1, ..., 1
  - chaque appel travaille successivement
  - chaque appel effectue

END CORR

(⋆) Complexité en espace.

Que pensez-vous de la complexité en espace de ces 2 versions du tri sélection ?

BEGIN CORR

END CORR

Tri d'un tableau 2D

On dispose maintenant de données stockées dans un "tableau" 2D $n \times p$, c-a-d. un tableau de n lignes et p colonnes.

Le tableau 2D `poudlard`

On a par exemple, le "tableau" `poudlard` suivant.

Nom	Courage	Loyauté	Sagesse	Malice	Maison
Adam	9	4	7	10	Serpentar
Ahmed	9	3	4	7	Griffondor
Albertine	10	6	5	9	Griffondor
André	2	8	8	3	Serdaigle
Angelina	10	4	2	5	Griffondor

Vous aurez reconnu 6 premiers élèves-sorciers imaginaires (ici : Adam, Ahmed, Albertine, André, Angelina) d'une série déjà citée, répartis dans 3 (des 4) *Maisons* (ici : Serpentar, Griffondor, Serdaigle) en fonction de leurs qualités (courage, loyauté, sagesse, malice). D'autres élèves-sorciers et la quatrième maison (Poufssouffle) apparaîtront prochainement.

Ce "tableau" peut être stocké par la liste de listes suivante.

```
poudlard = [
    ["Adam", 9, 4, 7, 10, "Serpentar"],
    ["Ahmed", 9, 3, 4, 7, "Griffondor"],
    ["Albertine", 10, 6, 5, 9, "Griffondor"],
    ["André", 2, 8, 8, 3, "Serdaigle"],
    ["Angelina", 10, 4, 2, 5, "Griffondor"]
]
```

Ainsi les guillemets utilisés autour de "tableau" s'expliquent : toutes les valeurs stockées ne sont pas d'un type unique : il s'agit bien de liste de listes. On se dispense donc maintenant de ces guillements.

Préalable. Les en-têtes des fonctions sont systématiquement définis. Ils explicitent le type des arguments de ces fonctions.

Dans ce sujet, on va écrire des fonctions qui manipulent des tableaux 2D **de valeurs de type différents**, ici des `int` ou des `str`. python 3.7 (version instalée à ce jour sur les machines) nécessite quelques contorsions pour définir le type de tels arguments, c-a-d. des listes de `int` ou `str`.

En début de sujet, on a donc défini "un nouveau type" `IntOrStr`. Ainsi on peut maintenant écrire `List[IntOrStr]` pour caractériser une liste de valeurs `int` ou `str`.

L'importation d'`Union` du module `typing` effectuée aussi en début de sujet permet de décrire certains cas "un peu compliqués". Ainsi `Union[int, str, None]` indique une valeur de type `int`, ou `str` ou la valeur `None`. Cette construction est par exemple utilisée pour préciser le type de la valeur retournée par une fonction.

Tri sélection d'un tableau 2D

On veut maintenant *trier les lignes* d'un tableau 2D *selon les valeurs d'une de ses colonnes*. Ceci suppose qu'une relation d'ordre existe sur les valeurs de cette colonne.

Par exemple, le tableau `poudlard` est actuellement trié selon sa colonne `0`, c-a-d. selon l'ordre lexicographique (*) des `Noms`.

(*) L'ordre lexicographique est l'ordre alphabétique étendu, par exemple appliqué pour ranger les mots d'un dictionnaire.

On modifie donc la signature de la fonction de tri sélection d'un tableau 1D pour indiquer **le numéro de la colonne** selon laquelle s'effectue **le tri des lignes** du tableau 2D.

```
def tri2D_selection(t: List[List[IntOrStr]], n: int, p: int, num_col:
int) -> List[List[IntOrStr]]:
    '''signature d'un tri par selection d'un tableau 2D d'int ou de
str
    selon l'ordre croissant de la colonne de numéro num_col'''
    pass
```

Remarque. On pourrait aussi ajouter un paramètre supplémentaire pour indiquer la relation d'ordre qui s'applique sur les valeurs de la "colonne de tri". Ici les colonnes 0 ("Nom") et 5 ("Maison") sont des valeurs de type `str` tandis que les 4 autres sont de type `int`. Ainsi, on pourrait spécifier des relations d'ordre adaptées selon le choix de la colonne de tri. Par exemple, l'ordre lexicographique (croissant) pour les colonnes 0 et 5, et l'ordre naturel (croissant) pour les 4 autres. Remarquons qu'un tel paramètre serait une fonction (et non une valeur).

On va simplifier cette question en utilisant, comme pour les tableaux 1D d'entiers traités précédemment, **les relations d'ordre notées `<`, `<=`, `>=` ou `>`** qui s'appliquent (en python) de façon adaptée à des valeurs `int` (ordre naturel) ou `str` (ordre lexicographique). Attention cependant à ne pas comparer aveuglément des `int` à des `str`.

