CAPES MATHS OPTION INFORMATIQUE

ALGORITHMIQUES DE TRI

Test si liste triée

Avant de commencer à écrire des algorithmes de tri, on s'intéresse à tester si une liste est triée (pas besoin de la triée si elle l'est déjà). Donner la fonction Python qui teste si une liste passée en paramètre est triée et donner la complexité de cette fonction.

```
def estTrie(l) :
    for i in range(len(l)-1):
        if l[i] > l[i+1] : return False
    return True
```

Cette fonction parcourt une seule fois chaque élément, elle est de compléxité O(n).

Tri par insertion

Le tri par insertion est l'algorithme utilisé par la plupart des joueurs lorsqu'ils trient leur « main » de cartes à jouer. Le principe consiste à prendre le premier élément du sous-tableau non trié et à l'insérer à sa place dans la partie triée du tableau.

a. Dérouler le tri par insertion du tableau [5.1, 2.4, 4.9, 6.8, 1.1, 3.0].

5.1 2.4 4.9 6.8 1.1 3.0

On commence par essayer de placer le deuxième élément, soit 2.4, dans le sous-tableau allant des cases 1 à 1. Comme 5.1 est supérieur à 2.4, on doit placer 2.4 avant 5.1. On passe ensuite au troisième élément, c'est-à-dire 4.9.

2.4 5.1 4.9 6.8 1.1 3.0

4.9 est inférieur à 5.1 mais supérieur à 2.4, on le place donc entre les deux. On passe ensuite au quatrième élément, soit 6.8.

2.4 4.9 5.1 6.8 1.1 3.0

6.8 ne bouge pas car il est supérieur à 5.1. On passe à 1.1 puis 3.0.

2.4 4.9 5.1 6.8 1.1 3.0

1.1 2.4 4.9 5.1 6.8 3.0

1.1 2.4 3.0 4.9 5.1 6.8

Le tableau est maintenant trié.

Le principe est simple à comprendre, mais la difficulté apparaît lorsqu'on essaie d'écrire l'algorithme avec des données structurées en tableau : il va parfois falloir déplacer plusieurs éléments pour en placer un, car il faut lui « faire de la place ».

b. Ecrire en Python la procédure de tri par insertion, par ordre croissant, d'un tableau de réels :

Procédure tri_par_insertion (tab : tableau de n réels)
Précondition : tab[0], tab[1], … tab[n-1] initialisés
Postcondition : tab[0] ≤ tab[1] ≤ … ≤ tab[n-1]

```
def tri_par_insertion (tab) :
   for j in range(1,len(tab)) :
      elt_a_placer = tab[j]
      i = j - 1
      while i >= 0 and tab[i] > elt_a_placer :
         tab[i+1] = tab[i]
         i = i - 1
      tab[i+1] = elt_a_placer
```

c. Un algorithme de tri est dit « stable » s'il préserve toujours l'ordre initial des ex-aequos. Dans notre exemple, l'algorithme est stable si des valeurs identiques restent dans leur ordre d'apparition avant le tri. L'algorithme de tri par insertion est-il stable ?

Oui, l'algorithme est stable.

d. Donner l'invariant de boucle correspondant à cet algorithme, en démontrant qu'il vérifie bien les 3 propriétés d'un invariant de boucle : initialisation, conservation, terminaison.

Invariant de boucle : Juste avant l'itération j, le sous-tableau tab[0…(j-1)] se compose des éléments qui occupaient initialement les positions tab[0…(j-1)], mais qui sont maintenant triés.

Initialisation : Il faut démontrer que la propriété est vraie juste avant l'itération j=1. Il faut donc montrer que « le sous-tableau tab[0…0] se compose des éléments qui occupaient initialement les positions tab[0…0], mais qui sont maintenant triés. » Avant l'itération j=1, on n'a pas modifié le tableau, donc l'élément tab[0] est bien l'élément tab[0] originel. Par ailleurs, un sous-tableau qui ne contient qu'un seul élément est forcément trié. La propriété est donc vérifiée.

