Travaux Pratiques 02 correction

---

1 Différentes méthodes

1.1 Tri par insertion

1.1.1. On obtient `[4]`, `[4,7]`, `[4,6,7]`, `[1,4,6,7]`, `[1,3,4,6,7]`.

1.1.2.
```
def insertion (T:list,x:float):
    '''insert l'élément x dans la liste T
    en conservant l'ordre croissant'''
    k=0
    while k<len(T) and T[k]<x:
            k+=1
    return(T[:k]+[x]+T[k:])
```

1.1.3. `insertion([1,3,7],2)` renvoie `[1, 2, 3, 7]`

1.1.4.
- `insertion([],1)` renvoie `[1]`
- `insertion([1,2,3],4)` renvoie `[1, 2, 3, 4]`
- `insertion([1,2,3],0)` renvoie `[0,1, 2, 3]`

1.1.5.
```
def tri_insertion(L:list):
    '''Trie par insertion la liste L'''
    T=[]
    for l in L:#on insère dans la liste T
        T=insertion(T,l)#Chaque élément de L en conservant l'ordre
    return(T)
```

1.1.6. `tri_insertion([4,7,6,1,3])` renvoie `[1, 3, 4, 6, 7]`.

1.2 Tri par sélection

1.2.1. On obtient :

(a) `[4,7,6,1,3]`
(b) `[1,7,6,4,3]`
(c) `[1,3,6,4,7]`
(d) `[1,3,4,6,7]`
(e) `[1,3,4,6,7]`

1.2.2.
```
def mini(L:list):
    '''->int| Détermine la position du minimum de L'''
    pos=0#position du minimum
    val=L[0]#valeur du minimum
    for k in range(len(L)):#On parcourt L
        if L[k]<val:#Si on trouve plus petit
            #On met à jour le minimum
            pos=k
            val=L[k]
    return(pos)
```

Ainsi `posmini([4,7,6,1,3])` renvoie 3.

1.2.3. Compléter le programme du tri par sélection :

```
def selection (L):
        for k in range(...):
                pos=mini(L[k:])+k
                #On rajoute k pour prendre en compte
                #le décalage d'indice dans la liste extraite.
                L[k],L[...]=
```

1.2.4. Le programme ne renvoie rien mais modifie la liste `L` en `[1,3,4,6,7]`.

2 Des tris récursifs

2.1 Tri fusion

2.1.1.
```
def fusion (L1,L2):
##    fusion fusionne les listes (triées) L1 et L2 en une liste triées L3.
##    Entrées : L1,L2 deux listes triées
##    Sorties : L3 une liste triée
#############################################
    L3=[]#on commence avec la liste vide.
    n1,n2=len(L1),len(L2)
    i,j=0,0#compteur de position dans la liste L1,L2
    while i<n1 and j<n2:# On parcourt les listes L1 et L2
        #On rajoute la plus petite valeur entre L1 et L2 dans L3 et on
        #met à jour le compteur correspondant
        if L1[i]<=L2[j]:
            L3.append(L1[i])
            i+=1
        else:
            L3.append(L2[j])
            j+=1
    #Une des listes est entièrement inséré dans L3 on rajoute maintenant
    #ce qui reste de l'autre liste
    if i<n1:
        return(L3+L1[i:])
    else:
        return(L3+L2[j:])
```

2.1.2.
- `fusion([1,4,7],[2,5,8,10])` renvoie `[1,2,4,5,7,8,10]`.
- `fusion([1,4,7,11],[2,3,6])` renvoie `[1,2,3,4,6,7,11]`.

2.1.3.
```
def tri_fusion(L):
##    Tri par ordre croissant la liste L en opérant un tri fusion. On va
##    trier chaque moitié du tableau et les fusionner ensuite.
##    Entrée : L une liste
##    Sortie : une liste
########################################
    n=len(L)
    if n==0:
        return([])
    elif n==1:
        return(L)
    else:
        return(fusion(tri_fusion(L[:(n//2)]),tri_fusion(L[(n//2):])))
```

L'exécution de `trifusion([2,4,1,5,7])` renvoie `[1,2,4,5,7]`.

