Diviser pour régner : le tri-fusion

Un algorithme de tri est une méthode permettant, à partir d'une liste d'éléments comparables (nombres, chaînes de caractères, …) dans le désordre, d'obtenir une liste comportant les mêmes éléments classés. L'intérêt de l'action de trier une liste est de permettre de rechercher très rapidement un élément dans celle-ci. Nous avons vu deux algorithmes de tri en classe de 1ère : le tri par insertion et le tri par sélection, et nous en avons profité pour introduire la notion de complexité en temps d'un algorithme. Les deux algorithmes de tri étudiés en classe de première sont de complexité quadratique, ce que l'on note en $O(n^2)$. Cela veut dire que si on multiplie la taille des données (typiquement le nombre d'éléments dans la liste) par un nombre N, le temps d'exécution de l'algorithme est multiplié par N^2. Une complexité quadratique n'est pas très bonne, car le temps d'exécution devient très grand quand le nombre de données à traiter augmente. Peut-on espérer trouver un algorithme de tri plus efficace ?

Note : dans ce cours, on ne parlera que de la complexité « dans le pire des cas » (voir cours de première pour plus de précisions).

I./ Principe de base de la méthode « Diviser pour régner » :

L'idée de base de la méthode est de partir d'un problème de grande taille, difficile à résoudre, et de le découper en plusieurs problèmes de taille plus petite, pouvant être résolus de façon plus simple. La solution du grand problème de départ peut être obtenue en combinant ensemble les différentes solutions des plus petits problèmes. Cette méthode sera utilisée pour des problèmes pouvant se résoudre de façon récursive : on coupe le gros problème en plusieurs parties, puis on coupe les parties en plusieurs parties plus petites, etc… jusqu'à obtenir des problèmes élémentaires que l'on peut résoudre directement. On peut donc décomposer la méthode en trois phases :

- Diviser : on divise les données initiales en plusieurs sous-problèmes, et ainsi de suite jusqu'à avoir des problèmes élémentaires.
- Régner : on résout les problèmes élémentaires obtenus.
- Combiner : on combine les solutions obtenues de façon à obtenir une solution au problème initial.

Il y a plusieurs problèmes pouvant être traités de la sorte, mais nous allons en étudier un en particulier, le tri-fusion, dont le rôle sera, à partir d'une liste d'éléments dans le désordre, d'obtenir la liste triée. Nous comparerons ensuite cet algorithme à ceux étudiés en classe de première pour voir s'il est plus efficace.

II./ Tri-fusion :

1./ Fusion de deux listes triées :

Soient deux listes L_1 et L_2 déjà triées dans l'ordre croissant de leurs éléments. La fusion des listes triées L_1 et L_2 est une opération qui va créer une liste L triée contenant tous les éléments de L_1 et de L_2. Les listes L_1 et L_2 étant déjà triées, l'algorithme de l'opération de fusion est très simple.

```
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:])
```

Si la liste L_1 est vide, la liste L se résume à la liste L_2, déjà triée. Si la liste L_2 est vide, la liste L se résume à la liste L_1, déjà triée. Si les deux listes sont non vides, on compare le premier élément de chacune d'entre elles : en effet, les listes L_1 et L_2 étant déjà triées, le plus petit élément de la liste L est forcément soit le plus petit élément de la liste L_1 (donc le premier élément de L_1), soit le plus petit élément de la liste L_2 (donc le premier élément de L_2).

Supposons que les deux listes de départ soient : `L1 = [1,3,5]` et `L2 = [2,4,6]`.

Les deux listes sont non vides, donc on compare leur premier élément. `L1[0] = 1` et `L2[0] = 2`. `L1[0] < L2[0]`, donc le premier élément de la liste `L` est `L1[0] = 1`. Pour trouver les éléments suivants de la liste `L`, il faut appliquer la même méthode avec la liste `L1` privée de son premier élément (soit `[3, 5]`) et la liste `L2`.

Entre la liste `[3,5]` et la liste `[2,4,6]`, le plus petit premier élément est `2`, qu'on rajoute à `L`, puis on applique la même méthode sur la liste `[3,5]` et la liste `[4,6]`.

Entre la liste `[3,5]` et la liste `[4,6]`, le plus petit premier élément est `3`, qu'on rajoute à `L`, puis on applique la même méthode sur la liste `[5]` et la liste `[4,6]`.

