Tri en Python

On considère ici des tableaux ou listes d'entiers ou de flottants.

En Python, on peut trier une liste à l'aide de la méthode `sort` : si a est une liste d'entiers ou de flottants, `a.sort()` modifie la liste en la liste triée. En revanche, la fonction `sorted(a)` est une fonction qui prenant une liste ou un tableau renvoie la liste (ou la tableau) des éléments triés par ordre croissant. Ainsi :

`a = [4,1,3,2] ; a.sort(a)` # modifie a en $[1,2,3,4]$

`sorted([4,1,3,2])` # renvoie la liste $[1,2,3,4]$

1) Tri par sélection (dans un tableau)

a) *Principe* : On détermine la position du plus petit élément, on le met en première position (en échangeant avec le premier élément), et on itère le procédé sur le tableau restant.

b) *Programme en Python* :

```
def triSelection(a) :                    # il s'agit d'une procédure : elle modifie a
    n = len(a)
    for i in range(n) :                  # on cherche k tel que a_k = min(a_j)_{j≥i}
        k = i
        for j in range(i+1,n) :
            if a[k] > a[j] :  k = j
        a[k],a[i] = a[i],a[k]            # on met par échange cet élément en première position
```

`a = [1,3,2] ; triSelection(a) ; print(a)` renvoie $[1,2,3]$

c) *Complexité* : $\sum_{k=1}^{n} k = O(n^2)$ dans tous les cas.

d) *Complément culturel* : *Le tri bulle* est une variante, consistant à comparer, en commençant par la fin de la liste, les couples d'éléments successifs : Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. On fait ainsi remonter le terme le plus petit (de même que les bulles les plus légères remontent le plus vite à la surface ...). D'où le nom donné à ce tri. Puis on itère le procédé sur le sous-tableau restant :

```
def triBulle(a) :
        for i in range(n-1) :
                for j in range(n-1,i,-1) :
                        if a[j] < a[j-1] then a[j],a[j-1] = a[j-1],a[j]
```

2) Tri rapide par insertion dans une liste

a) *Principe* : On insère au fur et à mesure les éléments dans une liste triée.

On va utiliser une procédure auxiliaire `ajouter(a,x)` qui étant donnés une liste triée a et un élément x, modifie a en insérant x dans la liste a à la bonne place (de sorte à obtenir une liste triée).

b) *Programme en Python* :

On suppose connue la fonction `a.insert(i,x)` qui permet d'insérer un élément x en position i.
La structure de listes en Python est particulièrement efficace, car l'ajout se fait en $O(1)$ opérations (en moyenne).

Dans la fonction `ajouter(a,x)`, il faut déterminer la position d'insertion.

Pour optimiser la recherche, on utilise une méthode par dichotomie.

```
def ajouter(x,a) :
    i = 0 ; j = len(a)
    while j>i :
        k = (i+j)//2
        if x < a[k] :  j = k
        else :  i = (k+1)
    a.insert(i,x)

def triInsertion(a) :
    b = copy(a)
    for x in a :  ajouter(x,a)
```

Variante récursive :

```
def triInsertion(a) :
    x = a.pop()
    triInsertion(a)
    ajouter(x,a)
```

Complexité : $O(n \log n)$ dans le pire cas et en moyenne, $O(n)$ dans le meilleur cas.

c) La variante sur les tableaux est de complexité $O(n^2)$.
En effet, l'ajout d'un élément dans un sous-tableau trié nécessite de décaler $O(n)$ éléments.
Par exemple, l'ajout de 3 dans le tableau `[1,5,6,*,*,*]` donne `[1,3,5,6,*,*,*]`.

3) Tri rapide par fusion de listes triées

a) *Principe* : Il s'agit d'un exemple d'algorithme de type "diviser pour régner".
On procède par fusions successives de listes déjà triées.
La première fonction, appelée `fusion`, fusionne deux listes supposées triées en une seule.

