10 – Les algorithmes de tri

I. Introduction

Nous allons traiter dans ce cours des algorithmes de tris.
Pour simplifier nous trierons des listes d'entiers, dans l'ordre croissant,

par exemple: > ***tri([9; 3; 7; 1; 2; 4])*** devra renvoyer ***[1; 2; 3; 4; 7; 9]***.

Ces algorithmes peuvent bien sûr s'appliquer à d'autres types de tris (décroissant, classement alphabétique, etc.), voir des classements beaucoup plus complexes : par exemple les réseaux sociaux utilisent un algorithme qui trie et ordonne l'ensemble des messages pouvant apparaître dans votre fil d'actualité (chaque message étant affecté d'une valeur numérique, correspondant à un score de pertinence spécifique pour chaque utilisateur.).

Il existe des dizaines d'algorithmes de tri et leurs variantes.
Conformément au programme nous présenterons ici quatre types d'algorithme de tri :

Le ***tri-rapide*** (ou quick-sort) , le ***tri par fusion***, le ***tri par insertion*** et enfin le ***tri par sélection*** .

Nous nous intéresserons de plus aux points suivants :

- ***Complexité temporelle*** des algorithmes : nous calculerons la complexité dans le pire et dans le meilleur des cas.
- ***Complexité spatiale*** des algorithmes : il est possible de trier une liste *en place* (la liste à trier est alors modifiée et aucune autre liste auxiliaire n'est utilisée), ou *non en place* (la liste initiale n'est pas modifiée, on retourne une nouvelle liste). Si on ne souhaite pas conserver la liste d'origine, le tri en place est préférable car il économise de l'espace mémoire.

Il faut garder en tête que ces algorithmes sont utilisés pour trier des listes très très grandes ...

Enfin, remarquons que Python possède des méthodes associées aux listes, déjà implémentées, qui permettent d'effectuer un tri. Celles-ci sont performantes (de complexité $O(nlog(n))$, dite quasi-linéaire), mais pas toujours autorisée dans une copie de concours.

Tri en place :

```
>>> ma_liste=[3,4,-9]
>>> ma_liste.sort()#trie en place la liste
>>> ma_liste
[-9, 3, 4]
```

Tri non en place :

```
>>> ma_liste=[3,4,-9]
>>> ma_liste_triée=sorted(ma_liste)#renvoie une copie triée de la liste
>>> ma_liste
[3, 4, -9]
>>> ma_liste_triée
[-9, 3, 4]
```

II. Tri rapide (ou QuickSort)

1) Présentation

On considère l'ensemble des éléments à trier, et on le partage en deux sous-ensembles, les éléments du premier étant plus petits que les éléments du second, puis on trie **récursivement** chaque sous-ensemble.
En pratique, on réalise le partage à l'aide d'un élément p arbitraire de l'ensemble à trier, appelé **pivot**.
Les deux sous-ensembles sont alors respectivement les éléments plus petits et plus grands que p.
Dans ce cours, *nous prendrons toujours comme pivot* ***le premier élément de la liste.***
Exemple :

8	1	5	14	4	15	12	6	2	11	10	7	9

2) Programmation

a) Fonction partition

Le but de ce paragraphe est de définir une fonction **`partition`**, ayant comme entrée une liste a, et qui partitionne en deux cette liste.
Comme annoncé dans le paragraphe précédent, on choisira comme pivot le premier élément de la liste à partitionner, et la fonction devra retourner une liste de trois éléments constituée : de la liste des valeurs plus petites que le pivot, du pivot, et de la liste des valeurs plus grandes que le pivot.
Exemple : **`partition`** ***([5,2,7,3,8,1])*** devra retourner [[2,3,1] , **5** , [7,8]]

Programmer une telle fonction partition, en utilisant deux listes auxiliaires b et c. Vous allez parcourir tous les éléments de la liste initiale a (à partir de l'indice 1) et stocker dans b les éléments plus petits que le pivot, et dans c les éléments plus grands. A la fin vous retournerez la liste b, le pivot et la liste c.

b) Fonction tri_rapide

Définir une fonction *récursive* **`tri_rapide`**, ayant comme entrée une liste a, et qui retourne la liste triée, en suivant les principes:

- si la liste est de taille 0 ou 1, alors elle est déjà triée.
- sinon, partitionner la liste grâce à la fonction **`partition`** puis effectuer un **`tri_rapide`** sur les deux sous listes b et c retournées par la fonction **`partition`**
  Le résultat à renvoyer sera la concaténation « b_trié + [pivot] + c_trié»

Remarquons que les appels récursifs s'arrêteront forcément puisque les deux sous listes sont de taille strictement inférieure au tableau initial.

3) Complexité spatiale

On voit que le tri présenté ici passe par la création de sous-listes, ce tri n'est donc pas effectué en place.
Cependant, il est possible d'effectuer un tri rapide en place mais l'algorithme est un peu plus difficile à construire.

