DS 2 : Tris
Éléments de correction

N°	Elts de rép.	Pts	Note
00-00	**Titre de l'exo**	0	0
0	éléments de réponse	0	0

01-07	**Algorithmes de Tris**		
1	Un alogorithme de tri consiste à modifier une liste L = [c_0, c_1, c_2, ...] en une liste contenant les mêmes éléments mais ordonné L' = [c_α, c_β, c_γ, ...] avec $c_\alpha < c_\beta < c_\gamma$	1	
2	`fonction_a`, oui, elle trie dans l'ordre décroissant `fonction_b`, oui, elle trie dans l'ordre croissant `fonction_c`, non, le deuxième exemple n'est pas trié comme les deux autres.	1	
3	Le tri A est un tri rapide. Dans la `fonction_1` on reconnaît la commande `p = L[debut]` qui choisit un élément pivot p comme premier élément de la sous liste. La boucle `for` de cette même fonction ré-arrange la liste de manière à ce que tous les éléments de cette liste inférieur au pivot se retrouvent au début et tous les éléments supérieur au pivot se retrouvent après. Le pivot est alors à sa place. La `fonction_2` est récursive, elle possède un cas d'arrêt si la liste à trier est vide. L'appel de la `fonction_1`, lui permet de s'appeler elle même pour deux listes de taille cumulé $n-1$, car le pivot est déjà bien placé. La terminaison est donc assuré. L'appel de la `fonction_1` permet aussi d'assurer la correction de l'algorithme car les éléments à trier avant le pivot sont tous inférieur aux éléments après le pivot.	1	

4	Le tri B est le tri fusion. L'expression diviser pour régner signifie que l'on "divise" le problème en triant séparement chaque moitié de liste, on "règne" en fusionnant les deux sous-listes plus facile à trier séparement.	1	
5	Le tri C est le tri par insertion. Le principe du tri par insertion est de prendre chaque élément de la liste, puis de l'insérer à sa place dans une liste triée. Pour insérer chaque élément de la liste initiale dans la liste déjà triée on le compare successivement aux éléments triés du plus grand au plus petit. Expliquons son fonctionnement à l'aide de l'exemple L = [14 1 4 3]. On prends d'abord le premier élément de la liste : 14 puis on place 14 dans la liste triée donc L' = [14]. On prends ensuite le deuxième élément de la liste : 1 on place 1 à la suite de la liste L' = [14 1], pour placer 1 à sa place on compare 1 avec 14, comme 1 < 14 alors on doit échanger les positions de 1 et 14, on modifie L' pour L' = [1 14] Le troisième élément est 4 on place 4 à la suite de la liste L' = [1 14 4], on compare 4 et 14, comme 4 < 14 alors on doit échanger les positions de 4 et 14, on modifie L' pour L' = [1 4 14], on compare 4 et 1, comme 1 < 4 alors on a rien à faire. Le quatrième élément est 3 on place 3 à la suite de la liste L' = [1 4 14 3], on compare 3 et 14, comme 3 < 14 alors on doit échanger ces deux éléments on a alors L' = [1 4 3 14], on compare 3 et 4, comme 3 < 4 on doit échanger ces deux éléments on a alors L' = [1 3 4 14], on compare 3 et 1, comme 1 < 3 on a rien à faire. On a parcouru tous les éléments de la liste, la liste est triée [1 3 4 14]	1	

6

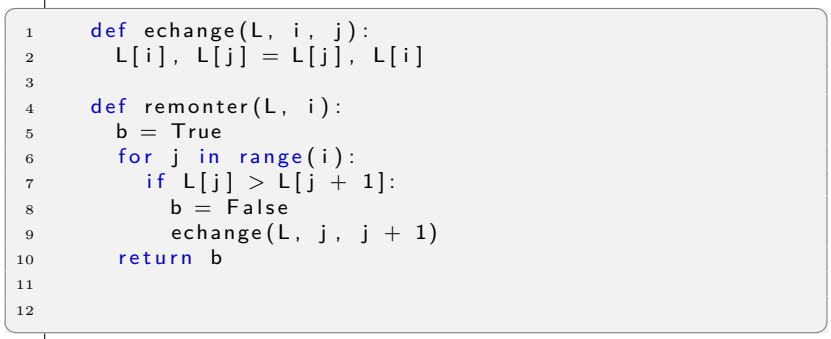
```
def tri_fusion(L):
  if len(L) == 1:
    return L
  else:
    L_1,L_2 = diviser(L)
    L_1 = tri_fusion(L_1)
    L_2 = tri_fusion(L_2)
    return fusion(L_1,L_2)

def diviser(L):
  n = len(L)
  milieu = n//2
  return L[:milieu], L[milieu:]

def fusion(L_1,L_2) :
  i=0
  j=0
  L = []
  while (i<len(L_1))and(j<len(L_2)):
    if L_1[i]<L_2[j] :
      L.append(L_1[i])
      i += 1
    else :
      L.append(L_2[j])
      j +=1
  if i == len(L_1) :
    L += L_2[j:]
  elif j == len(L_2) :
    L += L_1[i:]
  return L
```