Par exemple, `fusion([1,2,4],[2,3,8])` renvoie `1,2,2,3,4,8]`.

La fonction de tri, récursive, consiste à couper en deux la liste initiale, à trier (par appels récursifs) les deux sous-listes, puis à les fusionner en faisant appel à la fonction `fusion`.

Remarque : L'algorithme permettant la fusion de deux listes triées consiste à comparer les éléments de tête et à sélectionner le plus petit d'entre eux, à l'ajouter à la liste des éléments déjà sélectionnés, et à poursuivre sur les listes associées aux autres éléments.

b) *Programme en Python* :

```
def fusion(a,b) :
    c = [] ; n = len(a) ; m = len(b) ; i = 0 ; j = 0
    while i < len(a) and j < len(b) :    # on continue tant que les deux listes ne sont pas vides
        if a[i] < b[j] :  c.append(a[i]) ; i = i + 1
        else :  c.append(b[j]) ; j = j + 1
    # à ce stade, une (au moins) des deux listes est vide, et on ajoute les éléments de l'autre liste à c
```

```
    if i = len(a) :
        for j in range(j,len(b)) :  c.append(b[j])
    else :
        for j in range(i,len(a)) :  c.append(a[i])
    return c

def triFusion(a) :       # il s'agit ici d'une fonction renvoyant la liste des éléments de a triés
    n = len(a)
    if n <= 1 return a
    m = n // 2
    # on utilise ensuite le "slicing" pour définir les sous-listes, on les trie et on renvoie leur fusion :
    return fusion( triFusion(a[0:m]) , triFusion(a[m:n]) )
```

Si on veut obtenir une procédure, on peut par exemple considérer (il y a d'autres méthodes, plus directes) :

```
def triFusion_proc(a)
    b = triFusion(a)
    for i in range(len(a)) :  a[i] = b[i]
```

c) *Complexité* : $O(n \log n)$ dans tous les cas.

Notons n le nombre d'éléments de la liste à trier, et $T(n)$ le nombre d'opérations utilisées dans le tri fusion. Pour $n \geq 2$, on a $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n + c$, où c est une constante, car n représente le coût de la fusion de deux listes contenant au total n éléments.

Si $n = 2^p$, on a $T(n) = 2T(\frac{n}{2}) + n + c$. D'où on peut déduire $T(n) = 2^p T(1) + np + (1 + 2 + ... + 2^{p-1})c$.
Ainsi, sachant que $p = \log n$, on a : $T(n) = nT(1) + n \log n + (n-1)c$, donc $T(n) = O(n \log n)$.

Remarque : Pour majorer $T(n)$ dans le cas général, on montre d'abord par récurrence forte que $(T(n))_{n \in \mathbb{N}^*}$ est une suite croissante. On encadre alors n par des puissances de 2, en considérant $2^p \leq n < 2^{p+1}$, où $p = \log n$. On a donc $T(n) \leq T(2^{p+1})$, et comme $2^{p+1} \log(2^{p+1}) \leq 2n \log(2n)$, on peut en conclure $T_n = O(n \log n)$.

4) Tri rapide par pivot

a) *Principe* : Il s'agit de choisir un élément, appelé pivot, et de séparer en $O(n)$ opérations la liste en deux sous-listes comprenant respectivement les éléments inférieurs et les éléments supérieurs au pivot. On itère ensuite le procédé sur les deux sous-listes ainsi obtenues, jusqu'à obtenir des listes de longueur ≤ 1.

La procédure `partition(a,debut,fin)` prend en arguments une liste a de longueur n, un indice i de début, un indice j de fin tels que $0 \leq i \leq j < n$.

On choisit comme pivot la valeur de l'élément a_i, c'est-à-dire le premier élément de la sous-liste considérée.

Elle a pour effet de modifier (en temps linéaire) l'ordre des éléments de la sous-liste $(a_i, a_{i+1}, ..., a_j)$ de sorte que les éléments inférieurs au pivot se retrouvent avant le pivot, lequel précède ceux qui sont supérieurs au pivot.