Entre la liste `[5]` et la liste `[4,6]`, le plus petit premier élément est `4`, qu'on rajoute à `L`, puis on applique la même méthode sur la liste `[5]` et la liste `[6]`.

Entre la liste `[5]` et la liste `[6]`, le plus petit premier élément est `5`, qu'on rajoute à `L`, puis on applique la même méthode sur la liste `[]` et la liste `[6]`.

La liste `[]` étant vide, on renvoie `6`, qu'on rajoute à `L`. `L` contient alors `[1,2,3,4,5,6]` c'est-à-dire tous les éléments de `L1` et `L2` classés dans l'ordre croissant.

2./ Principe de la méthode du tri-fusion :

1ère étape : Diviser :

Soit une liste L de nombres à trier dans l'ordre croissant. Par exemple L = [8,6,1,5,4,7,3,2]. On va couper L en deux par le milieu. On obtient alors les deux listes suivantes : [8,6,1,5] et [4,7,3,2].

On découpe alors chacune des deux listes obtenues par le milieu : on obtient les quatre listes suivantes : [8,6], [1,5], [4,7] et [3,2].

On découpe à nouveau chacune de ces quatre listes par le milieu, en obtenant huit listes composées d'un seul élément : [8], [6], [1], [5], [4], [7], [3] et [2].

Cette étape s'arrête à ce stade, car on ne peut plus couper les listes restantes en deux.

Note : l'exemple précédent a été choisi pour que le partage tombe « juste ». La méthode marche de la même façon si jamais la liste contient un nombre impair d'éléments. Par exemple la liste [1,8,5] serait découpée en [1,8] et [5].

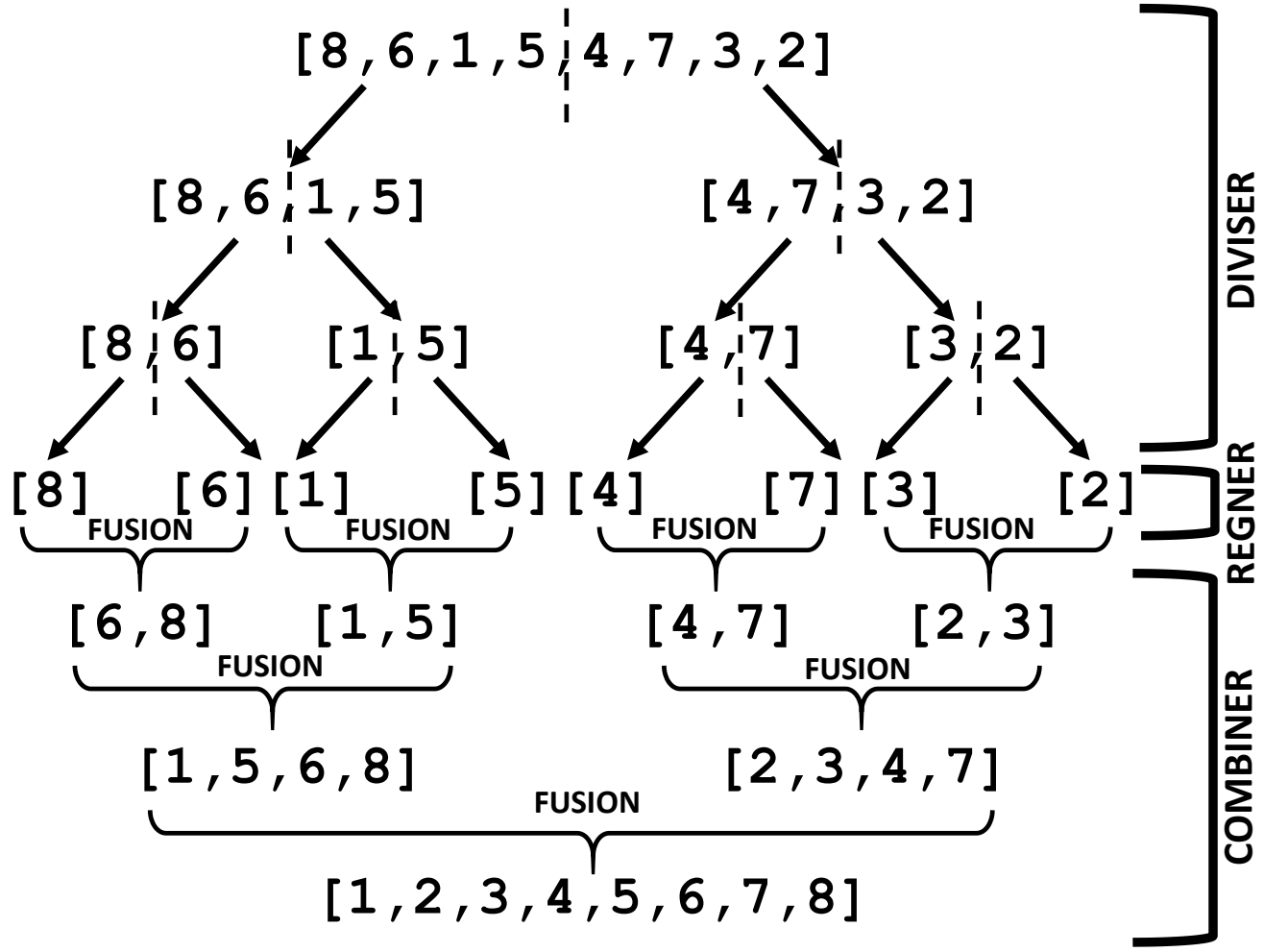
2ème étape : Régner :

