Tris et complexité

Un algorithme de tri sert à modifier la position des éléments d'une liste selon un ordre déterminé. Par exemple avec des nombres réels, on peut trier selon l'ordre $\leq$ pour que les valeurs apparaissent dans l'ordre croissant.
Trier une liste permet de rechercher plus rapidement un élément donné (complexité d'ordre $\log_2(n)$ plutôt que n), ou encore de rendre les données plus lisibles.

1 Tri par sélection

Il s'agit d'un des tris les plus simples. Voici le principe :

- On parcourt une liste `L` pour trouver son plus petit élément. On l'échange alors avec l'élément à la position 0. Ainsi, `L[0]` contient le minimum de `L`.
- Ensuite, on parcourt `L` à partir de l'indice 1. On trouve le plus petit élément de $\{L_1, L_2, \cdots\}$, et on l'échange avec l'élément à la position 1. Ainsi `L[1]` contient la deuxième plus petite valeur de `L`.
- Et ainsi de suite, jusqu'à ce que la liste soit triée.

Exercice 1. Coder une fonction `positMin` qui prend en argument une liste `L` de flottants et retourne un indice du minimum de cette liste.

Exercice 2. Coder une fonction `triSelect` qui prend en argument une liste `L` de flottants et retourne cette même liste triée par l'algorithme de tri par sélection.

Question 1. Montrer que cet algorithme a une complexité quadratique. (On l'exprime en fonction de la taille n de la liste `L`).

Outre la complexité, on peut également comparer deux algorithmes de tris selon leur *stabilité*, dont la définition est ci-dessous :

Définition

Un algorithme est dit stable si, lorsque plusieurs éléments sont considérés comme égaux par l'ordre considéré, le tri va renvoyer ces éléments dans le même ordre dans lequel ils étaient au départ.

Par exemple, on souhaite trier une liste de complexes par module croissant. Si on se donne une liste `L = [i,2i,-1,0,1]`, un algorithme stable devra renvoyer `[0,i,-1,1,2i]`. En revanche, un algorithme qui renvoie `[0,-1,i,1,2i]` n'est pas stable.

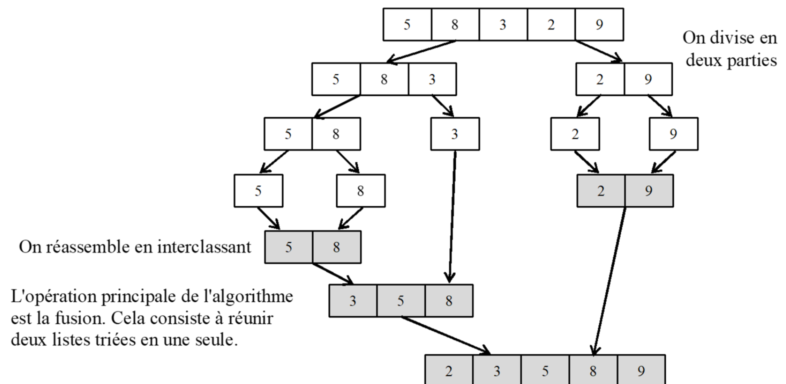
Question 2. Tester *à la main* le tri par sélection sur la liste `L = [5, 5.0, 2]` où `5` et `5.0` sont vus comme des éléments différents (bien qu'égaux pour l'ordre $\leq$). Le tri par sélection est-il stable ?

Parenthèse : complexité dans le meilleur cas. La complexité dans le pire cas est souvent celle qu'on considère par défaut. Mais on peut également regarder la complexité dans le meilleur cas (celui ou la liste est déjà triée). Normalement, il n'y aurait rien à faire, mais l'algorithme ne le sait pas, et va effectuer des opérations élémentaires superflues. On peut vérifier que le tri par sélection a une complexité quadratique dans le meilleur cas également. En réalité, cette complexité ne dépend pas de la liste donneé en argument et sera toujours quadratique. Cependant, on verra en approfondissement un autre tri qui est quadratique dans le pire cas et linéaire dans le meilleur cas. Par conséquent, pour une liste quelconque la complexité sera « intermédiaire » entre quadratique et linéaire, et donc ce tri est plus performant que le tri par sélection.

