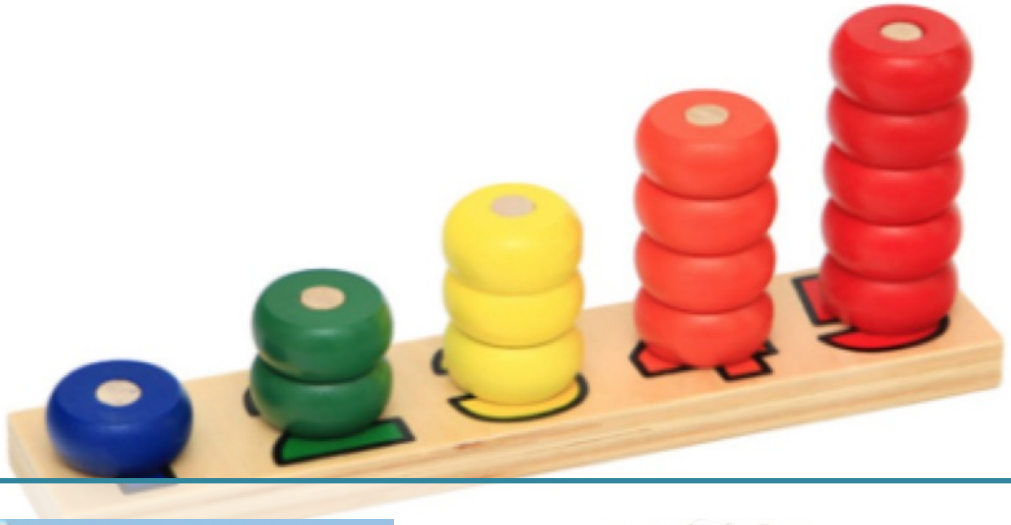
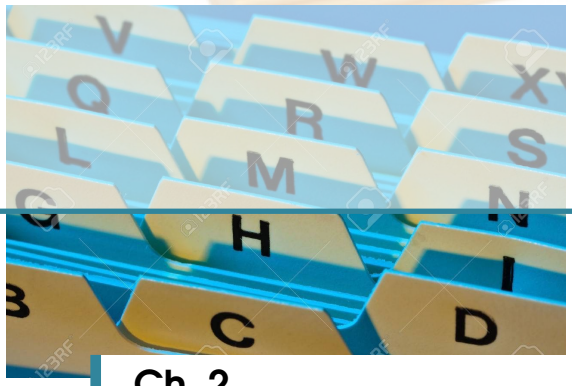


Sem. 2



Informatique

Cours

Ch. 2
Méthodes de tri

E. Durif – X. Pessoles
J.-P. Berne
MPSI

1 Introduction et objectifs

Objectif Un algorithme de tri est un algorithme permettant d'organiser une liste d'éléments selon un ordre fixé. On peut dire que les éléments à trier feront partie d'un ensemble E muni d'une relation d'ordre total noté $\leq$.

Les ensembles $\mathbb{N}$, $\mathbb{R}$... sont munis de l'ordre $\leq$.

L'ensemble des chaînes de caractères peut être muni de l'ordre lexicographique (ordre du dictionnaire). Ainsi en Python, `'a'<'b'`, `'aa'<'b'`, `'A'<'a'` (les lettres majuscules sont avant les lettres minuscules).

Les principales capacités développées ici sont :

- Comprendre un algorithme de tri et expliquer ce qu'il fait.
- S'interroger sur l'efficacité algorithmique temporelle d'un algorithme.
- Programmer un algorithme dans un langage de programmation moderne et général.

Définition **Effet de bord** – On dit qu'une fonction est à effet de bord lorsqu'elle modifie une variable en dehors de son environnement local. C'est par exemple le cas lorsqu'on donne une liste (objet mutable, passé par référénce) comme argument d'une fonction.

