TP 16- Proposition de solutions

Solution 1 Tri par sélection

1. Le script :

Tri par sélection

```
def tri_selection(L):
    n=len(L)
    for i in range(n-1):
        p=pmin(L,i,n)
        if p>i:
            L[i],L[p]=L[p],L[i]
```

2. Suivi de variables :

i	a	b	pmin(L,a,b)	L
X	X	X	X	[3,-1,1,4,1]
0	0	5	1	[-1,3,1,4,1]
1	1	5	2	[-1,1,3,4,1]
2	2	5	4	[-1,1,1,4,3]
3	3	5	4	[-1,1,1,3,4]

3. L'appel de l'instruction `pmin(L,a,b)` requiert $b-a-1$ tests.

4. Le nombres de tests du tri par sélection est

$$\sum_{i=0}^{n-2} n-i-1 =_{k=n-i+1} \sum_{k=1}^{n-1} k \sim \frac{n^2}{2}$$

La complexité est quadratique : $\mathcal{O}(n^2)$

Solution 2 Soit n la taille de la liste. Pour chaque élément, il y a $n-1$ tests. Au total, cela donne $n(n-1)$ tests pour déterminer la position dans la liste triée de chaque élément. La complexité est $\mathcal{O}(n^2)$.

Solution 3 Tri par insertion

1. Application de l'algorithme à `[2,5,0,1,4]` :

i	état de la liste
1	[2,5,0,1,4]
2	[0,2,5,1,4]
3	[0,1,2,5,4]
4	[0,1,2,4,5]

ou

i	j	aux	L
1	1	5	
			[2,5,0,1,4]
2	2	0	
	1		[2,5,5,1,4]
	0		[2,2,5,1,4]
			[0,2,5,1,4]
3	3	1	
	2		[0,2,5,5,4]
	1		[0,2,2,5,4]
	X	X	X
			[0,1,2,5,4]
4	4	4	
	3		[0,1,2,5,5]
			[0,1,2,4,5]

2. Description :

- à la ième étape, on considère que les i premiers éléments sont triés
- on considère l'élément suivant, le $i+1$-ième
- on le compare de proche en proche aux éléments qui le précèdent
- dès que l'on rencontre un élément qui lui est inférieur on s'arrête et l'insère.

En pratique, on décale les éléments les uns après les autres. La place est ainsi préparée lorsque la boucle s'arrête.

Méthode

➜ Le tri par **insertion** consiste à :
- parcourir la liste de gauche à droite ;
- à chaque étape, l'élément considéré, est classé parmi les éléments qui le précèdent (et donc qui sont déjà ordonnés).

3. Étude de la complexité temporelle : soit n la taille de la liste. Dans la configuration la pire (une liste décroissante), le nombre de test est :
- 1 test pour le 2nd élément
- 2 tests pour le 3ième élément
- ...
- $n-1$ tests pour le dernier élément

Le nombre de tests est : $1+2+3+\ldots+n-1 = \frac{n(n-1)}{2} \sim \frac{n^2}{2}$

La complexité de ce tri est $\mathcal{O}(n^2)$.

4. On peut envisager d'utiliser la dichotomie, dans la phase d'insertion d'un élément dans la sous-liste ordonnée le précédent. La complexité devient :

$$\sum_{k=1}^{n-1} \left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil \sim \frac{n\ln(n)}{\ln(2)}$$

C'est-à-dire : $\mathcal{O}(n\ln(n))$.

5. Un script est :

```
def tri_insertion_dicho(L):
    for i in range(1,len(L)):
        d,f=1,i
        if L[i]<=L[0]:
            v=L[i]
            L[1:i+1]=L[:i]
            L[0]=v
            continue
        elif L[i]>=L[i-1]: continue
        while d<f-1 :
            c=(f+d)//2
            if L[c]>L[i]: f=c-1
            else: d=c
        v=L[i]
        if L[f]<=L[i]:
            L[f+2:i+1]=L[f+1:i]
            L[f+1]=v
        else:
            L[f+1:i+1]=L[f:i]
            L[f]=v
```

Solution 4 Tri rapide

1. Suite des appels récursifs et les étapes de la remontée :

Profondeur	Liste
0	[2,-1,3,4,0,3,1,-2,-1]
1	[-1,0,1,-2,-1],[2],[3,4,3]
2	[-2,-1],[-1],[0,1],[2],[3],[3],[4]
3	[-2],[-1],[-1],[0],[1],[2],[3],[3],[4]
2	[-2,-1],[-1],[0,1],[2],[3],[3],[4]
1	[-2,-1,-1,0,1],[2],[3,3,4]
0	[-2,-1,-1,0,1,2,3,3,4]

2. Script basique :

tri_rapide.py

```
def tri_rapide(L):
    if len(L)<2:
        return(L)
    Li,Ls=[],[]
    for e in L[1:]:
        if e<=L[0]:
            Li.append(e)
        else:
            Ls.append(e)
    return(tri_rapide(Li)+[L[0]]+tri_rapide(Ls))
```

3. Étude de la complexité temporelle : soit n la taille de la liste.

