

Ds ALGORITHME des k plus proches voisins



Exercice 1 : Algorithme des k plus proches voisins

A l'entrée à l'école de Poudlard, le Choixpeau magique répartit les élèves dans les différentes maisons (*Gryffondor*, *Serpentard*, *Serdaigle* et *Poufsouffle*) en fonction de leur courage, leur loyauté, leur sagesse et leur malice).

L'objectif de ce DS est de réaliser un code qui permet de répartir un nouvel élève dans une des 4 maisons, en fonction des informations ci-dessous, qui concernent les élèves déjà inscrits à l'école et qui se sont vus attribuer une note sur 10 pour chacune de leur vertu :

Nom	Courage	Loyauté	Sagesse	Malice	Maison
Abdenour	9	4	7	10	Serpentard
Titouan	9	3	4	7	Gryffondor
Ilian	10	6	5	9	Gryffondor
Mathis	2	8	8	3	Serdaigle
Félix	10	4	2	5	Gryffondor
Corentin	10	4	9	9	Poufsouffle
Kylian	10	7	4	7	Gryffondor
etc					

Cette liste des élèves déjà répartis dans l'école est beaucoup plus longue. On en a ci-dessus qu'un petit extrait. La liste entière est contenue dans le fichier *poudlard.csv* mis à disposition.

Une nouvelle élève arrive à l'école de Poudlard. Elle est suédoise et s'appelle *Camilla*. Un test de positionnement lui a permis d'obtenir une note de 8 en Courage, 6 en Loyauté, 6 en Sagesse, 6 en Malice.

L'objectif de ce DS sera de réaliser un code qui reprend un algorithme du type « k plus proches voisins » et qui déterminera, parmi les élèves déjà inscrits à Poudlard, les maisons des 5 plus proches voisins de *Camilla*.

Avant de se lancer dans la réalisation de ce code, on donne des indications sur la distance à utiliser dans cet algorithme ...

On décide d'utiliser la relation suivante pour calculer la distance entre deux élèves 1 et 2 :

$$d = |\text{courage}_1 - \text{courage}_2| + |\text{loyauté}_1 - \text{loyauté}_2| + |\text{sagesse}_1 - \text{sagesse}_2| + |\text{malice}_1 - \text{malice}_2|$$

Par exemple la distance d entre *Camilla* la suédoise et *Abdenour* sera :

Nom	Courage	Loyauté	Sagesse	Malice	Maison
Abdenour	9	4	7	10	Serpentard
Camilla	8	6	6	6	?

$$d = |9 - 8| + |4 - 6| + |7 - 6| + |10 - 6|$$

$$d = |1| + |-2| + |1| + |4|$$

$$d = 1 + 2 + 1 + 4 = 8$$

En langage python, la fonction qui calcule la valeur absolue d'un nombre est **abs()** .

Par exemple :

```
>>> abs(-2)
2
>>> abs(4-100)
96
>>> abs(40-1)
39
```

TRAVAIL A REALISER :

⇒ Se loguer avec l'identifiant : **exam02.eleve** Mot de passe :


Le code à réaliser sera appelé *ds_mon_nom.py* . Il est à déposer en fin d'épreuves dans le répertoire : **Examens(Z :) /exam02/copies/**

⇒ **Copier** le dossier *NSI-4oct2022* : **Examens(Z :) /exam04/sujets/NSI-4oct2022** sur le disque dur de l'ordinateur (sur le *bureau* ou dans le répertoire *Mes documents*) :

Vous y trouvez 3 fichiers :



⇒ Ouvrez le fichier python dans pyzo. Assurez-vous que la case **Changer le répertoire courant lors de l'exécution d'un fichier** est bien cochée dans le menu *Exécuter* de Pyzo.

⇒ Le code contenu dans le fichier  *choixpeauMagique.py* comprend 3 fonctions et 1 programme principal.

- La fonction *lecture_csv()* est **complète**. Elle n'est pas à modifier et permet de créer une liste de liste.
- La fonction *tri_selection()* permet de trier une liste ℓ **simple**. Elle sera à modifier tout à l'heure ...
- La fonction *kProchesVoisins()* contient pour l'instant des appels à la fonction *print()* afin de pouvoir afficher dans le shell le contenu des 3 variables mises en argument.
- Le programme principal ci-contre permet d'initialiser les listes *echantillon* et *mystere* et ensuite d'exécuter la fonction *kProchesVoisins()*.

```
# Main
echantillon = lecture_csv("poudlard.csv")
mystere = ["Camilla",8,6,6,6]
kProchesVoisins(echantillon , mystere , 5)
```

⇒ Exécuter le code du fichier  *choixpeauMagique.py* et observer le contenu des listes *echantillon* et *mystere*.