Conséquence sur les algorithmes de tri : une fonction de tri ne renvoie rien la plupart du temps (on peut donc l'appeler une procédure). La liste passée en argument en entrée sera triée et cela, même en dehors du scope de la fonction.

Définition **Tri en place** – Un tri est effectué en place lorsque la liste à trier est modifiée jusqu'à devenir triée. Dans le cas contraire, la fonction de tri pourra renvoyer une novelle liste contenent les mêmes éléments, mais triés.

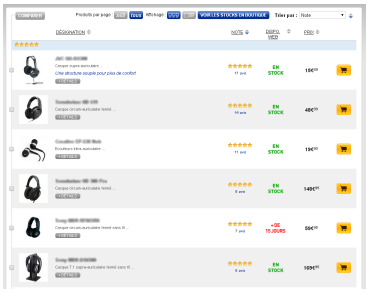
Définition **Tri stable** – Un algorithme est dit stable si les positions relatives de deux éléments égaux ne sont pas modifiées par l'algorithme : c'est-à-dire que si deux éléments x et y égaux se trouvent aux positions i_x et i_y de la liste avant l'algorithme, avec $i_x < i_y$, alors c'est également le cas de leurs positions après l'algorithme.

Définition **Tri comparatif** – Un tri est dit comparatif lorsqu'il s'appuie uniquement sur la comparaison deux à deux des éléments de la liste et pas sur la valeur ce ces éléments.

1.1 Exemple d'application

Exemple 1 : *tri d'avis sur un site marchand*

On ne s'intéresse ici qu'à une version simplifiée de l'exemple. On ne se focalise que sur le critère note et plus précisément sur le nombre d'avis.
Avant que l'utilisateur choisisse de classer par notes on a une liste non triée $T = [5, 14, 11, 8, 17, 7]$ et l'on souhaite obtenir $T = [5, 7, 8, 11, 14, 17]$.



1.2 Complément sur la complexité

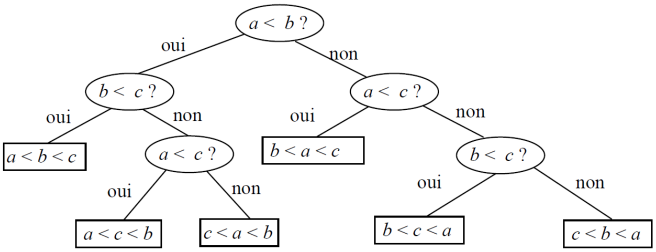
Définition 1 : *Notions de complexité*

La complexité permet de donner un ordre de grandeur du nombre d'opérations qu'effectue un algorithme.

- La complexité permet de comparer la performance des algorithmes.
- Elle s'estime en fonction d'une donné type d'un problème par exemple la taille d'une liste `n`.
- On parlera de complexité :
  - **linéaire** si le nombre d'opération de l'ordre de n et on la notera $o(n)$;
  - **quadratique** si le nombre d'opération de l'ordre de n^2 et on la notera $o(n^2)$;
  - **cubique** si le nombre d'opération de l'ordre de n^3 et on la notera $o(n^3)$;
  - **logarithmique** si le nombre d'opération de l'ordre de $log(n)$ et on la notera $log(n)$;

Exemple 2 : *Tri comparatif*

Un **algorithme comparatif** (à opposer au non-comparatif comme le tri baquet ou par paquet) est basé sur des comparaisons successives entre les données pour déterminer la permutation correspondant à l'ordre croissant des données. On va chercher ici à évaluer la complexité théorique des tris comparatifs en se basant sur le nombre de comparaisons.
Cet algorithme correspond au fonctionnement du tri insertion que nous verrons plus loin :



Résultat : *Tout algorithme comparatif fait dans le pire des cas au moins de l'ordre de* $n.log_2(n)$ ***comparaisons*** *soit une complexité de* $O(N \cdot log(N))$.