La cellule suivante qui s'exécute sans erreur vous aide à mieux comprendre si besoin.

```
# Exécuter cette cellule sans la modifier, ni écrire dedans
# Son exécution ne doit pas déclencher d'erreur
```

```
assert (2 < 9) == True
assert (9 <= 9) == True
assert (12 <= 9) == False
assert ("Adam" < "Angelique") == True
assert ("Angelique" <= "Angelique") == True
assert ("Albertine" <= "Adam") == False
```

Tri sélection 2D en place

Ecrire `tri2D_selection()` une version "en place" du tri par sélection d'un tableau 2D selon les valeurs d'une colonne.

```
def tri2D_selection(t: List[List[IntOrStr]], n: int, p: int, num_col:
int) -> List[List[IntOrStr]]:
    '''tri par selection d'un tableau 2D d'int ou de str
    selon l'ordre croissant de num_col'''
    assert n == len(t)
    for _ in range(n):
        assert p == len(t[_])
    assert 0 <= num_col < p or -p < num_col <= -1
    ### BEGIN CORR
    for ind_min_trie in range(0, n-1):
        ind_min = ind_min_trie
        for i in range(ind_min_trie, n):
            if t[i][num_col] < t[ind_min][num_col]:
                ind_min = i
        # permutation t[ind_min] et t[ind_min_trie]
        tmp = t[ind_min_trie] # tmp est une liste de taille p
        t[ind_min_trie] = t[ind_min]
        t[ind_min] = tmp
        #print(t)
    return t
    ### END CORR
```

Autovalidation

La validation suivante doit s'exécuter sans erreur. Et il est utile de comprendre *pourquoi* cette validation est ainsi construite.

```
# Exécuter cette cellule sans la modifier, ni écrire dedans
# Son exécution ne doit pas déclencher d'erreur
p1 = tri2D_selection(poudlard, len(poudlard), len(poudlard[0]), 0)
assert p1 == poudlard

for i in range(len(p1)):
    assert p1[i][0] == poudlard[i][0]

for i in range(len(p1)):
```

```
    for j in range(1, len(p1[0])):
            assert p1[i][j] == poudlard[i][j]
```

Votre validation (1)

Vérifier que le tri de `poudlard` selon la valeur `Courage` conduit bien à l'affichage suivant.

```
[['André', 2, 8, 8, 3, 'Serdaigle'], ['Ahmed', 9, 3, 4, 7,
'Griffondor'], ['Adam', 9, 4, 7, 10, 'Serpentar'], ['Albertine', 10,
6, 5, 9, 'Griffondor'], ['Angelina', 10, 4, 2, 5, 'Griffondor']]
```

Attention. Il s'agit d'un tri *en place* qui va donc modifier le tableau à trier (ici `poudlard`). Il peut donc être utile de conserver une copie du tableau initial ou de le re-générer si besoin.

```
### BEGIN CORR
poudlard = [
    ["Adam", 9, 4, 7, 10, "Serpentar"],
    ["Ahmed", 9, 3, 4, 7, "Griffondor"],
    ["Albertine", 10, 6, 5, 9, "Griffondor"],
    ["André", 2, 8, 8, 3, "Serdaigle"],
    ["Angelina", 10, 4, 2, 5, "Griffondor"]
]
res = [['André', 2, 8, 8, 3, 'Serdaigle'], ['Ahmed', 9, 3, 4, 7,
'Griffondor'], ['Adam', 9, 4, 7, 10, 'Serpentar'], ['Albertine', 10,
6, 5, 9, 'Griffondor'], ['Angelina', 10, 4, 2, 5, 'Griffondor']]
t1 = tri2D_selection(poudlard, len(poudlard), len(poudlard[0]), 1)
assert t1 == res
### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
t1 = tri2D_selection(poudlard, len(poudlard), len(poudlard[0]), 1)
assert t1 == res
### END TEST CACHE
```

Votre validation (2)

Vérifier que le tri de `poudlard` selon la `Maison` conduit bien à l'affichage suivant.

```
[['Ahmed', 9, 3, 4, 7, 'Griffondor'], ['Albertine', 10, 6, 5, 9,
'Griffondor'], ['Angelina', 10, 4, 2, 5, 'Griffondor'], ['André', 2,
8, 8, 3, 'Serdaigle'], ['Adam', 9, 4, 7, 10, 'Serpentar']]
```

Attention. Il s'agit d'un tri *en place* qui va donc modifier le tableau à trier (ici `poudlard`). Il peut donc être utile de conserver une copie du tableau initial ou de le re-générer si besoin.

```
### BEGIN CORR
t2 = tri2D_selection(poudlard, len(poudlard), len(poudlard[0]), 5)
```

```
res = [['Ahmed', 9, 3, 4, 7, 'Griffondor'], ['Albertine', 10, 6, 5, 9,
'Griffondor'], ['Angelina', 10, 4, 2, 5, 'Griffondor'], ['André', 2,
8, 8, 3, 'Serdaigle'], ['Adam', 9, 4, 7, 10, 'Serpentar']]
assert res == t2
### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
t2 = tri2D_selection(poudlard, len(poudlard), len(poudlard[0]), 5)
res = [['Ahmed', 9, 3, 4, 7, 'Griffondor'], ['Albertine', 10, 6, 5, 9,
'Griffondor'], ['Angelina', 10, 4, 2, 5, 'Griffondor'], ['André', 2,
8, 8, 3, 'Serdaigle'], ['Adam', 9, 4, 7, 10, 'Serpentar']]
assert res == t2
### END TEST CACHE
```

Etape 2 : continuons avec l'élément majoritaire de `n` éléments

Nous allons préciser deux notions différentes de la notion d'élément majoritaire parmi `n` éléments.

Elément majoritaire : sens absolu. *L'élément majoritaire* d'une liste de `n` éléments est, *lorsqu'il existe*, l'élément qui apparaît *strictement plus que* `n//2` fois.

Plusieurs algorithmes (de diverses complexités) existent pour déterminer, lorsqu'il existe, l'élément majoritaire d'un ensemble d'éléments.

Une seconde définition permet d'obtenir au moins une réponse dans tous les cas.

Elément majoritaire : sens relatif. L'*élément majoritaire* d'une liste de `n` éléments est l'élément qui apparaît le plus grand nombre de fois.

