**Introduction**

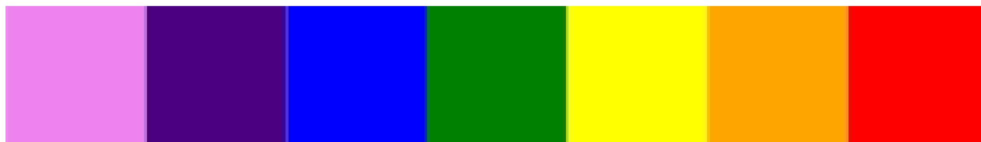
Imaginons que j'ai un ensemble d'individus pour lesquels je suis capable d'attribuer une qualité. Arrive un nouvel individu, pour lequel je n'ai pas de qualité qui soit attribuée. On cherche quelle qualité lui attribuer. Il apparaît cohérent de choisir pour ce nouvel individu la qualité de l' (ou des) individu(s) qui est (sont) le plus proche de lui...

Ce principe algorithmique s'appelle l'**algorithme des k plus proches voisins**.
C'est un bon exemple de **machine learning**, ou d'**intelligence artificielle** : on utilise une base de données pour catégoriser le plus finement possible une nouvelle entrée en cherchant ses voisins proches.

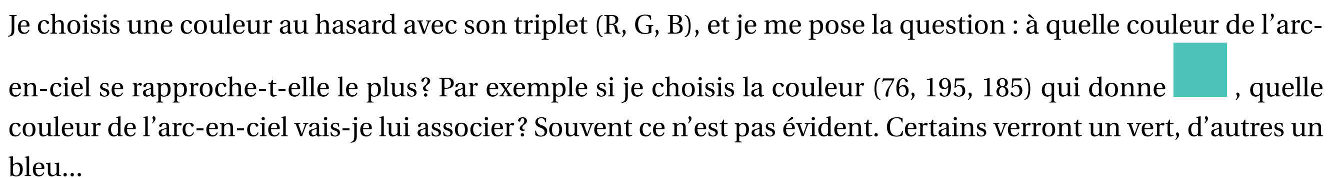
Les pré-requis :

- l'algorithmique de base, la notion de **list** Python et de dictionnaire;
- un algorithme de tri;
- les données en table, l'import de fichier CSV avec la bibliothèque **csv**;
- la gestion des données tabulées sous la forme d'une liste de dictionnaires.

Description du TP : j'ai choisi 7 couleurs dans l'arc-en-ciel : violet, indigo, bleu, vert, jaune, orange et rouge.



Je choisis une couleur au hasard avec son triplet (R, G, B), et je me pose la question : à quelle couleur de l'arc-en-ciel se rapproche-t-elle le plus? Par exemple si je choisis la couleur (76, 195, 185) qui donne , quelle couleur de l'arc-en-ciel vais-je lui associer? Souvent ce n'est pas évident. Certains verront un vert, d'autres un bleu...



**On va pour cela utiliser une base de donnée. J'ai proposé** 150 **couleurs (R, G, B) prises au hasard à une personne. Pour chaque valeur (R,G,B), je lui est demandé à quelle couleur de l'arc-en-ciel (qualité) elle l'associe. C'est l'apprentissage (le learning)**. Voir l'annexe 1 pour la description de l'expérience.
Par la suite, on pourrait utiliser cette base de donnée pour « prédire » quelle couleur de l'arc-en-ciel elle associerai à un autre triplet (R,G,B) quelconque.
La base de données est donc une table composée de deux champs :

- la **valeur** : ici une couleur avec son triplet (R, G, B)
- la **qualité** (une des couleurs de l'arc-en ciel)

Les données seront donc importées sous la forme d'une **liste** de dictionnaires. Chaque ligne correspond à une donnée, c'est à dire ici concrètement l'association d'une couleur, quantifiée avec son triplet « valeur » (R, G, B), à une couleur « qualite », une couleur de l'arc-en-ciel, selon le modèle :

Le code pour importer le fichier **csv** avec la bibliothèque Python **csv**.

```
table = [
{ 'valeur' : (R,G,B), 'qualite' : couleur }
..
]
```