2 Tri fusion

Le tri par partition-fusion (aussi appelé tri fusion) applique la stratégie dite « diviser pour régner » :

- On coupe la liste à trier en deux parties à peu près de même taille.
- On trie les données de chaque partie *en appliquant la méthode de tri fusion* (c'est une fonction récursive).
- On fusionne les deux parties (maintenant triées) en interclassant les données.



L'algorithme est donc récursif. La récursivité s'arrête car on finit par arriver à des listes composées d'un seul élément et le tri est alors trivial (cf les listes [5], [8], etc. dans l'exemple ci-dessus). L'algorithme *récursif* est celui ci-dessous :

```
1  def triFusion(L):
2      if len(L)==1:
3          return L
4      else:
5          g, d = 0, len(L)-1        # indices extrêmes de L
6          m = (g+d)//2              # m est l'indice du "milieu" de L
7
8          L1 = triFusion( L[:m+1] )  # on trie la première moitié de L
9          L2 = triFusion( L[m+1:] )  # on trie la seconde moitié de L
10         return fusion(L1,L2)      # cf ci-dessous pour la fonction "fusion"
```

La principale difficulté de cet algorithme est de coder la fonction `fusion`, qui va interclasser les éléments de deux listes `L1` et `L2` par ordre croissant.

```
1  def fusion(L1,L2):     # L1 et L2 sont des listes **déjà triées**
2      F=[]               # la liste fusionnée
3
4      i = 0              # i va parcourir les indices de L1
5      j = 0              # j va parcourir les indices de L2
6
7      while (i<len(L1)) and (j<len(L2)):
8          # tant qu'on n'est pas au bout d'une des deux listes
9          # on ajoute un élément à F et on met à jour i ou j
10
11     if ... :               # tous les éléments de L1 ont été placés dans F
12         F = F + L2[...]    # on place alors le reste des éléments de L2
13
14     else:                  # tous les éléments de L2 ont été placés dans F
15         F = F + L1[...]    # idem avec L1
16     return F
```

Exercice 3. Compléter et tester `fusion` puis `triFusion`.

Question 3. Si `L1` et `L2` sont deux listes de taille $\frac{n}{2}$, montrer que la complexité de l'algorithme `fusion` est linéaire en n.

Complexité du tri fusion. On note $c(n)$ le nombre d'opérations élémentaires de `triFusion` pour une liste de taille n. La relation de récurrence est du type [1] $c(n) = 2c(n/2) + n$. Pour simplifier la preuve, on suppose que $n = 2^p$. Alors en divisant par n, on a

$$\frac{c(2^p)}{2^p} = \frac{c(2^{p-1})}{2^{p-1}} + 1$$

donc la suite $u_p = \frac{1}{2^p} c(2^p)$ est arithmétique de raison 1. On en déduit que $u_p = p + u_0 = p + 1$, si bien que $c(2^p) = 2^p(p+1)$. Ainsi

$$c(n) = n\log_2(n) + n$$

ce qui implique que la complexité du tri fusion est d'ordre $\boxed{n\log_2(n)}$: on dit que la complexité est <u>log-linéaire</u>. C'est le mieux qu'on puisse faire avec un tri comparatif (cf définition en section 4).

3 Temps d'exécution

On veut comparer la vitesse d'exécution du tri fusion (complexité $n\log_2(n)$) et d'un tri quadratique (complexité n^2), afin de mettre en évidence l'importance de la complexité. Pour cela, on chronomètre le temps d'exécution des algorithmes avec une liste de très grande taille. On va générer cette liste aléatoirement, avec la fonction `random()` du module `random`. Puis, cela fait, on utilise la fonction `time()` du module `time` pour chronométrer le temps d'exécution.

- Rappel : la fonction `random()` du module `random` retourne un nombre aléatoire dans $[0,1]$.

```
import random
import time

N = 5000
L = ...      # Liste de N nombres aléatoires dans [0,1]

t1 = time.time()
triFusion(L)
t2 = time.time()
triSelection(L)
t3 = time.time()
print('temps tri fusion : ', t2-t1)
print('temps tri sélect : ', t3-t2)
```

Exercice 4. Compléter, compiler, admirer. Admirer encore avec `N = 10000`.