Cette définition assure l'existence d'*au moins un* élément majoritaire quelques soient les `n` éléments de départ. En revanche, l'unicité n'est plus assurée : `n` éléments, au plus, peuvent être majoritaires (au sens relatif).

Le choix de l'une ou l'autre des définitions ("absolu" *vs.* "relatif") dépend des contextes d'intérêt.

Dans la suite de ce tp, nous considérons **uniquement la formulation relative**.

Pour calculer un tel élément majoritaire, nous allons :

1. effectuer un parcours séquentiel des `n` éléments,
2. dénombrer les occurrences de chacun de ces éléments
3. en mettant à jour un dictionnaire `{(élément : nombre d'occurrences de cet élément)}`
   - la clé est un des `n` éléments

   - la valeur correspondante est le nombre d'occurrences de cet élément ;
4. puis effectuer un parcours de ce dictionnaire afin d'identifier le (ou les) élément(s) majoritaire(s).

Elément(s) majoritaire(s) d'un tableau 1D de `n` entiers

Ecrire la fonction `elemMajoritaire1D()` qui retourne sous la forme d'une liste (éventuellement réduite à un seul élément) l'élément majoritaire de n valeurs *entières* stockées dans un tableau 1D. Cette fonction retournera `None` si la liste des éléments majoritaires est vide.

Remarque. Cette fonction ne retourne pas le dictionnaire nécessaire à son exécution.

```
def elemMajoritaire1D(t : List[int], n: int) -> Union[List[int],
None]:
    '''retourne l'élément majoritaite de t tab de n entiers ou None
sinon'''
    assert n == len(t)
    ### BEGIN CORR
    if n == 0:
        return None

    nbocc = {}

    # construction et maj du dictionnaire des nb d'occ
    for v in t:
        if v in nbocc:
            nbocc[v] = nbocc[v] + 1
        else:
            nbocc[v] = 1


    # recherhce elem maj dans d
    max = nbocc[t[0]]
    res = []
    for v in nbocc:
        if nbocc[v] > max:
            max = nbocc[v]
            res = [v]
        elif nbocc[v] == max:
            res.append(v)
    return res
    ### END CORR
```

Auto-validation

La validation suivante doit s'exécuter sans erreur.

```
# Exécuter cette cellule sans la modifier, ni écrire dedans
# Son exécution ne doit pas déclencher d'erreur
```

```
t0 = [2, 4, 1, 3]
t1 = [3, 3, 1, 3]
assert sorted(elemMajoritaire1D(t0, len(t0))) == sorted(t0)
assert elemMajoritaire1D(t1, len(t1)) == [3]
```

Votre validation

Ecrire des tests de validation qui complètent les précédents : tableaux de taille impair, tableau vide.

```
### BEGIN CORR
assert elemMajoritaire1D([], 0) == None
assert sorted(elemMajoritaire1D([3, 3, 1, 3, 1], 5)) == [3]
assert sorted(elemMajoritaire1D([3, 3, 1, 1, 2], 5)) == [1, 3]
### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
assert elemMajoritaire1D([], 0) == None
assert sorted(elemMajoritaire1D([3, 3, 1, 3, 1], 5)) == [3]
assert sorted(elemMajoritaire1D([3, 3, 1, 1, 2], 5)) == [1, 3]
### END TEST CACHE
```

Elément(s) majoritaire(s) d'un tableau 2D

On étend naturellement aux tableaux 2D la notion d'élément majoritaire *selon une de ses colonnes*. L'élément majoritaire est alors l'élément majoritaire (au sens relatif) des `n` éléments qui composent la colonne choisie.

`elemMajoritaire2D()`

Ecrire la fonction `elemMajoritaire2D()` qui effectue ce traitement selon une colonne d'un tableau 2D similaire à `poudlard`.

```
def elemMajoritaire2D(t : List[List[IntOrStr]], n: int, p: int,
num_col: int) -> Union[int, str, None]:
    '''retourne l'élément majoritaite de t tab2D de n lignes d'int ou
str,
    retourne None si pas d'elemnt majoritaire'''
    assert n == len(t)
    for i in range(n):
        assert p == len(t[i])
    ### BEGIN CORR
    # extraction de la num_col de t comme t1D
    tmp = [v[num_col] for v in t]
    return elemMajoritaire1D( tmp, len(tmp) )
    ### END CORR
```

Autovalidation

La validation suivante doit s'exécuter sans erreur.

```
# Exécuter cette cellule sans la modifier, ni écrire dedans
# Son exécution ne doit pas déclencher d'erreur
maisonMaj = elemMajoritaire2D(poudlard, len(poudlard),
len(poudlard[0]), 5)
assert maisonMaj == ["Griffondor"]
```

Votre validation

Ecrire des tests de validation qui vérifient le(s) élément(s) majoritaire(s) pour chacune des qualités des 6 élèves-sorciers actuels de `poudlard`.

```
### BEGIN CORR
for q in range(1,5):
    q_maj = elemMajoritaire2D(poudlard, len(poudlard),
len(poudlard[0]), q)
    print(sorted(q_maj))
### END CORR
```

```
[9, 10]
[4]
[2, 4, 5, 7, 8]
[3, 5, 7, 9, 10]
```

```
# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
res = [
    [9, 10],
    [4],
    [2, 4, 5, 7, 8],
    [3, 5, 7, 9, 10]
]

for q in range(1,5):
    assert sorted(elemMajoritaire2D(poudlard, len(poudlard),
len(poudlard[0]), q)) == res[q-1]
### END TEST CACHE
```

Etape 3 : mesurons des proximités un peu compliquées

Il existe de nombreuses distances pour mesurer la proximité entre deux objets mathématiques. Intéressons-nous au cas suivant.

