TP 09
Tris rapides

Le but de ce T.P. est d'étudier des tris d'une compléxité en $O(n.\ln(n))$ alors que les tris par insertion ou par sélection du TP précédent sont de complexité $O(n^2)$, n désignant le nombre d'éléments dans la liste, ces tris sont donc ... plus rapides ! On utilisera le plus souvent des programmes récursifs.

On commence par écrire la fonction `PlusGrand(nb1,nb2)` qui renvoie le booléen traduisant $nb_1 > nb_2$ puis la procédure `Echanger(Liste,i,j)` qui échange les valeurs de `Liste[i]` et de `Liste[j]`.

On en profite pour faire des fonctions tests, si l'éxéction du programme `TestPlusGrand()` ci-contre ne renvoie pas un message d'erreur, il y a des chances que votre programme `PlusGrand` soit correct.

Proposer une fonction `TestEchanger()`.

```
def TestPlusGrand() :
    assert PlusGrand(3,1)
    assert not PlusGrand(3,3)
    assert not PlusGrand(3,5)
TestPlusGrand()
```

I Tri fusion

1) La fusion.

Si on se donne deux listes triées (ordre croissant), on peut les fusionner en une seule liste triée (ordre croissant), de façon simple : on prend le plus petit des premiers termes des deux listes et on recommence (récursivité).

Cela se traduit par concrètement par :

```
FusionRec([0,2,4],[1,3,5])  renverra à  [0]+FusionRec([2,4],[1,3,5])
 FusionRec([2,4],[1,3,5])   renverra à   [1]+FusionRec([2,4],[3,5])
  FusionRec([2,4],[3,5])        ...
```

Si l'une des deux listes est vide, ce sera la condition d'arrêt, le programme récursif donnera la concaténation des deux listes.

Compléter en conséquence la version récursive :

```
def FusionRec(Liste1,Liste2) :
    if ..... : #cas d'arrêt
        return Liste1[ :]+Liste2[ :]
    if PlusGrand(Liste2[0],Liste1[0]) :
        return [Liste1[0]] + FusionRec(.....)
    return .....
```

On pourra tester le programme en fusionnant la liste des dix premiers entiers impairs avec celles des dix premiers entiers pairs. Donner une fonction test pour ce programme avec, par exemple, ces deux listes.

Si on note n le nombre d'éléments, on fait exactement $n-1$ comparaisons (i.e. appels à `PlusGrand`) donc la complexité de ce programme est linéaire, i.e. en $O(n)$.

2) Le principe du tri fusion.

On prend une liste non triée, on la coupe en deux, on trie ces deux sous-listes (récursivité) et on les fusionne.

```
def TriFusion(L) :
    n = len(L)
    if n <= 1 :
        return L[ :]
    else :
        return FusionRec(TriFusion(L[ :n//2],TriFusion(L[n//2 :])
```

Le cas $n \leq 1$ est la condition d'arrêt du programme récursif, d'autre part, le programme renvoie une nouvelle liste, c'est une version externe et non en place, i.e. qui modifie la liste sans faire de copie.

Cela se traduit concrètement par :

```
TriFusion([7,8,3,5,6,3])  renverra à  FusionRec(TriFusion([7,8,3]),TriFusion([5,6,3]))
      TriFusion([7,8,3])  renverra à  FusionRec(TriFusion([7]),TriFusion([8,3]))
        TriFusion([8,3])  renverra à  FusionRec(TriFusion([8]),TriFusion([3]))
```

Comme `TriFusion([x])` renvoit `[x]`, on dépile alors les commandes :

```
                TriFusion([8,3])  renvoit  FusionRec([8],[3])=[3,8]
              TriFusion([7,8,3])  renvoit  FusionRec([7],[3,8])=[3,7,8]
              TriFusion([5,6,3])  renvoit  FusionRec(...)=[3,5,6]
TriFusion([7,8,3,5,6,3])  renvoit  FusionRec([3,7,8],[3̃,5,6])=[3̃,3,5,6,7,8]
```

On veut définir une fonction `TestTriFusion(Liste)` qui prend en argument une liste non triée et vérifie que l'on trouve une liste triée.

Pour cela, on va écrire une fonction `EstTriee(Liste)` qui prend en argument une liste et qui renvoie `True` si la liste est triée par ordre croissante, `False` sinon. Le programme ci-contre est faux, le corriger.

```
def EstTriee(Liste) :
    for i in range(1,len(Liste)) :
        if PlusGrand(Liste[i-1],L[i]) :
            return False
    return True
```

On pourra pousser le vice en écrivant une fonction `TestTriFusionAlea()` qui teste avec une liste créée aléaoirement.

3) La complexité

On part du prinicipe que la compléxité est croissante sur le nombre de termes de la liste à trier, on remarque ensuite que pour un entier non nul n, il existe un unique entier p tel que $2^p \leq n < 2^{p+1}$, on montre facilement que $p = \log_2(n) = \left\lfloor \dfrac{\ln(n)}{\ln(2)} \right\rfloor$, on peut montrer que $O(p) = O(\ln(n))$.

