1. Écrire une fonction `dist(x1,x2)` qui prend en arguments deux listes de même taille, et calcule la distance euclidienne entre les deux.

```
from math import sqrt
def dist(x1,x2) :
    S = 0
    for i in range (len(x1)) :
        S += (x1[i]-x2[i])**2
    return sqrt(S)
```

2. Écrire une fonction `distances_au_point(delits, x )` qui prend en arguments la matrice contenant les données criminelles des N individus et une liste de longueur p contenant les données criminelles d'un nouvel individu, et qui renvoie une liste distances de longueur N où la i-ème case contient la distance de x à l'observation i de délits.

```
def distances_au_point(delits, x ):
    distances = []
    N = delits.shape[0]
    for i in range (N):
        x_i = delits[i] # observation i
        distances.append(dist(x, list(x_i)) ) # dist n'accepte que des
            listes en entrée
    return distances
```

3. Écrire une fonction `trier_individus(distances)` qui prend en argument une liste contenant les distances de x à chaque observation et qui renvoie une liste `indices_tries` contenant les indices de la liste rangés par distance croissante. On pourra utiliser la fonction `sorted` de Python.

```
def trier_individus(distances):
        N = len ( distances )
        def f(i):
                return distances[i]
        return sorted ( range (N),key=f)
```

4. Écrire une fonction `moyenne_des_k_voisins(indices_tries, notes, k)` qui prend en arguments la liste contenant les indices des observations triées par distance croissante à x, la liste contenant les notes d'insoumission des individus et le nombre de voisins à considérer, et qui renvoie la moyenne des notes d'insoumission des k plus proches voisins de x.

```
def moyenne_des_k_voisins(indices_tries, notes, k ) :
        indices_voisins = indices_tries[:k]
        note = 0
        for i in indices_voisins :
                note += notes[i]
        note /=k # Division par k pour obtenir la moyenne
        return note
```

5. En déduire une fonction `k_plus_proches_voisins(delits, notes, x,k )` qui renvoie la note d'insoumission estimée pour le citoyen ayant x comme données criminelles.

```
def k_plus_proches_voisins(delits, notes, k) :
        distances = distances_au_point(delits, x )
        indices_tries = trier_individus(distances)
        note = moyenne_des_k_voisins(indices_tries, notes, k)
        return note
```

6. Pour fixer les valeurs initiales des k pôles, on procède de la manière suivante : pour chacun des k pôles, pour chacun de ses q descripteurs, on choisit une valeur aléatoire uniformément répartie entre la valeur minimale et la valeur maximale de ce descripteur calculée sur l'ensemble

des M données. Écrire une fonction `initialisation_poles(infos, k)` qui génère la matrice `pole` initiale. On pourra utiliser les fonctions `min` et `max` de Python, ainsi que la fonction `uniform(a,b)` du package `random`, qui renvoie un nombre aléatoire compris entre a et b.

```
from random import uniform
import numpy as np
def initialisation_poles(infos, k) :
        M,q = infos.shape
        poles = np.zeros((k,q))
        for qq in range(q):
                mini, maxi = min(infos[:,qq]), max(infos[:,qq])
                for kk in range(k) :
                        poles[kk,qq]= uniform(mini, maxi)
        return poles
```

7. Après l'étape d'initialisation, il faut affecter chaque observation à son pôle le plus proche. Écrire une fonction `pole_le_plus_proche(poles, x)` qui prend en arguments la matrice contenant les caractéristiques des pôles et une liste contenant les caractéristiques d'un individu, et qui renvoie l'indice du pôle le plus proche de x.

```
def pole_le_plus_proche(poles, x):
        k= poles.shape[0]
        distance_min = np.inf
        pole_x = 0
        for j in range(k) :
        distance = dist(x, list(poles[j]))
        if distance < distance_min:
                distance_min = distance
        return pole_x =j
```

8. Écrire une fonction `poles_les_plus_proches(poles, infos)` qui renvoie une liste qu'on appellera `repartition`, dont la i-ième case contient l'indice du pôle le plus proche de l'observation i.

```
def poles_les_plus_proches(poles, infos):
        M = infos.shape[0]
        repartition =[]
        for i in range(M):
                repartition.append(pole_le_plus_proche(poles, list(
                   infos[i])) )
        return repartition
```

9. Écrire une fonction `nouveaux_poles(infos, repartition, k)` qui renvoie une matrice `poles` contenant les caractéristiques des barycentres de chaque catégorie.

La principale difficulté réside dans le calcul du nombre d'observations n_ℓ affectées à chaque catégorie ℓ : en effet, on en a besoin pour le calcul des moyennes. Nous allons créer une liste de taille k dans laquelle nous allons stocker ces nombres n_ℓ.

```
def nouveaux_poles(infos, repartition, k) :
        M, q = infos.shape
        nb_observations = [0 for i in range(k)]
        poles = np.zeros((k,q))
        for i in range (M) :
                l = repartition[i] # catégorie à laquelle appartient
                   l'observation i
                poles[l]+= infos[i] # on l'ajoute aux autres
                   observations de même classe qu'elle
                nb_observations[l]+=1
        # Il ne nous reste plus qu'à diviser chaque pôle I par le
           nombre d'observations classées I
                for l in range(k) :
                        if nb_observations[l] != 0 :
```

```
                                poles[l] /= nb_observations[l]
        return poles
```

10. En déduire une fonction `k_moyennes(infos, k)` qui renvoie la matrice `poles` des pôles, correspondant aux k classes trouvées par l'algorithme des k-moyennes.

```
def k_moyennes(infos, k) :
        poles = initialisation_poles(infos, k)
        a_bouge = True
        while a_bouge :
                repartition = poles_les_plus_proches(poles, infos)
                poles0 = nouveaux_poles(infos, repartition, k)
                if
                a_bouge = (poles != poles0).any () # teste si les
                   pôles ont bougé
                poles = poles0
        return poles
```

11. Si un nouveau citoyen atteint la majorité, comment le classerons-nous ? On suppose qu'on dispose de ses données complètes, rangées dans une liste x de longueur q, et des caractéristiques `poles` des pôles calculées par l'algorithme des k-moyennes.

```
pole_le_plus_proche(poles, x)
```