Chaque ligne du tableau 2D `poudlard` contient un vecteur de 4 valeurs discrètes : courage, loyauté, sagesse, et malice. Ces valeurs sont des entiers positifs entre 0 et 10.

On peut donc extraire de la ligne i, qui décrit l'élève-sorcier h_i, les 4 valeurs (c_i, l_i, s_i, m_i) de **c**ourage, **l**oyauté, **s**agesse et **m**alice de l'élève-sorcier h_i.

Par exemple, la ligne d'`Adam` donne le vecteur (9, 4, 7, 10) et celle d'`Ahmed` le vecteur (9, 3, 4, 7).

Distance de Manhattan ... à Poudlard !

La **distance** (dite) **de Manhattan** est adaptée pour mesurer la proximité de 2 élèves-sorciers du tableau `poudlard` (c-a-d. de 2 lignes de ce tableau). On la définit comme suit pour 2 élèves-sorciers h_1 et h_2 du tableau `poudlard` :

$$d(h_1, h_2) = |c_1 - c_2) + |l_1 - l_2) + |s_1 - s_2) + |m_1 - m_2)$$

où $h_i = (c_i, l_i, s_i, m_i)$ pour $i = 1, 2$.

Par exemple, la distance de Manhattan entre `"Adam"` et `"Ahmed"` vaut $|9 - 9) + |4 - 3) + |7 - 4) + |10 - 7) = 7$.

`distM()`

Ecrire selon l'en-tête suivante, la fonction `distM()` qui retourne la distance de Manhattan entre 2 élèves-sorciers décrits par les lignes `i` et `j` du tableau `t` à `n` lignes et `p` colonnes. Le paramètre `cols` est le t-uple qui décrit les numéros des colonnes du tableau `t` qui forment le vecteur à considérer pour le calcul de la distance entre les 2 lignes.

Par exemple pour le tableau `poudlard`, le tuple (1, 2, 3, 4) décrit les 4 valeurs (c_i, l_i, s_i, m_i) mentionnées plus haut.

```
def distM(i: int, j: int, cols: Tuple[int],
          t: List[List[IntOrStr]], n: int, p: int) -> int:
    '''retourne la distance de Manhattan entre les vecteurs des
colonnes cols
    des lignes i et j du tableau t de taille n x p.'''
    assert n == len(t)
    for _ in range(n):
        assert p == len(t[_])
    assert 0 <= i < n or -n <= i < -1
    assert 0 <= j < n or -n <= j < -1
    ### BEGIN CORR
    d = 0
    for k in cols:
        d = d + abs(t[i][k] - t[j][k])
    return d
    ### END CORR
```

Autovalidation

La validation suivante doit s'exécuter sans erreur.

Attention. Le tableau `poudlard` est redéfini dans son état original (pour éviter les éventuels effets de bord des traitements précédents) .

```
# Exécuter cette cellule sans la modifier, ni écrire dedans
# Son exécution ne doit pas déclencher d'erreur
poudlard = [
    ["Adam", 9, 4, 7, 10, "Serpentar"],
    ["Ahmed", 9, 3, 4, 7, "Griffondor"],
    ["Albertine", 10, 6, 5, 9, "Griffondor"],
    ["André", 2, 8, 8, 3, "Serdaigle"],
    ["Angelina", 10, 4, 2, 5, "Griffondor"]
]
assert distM(0, 0, (1,2,3,4), poudlard, 5, 6) == 0  # distance entre
Adam et lui-même
assert distM(0, 1, (1,2,3,4), poudlard, 5, 6) == 7  # distance entre
Adam et Ahmed
```

Votre validation

Ecrire des tests qui valident votre fonction `distM()` sur `Adam`, `Albertine` et `Angelina`.

```
### BEGIN CORR
assert distM(0, 2, (1,2,3,4), poudlard, 5, 6) == 6
assert distM(0, 4, (1,2,3,4), poudlard, 5, 6) == 11
assert distM(2, 4, (1,2,3,4), poudlard, 5, 6) == 9
assert distM(4, 2, (1,2,3,4), poudlard, 5, 6) == 9
#
assert distM(2, 2, (1,2,3,4), poudlard, 5, 6) == 0
assert distM(4, 4, (1,2,3,4), poudlard, 5, 6) == 0
### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
assert distM(0, 2, (1,2,3,4), poudlard, 5, 6) == 6
assert distM(0, 4, (1,2,3,4), poudlard, 5, 6) == 11
assert distM(2, 4, (1,2,3,4), poudlard, 5, 6) == 9
assert distM(4, 2, (1,2,3,4), poudlard, 5, 6) == 9
#
assert distM(2, 2, (1,2,3,4), poudlard, 5, 6) == 0
assert distM(4, 4, (1,2,3,4), poudlard, 5, 6) == 0
### END TEST CACHE
```

Application à la proximité entre un nouvel élève et les élèves-sorciers de Poudlard

Un nouvel élève doit être intégré à `poudlard`. Pour des raisons que nous expliquerons dans la section suivante, nous préparons son intégration en réalisant le traitement suivant.

