1. La classe majoritaire

1. On utilise un dictionnaire pour compter le nombre d'occurrence de chaque classe avant d'en déterminer la plus grande

```
def majoritaire(L) :
    d = {}
    for elt in L :
        if elt[2] in d :
            d[elt[2]] += 1
        else :
            d[elt[2]] = 1
    Maj = 0
    for c in d :
        if d[c] > Maj :
            classeMaj = c
            Maj = d[c]
    return classeMaj
```

2. On modifie le début de la fonction précédente en prenant comme valeur associée à une clé le poids du point à rajouter

```
def majoritaire(x,dist,L) :
    d = {}
    for elt in L :
        if elt[2] in d :
            d[elt[2]] += 1/dist(x,elt[2])
        else :
            d[elt[2]] = 1/dist(x,elt[2])
    Maj = 0
    for c in d :
        if d[c] > Maj :
            classeMaj = c
            Maj = d[c]
    return classeMaj
```

2. Éviter un tri complet

1.
```
def posMax(x,L,dist) :
    ind = 0
    for k in range(len(L)) :
        if dist(x,L[k]) < dist(x,L[ind]) :
            ind = k
    return ind
```

2. On remplace le point de `L` (les k premiers points de E) le plus loin de `x` s'il est plus loin que le nouveau point de `E` que l'on examine. Il faut faire attention à ne pas garder la classe des points de `E` pour utiliser la fonction précédente.

```
def plusProche(x,E,dist,k) :
    L = E[:k]
    for y in E[:k] :
        L1 = [z[:2] for z in L]
        ind = posMax(x,L1)
        if dist(x,y[:2]) < dist(x,L1[ind]) :
            L[ind] = y      # ici il faut garder les classe
    return L
```

3. La complexité dans le code du cours, dus au tri de la liste des distances entière est en $O(n \ln n)$ si $n =$ `len(E)`. Ici le coût de `posMax` est en $O(k)$, on l'utilise $O(n)$ fois donc on a une complexité en $O(nk)$, ce qui est meilleur si k est petit devant n (ce qui est en général le cas)

3. Choisir k

1. On coupe l'ensemble `E` en deux (parts égales par exemple), on utilise la première partie comme ensemble de référence pour faire les prédictions sur la seconde partie puis on calcule le taux de bonnes réponses en fonction de la valeur de `k`. Pour un choix cohérent, il faut faire ce test plusieurs fois, donc en mélangeant `E` avant pour ne pas faire toujours le même découpage

2.
```
def choixK(E,dist) :
    K = [0] * 10          # on test les valeurs de k impairs de 1 à 21
    n = len(E)
    for _ in range(100) :         # pour faire 100 tests
        random.shuffle(E)         # E est mélangé sur place
        T = E[:n//2]
        for h in K :
            exact = 0
            k = 2*h+1
            for i in range(n//2,n) :
                y = kNN(E[i],T,k)
                if y[2] == E[i][2] :     # bonne prédiction
                    exact += 1
            K[h] += exact
    hMax = 0                      # on cherche la valeur de k qui a donné
        le plus de bonnes prédictions
    for i in range(len(K)) :
        if K[i] > H[hMax] :
            hMax = i
    return 2*hMax+1
```