2 Tri par insertion

2.1 Principe

Définition 2 : *Principe du tri par insertion*

Le **tri par insertion** est le tri que l'on effectue naturellement, par exemple pour trier un jeu de cartes. On trie les premières puis à chaque nouvelle carte on l'ajoute à l'ensemble déjà trié à la bonne place.
Ce trie s'effectue en place, c'est-à-dire qu'il ne demande pas d'autre tableau que celui que l'on trie. Son coût en mémoire est donc constant si on ne compte pas la place occupée par les données.

Exemple 3 : *Illustration du tri par insertion*

On représente en rouge les valeurs triées. La clef est en bleu. Les données décalées sont en vert.

Tableau T initial : 5 est la clef.	5	14	11	8
La clef est comparée avec la liste des valeurs déjà triées (vide ici).	5	14	11	8
La clef est 14	5	14	11	8
On compare la clef aux valeurs déjà triées (ici 5). Elle est plus grande et reste donc à sa position.	5	14	11	8
La clef est 11.	5	14	11	8
On compare la clef aux valeurs déjà triées : la première valeur comparée est 14. 14 est plus grand donc on décale 14 vers la droite pour laisser une place à la clef.	5	14	14	8
On compare la clef aux valeurs déjà triées : la valeur suivante est 5. 5 est plus petit donc on a trouvé la place de la clef et on l'écrit.	5	11	14	8
La clef est 8.	5	11	14	8
On compare la clef 8 à 14. 14 est plus grand donc on le décale en le copiant à la place de la clef.	5	11	14	14
On compare la clef 8 à 11. 11 est plus grand donc on le décale en le copiant à la place de la valeur supérieure (on écrase donc une des occurrences de 14).	5	11	11	14
On compare la clef 8 à 5. 5 est plus petit donc on est à la bonne place pour copier la clef (on écrase donc l'occurrence de 11 en doublon).	5	8	11	14

2.2 Implémentation

for $i \leftarrow 1$ **to** $n-1$ **do**
  $j \leftarrow i$;
  $clef \leftarrow L[i]$;
  while $j > 1$ *and* $clef < L[j-1]$ **do**
    $L[j] \leftarrow L[j-1]$;
    $j \leftarrow j-1$;
  end
  $L[j] \leftarrow clef$;
end
Output: L

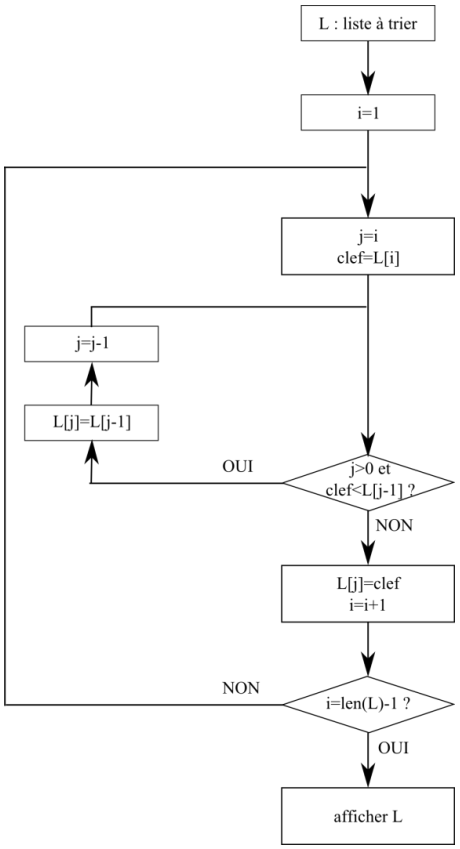


FIGURE 1: Algorigramme du tri par insertion

- On stocke les données dans un tableau L entre les indices 0 et $n-1$.
- On utilise deux variables notées i, j et une variable $clef$ (la clef) du même type que les données de la liste.
- On réalise une première boucle qui va permettre de parcourir tous les éléments $L[i]$ du tableau à trier. Chacun de ces éléments sera alors successivement la clef.