1

7	tri A = tri rapide, on remarque que `fonction_2` est appelé dans la définition de `fonction_2`, il est donc récursif. tri B = tri fusion, on remarque que `tri_fusion` est appelé dans la définition de `tri_fusion`, il est donc récursif. tri C = tri par insertion, son principe de fonctionnement demande de parcourir tous les éléments de la liste initiale pour les insérer, cette opération demande une boucle. L'opération d'insertion consiste à comparer l'élément à insérer avec tous les éléments de la liste trié, on utilise aussi une boucle. Il est donc itératif.	1	
08-13	**Performances des algorithmes**		
8	On compare la complexité temporelle et la complexité spatiale des algorithmes. La complexité temporelle correspond au nombre d'opération à effectuer pour trier une liste en fonction de sa taille. Donc au temps mis par un algorithme pour trier une liste en fonction de son nombre d'élément. La complexité spatiale correspond au nombre d'élément à garder en mémoire lors de l'exécution de l'algorithme de tri en fonction du nombre d'élément à trier. Il s'agit donc de l'espace utilisé par l'algorithme sur la mémoire vive de la machine en fonction du nombre d'élément à trier.	1	
9	Lorsque la complexité temporelle dépend de l'ordre initial des éléments, on parle de meilleur et pire des cas. On définit alors une complexité "dans le meilleur des cas", comme la complexité minimale, c'est-à-dire que l'ordre des éléments de la liste initiale permet à l'algorithme d'effectuer un minimum d'opération. On définit également une complexité "dans le pire des cas", comme la complexité maximale, c'est-à-dire que l'ordre des éléments de la liste initiale oblige l'algorithme à effectuer un maximum d'opération.	1	
10	Pour le tri A = tri rapide, le positionnement du pivot dépend de la liste initiale. on se retrouve dans le cas d'une étude de la complexité qui peut varier selon l'ordre initial de la liste. En effet selon le positionnement du pivot la taille des sous listes à trier diffèrent. Le meilleur des cas est une liste pour laquelle le pivot se place à chaque appel récursif au centre de la liste de taille n et la divise en deux listes de taille n/2. Le nombre d'appel récursif est alors minimal. Le pire des cas est une liste pour laquelle le pivot se place en début ou en fin de la liste initiale et la divise en deux listes de taille 0 et n-1. Le nombre d'appel récursif est alors maximal. Pour le tri B = tri fusion, le nombre d'opération à effectuer ne dépend pas de l'ordre initial des éléments, la liste est systématiquement divisé en deux listes de même taille à chaque appel récursif. L'opération de fusion, compare tous les éléments des deux listes. Pour le tri C = tri par insertion, le nombre de comparaison à effectuer lors de l'insertion, dépend de l'ordre des éléments à insérer. Le meilleur des cas est le cas où la liste est déjà triée, en effet lorsque l'algorithme parcoure la liste L il n'a pas besoin de déplacer les éléments car ils sont déjà en place. Le pire des cas se trouve pour une liste dans l'ordre strictement décroissant, en effet l'algorithme doit replacer au début chaque élément de la liste.	1	

