Chapitre 9 : Quelques méthodes de tris

Tri par sélection	**Tri par insertion**	**Tri fusion**	**Tri comptage**
Principe : Comme une photo de classe, la liste est lue afin de repérer le plus petit élément qui est placé en début de liste.	Principe du jeu de cartes : Pour une sous-liste déjà triée, on compare l'élément à trier (clé) aux valeurs précédentes.	Principe du diviser pour régner : on divise la liste par deux jusqu'à obtenir une liste de longueur unitaire (ou nulle) facile à classer. On utilise une autre fonction pour trier des sous listes triées. Ici tri pas en place et non stable.	Principe : Une liste intermédiaire *listeComptage* de longueur de $\max(L)+1$ est créée. On dénombre alors chaque élément de la liste à trier et on reporte cette valeur dans *listeComptage*. Il est alors facile de proposer une liste triée.

`def selection(L):`
`  for i in range(len(L)-1):`
`  index_min=i`
`  for j in range(i+1,len(L)):`
`  if L[index_min]>L[j]:`
`  index_min=j`
 	
`L[i],L[index_min]=L[index_min],L[i]`
`  return L`
	`def insertion(L):`
	`  for i in range(1,len(L)):`
	`  j=i-1`
	`  cle=L[i]`
	`  while cle<L[j] and j>=0:`
	`  j=j-1`
	`  L[j+2:i+1]=L[j+1:i]`
	`  L[j+1]=cle`
	`  return L`
		`def fusion1(L1,L2) :`
		`  if len(L1)<1 :`
		`  return L2`
		`  elif len(L2)<1:`
		`  return L1`
		`  else :`
		`  if L1[0]<L2[0]:`
		`  return [L1[0]]+fusion1(L1[1:],L2)`
		`  else:`
		`  return [L2[0]]+fusion1(L1,L2[1:])`
		
		`def fusion2(T):`
		`  if len(T)<2:`
		`  return T`
		`  else:`
		`  L1=fusion2(T[:len(T)//2])`
		`  L2=fusion2(T[len(T)//2:])`
		`  return fusion1(L1,L2)`
			`def TriComptage(L):`
			`  maxi = 0#on trouve la valeur max`
			`  for i in L:`
			`  if i>maxi:`
			`  maxi=i`
			`  N = maxi+1`
			`  listeComptage = [0]*(N)`
			`  for i in L:`
			`  listeComptage[i]=listeComptage[i]+1`
			`  liste_new=[]`
			`  for i in range(N):`
			`  for j in range(listeComptage[i]):`
			`  liste_new.append(i)`
			`  return liste_new`
			`rapide(L1)+[L[0]]+rapide(L2)`

Avec l'inégalité stricte, le tri est stable (>) car il ne modifie pas l'ordre initiale de deux valeurs identiques	Avec l'inégalité stricte, le tri est stable (>)	Ce découpage dichotomique n'assure pas la stabilité de l'algorithme	La notion de stabilité n'a pas trop de sens
Le tri est en place car pas de nouvelle liste créée	Le tri est en place car pas de nouvelle liste créée	Ici le tri n'est pas en place car à chaque appel, une nouvelle liste est créée	L'algorithme n'est pas en place : ce qui est retourné est la nouvelle liste *liste_new*
Complexité : Dans tous les cas, le nombre de comparaisons $T(n)$ est : $T(n)=(n-1)+(n-2)+\cdots 1=\frac{n(n-1)}{2}$ $T(n)=O(n^2)$	Complexité (unité de cout est la comparaison) : - Dans le pire des cas (liste inversée) : $T(n)=\frac{n(n-1)}{2}=O(n^2)$ - Dans le meilleur des cas (liste déjà triée) : $T(n)=O(n)$ C'est cette linéarité qui fait l'efficacité de cette méthode pour une liste presque triée.	Complexité : On peut remarquer que l'on a $log_2(n)$ niveaux et sur chaque niveau, on va comparer typiquement n valeurs : $T(n)=nlog_2(n)$ la complexité est quasi-linéaire.	Complexité (nombre d'itérations): - Dans le meilleur cas la liste à triée contient une fois tous les éléments entre 0 et maxi :$T(n)=O(n)$ - Le cas le plus défavorable est lorsque valeur maxi est très éloignée des autres valeurs $T(n)=O(maxi)$