- La deuxième boucle (*while*) va nous permettre pour chaque valeur de la clef de tester successivement les valeurs déjà triées jusqu'à trouver une valeur inférieure. On aura ainsi déterminé la place de la clef.
- A chaque fois qu'une valeur est plus grande que la clef on la décale vers la droite pour laisser la place d'écrire la clef. Dès que l'on arrive sur une valeur inférieure on a trouvé la bonne place pour écrire la clef.

Algorithme 1 :

Écriture sous Python de l'algorithme tri par insertion

```
def tri_insertion(T,n):
    for i in range(1,n):
        j=i
        v=T[i]
        while j>0 and v<T[j-1]:
            T[j]=T[j-1]
            j=j-1
        T[j]=v
    return T
```

Exemple 4 : ***Application avec affichage pas a pas sur l'exemple***

On insère les lignes *print* pour réaliser l'affichage

```
i= 1 j= 1 v= 14 T= [5, 14, 11, 8, 17, 7]
i= 1 j= 1 v= 14 T= [5, 14, 11, 8, 17, 7]
i= 2 j= 2 v= 11 T= [5, 14, 11, 8, 17, 7]
i= 2 j= 1 v= 11 T= [5, 14, 14, 8, 17, 7]
i= 2 j= 1 v= 11 T= [5, 11, 14, 8, 17, 7]
i= 3 j= 3 v= 8 T= [5, 11, 14, 8, 17, 7]
i= 3 j= 2 v= 8 T= [5, 11, 14, 14, 17, 7]
i= 3 j= 1 v= 8 T= [5, 11, 11, 14, 17, 7]
i= 3 j= 1 v= 8 T= [5, 8, 11, 14, 17, 7]
i= 4 j= 4 v= 17 T= [5, 8, 11, 14, 17, 7]
i= 4 j= 4 v= 17 T= [5, 8, 11, 14, 17, 7]
i= 5 j= 5 v= 7 T= [5, 8, 11, 14, 17, 7]
i= 5 j= 4 v= 7 T= [5, 8, 11, 14, 17, 17]
i= 5 j= 3 v= 7 T= [5, 8, 11, 14, 14, 17]
i= 5 j= 2 v= 7 T= [5, 8, 11, 11, 14, 17]
i= 5 j= 1 v= 7 T= [5, 8, 8, 11, 14, 17]
i= 5 j= 1 v= 7 T= [5, 7, 8, 11, 14, 17]
>>>
```

2.3 Complexité

Propriété Complexité

La complexité est dans le pire des cas quadratique : $C(n) = O(n^2)$**.**

	meilleur cas	pire cas
comparaisons	n	$n^2/2$
affectations	n	$n^2/2$

L'efficacité du tri par insertion est excellente lorsque le tableau est déjà trié ou « presque trié » ($C(n) = O(n)$). Il surpasse alors toutes les autres méthodes de tri qui sont au mieux en $O(n \times ln(n))$.

3 Tri rapide (ou "quicksort")

3.1 Principe

Définition 3 : *Tri rapide*

L'algorithme de tri rapide fait partie de la catégorie des algorithmes « diviser pour régner ».
A chaque appel de la fonction tri on choisit une valeur **"pivot"**, par exemple le premier élément. On effectue une partition des élément à trier. Un premier groupe est constitué de valeurs inférieures au pivot et un deuxième avec les valeurs supérieures. Le pivot est alors placé définitivement dans le tableau.
On traite alors chacun des groupes de façon indépendante. On peut les traiter avec le même algorithme.

Exemple 5 : *Illustration du tri rapide*

On représente en :
- bleu, le pivot ;
- vert, les valeurs triées ;
- rouge, les valeurs de gauche d'une segmentation ;
- jaune, les valeurs de droite d'une segmentation.