Conservation : Il faut démontrer que si la propriété est vraie juste avant l'itération j, alors elle reste vraie juste avant l'itération j+1. On suppose donc que « le sous-tableau tab[0…(j-1)] se compose des éléments qui occupaient initialement les positions tab[0…(j-1)], mais qui sont maintenant triés », et on doit montrer qu'après avoir exécuté le corps de la boucle Pour, « le sous-tableau tab[0…j] se compose des éléments qui occupaient initialement les positions tab[0…j], mais qui sont maintenant triés. » On sait que l'élément tab[j] va être inséré quelque part entre les positions 0 et j incluses, mais qu'il ne sera en aucun cas placé après la case j. Les éléments des cases 0 à j-1 peuvent être déplacés vers la droite, mais d'un cran seulement : aucun d'entre eux ne se retrouvera après la case j. Donc les éléments situés initialement dans les cases 0 à j restent bien dans ce bloc de cases. Par ailleurs, l'élément j va être bien être inséré de sorte à ce que le sous-tableau tab[0…j] reste trié. On a donc bien conservation de la propriété.

Terminaison : Il faut montrer qu'une fois la boucle terminée, l'invariant de boucle fournit une propriété utile pour montrer la validité de l'algorithme. Ici, la boucle prend fin quand j dépasse n-1, c'est-à-dire pour j=n. On sait alors que « le sous-tableau tab[0…n-1] se compose des éléments qui occupaient initialement les positions tab[0…n-1], mais qui sont maintenant triés. » Le tableau est donc trié, ce qui montre que l'algorithme est correct.

e. Evaluer le nombre d'affectations de réels pour un tableau de taille *n*, dans le cas le plus défavorable (tableau trié dans l'ordre décroissant).

Affectations de réels : Dans le cas le plus défavorable, on fait (j-1) décalages avant d'insérer l'élément.

$A = \sum_{j=1}^{n-1}(1 + (j-1) + 1)$ *(lignes 2, 5 et 8)*

$$A = \sum_{j=1}^{n-1}(1+j) = n-1+\sum_{j=1}^{n-1} j = n-1+\frac{n(n+1)}{n}-1 = \frac{n^2}{2}+\frac{3n}{2}-2$$

Le nombre d'affectations est donc $O(n^2)$.

Tri par sélection

Le principe du tri par sélection est le suivant. Pour chaque élément i de 1 à n-1, on échange tab[i] avec l'élément minimum de tab[i..n]. Nous devons donc rechercher, pour chaque itération le minium d'un sous-tableau de plus en plus petit. Les éléments à gauche de i sont à leur place définitive et donc le tableau est complémentent trié lorsque i arrive sur l'vant dernier élément (le dernier élément étant forcément le plus grand

a. Ecrire en Python la procédure de tri par sélection, par ordre croissant, d'un tableau de réels :

```
def tri_par_selection (tab) :
    for i in range(0,len(tab)-1) :    # de 0 à n-2
        indmin = i
        for j in range(i+1,len(tab)) : # de i+1 à n-1, recherche du min
            if tab[j] < tab[indmin] : indmin = j
        tab[indmin],tab[i]=tab[i],tab[indmin]  # echange de valeurs entre indmin et i
```

b. Donner la complexité de cet algorithme de tri.

On voit que l'on effectue une boucle sur tous les élements de tab sur i (moins le dernier). Et pour chaque itération la recherche du minimum demande, dans le pire des cas, de parcourir tous les éléments du sous-tableau. Nous avons donc une boucle sur i en O(n) et une boucle sur j en O(n), la complexité totale est donc $O(n^2)$.

c. Donner l'invariant de boucle correspondant à cet algorithme, en démontrant qu'il vérifie bien les 3 propriétés d'un invariant de boucle : initialisation, conservation, terminaison.

Invariant de boucle : Juste avant l'itération i, le tableau est trié entre les indices 0 et i-1, et tous les éléments restants sont supérieurs ou égaux à tab[i-1].

Initialisation : Pour i=1 (on vient de faire l'affectation i←1 mais on n'a pas exécuté le corps de la boucle pour i=1), a-t-on la propriété suivante: «tab est trié entre les indices 0 et 0, et tous les éléments restants sont supérieurs ou égaux à tab[0]»? Entre les indices 0 et 0, il n'y a qu'un seul élément donc ce sous-tableau est forcément trié. Par ailleurs, l'élément qui se trouve en position 0 est le min de tout le tableau, donc les éléments restants lui sont donc bien supérieurs ou égaux.

Conservation : Supposons que la propriété soit vraie à l'itération i, soit «tab est trié entre les indices 0 et i-1, et tous les éléments restants sont supérieurs ou égaux à tab[i-1]». On doit montrer que la propriété reste vraie pour i+1, soit «tab est trié entre les indices 0 et i, et tous les éléments restants sont supérieurs ou égaux à tab[i]». Or, lors de l'itération i, on prend un élément dans la partie non triée pour le mettre à la place i, et on ne touche pas aux éléments tab[0...i-1]. Cet élément est supérieur ou égal à tab[i-1] donc le début du tableau restera bien trié, jusqu'à i inclus maintenant. Comme par ailleurs, cet élément est le minimum de la partie non triée, les éléments restants sont bien supérieurs ou égaux à tab[i].