1. Ce nouvel élève est décrit par un t-uple de **6 valeurs** :
   - `(son_nom, son_courage, sa_loyauté, sa_sagesse, sa_malice, "")` ;
   - `son_nom` est une `str` et les 4 valeurs suivantes des `int` entre 0 et 10.
   - Remarquons qu'à la différence des élèves-sorciers de `poudlard`, ce nouvel élève n'appartient à aucune `Maison`. Ainsi la 6-ème valeur du t-uple est la chaîne vide `""`.
2. On calcule la distance de Manhattan entre ce nouvel élève et chaque élève-sorcier de `poudlard`
   - ce qui donne `n` distances (des entiers entre 0 et 40).
3. On ajoute chacune de ces distances à la ligne correspondant à chaque élève-sorcier dans le tableau `poudlard`
   - ce qui donne une colonne supplémentaire après celle de `Maison`
   - et modifie donc le tableau `poudlard`.

Indications.

- Il est commode d'ajouter *temporairement* ce nouvel élève à `poudlard` pour profiter du calcul réalisé par la fonction `distM()`.

- Ce nouvel élève est un t-uple et `poudlard` est une liste de listes.

Exemple. Préparons l'intégration aux 6 élèves-sorciers actuels de `poudlard` de :

```
eleve_bientot_celebre = ("Hermione", 8, 6, 6, 6, "")
```

Après le traitement décrit, le tableau `poudlard` aura la forme **différente** suivante :

```
[
    ['Adam', 9, 4, 7, 10, 'Serpentar', 8],
    ['Ahmed', 9, 3, 4, 7, 'Griffondor', 7],
    ['Albertine', 10, 6, 5, 9, 'Griffondor', 6],
    ['André', 2, 8, 8, 3, 'Serdaigle', 13],
    ['Angelina', 10, 4, 2, 5, 'Griffondor', 9]
]
```

- La dernière valeur de chaque ligne est la proximité entre "Hermione" et chaque élèves-sorciers.
- Cette valeur ajoute une colonne supplémentaire au tableau initial.
- En revanche, remarquons qu'aucune *nouvelle ligne*, c-a-d. aucun nouvel élèves-sorciers, n'a été ajouté à `poudlard` **après ce traitement** : "Hermione" n'est pas encore une héroïne !

`proximitesNouvelEleve()`

Ecrire ce traitement avec la fonction suivante `proximitesNouvelEleve()`.

- On choisit de laisser l'appel de cette fonction définir que le traitement s'effectue sur le tableau `poudlard` (comme paramètre effectif du tableau `t` et de ses dimensions `n`x`p`).
- Cette fonction modifie le tableau paramètre `t` en **ajoutant une dernière colonne supplémentaire** -- comme illustré dans l'exemple précédent pour un appel sur le tableau `poudlard`.

```
def proximitesNouvelEleve(eleve : Tuple[IntOrStr], t:
List[List[IntOrStr]],
          n: int, p: int) -> List[List[IntOrStr]]:
    '''calcule et ajoute en dernière colonne la distance de Manhattan
    entre eleve et les heros actuels de Poudlard'''
    assert n == len(t)
    for _ in range(n):
        assert p == len(t[_])
    ### BEGIN CORR
    # ajout temporaire du nouvel elève comme dernière ligne de t avec
conversion de type
    t.append(list(eleve))

    # ajout de la nouvelle colonne avant tout calcul (pour passer les
assert de distM())
    for i in range(n+1):
        t[i].append(0)

    # remarquons la modification des nombre de lignes et colonnes de
l'appel suivant
    for i in range(n+1):
        d = distM(n, i, (1,2,3,4), t, n + 1, p + 1)
        t[i][-1] = d

    # retrait de la derniere ligne temporaire
    t.pop(n)

    # post-condition sur les dimensions du tableau resultat
    assert n == len(t)
    for _ in range(n):
        assert p + 1 == len(t[_])

    return t
    ### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
poudlard = [
    ["Adam", 9, 4, 7, 10, "Serpentar"],
    ["Ahmed", 9, 3, 4, 7, "Griffondor"],
    ["Albertine", 10, 6, 5, 9, "Griffondor"],
```

```
    ["André", 2, 8, 8, 3, "Serdaigle"],
    ["Angelina", 10, 4, 2, 5, "Griffondor"]
]

res = [
    ['Adam', 9, 4, 7, 10, 'Serpentar', 8],
    ['Ahmed', 9, 3, 4, 7, 'Griffondor', 7],
    ['Albertine', 10, 6, 5, 9, 'Griffondor', 6],
    ['André', 2, 8, 8, 3, 'Serdaigle', 13],
    ['Angelina', 10, 4, 2, 5, 'Griffondor', 9]
]

assert res == proximitesNouvelEleve(eleve_bientot_celebre, poudlard,
len(poudlard), len(poudlard[0]))
### END TEST CACHE
```

Votre validation

Vérifier que l'application de votre fonction `proximitesNouvelEleve()` à `poudlard` est conforme à l'exemple précédent.

```
### BEGIN CORR
poudlard = [
    ["Adam", 9, 4, 7, 10, "Serpentar"],
    ["Ahmed", 9, 3, 4, 7, "Griffondor"],
    ["Albertine", 10, 6, 5, 9, "Griffondor"],
    ["André", 2, 8, 8, 3, "Serdaigle"],
    ["Angelina", 10, 4, 2, 5, "Griffondor"]
]

res = [
    ['Adam', 9, 4, 7, 10, 'Serpentar', 8],
    ['Ahmed', 9, 3, 4, 7, 'Griffondor', 7],
    ['Albertine', 10, 6, 5, 9, 'Griffondor', 6],
    ['André', 2, 8, 8, 3, 'Serdaigle', 13],
    ['Angelina', 10, 4, 2, 5, 'Griffondor', 9]
]

assert res == proximitesNouvelEleve(eleve_bientot_celebre, poudlard,
len(poudlard), len(poudlard[0]))
### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
poudlard = [
    ["Adam", 9, 4, 7, 10, "Serpentar"],
    ["Ahmed", 9, 3, 4, 7, "Griffondor"],
    ["Albertine", 10, 6, 5, 9, "Griffondor"],
    ["André", 2, 8, 8, 3, "Serdaigle"],
    ["Angelina", 10, 4, 2, 5, "Griffondor"]
```

```
]

res = [
    ['Adam', 9, 4, 7, 10, 'Serpentar', 8],
    ['Ahmed', 9, 3, 4, 7, 'Griffondor', 7],
    ['Albertine', 10, 6, 5, 9, 'Griffondor', 6],
    ['André', 2, 8, 8, 3, 'Serdaigle', 13],
    ['Angelina', 10, 4, 2, 5, 'Griffondor', 9]
]

assert res == proximitesNouvelEleve(eleve_bientot_celebre, poudlard,
len(poudlard), len(poudlard[0]))
### END TEST CACHE
```