4 Exercices d'approfondissement

Tris comparatifs ou non. Tous les tris précédents sont <u>comparatifs</u> : pour placer l'élément au bon endroit, on fait des comparaisons des divers éléments entre eux selon un ordre donné. L'algorithme qui suit est indépendant de toute notion d'ordre : il permet de trier sans effectuer de comparaison !

1. On peut être plus précis en remplaçant le « $+n$ » de la relation par le nombre exact d'opérations de `triFusion` hormis les appels récursifs, mais si on s'intéresse uniquement à l'ordre de la complexité, le résultat sera identique au final.

Exercice 5 (Tri par comptage)**.** On suppose que tous les éléments de la liste à trier sont des nombres *entiers* entre 0 et $M \in \mathbb{N}$. On veut là encore les trier par ordre croissant. L'algorithme de comptage construit d'abord un *histogramme* `H`, qui est une liste telle que `H[i]` donne le nombre d'occurences de l'entier `i` dans `L` (avec `i` allant de 0 à M). Ensuite, on reconstruit la liste triée à partir de cet histogramme. Par exemple, avec

`L = [0,2,2,5,2,5,3]`

on construit l'histogramme `H = [1,0,3,1,0,2]`. Ensuite on reconstruit une liste triée `T` à partir de `H` en ajoutant autant de fois qu'il le faut chaque entier compté par `H` : on obtient donc `T = [0,2,2,2,3,5,5]`.

```
1  def tri_comptage(L,M):
2      H = [0]*(M+1)
3      for i in range(len(L)):
4          H[...] += 1
5      print("L'histogramme est : ", H)   # pour visualiser
6  
7      T = []
8      for i in range(M+1):
9          # on ajoute i à T autant de fois que compté par l'histogramme H
10         print(T)   # idem, pour visualiser
11     return T
```

Exercice 6. Compléter et tester la fonction.

Question 4. Mettre toutes les instructions `print` en commentaire et exécuter la commande `tri_comptage([0],10**7)`. À votre avis, pourquoi est-ce si long ?

Exercice 7 (Tri par insertion)**.** Regarder le principe sur ce lien (vous pouvez cliquer sur ce lien en allant sur la version en ligne du document) : https://fr.wikipedia.org/wiki/Tri_par_insertion#Exemple. Écrire une fonction `triInsert` qui prend en argument une liste `L` et retourne cette liste triée en utilisant cet algorithme (si vous bloquez, un algorithme en pseudo-code est présenté en-dessous).

Exercice 8. Réaliser une étude théorique de l'algorithme par insertion :

1. Cet algorithme est-il stable ?
2. Évaluer la complexité dans le pire cas, qui est celui d'une liste triée par ordre décroissant.
3. Évaluer la complexité dans le meilleur cas, qui est celui d'une liste triée par ordre croissant.

Comparer avec l'algorithme de tri par sélection vu en TP.

Exercice 9. Réécrire une fonction `triFusion` qui trie des listes de complexes par module croissant. Le tri fusion est-il stable ?

Exercice 10. On peut montrer que la complexité du tri fusion dans le meilleur cas (i.e. une liste déjà triée) est, comme dans le pire cas, log-linéaire (i.e. d'ordre $n\log_2(n)$). Le tri par insertion en revanche a une complexité quadratique dans le pire cas et linéaire dans le meilleur cas. On peut donc se demander lequel de ces deux algorithmes est le meilleur pour trier une liste aléatoire. Mesurer le temps d'exécution pour des listes de grande taille. Lequel des deux algorithmes est le plus efficace ?

Exercice 11. Python dispose d'une fonction native appelée `sorted`, qui permet de trier une liste avec un algorithme très efficace. Calculer le ratio de la vitesse d'exécution de `sorted` sur celui de `triFusion` pour une liste aléatoire de taille $N = 100\,000$. Répéter pour 2, 4 et 8 fois ce nombre. Peut-on conclure que l'ordre de la complexité de `sorted` est plus petit que $n\log_2(n)$?