Terminaison : Pour i=n-1 (dernière valeur prise par i, qui va causer la sortie de la boucle), l'invariant de boucle s'écrit «tab est trié entre les indices 1 et n-2, et tous les éléments restants sont supérieurs ou égaux à tab[n-2]». Donc on a un tableau trié jusqu'à n-2, suivi d'une valeur supérieure ou égale à toutes les autres. Le tableau est donc bien trié de 0 à n-1, ce qui prouve que l'algorithme de tri est correct.

Tri à bulles

Le tri à bulles est un algorithme de tri qui s'appuie sur des permutations répétées d'éléments contigus qui ne sont pas dans le bon ordre.

Procédure tri_bulles(tab : tableau [1..n] de réels)
Précondition : tab est un tableau contenant n réels
Postcondition : les éléments de tab sont triés dans l'ordre croissant
Variables locales : i, j : entiers, e : réel
Début

```
1   i ← 1
2   Tant que (i <= n-1) Faire
3       j← n
4       Tant que (j >= i+1) Faire
5           Si tab[j] < tab[j-1] Alors
6               {on permute les deux éléments}
7               e ← tab[j-1]
8               tab[j-1] ← tab[j]
9               tab[j] ← e
10          Fin Si
11          j ← j-1
12      Fin Tant que
13      i ← i + 1
14  Fin Tant que
```

Fin tri_bulles

a. Soit le tableau suivant : {53.8, 26.1, 2.5, 13.6, 8.8, 4.0}. Donnez les premiers états intermédiaires par lesquels passe ce tableau lorsqu'on lui applique la procédure tri_bulles.

Valeur de i	Valeur de j	Etat du tableau juste après que j ait changé de valeur					
		tab[1]	tab[2]	tab[3]	tab[4]	tab[5]	tab[6]
1	6	53.8	26.1	2.5	13.6	8.8	4.0
1	5	53.8	26.1	2.5	13.6	4.0	8.8
1	4	53.8	26.1	2.5	4.0	13.6	8.8
1	3	53.8	26.1	2.5	4.0	13.6	8.8
1	2	53.8	2.5	26.1	4.0	13.6	8.8
1	1	2.5	53.8	26.1	4.0	13.6	8.8
2	6[1]	2.5	53.8	26.1	4.0	13.6	8.8
2	5	2.5	53.8	26.1	4.0	8.8	13.6
2	4	2.5	53.8	26.1	4.0	8.8	13.6
2	3	2.5	53.8	4.0	26.1	8.8	13.6
2	2	2.5	4.0	53.8	26.1	8.8	13.6

b. Complétez la phrase suivante de sorte à ce qu'elle corresponde à l'invariant de boucle du Tant que interne (boucle sur j, lignes 4 à 12 de l'algorithme).

[1] *le tableau est forcément inchangé depuis la ligne précédente car on n'est pas rentré dans le Tant que de la ligne 4 (j était < i+1).*

« Lorsqu'on vient de décrémenter j (ligne 11), le _____ du sous-tableau tab[_____**…** _____] se trouve en position _____. De plus, si j>1, les éléments du sous-tableau tab[_____**…**_____] occupent les mêmes positions qu'avant le démarrage de la boucle sur j. »

« Lorsqu'on vient de décrémenter j (ligne 11), le **minimum** du sous-tableau tab[**j…n**] se trouve en position **j**. De plus, si j>1, les éléments du sous-tableau tab[**1…j-1**] occupent les mêmes positions qu'avant le démarrage de la boucle sur j. »

c. Déduisez-en la propriété que présente le tableau lorsqu'on a terminé cette boucle interne, c'est-à-dire lorsqu'on arrive sur la ligne 13.

