```python
## Question 1
def hamming(s, t):
    n = len(s)
    d = 0
    for i in range(n):
        if s[i] != t[i]:
            d += 1
    return d

## Question 2
def d(i, j, s, t):
    if i == 0:
        return j
    if j == 0:
        return i
    if s[i] == t[j]:
        return d(i-1, j-1, s, t)
    return min(d(i-1, j-1, s, t), d(i-1, j, s, t), d(i, j-1, s, t)) + 1

## Question 3
Il peut y avoir plusieurs appels récursifs à la fonction d avec les mêmes arguments, ce qui est inefficace et
donne une complexité exponentielle.

## Question 4
def d(s, t):
    memo = {} # dictionnaire pour stocker les valeurs déjà calculées
    def aux(i, j):
        if (i, j) in memo:
            return memo[(i, j)]
        if i == 0:
            return j
        if j == 0:
            return i
        if s[i] == t[j]:
            memo[(i, j)] = aux(i-1, j-1)
        else:
            memo[(i, j)] = min(aux(i-1, j-1), aux(i-1, j), aux(i, j-1)) + 1
        return memo[(i, j)]
    return aux(len(s) - 1, len(t) - 1)

## Question 6
##      Méthode 1 : avec slicing
def split(L):
    n = len(L)
    return L[:n//2], L[n//2:]

##      Méthode 2 :
def split(L):
    n = len(L)
    L1, L2 = [], []
    for i in range(n):
        if i % 2 == 0:
            L1.append(L[i])
        else:
            L2.append(L[i])
    return L1, L2


## Question 7
L'algorithme est censé être utilisé sur de nouvelles données (que l'on ne connaît pas encore). Tester
l'algorithme sur des données qu'il a déjà vues ne permet pas de savoir s'il est efficace sur de nouvelles
données.

## Question 8
def plus_frequent(L):
    d = {}
    for x in L:
        if x in d:
            d[x] += 1
    else:
        d[x] = 1
    kmin, vmin = 0, 0
    for k in d:
        if d[k] > vmin:
            kmin, vmin = k, d[k]
    return kmin

## Question 9
def knn(x, k):
    L = voisins(x, X_train, k)
    return plus_frequent([y_train[i] for i in L])

## Question 10
def precision(k):
    n = len(X_test)
    nb_correct = 0
```

```python
    for i in range(n):
        if knn(X_test[i], k) == y_test[i]:
            nb_correct += 1
    return nb_correct / n

## Question 11
#Graphiquement, la précision maximum semble être 0.76. Donc l'erreur minimum est 1 - 0.76 = 0.24.

## Question 12
def meilleur_k(precisions):
    kmax = 1
    for k in precisions:
        if precisions[k] > precisions[kmax]:
            kmax = k
    return kmax

## Question 13
def bag(s):
    d = {}
        for mot in s.split():
            if mot in d:
                d[mot] += 1
            else:
                d[mot] = 1
    return d

## Question 14
def d_bag(s1, s2):
    d1, d2, d = bag(s1), bag(s2), 0
    for mot in d1:
        if mot in d2:
            d += abs(d1[mot] - d2[mot])
        else:
            d += d1[mot]
    for mot in d2:
        if mot not in d1:
            d += d2[mot]
    return d

## Question 15
def bag_ngram(s, n):
    d = {}
    words = s.split()
    for i in range(len(words) - n + 1):
        ngram = words[i]
        for j in range(1, n):
            ngram += " " + words[i + j]
        if ngram in d:
            d[ngram] += 1
        else:
            d[ngram] = 1
    return d


### Correction de la partie BDD
1) SELECT date,duree,score FROM JOUEURS JOIN PARTIES ON id_j=id_joueur
WHERE nom='Alice' ORDER BY date;
ou
SELECT date,duree,score FROM JOUEURS, PARTIES
WHERE id_j=id_joueur AND nom='Alice' ORDER BY date;

2) SELECT COUNT(*) +1 FROM PARTIES WHERE score>n;

3) SELECT MAX(score) FROM JOUEURS JOIN PARTIES ON id_j=id_joueur
WHERE pays='France';
ou bien :
SELECT MAX(score) FROM JOUEURS, PARTIES
WHERE id_j=id_joueur AND pays='France';

4) SELECT COUNT(*) +1 FROM JOUEURS JOIN PARTIES ON id_j=id_joueur
GROUP BY id_j HAVING MAX(score) >
(SELECT MAX(score) FROM JOUEURS JOIN PARTIES ON id_j=id_joueur WHERE nom='Alice');
ou bien
SELECT COUNT(*) +1 FROM
(SELECT id_j FROM PARTIES GROUP BY id_j HAVING MAX( score ) >
(SELECT MAX(score) FROM JOUEURS JOIN PARTIES ON id_j = id_joueur WHERE nom = 'Alice')
)
```