Les `k` plus proches élèves-sorciers d'un nouvel élève

En utilisant l'étape 1 (tri d'un tableau 2D selon l'une de ses colonnes), il est maintenant facile :

1. de trier le tableau résultat de `proximitesNouvelEleve()` selon les distances entre le nouvel élève et les élèves-sorciers,
2. pour extraire **les `k` élèves-sorciers les plus proches** de ce nouvel élève où `k` est un entier positif arbitraire -- mais inférieur au nombre d'élèves-sorciers, c-a-d. au nombre de lignes de `t`.

Remarque. Il n'y a pas nécessairement unicité du résultat si des proximités égales apparaissent de part et d'autres du k-ième rang.

On propose un traitement qui permet de profiter du tri en place du tableau résultat de `proximitesNouvelEleve()` en séparant ces deux étapes.

`proximitesNouvelEleve()` : version triée

Reprendre la fonction `proximitesNouvelEleve()` de façon à retourner le tableau trié selon les distances entre le nouvel élève et les élèves-sorciers.

On rappelle que la préparation de l'intégration d'un nouvel élève ajoute une nouvelle dernière colonne au tableau : la colonne des distances entre les élèves-sorciers et cet élève. Ce traitement modifie donc le tableau paramètre `t`.

```
def proximitesNouvelEleve(eleve : Tuple[IntOrStr], t:
List[List[IntOrStr]],
          n: int, p: int) -> List[List[IntOrStr]]:
    '''version triee selon derniere colonne de :
    du calcul et ajout en dernière colonne la distance de Manhattan
    entre eleve et les heros actuels de Poudlard'''
    assert n == len(t)
    for _ in range(n):
        assert p == len(t[_])
    ### BEGIN CORR
    # ajout temporaire du nouvel elève comme dernière ligne de t avec
```

```
    conversion de type
    t.append(list(eleve))

    # ajout de la nouvelle colonne avant tout calcul (pour passer les
assert de distM())
    for i in range(n+1):
        t[i].append(0)

    # remarquons la modification des nombre de lignes et colonnes de
l'appel suivant
    for i in range(n+1):
        d = distM(n, i, (1,2,3,4), t, n + 1, p + 1)
        t[i][-1] = d

    # retrait de la derniere ligne temporaire
    t.pop(n)

    #  tri dselon la dernière colonne ajoutée
    tri2D_selection(t, n, p+1, -1)

    # post-condition sur les dimensions du tableau resultat
    assert n == len(t)
    for _ in range(n):
        assert p + 1 == len(t[_])

    return t
    ### END CORR
```

Autovalidation

La validation suivante qui modifie le tableau `poudlard` pour préparer l'intégration de la nouvelle élève "Hermione", doit s'exécuter sans erreur.

```
# Exécuter cette cellule sans la modifier, ni écrire dedans
# Son exécution ne doit pas déclencher d'erreur
poudlard = [
    ["Adam", 9, 4, 7, 10, "Serpentar"],
    ["Ahmed", 9, 3, 4, 7, "Griffondor"],
    ["Albertine", 10, 6, 5, 9, "Griffondor"],
    ["André", 2, 8, 8, 3, "Serdaigle"],
    ["Angelina", 10, 4, 2, 5, "Griffondor"]
]

eleve_bientot_celebre = ("Hermione", 8, 6, 6, 6, "")

poudlard = proximitesNouvelEleve(eleve_bientot_celebre, poudlard,
len(poudlard), len(poudlard[0]))

res = [
```

```
    ['Albertine', 10, 6, 5, 9, 'Griffondor', 6],
    ['Ahmed', 9, 3, 4, 7, 'Griffondor', 7],
    ['Adam', 9, 4, 7, 10, 'Serpentar', 8],
    ['Angelina', 10, 4, 2, 5, 'Griffondor', 9],
    ['André', 2, 8, 8, 3, 'Serdaigle', 13]
]

assert res == poudlard
```

`kPlusProches()`

On effectue maintenant l'extraction des `k` élèves-sorciers les plus proches du nouvel élève.