# 1 Importer les données à partir d'un fichier CSV

Récupérer le fichier **couleurs.csv** et en reprenant le code ci-dessous, importer les données et les mettre sous la forme souhaitée, c'est à dire avec la variable **table** décrite ci-dessous.

```
import csv
fichier = open("couleurs.csv")
data = csv.DictReader(fichier, delimiter = ",")
table_brute = list(data)
```

La structure du fichier CSV est la suivante :

```
R,G,B,qualite
165,249,99,vert
131,23,221,indigo
```

Penser à utiliser la fonction **int** pour transformer en entier les valeur (R,G,B). Vérifier que la variable **table** commence par :

```
table = [
{ 'valeur': (165, 249, 99), 'qualite': 'vert'},
{ 'valeur': (131, 23, 221), 'qualite': 'indigo'},
...
```

Et si vous n'y arrivez pas, copier et coller la variable **table** donnée en annexe...

Quelques questions sur la manipulation de la variable **table**.

1. Que renvoie la commande **table**[**3**] ?
2. Que renvoie la commande **table**[**3**][**'valeur'**] ?
3. Que renvoie la commande **table**[**3**][**'valeur'**][**2**] ?
4. Que renvoie la commande **table**[**3**][**'qualite'**] ?

## 2 Mesurer la distance entre deux données

Si on veut savoir à quelle(s) donnée(s) rapprocher une nouvelle entrée, il faut être en capacité de « mesurer » l'écart entre les deux, c'est à dire la distance. Ici chaque donnée est quantifiée avec un triplet (R, G, B).

Observer la fonction **distance** ci-dessous, qui prend comme argument deux valeurs (ici des triplets), et renvoie une information.

```
def distance(valeur_1, valeur_2):
  dist = (valeur_2[0] - valeur_1[0])**2 + (valeur_2[1] - valeur_1[1])**2 +
   (valeur_2[2] - valeur_1[2])**2
  return dist
```

1. Pourquoi cette information peut être qualifiée de distance ?
2. Vérifier que pour (5,8,6) et (10,51, 23) la distance renvoyée est 2163. Calculer d'abord à la main, puis en utilisant l'algorithme.
3. Tester cette fonction en calculant la distance entre les deux premières lignes de la variable table. Vérifier que l'on obtient 67116.

## 3 Le choix du plus proche voisin

On propose une nouvelle entrée, c'est à dire ici un nouveau triplet (R, G, B).
**On décide de lui attribuer la couleur (qualité) du plus proche de lui dans la table.**

Écrire une fonction **plusProcheVoisin** qui prend comme argument la table, le triplet, et qui renvoie la qualité du plus proche voisin dans la table.

Effectuer les tests ci-dessous pour vérifier le fonctionnement de votre algorithme.

Et si vous n'y arrivez pas, vous pouvez vous laisser guider par la méthodologie décrite ci-dessous.

```
def plusProcheVoisin(table, triplet):
    # on garde en mémoire le proche = (num, distance) du plus proche de triplet
    # on initialise proche avec le premier de la table
    proche = (table[0], distance(table[0]['valeur'], triplet)
    pour chaque elt dans le reste de la table :
        on calcule sa distance au triplet,
        si elle est inférieure au minimum stocké dans proche, on le recopie dans proche
    on renvoie la qualite de proche.
```

Et si vous n'y arrivez pas, vous pouvez regarder une solution en annexe 3.

Quelques tests à réaliser :

1. vérifier que **plusProcheVoisin(table, (139, 167, 242) )** renvoie **'bleu'** ;
2. vérifier que **plusProcheVoisin(table, (12, 245, 25) )** renvoie **'vert'** ;
3. vérifier que **plusProcheVoisin(table, (186, 18, 32) )** renvoie **'rouge'** ;

# 4 Les k plus proches voisins

Quand on choisit un nouveau triplet, choisir le plus proche n'est pas forcément optimal : en effet si le plus proche est le "vert" et qu'ensuite vient trois fois le "bleu", il aurait mieux valu lui attribuer le "bleu" ...
L'idée dans cette partie n'est plus de se contenter du plus proche voisin, mais des 2 ou 3 ou 4 plus proches voisins... et plus généralement des $k$ plus proches voisins, où $k$ sera une valeur choisie, donc passée en paramètre.
Pour y parvenir, on va d'abord programmer deux fonctions utiles : l'une qui, à partir de la liste qualités trouvées sur les $k$ plus proches, renvoie la plus présente, puis une autre qui triera les distances mesurées pour ensuite ne retenir que les plus $k$ plus proches.