Tableau T initial : 5 est le pivot	5	14	11	8	17	7
On partitionne pour trouver une partie droite (en jaune) et une partie gauche (en rouge). Sauf qu'ici il n'y aura pas de partie gauche car 5 est la plus petite valeur. 5 est donc à sa place définitive.	5	14	11	8	17	7
On traite de façon indépendante les partie gauche et droite. Ici pas de partie gauche. On cherche alors les nouveaux pivots (ici 14).	5					
		14	11	8	17	7
On segmente alors les parties droite et gauche autour du pivot 14 dont on a ainsi déterminé la position définitive.	5					
		7	8	11	14	17
On traite de façon indépendante les partie gauche et droite. On recherche pour chacune le pivot (ici 7 et 17).	5				14	
		7	11	8		
						17
Pour la chaine de pivot 7 on segmente mais il n'y a pas de partie gauche. Le pivot 17 est seul dans sa chaine il est donc directement à la bonne position.	5				14	
		7	11	8		
						17
On traite de façon indépendante les partie gauche et droite. Ici pas de partie gauche. On cherche alors les nouveaux pivots (ici 8).	5	7			14	17
			8	11		
On classe le seul élément restant par rapport au pivot.	5	7			14	17
			8	11		

3.2 Implémentation

- On crée tout d'abord une fonction "**partition**".
  - Les arguments de cette fonction seront le tableau a et deux indices correspondant au début et à la fin de la partie à segmenter du tableau (par convention g inclus et **d** exclu). En effet on ne segmente pas à chaque fois l'ensemble du tableau.
  - On suppose qu'il y a au moins un élément dans ce segment ce que l'on peut vérifier avec $assert$:
    ```
    assert g<d
    ```
  - On choisit $a[g]$ comme pivot que l'on peut stocker dans une variable v.
  - Le principe consiste alors à parcourir le tableau de la gauche vers la droite, entre les indices $g+1$ et d avec une boucle for.
  - on stockera dans deux sous listes $ainf$ et $asup$ toutes les composantes respectivement inférieures ou supérieures au pivot.

- on rassemblera les deux sous listes ainsi que le pivot pour donner la liste a partitionnée (en assemblant également le reste de la liste a).
- On note m l'indice qui correspond au partitionnement de a.
- On sortira de cette fonction m et le tableau a.

Algorithme 2 : *Algorithme de partitionnement pour le tri rapide*

```
def partition(a,g,d):
    assert g<d
    v=a[g]
    ainf=[]
    asup=[]
    for x in a[g+1:d]:
        if x<v:
            ainf.append(x)
        elif x>v:
            asup.append(x)
    a=a[0:g]+ainf+[v]+asup+a[d:len(a)]
    m=len(ainf)+g
    return m,a
```

Algorithme 3 :

Algorithme récursif du tri rapide

```
def tri_rapide(a,g,d):
    if g>=d-1:
        return
    else:
        m,a=partition(a,g,d)
        tri_rapide(a,g,m)
        tri_rapide(a,m+1,d)
```

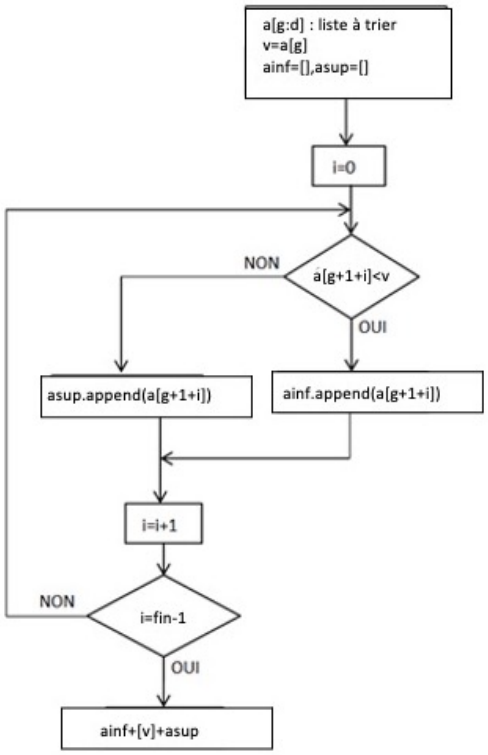


FIGURE 2: Algorigramme du tri par insertion

La fonction tri rapide est alors implémentée de façon récursive.