De plus, la fonction `partition` renvoie la valeur de la position du pivot dans la liste obtenue.

Par exemple, avec $a = [\mathbf{3}, 5, 1, 6, 2, 4, 7]$, $d = 0$, $f = 6$, `partition(a,i,j,p)` a pour effet de transformer a par exemple en la liste $[2, 1, \mathbf{3}, 5, 6, 4, 7]$.

L'algorithme le plus élégant consiste à utiliser deux indices permettant de parcourir en parallèle les éléments de la liste (hormis le pivot) par la gauche et par la droite : dès qu'on obtient à gauche un élément plus grand que le pivot et à droite un élément plus petit, on effectue un échange.

A la fin, on permute le pivot avec le dernier élément de la première sous-liste, de sorte à le mettre à sa place, et on renvoie la position du pivot ainsi modifiée.

Remarque : Une solution plus facile à programmer mais utilisant un tableau auxilaire consisterait à construire les deux listes contenant respectivement les éléments inférieurs au pivot et les éléments supérieurs au pivot, puis à concaténer les deux listes.

b) *Programme en Python* :

```
def partition(a,debut,fin) :
    if debut >= fin :  return debut
    i = debut + 1
    j = fin
    while i<j :
        if a[i] <= pivot :  i = i + 1
        elif a[j] > pivot :  j = j - 1
        else :  a[i],a[j] = a[j],a[i] ; i = i + 1 ; j = j - 1
        if a[i] < pivot :  a[debut],a[i] = a[i],a[debut] ; return(i)
        else :  a[debut],a[i-1] = a[i-1],a[debut] ; return(i-1)

def triPivot(a,debut,fin) :       # tri du sous-tableau défini par les indices de debut et de fin.
    if debut <= fin :
         k = partition(a,debut,fin)
         triPivot(a,debut,k-1) ; triPivot(a,k+1,fin)

def triPivot2(a) :       # tri du tableau (on appelle la fonction précédente avec les bonnes bornes)
    triPivot(a,0,len(a)-1)
```

c) *Complexité* : $O(n \log n)$ en moyenne (*difficile à prouver*), $O(n^2)$ dans le pire cas.

Notons n le nombre d'éléments de la liste à trier, et T_n le nombre d'opérations utilisées dans le tri rapide.

Dans le cas le pire, le choix du pivot correspond à chaque itération à un des extrema de la liste considérée. On obtient alors T_n de l'ordre de n^2.

5) Complexité optimale d'un algorithme de tri (*complément culturel*)

Il y a $n!$ ordres totaux possibles dans un ensemble de cardinal n. L'algorithme de tri doit réaliser des opérations différentes pour chacune de ces possibilités (puisque les échanges sont différents pour deux configurations initiales différentes). Or, chaque test `if` distingue les permutations qui le vérifient de celles qui ne le vérifient pas. Ainsi, après N tests, on distingue donc au plus 2^N classes de permutations.

De ce fait, tout algorithme de tri doit au moins comporter un nombre de tests N vérifiant $2^N \geq n!$

On peut aussi interpréter les échanges effectués (selon les résultats aux tests) par un arbre (binaire) de décision : un arbre de hauteur N comporte au plus 2^N feuilles.

En conclusion, si T_n est la complexité (en temps) d'un algorithme de tri, alors $2^{T_n} \geq n!$

Comme $\log(n!) \sim n \log n$, alors T_n est nécessairement au moins de l'ordre de $n \log n$.

Remarque : Le raisonnement précédent peut s'avérer faux lorsqu'on dispose d'informations supplémentaires. Si par exemple, on sait que les éléments à trier sont des entiers compris entre 1 et p, on peut effectuer le tri en $O(p)$ opérations : il suffit d'utiliser un tableau de longueur p donnant les occurrences de chaque élément.