## 4.1 L'élément le plus présent dans une liste

Imaginons que parmi les trois plus proches voisins d'un triplet, les qualités notées sont dans cet ordre : ['bleu', 'vert','vert']. La qualité la plus présente est 'vert', donc c'est elle qu'il faut retenir. Nous allons donc commencer par programmer une fonction qui réalise cette recherche dans une liste.

Écrire une fonction **plusForteOccurrence(maListe)** qui prend comme argument une liste **maListe**, et qui renvoie l'élément de cet liste qui a la plus forte occurrence.

Et si vous n'y arrivez pas, vous pouvez vous laisser guider par la méthodologie ci-dessous.

```
def plusForteOccurrence(liste):
    """
    fonction qui renvoie l'élément qui a la plus forte occurrence dans une liste
    """
    dico est un dictionnaire vide et plusFort est nul
    pour chacun des éléments de la liste :
        si élément fait partie du dico
            alors augmenter de 1 la clé élément du dico
        sinon
            mettre la clé élément du dico à 1
        si la clé élément du dico est supérieur à plusFort alors :
            plusFort prend la valeur de élément correspondant dans le dico
            et la réponse prend la valeur de élément
    retourner la reponse
```

Et si vous n'y arrivez pas, voir une solution an annexe 4.

Quelques tests à réaliser :

1. vérifier que **plusForteOccurrence(['bleu', 'vert', 'vert'])** renvoie **'vert'** ;
2. vérifier que **plusForteOccurrence(['bleu', 'bleu', 'vert', 'vert'])** renvoie **'bleu'** ;
3. vérifier que **plusForteOccurrence(['vert', 'bleu', 'bleu', 'vert', 'bleu'])** renvoie **'bleu'** ;

### 4.2 Trier une table de données selon un champ

Pour la recherche des $k$ plus proches voisins d'une entrée, on va commencer par dresser une table de distances que l'on triera. On écrira cette table de distance sous la forme suivante.

```
table_distance = [
{  'dist' : 35 , 'qualite' : 'bleu'} ,
{  'dist' : 7 , 'qualite' : 'vert'} ,
{  'dist' : 18 , 'qualite' : 'jaune'}
]
```

Écrire une fonction **trierTable(tableATrier, champ)** qui prend comme argument une table, et qui la renvoie triée selon le champ indiqué.

Pour faire simple, choisir l'ordre croissant... Et si vous n'y arrivez pas, voir une solution en annexe 5.

Test : vérifier que la commande **trierTable(table_distance, 'dist')** renvoie :

```
[
{'dist': 7, 'qualite': 'vert'},
{'dist': 18, 'qualite': 'jaune'},
{'dist': 35, 'qualite': 'bleu'}      ]
```

On peut constater que la table est triée. Vous pouvez réaliser des tests sur d'autres tables de votre choix...

### 4.3 Les k plus proches voisins

Et maintenant, en utilisant les deux fonctions suivantes, nous allons pouvoir programmer des $k$ plus proches voisins d'une entrée.

Écrire une fonction **kPlusProchesVoisins(table, entree, k)** qui prend comme argument la table de « données », le triplet « nouvelle entrée » et un entier $k > 1$, et qui renvoie **LA** qualité la plus présente chez ses $k$ plus proches voisins.

On utilise pour cela les deux fonctions précédentes. Et si vous n'y arrivez pas, vous pourrez vous laisser guider par la méthodologie décrite ci-dessous.

```
def kPlusProchesVoisins(table, entree, k):
    """
    fonction qui renvoie LA qualité la plus présente chez les k plus proches voisins.
    Pour faire on crée une liste dans laquelle on insère les k > 1 plus proches
    chaque élément de la liste est un dictionnaire avec : distance et qualite
    """
    Dresser la table avec les champs "distance - qualité"
    Trier cette table selon la distance
    Extraire les k premiers
    Faire la liste des qualités sur les k premiers
    Renvoyer la plus forte occurrence
```

Effectuer le test ci-dessous pour vérifier le fonctionnement de votre algorithme.
La commande **print(kPlusProchesVoisins(table , (15, 68, 152) , 5))** renvoie **'bleu'**.
Pour information, les 6 plus proches voisins de la couleur (15, 68, 152) dans la table sont **['bleu', 'bleu', 'bleu', 'indigo', 'bleu']**.