Si on note $C(n)$ est la compléxité pour une liste de taille n, on a $C(2) = 1$ et on a donc $C(n) \leq C(2^{p+1})$. On pose alors $u_p = C(2^p)$, on a $u_1 = 1$

Expliquer pourquoi on a pour $p \geq 2$: $u_p = 2u_{p-1} + 2^p - 1$.

On pose la suite (α_p) telle que $\forall p \in \mathbb{N}^*$, $u_p = \alpha_p.2^p$, on a $\alpha_1 = \frac{1}{2}$, la relation donne alors

$$\alpha_p.2^p = \alpha_{p-1}.2^p + 2^p - 1 \leq \alpha_{p-1}.2^p + 2^p$$

ce qui nous permet de conclure que $\alpha_p - \alpha_{p-1} \leq 1$ et donc

$$\alpha_p = \alpha_1 + \sum_{k=2}^{p} \alpha_k - \alpha_{k-1} \leq \frac{1}{2} + \sum_{k=2}^{p} 1 = p - \frac{1}{2} \leq p$$

On en déduit donc $u_p \leq p.2^p$ et donc $C(n) \leq u_{p+1} \leq (p+1)\,2^{p+1}$, par conséquent

$$C(n) = O(2(p+1)2^p) = O(p.2^p) = O(\ln(n).n)$$

4) Version itérative de la fusion

On va s'intéresser maintenant à une version itérative externe du programme fusion. Tout d'abord la boucle de variable i fera n_1+n_2 tours (n_1 et n_2 étant la longueur des listes). Ensuite, on utilise deux variables auxiliaires qui donne les indices des termes à comparer, concrètement : pour `Liste1=[0,2]` et `Liste2=[1,3,5]`, avec `res` la liste que donnera la fusion, on a

i	`PlusGrand(Liste2[pos2],Liste1[pos1])`	`res`	`pos1`	`pos2`	
		`[ ]`	0	0	
0	True	[0]	1	0	(1)
1	False	[0,1]	1	1	(2)
2	True	[0,1,2]	2	1	(3)
3		[0,1,2,3]	2	2	
4		[0,1,2,3,5]	2	3	

(1) : 0 =`Liste1[pos1]`<`Liste2[pos2]`= 1, on ajoute donc 0 à `res` et on incrémente donc `pos1`
(2) : 1 =`Liste2[pos2]`<=`Liste1[pos1]`= 2, on ajoute donc 1 à `res` et on incrémente donc `pos2`
(3) : on a `pos1` qui vaut la longueur de la liste `Liste1`, donc on ne compare plus `Liste1[pos1]` et `Liste2[pos2]`, il suffit d'écluser la liste `Liste2`.

Pour écrire le programme, dans la boucle, on commencera par tester si l'on a déjà parcouru totalement une des deux listes sinon, on comparera `Liste1[pos1]` et `Liste2[pos2]`.
Compléter le script ci-contre :
Pour aller plus loin, on peut chercher à un invariant de boucle de ce programme.
Vérifier que $\mathcal{P}(i)$ ci-dessous en est un :

$\mathcal{P}(i)$ {
(1) `pos1<=n1`
(2) `pos2<=n2`
(3) `pos1+pos2=i`
(4) `res+Liste1[pos1 :]` est triée
(5) `res+Liste2[pos2 :]` est triée
(6) `res+Liste1[pos1 :]+Liste2[pos2 :]` contient les mêmes éléments que `Liste1+Liste2`
}

```
def FusionIter(Liste1,Liste2) :
    n1,n2,res,pos1,pos2 =.............
    for i in range(n1+n2) :
        if pos1==n1 :
            res.append(Liste2[....
            pos.....
        elif pos2==n2 :
            .....
            .....
        elif PlusGrand(Liste2[......
            res.append(Liste......
            pos.....
        else :
            ..........
            ..........
    return res
```

Pour s'amuser, vérifier que le programme suivant est un autre script possible :

```
def FusionIter2(Liste1,Liste2) :
    n1,n2,res,pos1,pos2 =len(Liste1),len(Liste2),[ ],0,0
    for i in range(n1+n2) :
        if pos1==n1 or pos2 == n2 :
            return res+Liste1[pos1 :]+Liste2[pos2 :]
        bool = PlusGrand(Liste2[pos2],Liste1[pos1])
        res = res + bool*[Liste1[pos1]]+(1-bool)*[Liste2[pos2]]
        pos1,pos2 = pos1+bool,pos2+(1-bool)
```

5) Version en place

Pour ceux qui sont plus à l'aise, reprendre ce tri avec des versions en place et non externe.
On peut également comparer le temps mis pour les tris quadratiques et le tri fusion...

II Tri rapide

1) Version externe.

On commence par une version externe, c'est-à-dire que l'on va écrire une fonction récursive `TriRapide(Liste)` qui prend en entrée une liste non triée et qui retourne une nouvelle liste triée, contenant les mêmes éléments.