- Dans la cas le meilleur,

➢ la découpe donne deux sous-listes de tailles comparables ;

➢ la profondeur de récursivité est k tel que

$$\min\left\{k\in\mathbb{N}^*; 2^k\geq n\right\}$$

ce qui donne $\log_2(n)$ étapes ;

➢ à chaque niveau de profondeur, les sous-listes formant une partition de la liste de départ, il y a au maximum n comparaisons.

La complexité est donc $\Omega(n\ln(n))$.

- Dans le cas le pire,

➢ la découpe donne une sous-liste vide ;

➢ la profondeur de récursivité est donc de $n-1$

➢ au niveau i de profondeur, la taille de la liste à traiter est $n-i-1$ ce qui donne autant de comparaisons.

Or $\sum_{i=1}^{n-1}(n-i-1)\sim\frac{n^2}{2}$; la complexité est donc $\mathcal{O}(n^2)$.

Solution 5 Fonction `fusion(L1,L2)`

Version itérative

```
def fusion(L1,L2):
    L=[]
    while len(L1)*len(L2)>=1:
        if L1[0]<L2[0]:
            L.append(L1.pop(0))
        else:
            L.append(L2.pop(0))
    return(L+L1+L2)
```

Version récursive

```
def fusion(L1,L2):
    if len(L1)==0:
        return(L2)
    if len(L2)==0:
        return(L1)
    if L1[0]<=L2[0]:
        return([L1[0]]+fusion(L1[1:],L2))
    else:
        return([L2[0]]+fusion(L1,L2[1:]))
```

Solution 6 Tri fusion

1. La suite des étapes de découpe, puis celles de fusions :

Profondeur	Liste
0	[2,-1,3,4,0,3,1,-2,-1]
1	[2,-1,3,4],[0,3,1,-2,-1]
2	[2,-1],[3,4],[0,3],[1,-2,-1]
3	[2],[-1],[3],[4],[0],[3],[1],[-2,-1]
4	[2],[-1],[3],[4],[0],[3],[1],[-2],[-1]
3	[2],[-1],[3],[4],[0],[3],[1],[-2,-1]
2	[-1,2],[3,4],[0,3],[-2,-1,1]
1	[-1,2,3,4],[-2,-1,0,1,3]
0	[-2,-1,-1,0,1,2,3,3,4]

2. La fonction `tri_fusion` :

Tri fusion

```
def tri_fusion(L):
 if len(L)<2:
     return(L)
 k=len(L)//2
 return(fusion(tri_fusion(L[:k]),tri_fusion(L[k:])))
```

3. Étude de la complexité temporelle : soit n la taille de la liste.

- La découpe par dichotomie donne que la profondeur des appels récursifs est $\left\lfloor\frac{\ln n}{\ln 2}\right\rfloor$;
- La complexité de fusion est linéaire en la taille des listes entrées. Or, lors des appels d'un même niveau, le travail se fait sur une partition de la liste de départ et donc on peut estimer que le travail reste linéaire en n.

Conclusion, la complexité temporelle du tri fusion est toujours la même (dans le pire comme dans le meilleure des cas) :

$$\mathcal{O}(n\ln n)$$

Solution 7 Tri à bulles

1. Compléter le tableau décrivant chaque échange d'éléments du tri :

Parcours	Liste
1	[3,1,0,8,5,2,2,2]
	[1,3,0,8,5,2,2,2]
	[1,0,3,8,5,2,2,2]
	[1,0,3,5,8,2,2,2]
	[1,0,3,5,2,8,2,2]
	[1,0,3,5,2,2,8,2]
	[1,0,3,5,2,2,2,8]
2	[0,1,3,5,2,2,2,8]
	[0,1,3,2,5,2,2,8]
	[0,1,3,2,2,5,2,8]
	[0,1,3,2,2,2,5,8]
3	[0,1,2,3,2,2,5,8]
	[0,1,2,2,3,2,5,8]
	[0,1,2,2,2,3,5,8]
4	[0,1,2,2,2,3,5,8]

2. La fonction `tri_bulles(L)` :

```
1 def tri_bulles(L):
2     n=len(L)
3     test=True
4     while test:
5         test=False
6         for i in range(n-1):
7             if L[i]>L[i+1]:
8                 L[i],L[i+1],test=L[i+1],L[i],True
9     return(L)
```

La variable `test` est de type booléenne, elle détermine si on continue le parcours ou non, c'est-à-dire si la liste est triée ou non : `True` pour *continuer le tri* et `False` pour *arrêter le tri* :

- si `test` contient `True`, alors soit c'est le début du premier tour, soit la liste à subit une modification : il faut parcours la liste pour vérifier si elle est triée ;
- si `test` contient `False` la liste est triée.

Étude de la complexité temporelle : soit n la taille de la liste.

- La configuration la pire est lorsque la liste est triée dans l'ordre inverse. Notant n sa taille, il y a n parcours de $n-1$ tests soit $\dfrac{n(n-1)}{2}$ comparaisons.
- La configuration la meilleure est quand la liste est déjà triée, un simple parcours permet de le vérifier : soit $n-1$ comparaisons.

La complexité du tri à bulles est $\mathcal{O}(n^2)$