Et si vous n'y arrivez pas, voir une solution en annexe 6.

## 5 Quelle valeur de k choisir?

On se pose la question : pour notre table, quelle valeur de k choisir? A-t-on de meilleurs résultats si on prend les $k = 3$, les $k = 16$ où les $k = 30$ plus proches voisins?

Pour répondre à cette question, on va se servir de la table que l'on va séparer en deux de manière arbitraire : une première partie (20 % environ) qui va nous servir de tableAtester, et une seconde partie (80 % restants environ) qui vont nous servir de tableDeReference.

Pour chaque ligne de la tableATester, on compare le résultat de l'algorithme des k plus proches voisins avec la qualité effective de la ligne, et ce pour différentes valeurs de $k$.

Donc, pour différentes valeurs de $k$, on va calculer le taux de réussite des $k$ plus proches voisins de tableATester dans tableDeReference. On pourra donner le résultat sous forme d'un dictionnaire :

```
resultat = {k : taux de réussite}
```

Écrire une fonction **testDesValeursDeK(tableATester, tableDeReference, kmax)** qui renvoie pour chaque valeur de k le taux de réussite de tableATester dans TableDeReference.

Et si vous n'y arrivez pas, voir une solution en annexe 7.

Pour les tests, on prendra les variables suivantes :

```
tab1 = list(table[0:40]) # les éléments à tester, on prend les 40 premiers
tab2 = list(table[40:]) # la table de référence, le reste de la table


#le test à effectuer :
print(testDesValeursDeK(tab1, tab2, 30))
```

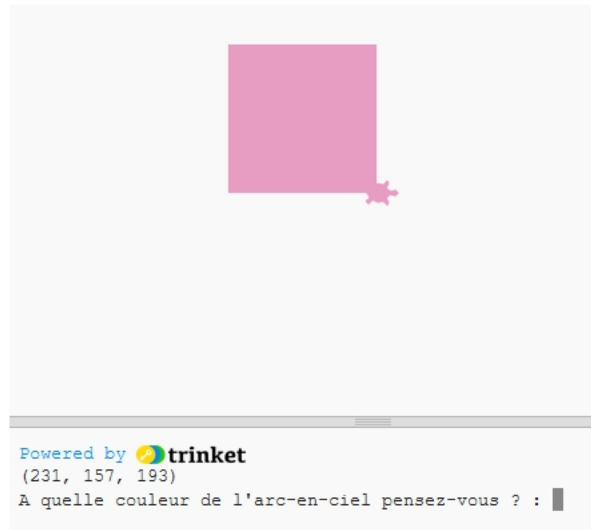
Voici les taux obtenus. Quel est la valeur de $k$ qui donne le meilleur résultat pour la table?

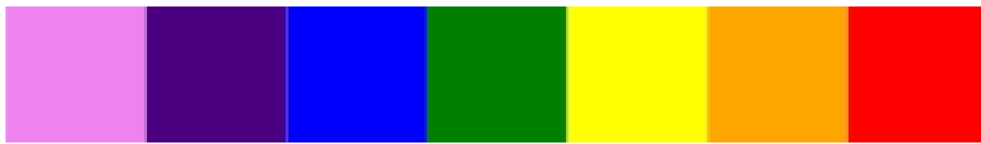
```
{1: 0.8, 2: 0.8, 3: 0.85, 4: 0.875, 5: 0.825, 6: 0.825,
7: 0.825, 8: 0.825, 9: 0.85, 10: 0.85, 11: 0.85, 12: 0.85,
13: 0.875, 14: 0.875, 15: 0.875, 16: 0.925, 17: 0.9, 18: 0.875,
19: 0.875, 20: 0.875, 21: 0.875, 22: 0.85, 23: 0.85,
24: 0.875, 25: 0.875, 26: 0.875, 27: 0.85, 28: 0.825,
29: 0.825, 30: 0.8}
```

Prolongement : réaliser un graphique avec les taux de succès en fonction de $k$. Voir l'annexe 8 pour exemple avec la **table** de ce TP.