2.2 Tri rapide

2.2.1.
```python
def Pivot(L:list,x:float):
    '''->(list,list) | Sépare la liste L en deux sous-listes Lp,Lg contenant
    respectivement les élements plus petit (plus grand) que x.
    '''
    Lp,Lg=[],[]
    for l in L:#On parcourt la liste L
        if l<x:#On teste la position par rapport au pivot
        #puis on ajoute dans la bonne liste
            Lp.append(l)
        else:
            Lg.append(l)
    return(Lp,Lg)
```

2.2.2. On obtient les résultats suivants :

- `Pivot(L,4)` renvoie `([3, 1, 0], [4, 6, 10])`
- `Pivot(L,-1)` renvoie `([], [3, 4, 6, 1, 0, 10])`
- `Pivot(L,11)` renvoie `([3, 4, 6, 1, 0, 10], [])`

2.2.3.
```python
def trirapide(L:list):
    '''->list|Tri la liste par tri rapide
    '''
    if len(L)<= 1:
        return(L)
    else:
        Lp,Lg=Pivot(L[1:],L[0]) #On coupe la liste par rapport à son
        #premier élément
        Lp=trirapide(Lp)#On trie les sous-listes
        Lg=trirapide(Lg)
        return(Lp+[L[0]]+Lg)#On retourne la liste finale
```

2.2.4. Le programme renvoie la liste `[-1, 1, 4, 5, 8]`.

3 Comparaison de rapidité d'exécution

3.0.1. Compléter les lignes suivantes pour obtenir une liste contenant les temps d'exécutions du tri par insertion sur des listes contenant les nombres de 1 à n.

```python
import time
import random as rd

ListeTempsI=[]
for k in range(1,1000):
    L=[i for i in range(k)]# Création de la liste
    rd.shuffle(L)# Mélange
    t1=time.time()# Temps initial
    tri_insertion(L)# Tri
    t2=time.time()#Temps final
    ListeTempsI.append(t2-t1)# Ajout du temps passé pour trier
```

3.0.2.
```python
ListeTempsS=[]
for k in range(1,1000):
    L=[i for i in range(k)]# Création de la liste
    rd.shuffle(L)# Mélange
    t1=time.time()# Temps initial
    selection(L)# Tri
```

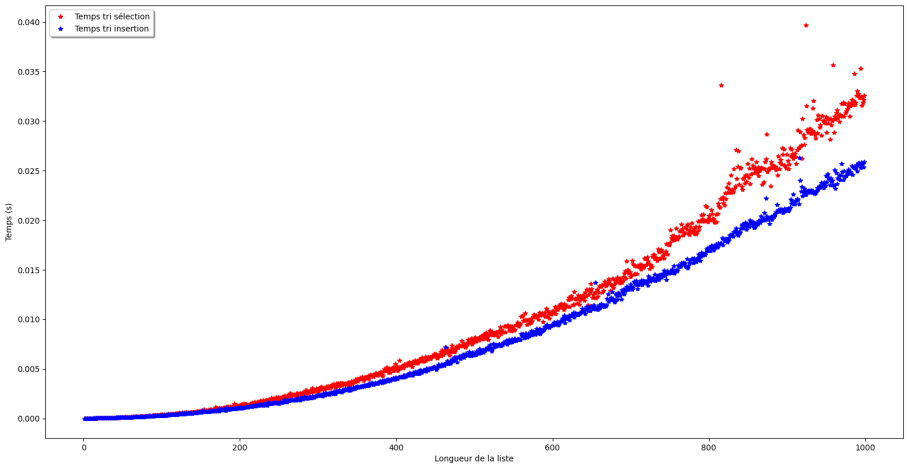

FIGURE 1 – Temps d'exécution tri sélection et tri insertion

```
7       t2=time.time()#Temps final
8       ListeTempsS.append(t2-t1)# Ajout du temps passé pour trier
```

3.0.3.

```
import matplotlib.pyplot as plt

N=[k for k in range(1,1000)]

plt.figure('Comparaison tri sélection vs insertion')
plt.plot(N,ListeTempsI,'r*',label='Temps tri sélection')
plt.plot(N,ListeTempsS,'b*',label='Temps tri insertion')

plt.suptitle('')
plt.xlabel('Longueur de la liste')
plt.ylabel('Temps (s)')
legend = plt.legend(loc='upper left', shadow=True, fontsize='medium')

plt.show()
```

On obtient le graphique affiché en figure 1.

On remarque que les temps d'exécution des deux programmes sont relativement similaire avec un léger avantage de vitesse pour le tri par insertion.

