08 - Les algorithmes de tri

1 Introduction

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.
Nous présenterons ici cinq types d'algorithme de tri :

- Le **tri-rapide** et le **tri par fusion**. (qui utilisent tous les deux le principe *"diviser pour régner"*)
- le **tri par insertion**, le **tri à bulle** 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).
  Lorsqu'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 $O(n\log(n))$, dite quasi-linéaire), mais pas toujours autorisées 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]
```

2 Tri rapide (ou QuickSort)

2.1 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****.*

2.2 Exemple :

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

2.3 Programmation

2.3.1 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`.

2.3.2 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_triée+ [pivot] + c_triée`»
  Remarquons que les appels récursifs s'arrêteront forcément puisque les deux sous listes sont de taille strictement inférieure à la liste initiale.

2.4 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.

Il est cependant possible d'effectuer un tri rapide en place mais l'algorithme est un peu plus difficile à construire.

2.5 Complexité temporelle

- Déterminons la complexité $P(n)$ de la fonction partition dans le cas d'une liste de taille n :

- 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".

  Notons $C(n)$ la complexité dans le pire des cas de la fonction tri_rapide pour trier une liste de taille n.

  Déterminons une relation de récurrence entre $C(n)$ et $C(n-1)$:

- Si on note $\mathrm{C}(0) = \lambda$, on pourrait alors vérifier, en calculant de proche en proche les $\mathrm{C}(n)$, que $C(n) = na + \dfrac{n(n+1)}{2}b + \lambda$.

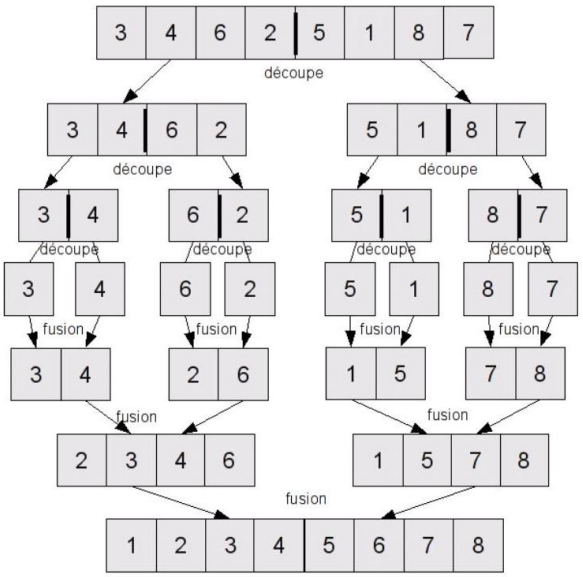
On en déduit donc que la complexité dans le pire des cas est **quadratique**.
Cependant, on pourrait montrer que sa **complexité moyenne** (notion hors programme) est un $\boldsymbol{O(nlog(n))}$, ce qui est en pratique assez rapide (d'où le nom de cette méthode de tri !). On parle de **complexité quasi-linéaire**.

3 Tri par fusion

3.1 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 :



3.2 Programmation

3.2.1 Fonction fusion

Il s'agit de créer une fonction qui à partir de deux listes `a` et `b` **déjà triées**, retourne une liste triée contenant tous les élemetns de `a` et tous els éléments de `b` .

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 $a[i] \leq b[j]$, on ajoute $a[i]$ à la liste c et on incrémente l'indice i de 1.
  - Si $a[i] > b[j]$, on ajoute $b[j]$ à la liste c, et on et on incrémente l'indice j de 1 .
- 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] :

Programmer une fonction `fusion`, ayant comme entrée deux listes `a` et `b`, supposés déjà triées, et qui les fusionne en retournant la nouvelle liste obtenue.

3.2.2 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 obtenues.

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

3.3 Complexité

- Déterminons la complexité $\mathrm{F(m)}$ de la fonction fusion dans le cas où on fusionne deux listes de taille m

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 $C\left(2^p\right)$ cette complexité.

- Exprimons $C\left(2^p\right)$ en fonction de p et de $C\left(2^{p-1}\right)$.

- En notant $\mathrm{C(1)} = \lambda$ on pourrait alors vérifier, en calculant de proche en proche les $C\left(2^p\right)$, que pour tout entier naturel p, $C\left(2^p\right) = (a+\lambda)2^p + bp2^p - a$
- En supposant que n est une puissance de 2 , déterminons $C(n)$ en fonction de n :

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

4 Tri par insertion

4.1 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

4.2 Programmation

Description de l'algorithme :

- 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.