### ANNEXE 1 : LA SAISIE DES DONNÉES AVEC TURTLE

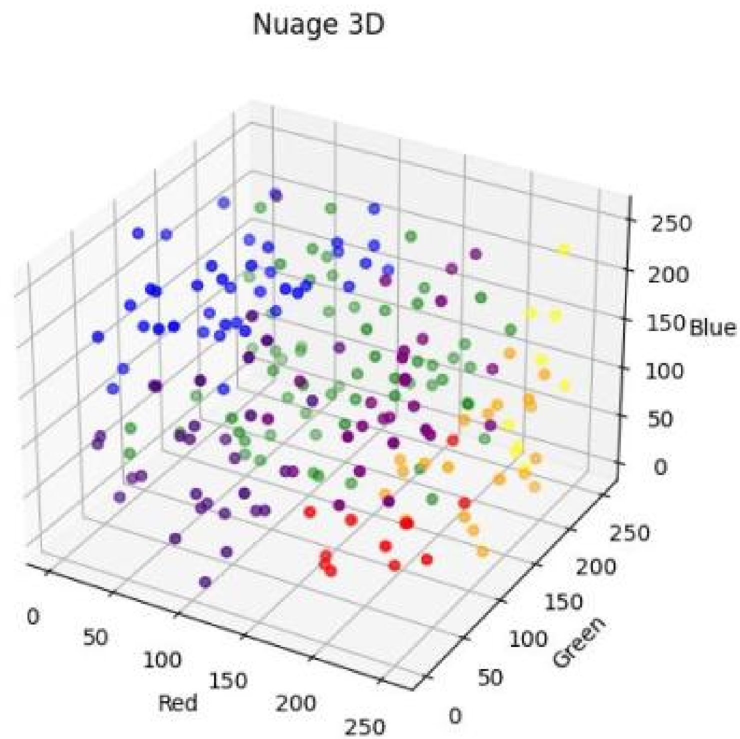
La copie d'écran ci-dessous montre la routine réalisée avec **Turtle** qui permet d'acquérir les données : on propose au sujet un triplet (R, G, B), et il indique à quelle couleur de l'arc en ciel il pense. C'est la phase d'apprentissage (learning). L'idée est ensuite d'utiliser la base de données pour ensuite prédire ce que la personne donnerait comme couleur si on lui propose un autre triplet (R, G, B).