Cette étape est un peu particulière dans l'exemple du tri-fusion. En effet en divisant le problème, on est arrivé à des listes contenant un seul élément. Une liste contenant un seul élément est forcément triée. Il n'y a donc rien à faire : les sous-problèmes sont directement résolus.

3ème étape : Combiner :

Pour obtenir la liste globale triée, on va combiner les sous-listes deux à deux en utilisant la fonction « fusion » vue précédemment. On a le droit d'utiliser cette fonction car les sous-listes sont triées.

On peut représenter les étapes précédentes par le schéma suivant :



3./ Implémentation en Python :

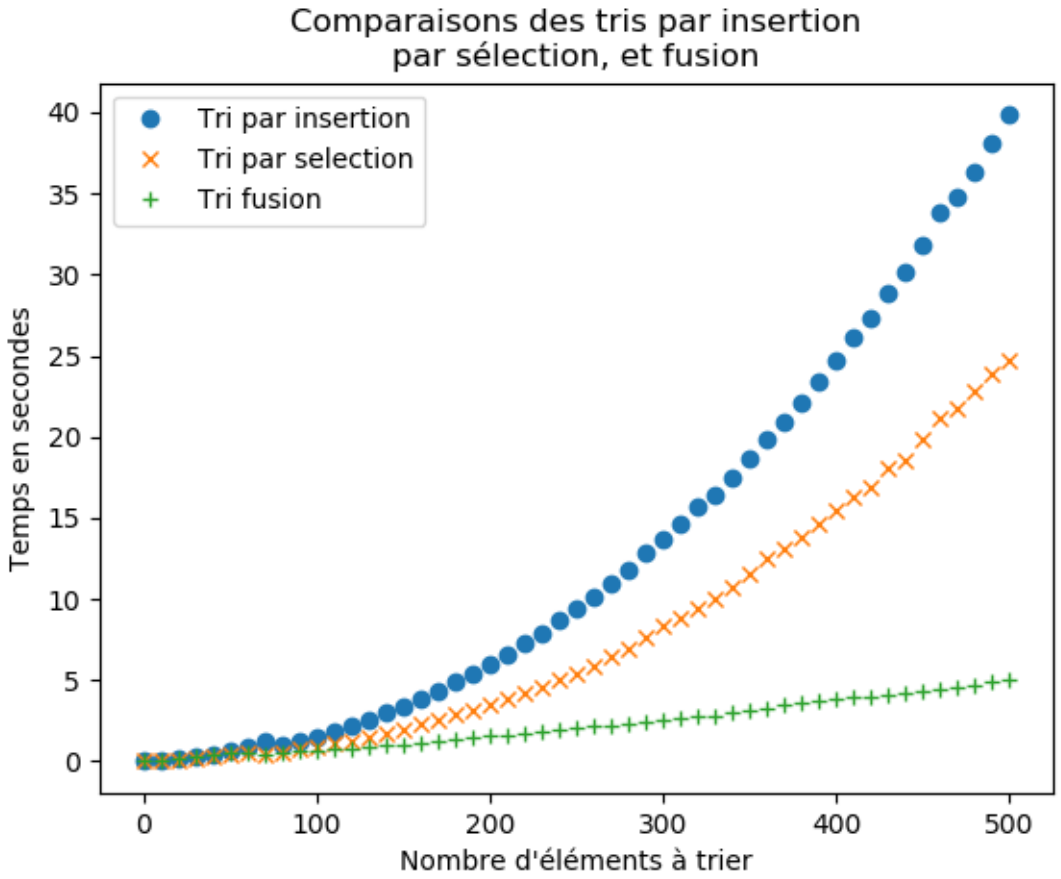
```
def tri_fusion(L):
    longueur = len(L)
    if longueur <= 1:
        return L
    milieu = longueur // 2
    L1 = L[:milieu]
    L2 = L[milieu:]
    return fusion(tri_fusion(L1),tri_fusion(L2))
```

