Cours 8 – Tris

Problématique : On a vu dans le cours sur la complexité qu'une recherche dans un tableau non trié avait une complexité linéaire, alors que cette complexité était seulement logarithmique dans le cas de tableaux triés.

On s'intéresse donc ici à différents algorithmes permettant de trier efficacement des tableaux, afin de pouvoir les exploiter efficacement.

I Algorithme naïf : tri par sélection

Principe. On parcourt la liste L :

— On cherche le plus petit élément de liste et on l'échange avec $L[0]$

— On cherche ensuite le deuxième plus petit élément et on l'échange avec $L[1]$

— On continue de cette façon jusqu'à ce que le tableau soit entièrement trié

Exemple. Dérouler l'algorithme de tri par sélection sur la liste $L = [9, 7, 3, 1, 2]$.

Implémentation en Python.

```
def tri_selection(L):
```

.

Analyse de complexité.

II Tri par insertion

Principe. On insère les éléments du tableau un par un dans l'ensemble des éléments déjà triés. (C'est par exemple ce qu'on fait souvent pour trier un jeu de cartes.)

1. On prend la première carte
2. On prend la 2ème carte et on la compare à la 1ère.
   - Si la 2ème est plus petite, on la place avant la 1ère
   - Sinon, on la place après.
3. On prend la 3ème carte et on la compare à la 2ème
   - Si la 3ème est plus grande, on la place après la 2ème.
   - Sinon, on la compare à la 1ère.
     - Si la 3ème est plus petite, on la place avant la 1ère.
     - Sinon, on la place entre la 1ère et la 2ème.
4. on continue ainsi de suite jusqu'à avoir placé toutes les cartes.

On insère `7` dans `L[0:1]`

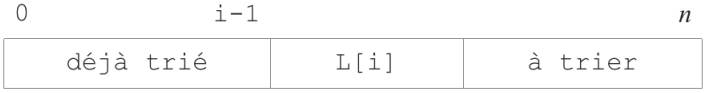
On insère `3` dans `L[0:2]`

On insère `1` dans `L[0:3]`

On insère `2` dans `L[0:4]`

9	7	3	1	2

Implémentation On utilise une boucle pour parcourir tous les éléments de la liste (à partir de l'indice $i = 1$.). Avant chaque itération, on se trouve dans la situation suivante :



Pour insérer $L[i]$ au bon emplacement, on décale les éléments précédents vers la droite, tant qu'ils sont supérieurs à $L[i]$.

```
def tri_insertion(L):
```

.

Exemple Dérouler l'algorithme précédent sur la liste $L = [15, 4, 2, 55]$. Faire apparaître chaque modification de la liste L.

Analyse de Complexité

— Pire des cas

— Meilleur des cas

III Tri rapide (quicksort)

Principe. On utilise le principe de « diviser pour régner ». On partage la liste en deux sous-ensembles, les éléments du 1er sous-ensemble étant tous plus petits que ceux du 2ème, puis on applique récursivement le tri à chacun des sous-ensembles. (le cas de base étant une liste à 0 ou 1 élément, nécessairement triée).

En pratique, pour effectuer le partage, on choisit arbitrairement un élément de la liste (par exemple le premier), qu'on appelle **pivot**. On met dans le premier sous-ensemble les valeurs plus petites que p et dans le second celles qui sont plus grandes que p.

Exemple. On applique l'algorithme de tri rapide à la liste $L = [4, 6, 7, 3, 5, 2, 1]$.

Pour trier `L[0:7]`, on choisit `4` comme pivot :

On partage les deux sous-ensembles par rapport à `4` :

Pour trier `L[0:3]`, on choisit `3` comme pivot :

On partage les deux sous-ensembles par rapport à `3` :

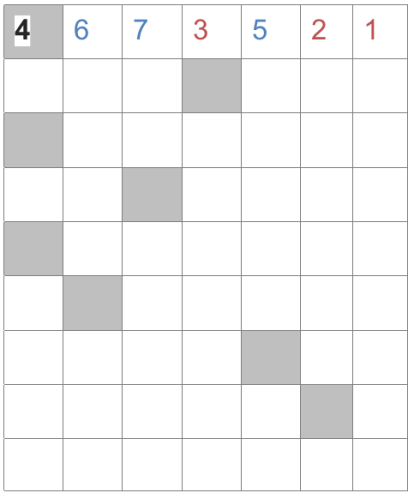
Pour trier `L[0:2]`, on choisit `2` comme pivot :

On partage les deux sous-ensembles par rapport à `2` :

Pour trier `L[4:7]`, on choisit `6` comme pivot :

On partage les deux sous-ensembles par rapport à `6` :

On obtient finalement :



Implémentation. Pour simplifier notre implémentation, on ne va pas trier la liste sur place comme dans l'exemple précédent (un appel à la fonction `tri_rapide(L)` laissera donc L inchangée, mais renverra une liste triée contenant tous les éléments de L).