1. Créer une fonction *distance()* .

On a vu précédemment que la distance *d* entre Camilla la suédoise et Abdenour était

$$d = |\text{courage1} - \text{courage2}| + |\text{loyauté1} - \text{loyauté2}| + |\text{sagesse1} - \text{sagesse2}| + |\text{malice1} - \text{malice2}| = 8$$

Nom	Courage	Loyauté	Sagesse	Malice	Maison
Abdenour	9	4	7	10	Serpentard
Camilla	8	6	6	6	?

⇒ Créer une fonction nommée *distance()* qui prend en argument une liste *l* et une liste *mystere*. Cette fonction renvoie la distance *d* entre la personne de l'échantillon représentée dans *l* et la suédoise Camilla représentée dans *mystere*.

```
def distance(l,mystere):
    """
    Arguments :
    l : liste du type [Nom (string) , Courage(int) , Loyaute(int) , Sagesse(int) , Malice(int), Maison(string) ]
    mystere : liste du type [Nom (string) , Courage(int) , Loyaute(int) , Sagesse(int) , Malice(int) ]
    Renvoie la distance (float) entre "l" et "mystère" : distance = |courage1-courage2| +
    |loyaute1-loyaute2| + |sagesse1-sagesse2| + |malice1-malice2|
    """
    d = 0
    for i in range(1,5) :
        d += abs(l[i]-mystere[i])
    return d
```

Vous pourrez valider le code de cette fonction en réalisant le test suivant qui affichera 8 dans le shell :

```
mystere = ["Camilla",8,6,6,6]
l = ['Abdenour', 9, 4, 7, 10, 'Serpentar']
d = distance(l,mystere)
print(d)
```

2. Créer une fonction *ajout_distance()* .

⇒ Créer une fonction nommée *ajout_distance()* qui prend en argument la liste de listes *echantillon* et la liste *mystere*. Cette fonction ajoute en fin de chacune des listes qui composent la liste de listes *echantillon*, la distance entre la personne représentée dans *echantillon* et la suédoise Camilla représentée dans *mystere*. On utilisera dans cette fonction, la fonction *distance()* créée précédemment.

```
def ajout_distance(echantillon,mystere) :
    """
    Arguments :
    echantillon : listes contenant plusieurs listes du type :
    [Nom (string) , Courage(int) , Loyaute(int) , Sagesse(int) , Malice(int), Maison(string) ]
    mystere : liste du type [Nom (string) , Courage(int) , Loyaute(int) , Sagesse(int) , Malice(int) ]
    Modifie chaque liste de la liste echantillon en rajoutant sur l'index 6, la distance d avec mystere
    """
    for i in range(len(echantillon)) :
        d = distance(echantillon[i],mystere)
        echantillon[i].append(d)
```

Vous pourrez valider le code de cette fonction en réalisant le test suivant :

```
mystere = ["Camilla",8,6,6,6]
echantillon = [['Abdenour', 9, 4, 7, 10, 'Serpentar'] , ['Titouan', 9, 3, 4, 7, 'Griffondor']]
ajout_distance(echantillon,mystere)
print(echantillon)
```

A l'exécution, on doit trouver :

```
[['Abdenour', 9, 4, 7, 10, 'Serpentar', 8], ['Titouan', 9, 3, 4, 7, 'Griffondor', 7]]
```

3. Modifier la fonction `tri_selection()` .

⇒ Le code donné dans cette fonction est adapté à des listes simples. L'utiliser pour trier la liste *echantillon* complète `echantillon = lecture_csv("poudlard.csv")` par ordre croissant des distances *d* .

```
def tri_selection(l) :
    """
    Arguments :
        l : liste échantillon dont les éléments sont des listes du type
            [Nom (string) , Courage(int) , Loyaute(int) , Sagesse(int) , Malice(int), Maison(string), distance(float) ]
    Que fait cette fonction ? :
        Modifie l'ordre de l, en repositionnant les listes l[i] par ordre croissant de la distance l[i][6]
    """
    for i in range(len(l)-1) :
        iMin = i
        for j in range(i+1,len(l)) :
            if l[j][6] < l[iMin][6] :
                iMin = j
        # on échange
        tmp = l[i]
        l[i] = l[iMin]
        l[iMin] = tmp
```