La condition d'arrêt sera une longueur de la liste inférieure ou égale à un, la fonction retournera la liste telle quelle, cela permet de prendre en entrée la liste vide.

Ensuite, on choisit un élément de la liste, le premier par exemple qui sera le pivot, on considère alors les éléments du reste de la liste en les comparant avec la valeur pivot, les ajoutant ainsi à deux nouvelles listes, celle des éléments plus petits nommée `ListeDesPlusPetits` et celle des éléments plus grand, nommée ...

Par exemple, si l'on prend `L = [7,8,5,6,3,9]`, `pivot=7` et on trouvera `ListeDesPlusPetits=[5,6,3]` et `ListeDesPlusGrands=[8,9]`. A noter que la liste `ListeDesPlusPetits+[pivot]+ListeDesPlusGrands` contient les mêmes éléments que la liste en entrée et que le pivot est à sa place au sens du tri. Il suffit donc de trier les deux sous-listes `ListeDesPlusPetits` et `ListeDesPlusGrands`, ce qui s'implémente facilement grâce à la récursivité.
Compléter le script ci-contre :

```
def TriRapide(Liste) :
    if ............. : #cas d'arrêt
        return .....
    Pivot,ListeDes... # Initialisation
    for x in Liste[...
        if PlusGrand(......
            ....
        else :
            .....
    return TriRapide(...)+[Pivot]+...
```

On remarquera que la relation de récurrence vérifiée par la compléxité de ce programme est la même si l'on suppose que les longueurs des deux sous-listes `ListeDesPlusPetits` et `ListeDesPlusGrands` sont les mêmes.

On peut définir une fonction `TestTriRapide(Liste)` ou plus généralement `TestTriRapideAlea()`.

2) Description du programme en place.

Pour le tri rapide en place, l'idée la même, si ce n'est que le pivot choisi est à la fin de la liste.

On va utiliser un programme intermédiaire `Decoupe(Liste,deb,fin)` qui va modifier la liste des indices `deb` à `fin` inclus et retourner `IndicePivot` de telle sorte qu'**une fois le programme exécuté** :
• le pivot sera à l'indice `IndicePivot` de `Liste` (modifiée !)
• les éléments d'indices `deb` à `IndicePivot-1` sont plus petits que le pivot (la `ListeDesPlusPetits` de tantôt)
• les éléments d'indices `IndicePivot+1` à `fin` sont plus grands que le pivot (la `ListeDesPlusGrands` de tantôt).

Par exemple, si `L = [7,9,5,8,3,6]`, `Decoupe(L,0,5)` renvoit 2 et `L` vaut `[5,3,6,8,9,7]`.
Le pivot 6 est maintenant à sa place, précédé des plus petits et suivi des plus grands (non triés, l'ordre a changé mais cela n'est pas important).
On exécute ensuite `Decoupe(L,0,1)` afin de trier les plus petits, on obtient comme retour 0 et la liste devient `[3,5,6,8,9,7]`, le pivot 5 est maintenant à sa place.
On exécute ensuite `Decoupe(L,3,5)` afin de trier les plus petits, on obtient comme retour 3 et la liste devient `[3,5,6,7,9,8]`, le pivot 7 est maintenant à sa place.
On exécute ensuite `Decoupe(L,4,5)` afin de trier les plus petits, on obtient comme retour 4 et la liste devient `[3,5,6,7,8,9]`.

Ensuite, on va utiliser un programme récursif `TriRapidePartiel(Liste,deb,fin)` qui prend en entrée une liste à trier entre les indices `deb` et `fin` inclus.
On utilise la programme `Decoupe` précédent, l'indice du pivot obtenu permet de mettre en place la récursivité.

```
def TriRapidePartiel(Liste,deb,fin) :
    if deb>fin : #au moins deux éléments !
        IndPiv = Decoupe(Liste,deb,fin)
        TriRapidePartiel(Liste,deb,IndPiv-1)
        TriRapidePartiel(Liste,IndPiv+1,fin)
def TriRapideEnPlace(Liste) :
    TriRapidePartiel(Liste,0,len(Liste)-1)
```

3) Implémentation du programme Decoupe.

Pour la programmation de `Decoupe(Liste,deb,fin)`, la valeur du pivot est donc `Liste[fin]` et on initialise la valeur de `IndicePivot` à `deb`. On va considérer les termes de la liste des indices `deb` à `fin-1`, en les comparant au pivot, si le pivot est plus grand, on permute les termes d'indices `i` et `IndicePivot` et on incrémente `IndicePivot`.
À la fin de chaque étape de la boucle, `IndicePivot` est tel que
• les éléments d'indices `deb` à `IndicePivot-1` sont plus petits que le pivot,
• les éléments d'indices `IndicePivot` à `i` sont plus grands que le pivot.

```
def Decoupe(Liste,deb,fin) :
    IndicePivot = ...
    Pivot = ...
    for i in range(....
        if PlusGrand(.....
            Echanger(....
            IndicePivot+=1
    Echanger(......
    return IndicePivot
```