4) Complexité temporelle

a) Déterminer la complexité P(n) de la fonction **`partition`** dans le cas d'une liste de taille n.

b) Nous admettrons que le « pire des cas », pour la complexité de la fonction ***tri_rapide***, est celui où à chaque appel récursif, la partition effectuée retourne une liste vide et une liste de taille « un de moins ».
(i)Si on note C(n) la complexité dans le pire des cas de la fonction ***tri_rapide*** pour trier une liste de taille n, déterminer tout d'abord une relation de récurrence entre C(n) et C(n-1).
Vous vérifierez que vous obtenez une expression de la forme $C(n) = a+bn+C(n-1)$, où a et b sont des constantes entières.
(ii) Si on note $C(0)=\lambda$, on pourrait alors vérifier, en calculant de proche en proche les C(n), que $C(n)=na+\frac{n(n+1)}{2}b+\lambda$.

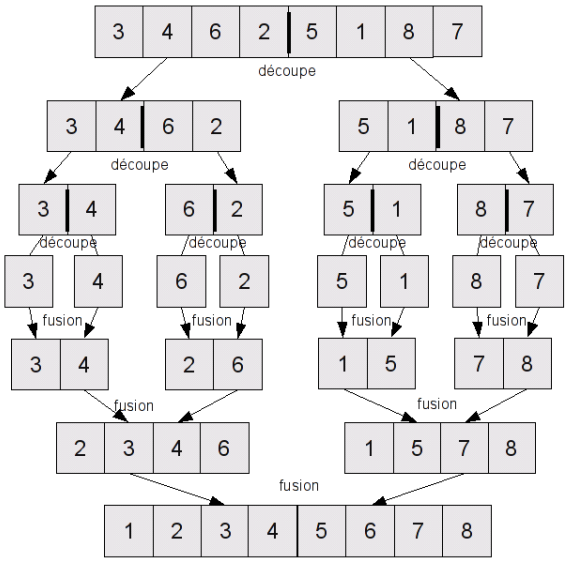
On en déduit donc que la complexité dans le pire des cas est **quadratique**.
Cependant, on pourrait montrer que **sa complexité moyenne** (notion hors programme) est un $\boldsymbol{O(n\log(n))}$**,** ce qui est en pratique assez rapide (d'où le nom de cette méthode de tri !). On parle de **complexité quasi-linéaire.**

III. Tri par fusion

1) Présentation

Comme pour le tri rapide, on coupe la liste initiale en deux, mais au lieu de couper la liste autour d'un pivot, on coupe la liste en deux parties égales par le milieu. On trie ensuite ces deux listes (par récursivité) et on les fusionne (la fusion consiste à fusionner deux listes déjà triées en une seule liste triée).

Exemple :



2) Programmation

a) Fonction fusion

Définir une fonction **`fusion`**, ayant comme entrée deux listes a et b, **supposés déjà triées**, et qui les fusionne.
Cette fonction devra donc retourner une seule liste, qui contiendra tous les éléments de a et de b, triés.
Remarquons que cette fusion ne va pas s'effectuer en place, c'est-à-dire que nous allons créer une nouvelle liste qui contiendra les deux listes fusionnées.

Voici le principe de construction de cette fonction :

- Créer une liste vide c.
- Utiliser deux indices i et j pour parcourir les listes a et b tant qu'elles ne sont pas épuisées.
  Si $a[i]<b[j]$, on ajoute $a[i]$ à la liste c et on passe à l'élément suivant de a.
  Si $a[i]\tilde{A}b[j]$, on ajoute $b[j]$ à la liste c, et on passe à l'élément suivant de b.
- Dès qu'une liste est épuisée, on complète c avec tous les éléments restants de l'autre liste.

Illustration pour fusionner [2,5,7] et [1,3,8,11] :

c) Fonction tri_fusion

Définir une fonction récursive **`tri_fusion`**, ayant comme entrée une liste a, et qui retourne la liste triée, en suivant les principes:

- si la liste est de taille 0 ou 1, alors elle est déjà triée.
- sinon, couper cette liste en deux parties égales ou presque (prendre n//2 pour la taille de la première sous-liste, ce qui règle le cas du nombre impair d'éléments).
- effectuer un **`tri_fusion`** sur chacune de ces sous-listes, et fusionner (grâce à la fonction ***fusion***) ces deux listes triées obtenus.

Remarquons que les appels récursifs s'arrêteront forcément puisque les deux sous-listes sont de taille strictement inférieure à la liste initiale.