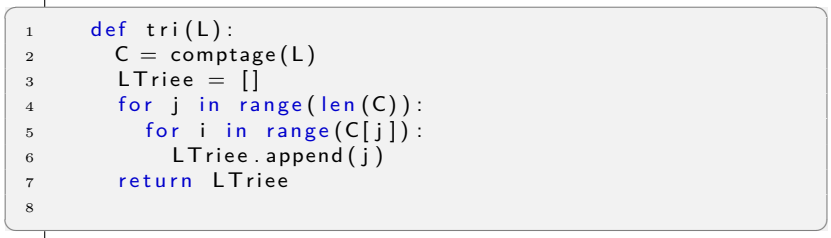
11	Les figures représentent les temps d'execution en fonction de la taille des listes à trier, il s'agit donc de la complexité temporelle. Le tri B = tri fusion ne présente pas de meilleur ni de pire des cas, on a donc une seule courbe. Il s'agit donc de la figure 1. Les figures 2 et 3 présentent toutes les deux des complexités dans le meilleur et le pire des cas qui sont quasi-linéaire et quadratique. Il faut donc comparer les complexité moyenne. Pour le tri A = tri rapide, le pivot se trouve rarement aux extrémités on a donc en moyenne le meilleur des cas. Donc il s'agit de la figure 2. Pour le tri C = tri par insertion, la liste est rarement déjà trié, on a donc un nombre n de comparaison a effectuer, il s'agit donc en moyenne du pire des cas. Donc il s'agit de la figure 3.	1	
12	Un tri en place modifie directement la liste d'entrée L sans utiliser de fonction auxiliaire.	1	
13	Un tri en place permet de ne pas allouer d'espace mémoire supplémentaire. Il permet d'améliorer la complexité spatiale de l'algorithme. Le tri A = tri rapide, si le tri n'est pas en place à chaque appel récursif on créé deux listes, dont la taille cumulé est de n. Or on fait $\log n$ appel, donc la complexité est de $O(n \log n)$ quasi-linéaire. Le tri en place améliore cela pour une complexité linéaire. Le tri B = tri fusion, procède de la même façon, donc le tri en place améliore la complexité. Le tri C = tri par insertion, crée seuleument une seconde liste de taille n. Sa complexité est donc linéaire qu'il soit en place ou pas.	1	
14-17	**Tri à bulles**		
14		1	

```
1   def echange(L, i, j):
2     L[i], L[j] = L[j], L[i]
3
4   def remonter(L, i):
5     b = True
6     for j in range(i):
7       if L[j] > L[j + 1]:
8         b = False
9         echange(L, j, j + 1)
10    return b
11
12
```

15	
	`def tri_bulle(L):`
	`  i = len(L)  1`
	`  b = False`
	`  while not(b) and i>0 :`
	`  b = remonter(L, i)`
	`  i = i moins 1`
	`  return L`
		1	

16	La complexité temporelle de remonter(L,i) est de i, car on fait une opération par étape de boucle et il y a i étape de boucle. Dans le meilleur des cas, quand la liste est déjà trié, on a besoin d'appeler remonter une seule fois avec remonter(L,n-1), donc la complexité temporelle est de n-1 = O(n), c'est une complexité linéaire. Dans le pire des cas, quand la liste est en ordre décroissant, on a besoin d'appeler remonter pour tous les indices i allant de (n - 1) à 1 donc la compléxité temporelle est de (n 1) + (n 2) + ... + 1 = O(n^2), c'est une complexité temporelle quadratique. Le tri à bulles trie la liste en place, il manipule directement la liste en entrée L, l'espace mémoire utilisé est donc de même taille que L donc O(n), c'est une complexité spatiale linéaire.	1	
17	C'est un tri par insertion car on compare successivement chaque élément avec le reste de la liste. Il est donc analogue au tri C.	1	
18-20	**Recherche de la médiane**		

18	Une application du tri d'une liste est la recherche de la médiane de ses éléments. Une première méthode directe serait de prendre après avoir trié la liste l'élément au centre de la liste. En effet on aura alors autant d'élément supérieur que inférieur à celui placé au centre, c'est la définition de la médiane. Un premier exemple simple d'algorithme de recherche de la médiane serait donc le suivant.
	`def mediane_simple(L):`
	`  L = fonction_3(L)`
	`  return L[len(L)//2]`
		1	

19	Cet algorithme profite de la structure du tri en pivot pour accélérer la recherche de la médiane. En effet on a juste besoin de trouver l'élément au centre de la liste triée. Il n'est alors pas nécessaire de trier la liste en totalité. La modification est la suivante : - on prend à nouveau un élément pivot, - on le positionne correctement dans la liste, - si le pivot se trouve au centre de la liste on s'arrête, - sinon on re-applique la même procédure pour la liste qui contient l'élément au centre de la liste. Il s'agira à nouveau d'un algorithme récursif mais avec uniquement un appel récursif dans la fonction, et non deux. Ce qui accélère l'algorithme.	1	
20	Pour le tri B = tri fusion, on est obligé de trier l'ensemble de la liste car seule la dernière étape de fusion ordonne les éléments. De même pour le tri C = tri rapide, on est obligé de trier l'ensemble de la liste car le dernier élément inséré peut déterminer la médiane de la liste.	1	
21-24	**Un autre algorithme de tri**		
21		1	
22		1	

23	Cette fonction de tri a une complexité linaire, car on ne parcours qu'une fois la liste avec la fonction comptage. Cette complexité est meilleure que les trois tris vu en cours. Ce type de tri n'est possible que car on connait à l'avance les éléments présents dans la liste.	1	
24	On peut chercher les mots de la citation en complexité logarithmique à l'aide d'une méthode par dichotomie sur une liste trié de mots. On peut trier les mots en complexité linaire à l'aide du tri précédent, car on connait tous les mots possibles ce sont ceux du dictionnaire.	1	