La dernière valeur de j testée est j = i. C'est la valeur qui rend le test faux et qui va nous faire sortir du Tant que interne. A ce moment là, notre invariant de boucle nous dit que :
« Le minimum du sous-tableau tab[i…n] se trouve en position i. De plus, si i>1, les éléments du sous-tableau tab[1…i-1] occupent les mêmes positions qu'avant le démarrage de la boucle sur j. »
Ainsi, la boucle sur j n'affecte pas le sous-tableau tab[1…i-1], mais elle modifie le sous-tableau tab[i…n] de sorte à ce que son minimum se trouve en position i.

d. En utilisant la propriété précédente, on peut montrer que l'invariant de boucle du Tant que externe (boucle sur i) est le suivant : « Juste avant d'incrémenter i (ligne 13) : tab est trié entre les indices 1 et i, et tous les éléments restants sont supérieurs ou égaux à tab[i] ». Donnez la première étape de cette démonstration (initialisation). Les étapes de conservation et de terminaison ne sont pas demandées.

Initialisation : il faut vérifier que juste avant le premier passage par la ligne 13, la propriété est vraie. On remplace donc i par 1. Il faut montrer que « tab est trié entre les indices 1 et 1, et tous les éléments restants sont supérieurs ou égaux à tab[1] ».
Première partie : tab est trié entre les indices 1 et 1 car un tableau de taille 1 est forcément trié. Deuxième partie : on utilise la question précédente en remplaçant i par 1. On sait donc que lorsqu'on a terminé la boucle interne, le minimum du sous-tableau [1…n] (donc le minimum de tout le tableau) se trouve en position 1. Par définition du minimum, les éléments occupant les positions 2…n sont supérieurs ou égaux à tab[1].

e. Calculez le nombre total d'affectations de réels réalisées par la procédure tri_bulles lors du tri complet d'un tableau de *n* réels, dans le cas le plus défavorable.

Le cas le plus défavorable est celui où l'on réalise la permutation à chaque passage dans la boucle interne. On a alors 3 affectations de réels par passage. Le nombre total d'affectations de réels est alors :

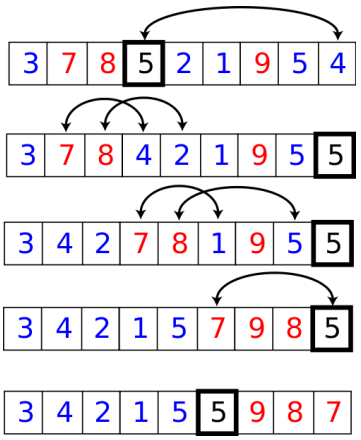
$$A = \sum_{i=1}^{n-1}\left(\sum_{j=i+1}^{n} 3\right) = \sum_{i=1}^{n-1}(3(n-1)) = 3\left(\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i\right) = 3\left(n(n-1) - \frac{(n-1)n}{2}\right) = \frac{3}{2}n(n-1)$$

Tri rapide *(quick sort)*

Cette méthode de tri consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite. Cette opération s'appelle le *partitionnement*. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

Concrètement, pour partitionner un sous-tableau :

- on place le pivot arbitrairement à la fin (peut être fait aléatoirement), en l'échangeant avec le dernier élément du sous-tableau (étape 1 ci-contre)
- on place tous les éléments inférieurs au pivot en début du sous-tableau (étapes 2 et 3)
- on place le pivot à la fin des éléments déplacés (étapes 4 et 5)



Ecrire en Python la procédure de partitionnement et la procédure récurvise de tri rapide.

```
def triRapide (tab) :
   triRapideTableau(tab,0,len(tab)-1)

def triRapideTableau (tab, premier, dernier) :
   if premier < dernier :
      pivot = partitioner(tab,premier,dernier)
      triRapideTableau(tab,premier,pivot-1)
      triRapideTableau(tab,pivot+1,dernier)

def partitioner (tab, premier, dernier) :
   j = premier        # décalage/compteur du premier élément inférieur au pivot
   for i in range(premier,dernier) :
      if tab[i] <= tab[dernier] :
         tab[i],tab[j] = tab[j],tab[i]
         j = j + 1
   tab[dernier],tab[j] = tab[j],tab[dernier]
   return j
```

Tri par fusion interne

a. Ecrire en Python une version **récursive** de l'algorithme du tri par fusion d'un tableau de réels.

```
def fusion(gauche, droite):
    igauche, idroite = 0,0
    result = []
    while igauche < len(gauche) and idroite < len(droite):
        if gauche[igauche] < droite[idroite]:
            result.append(gauche[igauche])
            igauche += 1
        else:
            result.append(droite[idroite])
            idroite += 1

    result += gauche[igauche:]
    result += droite[idroite:]
    return result

def triFusion (tab):
    if len(tab) <= 1 :  return tab
    milieu = len(tab) // 2
    gauche = triFusion(tab[:milieu])
    droite = triFusion(tab[milieu:])
    return fusion(gauche,droite)
```