```
1  def tri_insertion (a):
2      ''' retourne un tableau avec les mêmes éléments triés par ordre croissant '''
3
4      n=len(a)
5
6      for i in range(1,n):   #on parcourt un par un tous les éléments de la liste
7          x=a[i]                  #x est l'élément à insérer dans la partie de liste déjà triée
8          j=i                        #on va parcourir la partie de liste déjà triée ,
9                                          #de la droite vers la gauche
10         while j>0 and x<a[j-1]: #tant qu'on est pas au début de la liste
11                                         #et que les elts rencontrés sont plus grand que x
12             a[j]=a[j-1]           #on décale vers la droite les éléments
13             j=j-1
14         a[j]=x    #on insère x à la bonne place
15     return a
```

4.3 Complexité

- Dans le meilleur des cas : la liste est déjà triée, la boucle for s'exécute $(n-1)$ fois, mais la boucle while ne s'exécute jamais, la complexité est donc **linéaire**.
- Dans le pire des cas : la liste est triée par ordre décroissant, la boucle for s'exécute $(n-1)$ fois, et la boucle while va tourner un maximum de fois à chaque fois (car il faudra remonter toute le liste pour bien placer l'élément x). Plus précisément, pour la boucle for d'indice i (i compris entre 11 et $(n-1)$), la boucle while va tourner i fois.
  La complexité sera donc en $O\left(\sum_i = 1^{n-1} i\right) = O\left(\frac{(n-1)n}{2}\right) = O\left(n^2\right)$, donc **quadratique**.

Remarque : Même si dans le pire des cas, la complexité est quadratique, 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).

5 Tri à bulle

5.1 Présentation

Voici le principe du **tri à bulle** : on parcourt la liste plusieurs fois de gauche à droite, en comparant chaque paire d'éléments consécutifs. Si les éléments ne sont pas dans l'ordre souhaité, on les échange.

Ainsi, à chaque passage, le plus grand élément "remonte" vers la fin de la liste, comme une bulle d'air dans l'eau.
Ce processus est répété jusqu'à ce que la liste soit entièrement triée.

Plus précisément, étant donné une liste de taille n, pour i allant de 0 à $n-2$:

- Pour chaque indice j allant de 0 à $n-i-2$:
  - Comparer $a[j]$ et $a[j+1]$.
  - Si $a[j] > a[j+1]$, échanger $a[j]$ et $a[j+1]$.

Exemple :

5	3	8	4	2
3	5	4	2	8
3	4	2	5	8
3	2	4	5	8
2	3	4	5	8

Remarque : Le tri à bulle s'effectue en place, son coût en mémoire est donc constant.

5.2 Programmation

```
1  def tri_bulle(t):
2      n = len(t)
3      for i in range(n-1):
4          for j in range(n-i-1):
5              if t[j] > t[j+1]:
6                  t[j], t[j+1] = t[j+1], t[j]
```

5.3 Complexité

La complexité du tri à bulle est dans le pire des cas **quadratique**, c'est-à-dire $O(n^2)$. En effet, pour chaque passage i, on effectue $n-i-1$ comparaisons, ce qui donne un total de $\frac{n(n-1)}{2}$ comparaisons. Ainsi, la complexité est bien $O(n^2)$.

Remarque : Dans sa version naïve, le tri à bulles effectue systématiquement un nombre quadratique de comparaisons, même lorsque le tableau est déjà trié. Il est cependant possible d'améliorer cet algorithme en introduisant un *drapeau* (variable booléenne) permettant de détecter si un échange a eu lieu lors d'un parcours du tableau.

À chaque itération, le drapeau est initialisé à `faux`. Si au moins un échange est effectué pendant le parcours, le drapeau est mis à `vrai`. Dans le cas contraire, cela signifie que le tableau est déjà trié et l'algorithme peut être arrêté prématurément.

Grâce à cette optimisation, le tri à bulles a une complexité linéaire $O(n)$ dans le meilleur des cas (tableau déjà trié), tout en conservant une complexité quadratique $O(n^2)$ dans le pire des cas.

6 Tri par sélection

6.1 Présentation

Voici le principe du tri par sélection : on trie progressivement la liste, 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 une liste de taille n , pour k allant de 0 à $n-2$:

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

Exemple :

3	7	2	6	5
2	7	3	6	5
2	3	7	6	5
2	3	5	6	7
2	3	5	6	7

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

6.2 Programmation

- **Création d'une fonction auxiliaire :** Ecrire une fonction `indice_min` recevant comme paramètre une liste `a` de longueur n et deux entiers g et d tels que $0 \leq g \leq d < n$, et renvoyant j tel que $a[j]$ soit la plus petite valeur de la partie de la liste `a` comprise entre les indices g et d inclus.
- Programmer une version itérative du tri par sélection.

6.3 Complexité

Montrer que la complexité du tri par sélection est dans le pire des cas **quadratique**.
(il existe aussi une version récursive, mais la complexité sera la même)

7 BILAN :

Algorithme	**Type**	**Meilleur cas**	**Cas moyen**	**Pire cas**
Tri rapide	Récursif	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Tri fusion	Récursif	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Tri par insertion	Itératif	$O(n)$	$O(n^2)$	$O(n^2)$
Tri par sélection	Itératif / récursif	$O(n^2)$	$O(n^2)$	$O(n^2)$
Tri à bulles	Itératif	$O(n)$	$O(n^2)$	$O(n^2)$