- Ce traitement prendra comme argument `t` **le résultat de la version avec tri de `proximitesNouvelEleve()`**
  - Attention aux dimensions de ce tableau.
- Ce traitement donnera un **nouveau** tableau uniquement constitué des `k` lignes extraites du tableau `t`.
  - Ainsi celui-ci ne pourra être trié qu'une seule fois si plusieurs tentatives d'extraction pour différentes valeurs de `k` sont nécessaires.

```
def kPlusProches( k: int, t: List[List[IntOrStr]], n: int, p: int  ) -
> List[List[IntOrStr]]:
    '''retourne sous la même forme que t les k plus proches lignes de
t'''
    assert n == len(t)
    for _ in range(n):
        assert p == len(t[_])
    ### BEGIN CORR
    # construction du tab des k premieres lignes de tmp
    return [v for v in t[0:k]]
    ### END CORR
```

Votre validation

Vérifier que les 3 plus proches élèves-sorciers d'`"Hermione"` sont `"Albertine"`, `"Ahmed"`et `"Adam"`.

```
### BEGIN CORR
poudlard = [
    ["Adam", 9, 4, 7, 10, "Serpentar"],
    ["Ahmed", 9, 3, 4, 7, "Griffondor"],
    ["Albertine", 10, 6, 5, 9, "Griffondor"],
    ["André", 2, 8, 8, 3, "Serdaigle"],
    ["Angelina", 10, 4, 2, 5, "Griffondor"]
]

eleve_bientot_celebre = ("Hermione", 8, 6, 6, 6, "")
```

```
poudlard = proximitesNouvelEleve(eleve_bientot_celebre, poudlard,
len(poudlard), len(poudlard[0]))
proxHermione = kPlusProches(3, poudlard, len(poudlard),
len(poudlard[0]))

res = {'Albertine', 'Ahmed', 'Adam'}
for e in proxHermione:
    assert e[0] in res
    res = res - {e[0]}
assert len(res) == 0
### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
poudlard = [
    ["Adam", 9, 4, 7, 10, "Serpentar"],
    ["Ahmed", 9, 3, 4, 7, "Griffondor"],
    ["Albertine", 10, 6, 5, 9, "Griffondor"],
    ["André", 2, 8, 8, 3, "Serdaigle"],
    ["Angelina", 10, 4, 2, 5, "Griffondor"]
]

eleve_bientot_celebre = ("Hermione", 8, 6, 6, 6, "")

poudlard = proximitesNouvelEleve(eleve_bientot_celebre, poudlard,
len(poudlard), len(poudlard[0]))
proxHermione = kPlusProches(3, poudlard, len(poudlard),
len(poudlard[0]))

res = {'Albertine', 'Ahmed', 'Adam'}
for e in proxHermione:
    assert e[0] in res
    res = res - {e[0]}
assert len(res) == 0
### END TEST CACHE
```

Etape 4 : classifions Poudlard maintenant !

On dispose maintenant des outils nécessaires à l'algorithme des k plus proches voisins pour l'apprentissage automatique (*machine learning*) supervisé

On suppose un ensemble de données dont on connaît *toutes les* caractéristiques. Ces caractéristiques peuvent être discrètes et de type divers (numériques ou alphanumérique).

Par exemple, nos 6 élèves-sorciers de `poudlard` sont connus par des caractéristiques de type `int` pour courage, loyauté, sagesse, ... ou des `str` comme "Griffondor", "Serpentar", ... pour leur maison. Cette dernière caractéristique peut être vue comme une **étiquette** qui regroupe certains élèves-sorciers : les élèves-sorciers de la maison "Griffondor" sont ..., les élèves-sorciers de la maison "Serpentar" sont ..., ...

Implicitement, on suppose qu'il y a moins d'étiquettes différentes que de combinaisons des autres valeurs des caractéristiques. Ici, il y a 4 maisons et 11^4 combinaisons possibles des autres caractéristiques.

La classification consiste à identifier *l'étiquette d'une nouvelle donnée* dont on connaît les autres caractéristiques. Cette détermination s'effectue à partir de la connaissance de l'ensemble des données connues.

L'algorithme des k plus proches voisins choisit pour *étiquette d'une nouvelle donnée*, l'étiquette majoritaire des k données les plus proches de cette nouvelle donnée (dans l'ensemble des données connues).

Dans notre exemple, cela pourrait être l'algorithme du *Choixpeau magique* de Poudlard lorsque celui-ci affecte un nouvel élève dans une des 4 maisons Poudlard.

Bien sûr, la pertinence de cette classification dépend :

- du nombre de données connues,
- de la valeur `k` du nombre de voisins considérés,

et d'autres paramètres que nous oublierons aujourd'hui (la distance entre 2 éléments, ...)

4 nouveaux élèves arrivent à Poudlard ...

Nous allons donc maintenant mettre en oeuvre cet algorithme pour affecter chacun des 4 nouveaux élèves suivants dans une maison qui correspondant à leurs qualités.

Nom	Courage	Loyauté	Sagesse	Malice
Hermione	8	6	6	6
Drago	6	6	5	8
Cho	7	6	9	6
Cedric	7	10	5	6

Regrouper ces 4 nouveaux élèves dans `nouveauxEleves` : un tableau 1D de 4 t-uples. Chaque t-uple est d'une forme similaire à `eleve_bientot_celebre` (c'est un t-uple de longueur 6).

```
### BEGIN CORR
nouveauxEleves = [
    ("Hermione", 8, 6, 6, 6, "" ),
    ("Drago", 6, 6, 5, 8, ""),
    ("Cho", 7, 6, 9, 6, ""),
    ("Cedric", 7, 10, 5, 6, "")
]
### END CORR

# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
res  = [
    ("Hermione", 8, 6, 6, 6, "" ),
```

```
    ("Drago", 6, 6, 5, 8, ""),
    ("Cho", 7, 6, 9, 6, ""),
    ("Cedric", 7, 10, 5, 6, "")
]
assert res == nouveauxEleves
### END TEST CACHE
```

Première tentative pour valider cette approche

Notre connaissance actuelle des données connues se limite aux 6 élèves-sorciers du tableau `poudlard`. Bien sûr, ce n'est pas satisfaisant pour une classification pertinente : il manque par exemple des représentants de la maison "Poufsouffle" dans les données connues !