b. Ecrire en notation algorithmique une version **itérative** de l'algorithme du tri par fusion d'un tableau de réels.

```
Procédure tri_par_fusion (tab : tableau [1...n] de réels)
Précondition : tab[1], tab[2], … tab[n] initialisés
Postcondition : tab[1] ≤ tab[2] ≤ … ≤ tab[n]
Variables locales :
   i, j, k : entiers
   lgMono, nbMono1traites, nbMono2traites : entiers
   debutMono1, debutMono2 : entiers
   tmp: tableau de réels
Début
  lgMono ← 1

  Tant que (lgMono < n)
      tmp ← réserve tableau [1...lgMono] de réels
      debutMono1 ← 1
      debutMono2 ← debutMono1 + lgMono

      Tant que (debutMono2 ≤ n) /* tant qu'on a une paire de monotonies à fusionner */
          {On recopie la premiere monotonie dans tmp. On est sûr qu'elle est complète, vue
la condition de bouclage de ce Tant que}
          k ← debutMono1   {indice dans tab}
          i ←  1                      {indice dans tmp}
          Tant que (k < debutMono1 + lgMono)
              tmp[i] ← tab[k]
              k ← k + 1
              i ← i + 1
          Fin Tant que

          {On fusionne les deux monotonies (attention, la 2ème peut être incomplete)}
          i ← 1                       {indice dans tmp, monotonie 1}
          j ← debutMono2    {indice dans tab, monotonie 2}
          k ← debutMono1    {indice dans tab, monotonie fusionnée}
          nbMono1traites ← 0
          nbMono2traites ← 0
          Tant que ((nbMono1traites < lgMono) et (nbMono2traites < lgMono) et (j ≤ n))
                {tant qu'aucune des deux monotonies n'est epuisée}
                Si (tmp[i] < tab[j]) Alors
                  tab[k] ← tmp[i]
                  nbMono1traites ← nbMono1traites + 1
                  i ← i +1
                Sinon
                  tab[k] ← tab[j]
                  nbMono2traites ← nbMono2traites + 1
                  j ← j + 1
                Fin Si
              k ← k + 1
          Fin Tant que
          Si ((nbMono2traites = lgMono) ou (j > n)) Alors
              {On a épuisé la seconde monotonie, il reste à recopier la première}
              Tant que (nbMono1traites < lgMono)
                  tab[k] ← tmp[i]
                  nbMono1traites ← nbMono1traites + 1
                  i ← i + 1
                  k ← k +1
               Fin Tant que
          Sinon
             {Dans le cas où l'on est sorti du « Tant que » parce que la monotonie 1 est
épuisée, il n'y a rien à faire, les éléments de la monotonie 2 sont déjà en place}
          Fin Si

          {On passe à la paire de monotonies suivante}
```

```
          debutMono1 ← debutMono1 + (2*lgMono);
          debutMono2 ← debutMono2 + (2*lgMono);
        Fin Tant que

      Libère tmp
      lgMono ← lgMono * 2 {pour la prochaine passe sur le tableau, les monotonies seront 2x
plus longues}
    Fin Tant que

Fin tri_par_fusion
```

Tri par comptage

Soit N un entier naturel non nul. On cherche à trier une liste L d'entiers naturels strictement inférieurs à N.

a. Ecrire une fonction comptage, d'arguments L et N, renvoyant une liste P de longueur N dont l'élément d'indice k désigne le nombre d'occurrences de l'entier k dans la liste L.

```
def comptage (L,N) :
   P = [0 for i in range(N)]
   for k in L :
      P[k] += 1                # on incrémente P[k] à chaque occurrence de k
   return P
```

b. Utiliser la liste P pour déduire une fonction triComptage, d'arguments L et N, renvoyant une liste M triée dans l'ordre croissant.

```
def triComptage (L,N) :
   M = []
   P = comptage(L,N)
   for k in range(N) :
      for i in range(P[k]) :
         M.append(k)
   return M
```

c. Tester la fonction triComptage sur une liste de 20 entiers inférieurs ou égaux à 5, choisis aléatoirement.

```
from random import randrange                    # ou randint
L = [randrange(0,6) for j in range(20)]
print(L)
print(triComptage(L,6))
```

On obtient :

```
[3,2,3,0,4,3,2,0,2,4,5,3,5,2,0,2,1,4,1,3]
[0,0,0,1,1,2,2,2,2,2,3,3,3,3,3,4,4,4,5,5]
```

d. Quelle est la complexité temporelle de cet algorithme ? La compararer à la complexité d'un tri par insertion ou d'un tri fusion.