3) complexité

a) Déterminer la complexité F(m) de la fonction **`fusion`** dans le cas où on fusionne deux listes de taille m.

b) Nous allons nous intéresser à la complexité de la fonction **`tri_fusion`** dans le cas d'une liste de départ de taille 2^p. Notons $C(2^p)$ cette complexité.
(i) Exprimer $C(2^p)$ en fonction de p et de $C(2^{p-1})$. Vous vérifierez que vous obtenez une expression de la forme $C(2^p) = a + b2^p + 2C(2^{p-1})$, où a et b sont des constantes entières.
(ii) En notant C(1)=λ on pourrait alors vérifier, en calculant de proche en proche les $C(2^p)$, que pour tout entier naturel p, $C(2^p) = (a + \lambda)2^p + bp2^p - a$
En supposant que n est une puissance de 2, en déduire C(n) en fonction de n.

On a donc montré que la complexité, même dans le pire des cas, est en $O(n\log(n))$ (**complexité quasi-linéaire).**

IV. Tri par insertion

1) Présentation

Le tri par insertion est sans doute le plus naturel. Il consiste à insérer successivement chaque élément dans l'ensemble des éléments déjà triés. C'est le procédé que l'on utilise, en général, pour classer un jeu de cartes (on pioche les cartes une par une et on range chaque carte piochée dans son jeu déjà trié)
Le tri par insertion s'effectue ***en place***, son coût en mémoire est donc constant.
Son principe est d'insérer successivement chaque élément a[i] dans la portion de la liste située avant l'indice i, qui est déjà triée.

Exemple :

3	7	2	6	5	1	4

2) Programmation

Créer une fonction **`tri_insertion`** d'argument une liste **`a`**, en suivant les principes :

- Utiliser une boucle ***for*** pour parcourir les éléments successifs de la liste qui sont à trier.
- A chaque tour de cette boucle ***for,*** pour insérer l'élément a[i] au bon endroit dans la partie de liste déjà triée :
  - Stocker l'élément a[i] à insérer dans une variable x
  - Parcourir les éléments de la liste situées avant l'indice i de la droite vers la gauche, et tant que vous n'avez pas atteint le début de la liste, et tant que les éléments rencontrés sont plus grands que x, vous les décalez d'un cran vers la droite.
  - Vous insérez alors x au bon endroit

Cette fonction retourne la liste **`a`** qui a été modifiée.

3) complexité

Déterminer la complexité de ce programme dans le meilleur et dans le pire des cas. (vous l'exprimerez en fonction de la taille n du tableau à trier)
Bilan : Dans le pire des cas, la **complexité est quadratique**, d'où un temps d'exécution qui peut être long.
Cependant, le tri par insertion présente des avantages : il peut être facilement mis en œuvre pour **trier des valeurs au fur et à mesure de leur apparition** (cas des algorithmes en temps réel où il faut parfois exploiter une série de valeurs triées qui vient s'enrichir, au fil du temps, de nouvelles valeurs).
Il peut également facilement s'adapter si on ne veut pas trier toute une liste mais seulement récupérer les k plus petits éléments (algorithme des k plus proches voisins par exemple).
Notons enfin que la complexité est meilleure quand le tableau initial possède un "certain ordre".

V. Tri par sélection

1) Présentation

Voici le principe du tri par sélection : on trie progressivement le tableau, en le parcourant de gauche à droite, en déterminant la plus petite des valeurs non encore triées et en la mettant à sa place.
Plus précisément, étant donné un tableau t de taille n, pour k allant de 0 à (n-2) :

- Déterminer un indice j tel que t[j] soit la plus petite des valeurs de la portion du tableau t entre les indices k et n-1 inclus.
- Echanger t[k] et t[j]

Exemple :

3	7	2	6	5

Remarque : Le tri par insertion s'effectue ***en place***, son coût en mémoire est donc constant.

2) Programmation

1ère étape : Création d'une fonction auxiliaire
Ecrire une fonction `indice_min` recevant comme paramètre un tableau t de longueur n et deux entiers g et d tels que $0 \leq g \leq d < n$, et renvoyant j tel que t[j] soit la plus petite valeur de la partie du tableau t comprise entre les indices g et d inclus.
2ème étape: Version itérative du tri par sélection
Programmer une version itérative du tri par sélection. Evaluer la complexité de cette fonction.
(il existe aussi une version récursive, mais la complexité sera la même)

3) complexité

Montrer que la complexité du tri par sélection est dans le pire des cas quadratique.

BILAN :

Tri rapide	
Algorithme récursif	
Meilleur des cas et en moyenne	$O(nlog(n))$
Pire des cas	$O(n^2)$

Tri fusion
Algorithme récursif
Dans tous les cas : $O(nlog(n))$

Tri par insertion	
Non récursif	
Meilleur des cas (tableau déjà trié).	$O(n)$
Pire des cas (tableau dans l'ordre décroissant)	$O(n^2)$

Tri par sélection
Itératif ou récursif
Dans tous les cas : $O(n^2)$