Mais c'est suffisant pour mettre en oeuvre *un prototype* de traitement en attendant de pouvoir disposer d'une base de connaissance suffisante. Ce qui sera l'objet de la dernière étape de ce tp.

Votre validation

On fixe arbitrairement `k = 2` pour cette première expérience.

Vérifier que cette première expérience affecte 3 des 4 nouveaux élèves dans une même maison et hésite entre 2 maisons pour le quatrième.

```
k = 2
### BEGIN CORR
for e in nouveauxEleves:

    # on repart d'un poudlard "propre"
    poudlard = [
    ["Adam", 9, 4, 7, 10, "Serpentar"],
    ["Ahmed", 9, 3, 4, 7, "Griffondor"],
    ["Albertine", 10, 6, 5, 9, "Griffondor"],
    ["André", 2, 8, 8, 3, "Serdaigle"],
    ["Angelina", 10, 4, 2, 5, "Griffondor"]
    ]

    # les distances triées au nouvel eleve e -> modifie poudlard
    poudlard = proximitesNouvelEleve(e, poudlard, len(poudlard),
len(poudlard[0]))

    # extraction des k plus proches heros de e
    kProxE = kPlusProches(k, poudlard, len(poudlard),
len(poudlard[0]))

    # classification par maison majoritaire de kProxE (maison toujours
en colonne 5)
    maisonE = elemMajoritaire2D(kProxE, len(kProxE), len(kProxE[0]),
5)
```

```
    print(e[0], maisonE)
    ### END CORR
```

Bien sûr, les résultats obtenus ne sont pas pertinents : ils sont déduits d'un échantillon trop incomplet pour représenter une base de connaissances représentative de Poudlard.

Etape 5 : et comme le *Choixpeau magique*, affectons les nouveaux élèves dans la maison qui leur convient le mieux.

Un Poudlard plus complet

Le fichier `poudlard2022.csv` contient les 50 élèves-sorciers de Poudlard avant la rentrée 2022.

Ce fichier est au format "csv" (*Comma Separated Values*, valeurs séparées par des virgules). Le module python `csv` permet de lire de tels fichiers de façon assez similaire à la lecture des fichiers textes vus en cours. Les curieux pourront consulter cette partie de la documentation python.

```
import csv
```

La fonction suivante lit ce fichier et retourne son contenu sous la forme d'un tableau 2D similaire à `poudlard`-- mais avec 50 lignes maintenant.

```
def lirePoudlard(nomDuFichier: str) -> List[List[IntOrStr]]:
    '''lit nomDuFichier au format csv et retourne un tableau 2D'''
    res = []
    with open(nomDuFichier, 'r', encoding="utf-8") as donnees_csv:
        eleves = csv.reader(donnees_csv, delimiter=";")
        eleves.__next__() # saut ligne 0
        for e in eleves:
            val_e = (str(e[0]), int(e[1]), int(e[2]), int(e[3]),
int(e[4]), str(e[5]))
            res.append(list(val_e))
    return res
```

Initialiser le tableau 2D `poudlard2022` avec la base de connaissances donnée par le fichier "poudlard2022.csv".

```
### BEGIN CORR
poudlard2022 = lirePoudlard("./poudlard2022.csv")
### END CORR
```

```
# Ne pas écrire dans cette cellule
### BEGIN TEST CACHE
poudlard2022_ref = lirePoudlard("./poudlard2022.csv")
assert poudlard2022 == poudlard2022_ref
### END TEST CACHE
```

Votre validation

Vous pouvez par exemple vérifier la présence de l'élève-sorcier suivant.

```
['Demelza', 10, 6, 5, 3, 'Griffondor']

### BEGIN CORR
print(poudlard2022[10])
### END CORR
```

Le *Choixpeau magique*

Vous pouvez enfin reprendre l'expérimentation de l'étape 4 pour affecter les 4 nouveaux élèves dans leurs maisons. Faire varier `k` jusqu'à obtenir le même résultat que celui obtenu par le *Choixpeau magique* de Poudlard pour ces 4 nouveaux, et bientôt célèbres, élèves.

Soit (pour celles et ceux qui ne s'en souviennent pas) :

```
Hermione -> 'Griffondor'
Drago    -> 'Serpentar'
Cho      -> 'Serdaigle'
Cedric   -> 'Poufsouffle'

### BEGIN CORR
# les données
nouveauxEleves = [
    ("Hermione", 8, 6, 6, 6, "" ),
    ("Drago", 6, 6, 5, 8, ""),
    ("Cho", 7, 6, 9, 6, ""),
    ("Cedric", 7, 10, 5, 6, "")
]

# le traitement
for k in range(2, 7):
    print("k = ", k)

    for e in nouveauxEleves:
        #print(e)

        # on repart d'un poudlard "propre"
        poudlard2022 = lirePoudlard("./poudlard2022.csv")

        # les distances triées au nouvel eleve e -> modifie poudlard
```

```
        poudlard2022 = proximitesNouvelEleve(e, poudlard2022,
len(poudlard2022), len(poudlard2022[0]))

        # extraction des k plus proches heros de e
        kProxE = kPlusProches(k, poudlard2022, len(poudlard2022),
len(poudlard2022[0]))

        # classification par maison majoritaire de kProxE (maison
toujours en colonne 5)
        maisonE = elemMajoritaire2D(kProxE, len(kProxE),
len(kProxE[0]), 5)

        print(e[0], maisonE)
    print("----")
### END CORR
```