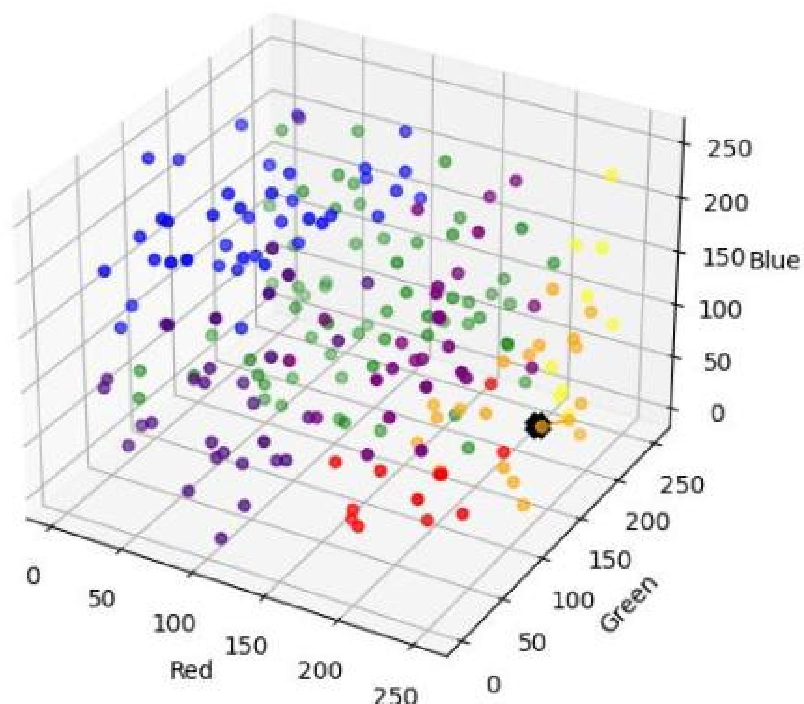



**ANNEXE 2 : VISUALISATION DES DONNÉES ET DE LA RECHERCHE**

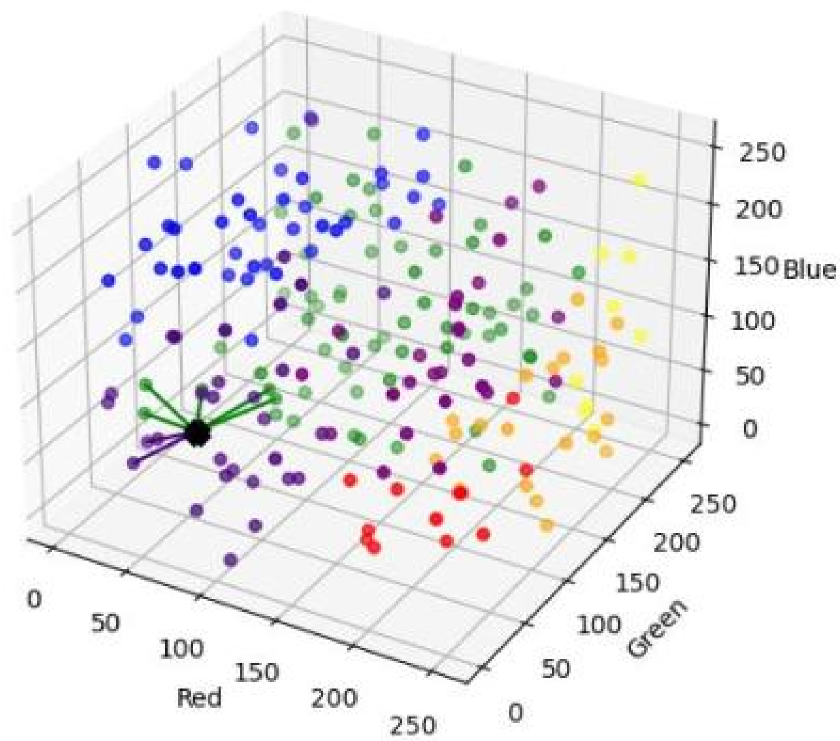
Chaque point représente en 3D les triplets (R, G, B) qui ont été présentés à la personne, et la couleur qui a été associée. C'est une visualisation de la **table**.




On a rajouté un point au hasard, et on l'a relié au plus proche voisin.



On a relié un point au hasard à ses 7 plus proches voisins. C'est l'idée de la recherche de ce TP.




En approfondissement, vous pourrez essayer de réaliser avec ces graphiques avec le mode 3D de **matplotlib** : il s'agit de « scatter 3D ».

**ANNEXE 3 : UNE SOLUTION POUR LE PLUS PROCHE VOISIN**

```
#########################################
# LE plus proche voisin
#########################################

def plusProcheVoisin(table, valeur):
  """
  fonction qui renvoie LA qualité du plus proche voisin de la valeur dans la table
  """
  # on utilise couple pour retenir le plus proche sous la forme (distance, qualité)
  # on initialise ce couple avec le premier de la table
  couple = ( distance(table[0]['valeur'], valeur) , table[0]['qualite'] )
  # pour chaque élément dans la table pour 1 et suivant
  for elt in table[1:]:
          # on calcule la distance de cet élément au triplet
    dist = distance( elt['valeur'], valeur )
    # si la distance entre elt et la valeur est inférieure au min,
    # on redéfinit le couple
    if dist < couple[0] :
      couple = ( dist, elt['qualite'] )
  # on renvoie la qualité du couple
  return couple[1]
```