III./ Complexité du tri-fusion :

Pour comparer l'efficacité du tri-fusion par rapport aux tris vus en première, nous avons chronométré (grâce au module time) le temps nécessaire pour trier un certain nombre d'éléments, de 0 à 500.

Note : comme le tri de 500 éléments est extrêmement rapide en Python, quelque soit la méthode, le chronométrage a été fait sur le tri de 2000 listes tirées au hasard pour chaque valeur de N, nombre d'éléments de la liste. Les trois algorithmes de tri (insertion, sélection, fusion) ont été appliqués sur les mêmes listes, afin de pouvoir faire une comparaison représentative.

On obtient les courbes suivantes :



On remarque que le tri-fusion est beaucoup plus efficace que les deux autres algorithmes testés. On remarque bien l'augmentation du temps en $O(n^2)$ pour les tris insertion et sélection. Le tri-fusion ne suit manifestement pas une loi quadratique. Une étude théorique plus poussée montrerait que la complexité en temps du tri-fusion est en $o(n \times \log_2(n))$, avec $\log_2(n)$ le logarithme en base 2 de n. Le logarithme en base 2 d'un nombre n se calcule de la façon suivante :

$$\log_2(n) = \frac{\ln(n)}{\ln(2)}$$

Où $\ln(n)$ est le logarithme népérien de n.

Cela veut donc dire que si on multiplie le nombre d'éléments d'une liste par 10, le temps d'exécution de l'algorithme de tri-fusion sera multiplié par $10 \times \log_2(10) \approx 33{,}2$. Pour les tris par insertion ou par sélection, le temps d'exécution sera multiplié par $10^2 = 100$.

Si on multiplie le nombre d'éléments d'une liste par 100, le temps d'exécution de l'algorithme de tri-fusion sera multiplié par $100 \times \log_2(100) \approx 664$, alors que pour les tris par insertion ou par sélection, le temps d'exécution sera multiplié par $100^2 = 10\,000$.

Si on multiplie le nombre d'éléments d'une liste par 1000, le temps d'exécution de l'algorithme de tri-fusion sera multiplié par $1000 \times \log_2(1000) \approx 9966$, alors que pour les tris par insertion ou par sélection, le temps d'exécution sera multiplié par $1000^2 = 1\,000\,000$.

On se rend compte que plus le nombre N d'éléments à trier est important, plus l'écart se creuse entre le tri-fusion et ses concurrents.

Remarques :

- On peut démontrer de façon théorique (mais c'est largement hors programme) que pour un algorithme de tri on ne peut pas faire mieux que $o(n \times \log(n))$ comme complexité. Notamment il est impossible de faire un algorithme de tri en $o(n)$.
- Le tri-fusion faisant appel à des fonctions récursives, il va donc être très gourmand en mémoire.
- Pour une comparaison de différents algorithmes de tris, revoir le cours de première, ainsi que la page suivante : https://fr.wikipedia.org/wiki/Algorithme_de_tri#Comparaison_des_algorithmes. Vous pourrez aussi trouver des informations et des simulations sur cette page : http://lwh.free.fr/pages/algo/tri/tri.htm.
- Pour comparaison, voici les fonctions $y=x$ (linéaire), $y=x \times \log(x)$ (linéarithmique) et $y=x^2$ (quadratique ou carrée) :