Notons qu'il n'y a pas de comparaisons entre éléments de la liste dans ce tri.
Dans la fonction comptage, on crée une liste de longueur N puis on effectue n passages dans une boucle for, avec une addition/affectation à chaque passage. La complexité est donc N+n.
Dans la fonction triComptage, en plus de l'appel à comptage, on a deux boucles for imbriquées permettant de remplir la liste triée. Le nombre de passages dans ces boucles est égal au nombre n d'éléments de la liste de départ. Donc la complexité du tri est en O(n+N). On a obtenu un tri dont la complexité est linéaire en le nombre d'éléments de la liste !
Les meilleurs tris utilisant des comparaisons d'éléments était en O(n log(n)). Mais le tri comptage a aussi des

inconvénients :
- il ne fonctionne qu'avec des nombres entiers naturels, et nécessite de connaître le max
- si N est grand, l'usage d'une liste auxiliaire de taille N contenant les occurrences peut s'avérer problématique pour la mémoire

Tri stupide

Analyser la complexité du tri suivant :

```
def triStupide (tab) :
   while not estTrie(tab) :
      melangeAleatoire(tab)
```

Où `estTrie` teste si le tableau est trié et `melangeAleatoire` mélange les éléments du tableau de manière aléatoire.

Le pire cas n'est pas borné ! Le temps d'exécution est tout de même supposé fini quelque soit la taille du tableau.

Tri par base (tri radix)

Dans le tri par base, on trie les éléments par clés selon l'ordre lexicographique. Avec des nombres, les éléments sont triés avec les clés définies comme les chiffres qui composent les nombres :

1. Prendre le chiffre le moins significatif comme clé
2. Trier (par comptage) les éléments
3. Répéter avec les chiffres plus significatifs

Exemple :

liste	6**1**	60**1**	11**1**	1**1**	**1**	6**0**	1**6**	**6**	6**6**
chiffre	0	1	2	3	4	5	6	7	8	9
compt. chiffre	1	5	0	0	0	0	3	0	0	0
compt. cumulé	1	6	6	6	6	6	9	9	9	9

liste	**6**0	**6**1	6**0**1	1**1**1	**1**1	**0**1	**1**6	**0**6	**6**6
chiffre	0	1	2	3	4	5	6	7	8	9
compt. chiffre	3	3	0	0	0	0	3	0	0	0
compt. cumulé	3	6	6	6	6	6	9	9	9	9

liste	**6**01	**0**01	**0**06	**1**11	**0**11	**0**16	**0**60	**0**61	**0**66
chiffre	0	1	2	3	4	5	6	7	8	9
compt. chiffre	7	1	0	0	0	0	1	0	0	0
compt. cumulé	7	8	8	8	8	8	9	9	9	9

liste	1	6	11	16	60	61	66	111	601

a. Ecrire en Python une fonction `valeurBase` qui prend en paramètre deux nombres entiers positifs n et i et qui calcule et retourne la valeur du ième chiffre de n en partant des unités. Exemple valeurBase(61,0)=1 et valeurBase(11,2)=0.

```
def valeurBase(n,i) :
    return (n // (10**i)) % 10
```

b. Ecrire la fonction Python de tri par base.

```
def TriRadix (tab) :
    for digit in range(3):                          # suppose que les nombres sont [0,999]
        cpt = [0] * 10                              # init compteur
        temp = [0] * len(tab)                       # liste résultat itération
        for val in tab:                             # parcours de tab
            cpt[valeurBase(val,digit)] += 1         # compteur clé (unité, dizaine, etc.)
        for i in range(1,10):                       # cumul compteur => position
            cpt[i] += cpt[i-1]
        for i in range(len(tab)-1,-1,-1):           # placement position trié par clé
            indTemp = valeurBase(tab[i],digit)      # indice placement
            temp[cpt[indTemp]-1] = tab[i]           # décalage déplacement (cpt != pos)
            cpt[indTemp] -= 1                       # décrément pos prochain même clé
        tab[:] = temp[:]                            # recopie résultat
```

c. Donner la complexité de ce tri.

Dans ce tri, nous faisons une boucle sur le nombre de clés k (c-à-d le nombre max de chiffres dans les nombres présents). Puis pour chaque clé, on fait un tri par comptage en O(n). Donc la complexité est en $O(n \times k)$. Si k est proportionnel à n (généralement le cas), alors la complexité est en O(n log n) car on aura log(n) clés.