3.0.4. Comme dans la partie 3, on commence par récupérer les temps d'exécution du tri fusion sur les listes contenant les entiers de 1 à n.

```
ListeTempsF=[]
for k in range(1,1000):
    L=[i for i in range(k)]# Création de la liste
    rd.shuffle(L)# Mélange
    t1=time.time()# Temps initial
    tri_fusion(L)# Tri
    t2=time.time()#Temps final
    ListeTempsF.append(t2-t1)
```

Puis on recréer un graphique contenant les temps d'exécutions des trois programmes de tri :

```
N=[k for k in range(1,1000)]

```

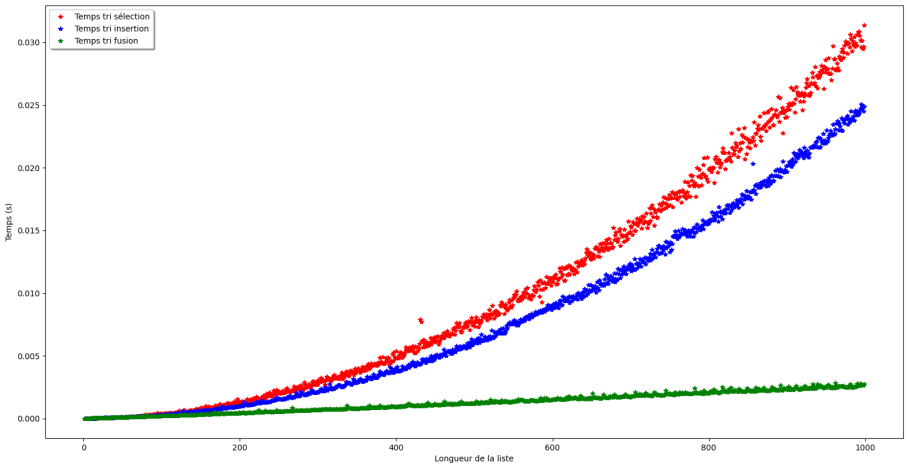

FIGURE 2 – Temps d'exécution tri sélection, tri insertion, tri fusion

```
plt.figure('Comparaison tri sélection, insertion, fusion')
plt.plot(N,ListeTempsI,'r*',label='Temps tri sélection')
plt.plot(N,ListeTempsS,'b*',label='Temps tri insertion')
plt.plot(N,ListeTempsF,'g*',label='Temps tri fusion')

plt.suptitle('')
plt.xlabel('Longueur de la liste')
plt.ylabel('Temps (s)')
legend = plt.legend(loc='upper left', shadow=True, fontsize='medium')

plt.show()
```

On obtient ainsi le graphique 2.

On remarque ainsi que le tri fusion est beaucoup plus rapide que les deux tris précédents.

3.0.4. De la même manière que précédemment on récupère les temps d'exécutions du tri rapide

```
ListeTempsR=[]
for k in range(1,1000):
    L=[i for i in range(k)]# Création de la liste
    rd.shuffle(L)# Mélange
    t1=time.time()# Temps initial
    tri_fusion(L)# Tri
    t2=time.time()#Temps final
    ListeTempsR.append(t2-t1)
```

Puis on affiche le graphique

```
N=[k for k in range(1,1000)]

plt.figure('Comparaison tri rapide, fusion')
plt.plot(N,ListeTempsR,'r*',label='Temps tri rapide')
plt.plot(N,ListeTempsF,'g*',label='Temps tri fusion')

plt.suptitle('')
plt.xlabel('Longueur de la liste')
plt.ylabel('Temps (s)')
```

```
10 legend = plt.legend(loc='upper left', shadow=True, fontsize='medium')
11
12 plt.show()
```

On obtient ainsi le graphique 3 qui montre que le tri rapide est, comme son nom l'indique, plus rapide que les autres tri présentés ici.

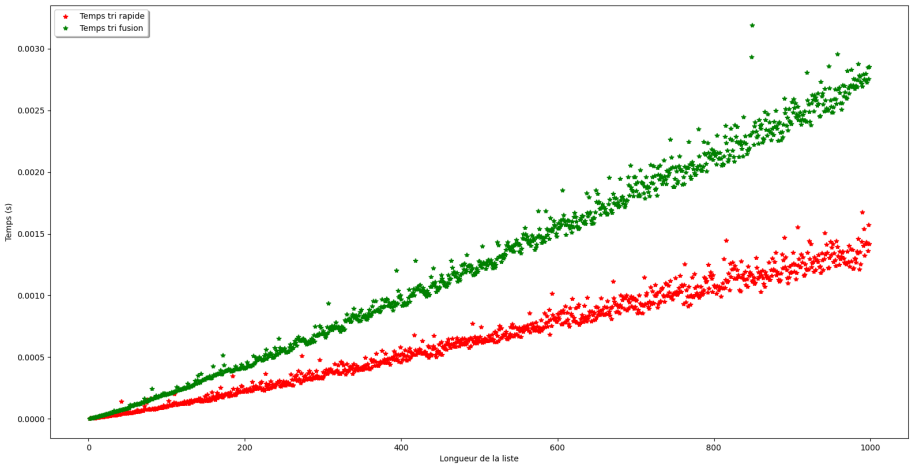

FIGURE 3 – Temps d'exécution tri rapide, tri fusion