**ANNEXE 4 : UNE SOLUTION POUR plusForteOccurrence**

```
def plusForteOccurrence(li):
  """
  fonction qui renvoie l'élément qui a la plus forte occurrence dans une liste
  """
  dico = {}
  plusFort = 0
  for elt in li :
    if elt in dico :
      dico[elt] +=1
    else :
      dico[elt] = 1
    if dico[elt] > plusFort :
        plusFort = dico[elt]
        reponse = elt
  return reponse

print(plusForteOccurrence(['bleu', 'vert','vert'])
# doit donner 'vert'
```

**ANNEXE 5 : UNE SOLUTION POUR trierTable**

```python
def trierTable(tableATrier, champ):
    """
    renvoie la tableTriee
    par ordre croissant selon un champ
    en utilisant la méthode par insertion
    par ordre croissant
    """
    # on initialise le renvoi avec le premier élément de la table_a_trier
    tableTriee = [ tableATrier[0] ]
    # pour chaque elt restant dans tableAtrier :
    for ligne in tableATrier[1:] :
        # on l'insere dans la tableTriee en utilisant l'insertion
        # pour cela on dépile tant que c'est possible
        # et la valeur du nouveau champ reste inférieure au dernier...
        depile = []
        while tableTriee and tableTriee[-1][champ] > ligne[champ]:
            depile.append(tableTriee.pop())
        tableTriee.append(ligne)
        while depile :
            tableTriee.append(depile.pop())
    # on renvoie la tableTriee
    return tableTriee
```

**ANNEXE 6 : UNE SOLUTION POUR kPlusProchesVoisins**

```
def kPlusProchesVoisins(table, entree, k):
  """
  fonction qui renvoie LA qualité la plus présente chez les k plus proches voisins.
  Pour faire on crée une liste dans laquelle on insère les k > 1 plus proches
  chaque élément de la liste est un dictionnaire avec : distance et qualite
  """
  # table_distance est une liste de dictionnaires {'dist' : float , 'qualite' : str}
  # on la génére avec chaque élément de la table
  table_distance = []
  for ligne in table :
      dico = {}
      dico['dist'] = distance(ligne['valeur'], entree)
      dico['qualite'] = ligne['qualite']
      table_distance.append(dico)
  print(table_distance)
  # on trie table distance selon le champ dist
  table_distance_triee = trierTable(table_distance, 'dist')
  print(table_distance_triee)
  # on dresse la liste des k premières qualités
  k_qualite = []
  for elt in table_distance_triee[0:k] :
    k_qualite.append(elt['qualite'])
  print(k_qualite)
  # et enfin on recherche la plus présente dans cette liste
  return plusForteOccurrence(k_qualite)
```

**ANNEXE 7 : UNE SOLUTION POUR testDesValeursDeK**

```python
##########################################################
# CALCUL DES FREQUENCES DE REUSSITE POUR DES VALEURS DE K
##########################################################


def testDesValeursDeK(tableATester, tableDeReference, kmax):
  """
  renvoie pour chaque valeur de k le taux de réussite
  de la tableATester dans la tableDeReference.
  On utilise un  dictionnaire { k : taux de succes }
  pour les valeurs de k choisies
  La valeur de k ne doit pas dépasser le nombre d'éléments
  de la TableDeReference....
  """
  resultat = {}
  # pour chaque valeur de k,
  # on teste le nombre de succes de TableATester dans TableDeReference
  for k in range(1,kmax+1):
    succes = 0
    # pour chaque elt dans tableATester,
    # on recherche la qualité attribuée parmi les k plus proches voisins
    for elt in tableATester :
      qualite_attribuee = kPlusProchesVoisins(tableDeReference, elt['valeur'], k)
      # print(qualite_attribuee, elt['qualite'])
      # on le compare à la qualité effective de l'élément
      if qualite_attribuee == elt['qualite'] :
        succes +=1 # si ça match, on ajoute 1
    # pour cette valeur de k on calcule le taux de réussite.
    taux = succes / len(tableATester)
    # print('fin')
    # on ajoute ce taux dans le dictionnaire
    resultat[k] = taux
  # enfin on renvoie le dictionnaire construit
  return resultat
```

**ANNEXE 7 : UNE SOLUTION POUR testDesValeursDeK**

Pour avoir essayé avec plusieurs valeurs d $k$ (en abscisses), on remarque que le meilleur taux de réussite est obtenu pour $k = 16$ AVEC CETTE TABLE.