4. Finaliser le code complet .

⇒ Créer la fonction `kVoisins()` qui prend en argument la liste de listes *echantillon* et le nombre *k* de voisins (on prendra ensuite *k* = 5). Elle permet d'imprimer dans le shell les maisons des *k* plus proches voisins afin de pouvoir définir la maison à laquelle appartiendra Camilla.

```
def kVoisins(l,k) :
    """
    Arguments :
        l : liste échantillon dont les éléments sont des listes du type
            [Nom (string) , Courage(int) , Loyaute(int) , Sagesse(int) , Malice(int), Maison(string), distance(float) ]
    Que fait cette fonction ? :
        Affiche les k voisins les plus proches
    """
    for i in range(k) :
        print(l[i][5] , end=" ")
```

```
def kProchesVoisins(echantillon , mystere , k) :
    """
    Arguments :
        echantillon : liste dont les éléments sont des listes du type
            [Nom (string) , Courage(int) , Loyaute(int) , Sagesse(int) , Malice(int), Maison(string) ]
        mystere : liste de type [prenom (string) , Courage(int) , Loyaute(int) , Sagesse(int) , Malice(int)]
    Que fait cette fonction ? :
        Cette fonction imprime dans le shell les maisons des k plus proches voisins de mystere
    """
    ajout_distance(echantillon,mystere)
    tri_selection(echantillon)
    kVoisins(echantillon , k)

# Main
echantillon = lecture_csv("poudlard.csv")
mystere = ["Camilla",8,6,6,6]
kProchesVoisins(echantillon , mystere , 5)
```

L'exécution de ce code donnera dans le shell :

```
Griffondor Serpentar Griffondor Serdaigle Poufsouffle
```

Exercice 2 : Algorithme d'une recherche dichotomique

On travaille toujours sur la liste de liste *echantillon* de l'exercice précédent qui est normalement à présent triée par ordre croissant des distances. Pour ceux qui n'ont pas eu le temps de traiter correctement l'exercice précédent, vous travaillerez sur la liste suivante :

```

echantillon = [
    ['Robin', 9, 6, 5, 4, 'Griffondor', 4]
    ['Milicent', 9, 3, 5, 6, 'Serpentar', 5]
    ['Neville', 10, 5, 6, 4, 'Griffondor', 5]
    ['Padma', 6, 6, 6, 9, 'Serdaigle', 5]
    ['Susan', 5, 6, 5, 5, 'Poufsouffle', 5]
    ['Kyliau', 10, 7, 4, 7, 'Griffondor', 6]
    ['Benjamin', 9, 8, 4, 7, 'Griffondor', 6]
    ['Shun', 10, 6, 5, 3, 'Griffondor', 6]
    ['Renan', 5, 4, 6, 5, 'Serpentar', 6]
    ['Lavande', 10, 8, 8, 6, 'Griffondor', 6]
    ['Ilian', 10, 6, 5, 9, 'Griffondor', 6]
    ['Titouan', 9, 3, 4, 7, 'Griffondor', 7]
]

```

La fonction ci-contre permet de tester si la distance contenue dans la variable d est présente dans la liste de listes *echantillon*.

```

def recherche_naif(echantillon,d) :
    for l in echantillon :
        if l[6] == d :
            return True
    return False

```

L'exécution de cette fonction donnera :

```

a = recherche_naif(echantillon,6)
if a : print('oui')
else : print('non')
                                oui

a = recherche_naif(echantillon,99)
if a : print('oui')
else : print('non')
                                non

```

⇒ Créer la fonction *recherch_dicho()* qui retourne la même chose, mais en réalisant une recherche de type dichotomique.

```

def recherche_dicho(echantillon,d)

```

```

def recherche_dicho(echantillon,d) :
    iDeb = 0
    iFin = len(echantillon)-1
    while iDeb <= iFin :
        m = (iDeb + iFin)//2
        if d == echantillon[m][6] :
            return True
        elif d > echantillon[m][6] :
            iDeb = m+1
        elif d < echantillon[m][6] :
            iFin = m-1
    return False

```

⇒ Quelle est la complexité du code de la fonction *recherche_naif()* et celui de la fonction *recherche_dico()* ?

Dans le code de la fonction *recherche_naif()* la liste est parcourue en totalité si la valeur à trouver n'est pas présente dans la liste. La complexité de ce code est donc **linéaire**, en $O(n)$. Dans celui de la fonction *recherche_dico()*, la liste est divisée à chaque itération en 2. Cet algorithme a une complexité en $O(\log_2(n))$.