On rappelle que la fonction est récursive. Après avoir sélectionné le pivot, on va donc créer deux listes contenant les éléments plus petits/plus grands que le pivot et appeler récursivement la fonction sur ces deux listes

```
def tri_rapide(L):



.
```

Analyse de complexité.

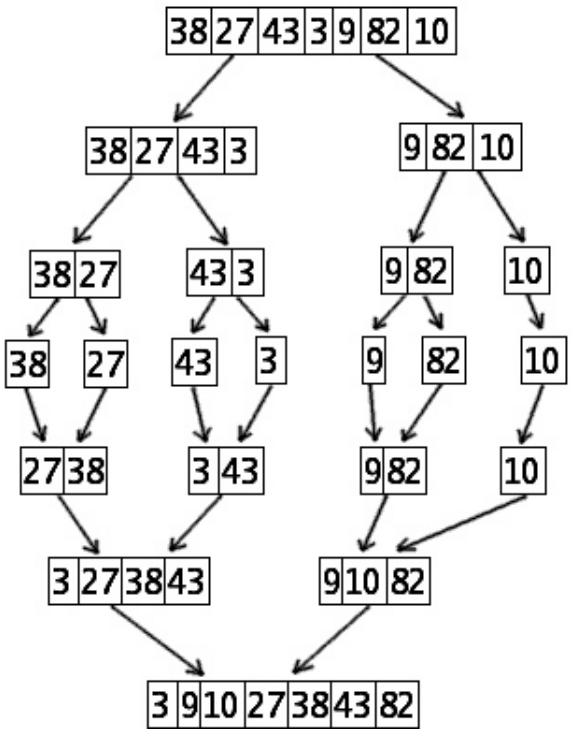
— Meilleur des cas :

— Pire des cas :

IV Tri par fusion

Principe. On utilise une nouvelle fois le principe de diviser pour régner. Le partage de la liste en deux se fait cette fois sans comparer les éléments, et les deux parties sont donc prises de même taille. On appelle récursivement l'algorithme de tri sur chacune des deux sous-listes. On fusionne alors les deux sous-listes triées en respectant l'ordre.

(Le fait de partager en deux sous-listes de tailles égales permet d'éviter la situation du pire des cas du tri rapide.)



Exemple. Appliquer l'algorithme de tri par fusion à la liste $L = [7, 6, 3, 5, 4, 2, 1, 8]$.

Implémentation. On commence par écrire une fonction `fusion(L1,L2)` qui prend en arguments deux listes triées $L1$ et $L2$ et renvoie la liste triée composée de tous les éléments de $L1$ et $L2$.

```
def fusion(L1,L2):




.
```

On peut alors écrire une fonction récursive mettant en œuvre l'algorithme de tri par fusion sur une liste L. (Pour le tri fusion, le tri ne peut pas se faire sur place, on est obligés de travailler sur des nouvelles listes)

```
def tri_fusion(L):
```

.

Analyse de complexité

V Comparaison

Comparaison des complexités temporelles.

	Tri par insertion	Tri rapide	Tri par fusion
Meilleur des cas	$O(n)$	$O(n\log(n))$	$O(n\log(n))$
Pire des cas	$O(n^2)$	$O(n^2)$	$O(n\log(n))$
Cas moyen	$O(n^2)$	$O(n\log(n))$	$O(n\log(n))$
Liste presque triée	$O(n)$	$O(n^2)$	$O(n\log n)$

Attention : les meilleurs/pires cas ne sont pas les mêmes selon les algorithmes, cf. dernière ligne du tableau.

Complexité spatiale. Quand le tri peut être fait sur place (sélection, insertion, tri rapide), le coût est constant.

Pour les versions de tri non en place (notre implémentation du tri rapide, tri fusion), on doit stocker l'équivalent d'un deuxième tableau de la même taille que le tableau à trier, ce qui peut être rédhibitoire lors d'une tri de grosses bases de données. C'est l'inconvénient majeur du tri par fusion.

Recherche(s) dans un tableau.

— Recherche dans un tableau non trié : $O(n)$

— Tri puis recherche dans un tableau trié : $O(n\log(n)) + O(\log(n)) = O(n\log(n))$

Asymptotiquement, pour une seule recherche, la 1ère solution est plus intéressante (mais pour des valeurs concrètes, $\log(n)$ reste petit, donc la différence n'est pas forcément importante).

Si on effectue k recherches :

— La recherche sans tri a un coût de l'ordre de kn

— La recherche avec tri a un coût de l'ordre de $n\log n + k\log n$

Pour une valeur importante de k, trier la liste peut ainsi améliorer la complexité.