- On utilise pour cela une fonction tri_rapide qui prend les même arguments que la fonction $partition$.
- **Si** $g \geq d-1$, il y a au plus un élément à trier donc rien à faire (initialisation de la récursivité),
- **Sinon** il faut
  - partitionner les élément entre g et d,
  - faire un appel récursif pour trier $a[g \cdots m[$ (partie gauche),
  - faire un appel récursif pour trier $a[m+1 \cdots d[$ (partie droite),

Pour trier un tableau il suffit d'appeler tri_rapide sur la totalité des éléments.

3.3 Complexité

Propriété Efficacité de l'algorithme

Cette méthode de tri est très efficace lorsque les données sont distinctes et non ordonnées. La complexité est alors globalement en O(n ln(n)). Lorsque le nombre de données devient petit (<15) lors des appels récursifs de la fonction de tri, on peut avantageusement le remplacer par un tri par insertion dont la complexité est linéaire lorsque les données sont triées ou presque. [**Beynet**]

- D'autre part si le tableau est déjà trié avec le code mis en place on tombe sur la complexité "dans le pire des cas". Une solution simple consiste à ne pas choisir systématiquement le premier élément du segment comme pivot, mais plutôt un élément au hasard. On peut par exemple choisir la valeur de façon aléatoire. [**wack**]

meilleur cas	pire cas
$nlogn$	$n^2/2$

4 Tri par fusion

4.1 Principe

Cet algorithme fait aussi partie des algorithmes "diviser pour régner".

Le principe consiste à couper le tableau de départ en deux. On trie chacun des groupes indépendamment. Puis on fusionne les deux groupes en utilisant le fait que chacun des groupe est déjà ordonné.

Il est possible pour réaliser l'ordonnancement de chacun des groupes d'utiliser à nouveau l'algorithme de tri de façon récursive.

Exemple 6 : *Illustration sur l'exemple*

On représente en rouge le groupe de gauche, en jaune le groupe de droite, en bleu la valeur testée du groupe de gauche et en gris la valeur testée du groupe de droite. Les valeurs en vert sont les valeurs définitivement triées.

On notera que pour chaque étape, on utilise deux tableaux, le tableau de départ sur la ligne supérieure et le tableau d'arrivée sur la ligne inférieure.

Tableau T initial : On découpe en une deux moitiés (rouge et jaune).	5	14	11	8	17	7
Chacune de ces deux listes est triée de façon indépendante (pas illustré ici) soit par un autre algorithme soit de façon récursive.	5	11	14	7	8	17
On procède alors à l'opération de fusion en comparant les premiers nombres de chaque bloc. On a besoin d'un second tableau que l'on	5	11	14	7	8	17
remplit après les comparaisons successives des éléments des listes L_1 (rouge) et L_2 (jaune). 5<7 on le stock de façon définitive dans T_2.		11	14	7	8	17
	5					
On continue la fusion en comparant les premiers nombres restants dans chaque bloc (ici 7 et 11).		11	14	7	8	17
	5					
7<11 on le stocke dans le deuxième tableau.		11	14		8	17
	5	7				
On continue la fusion en comparant les premiers nombres restants dans chaque bloc (8 et 11).		11	14		8	17
	5	7				
8<11 on le stocke dans le deuxième tableau.		11	14			17
	5	7	8			
On continu la fusion en comparant les premiers nombres restants dans chaque bloc (11 et 17).		11	14			17
	5	7	8			
11<17 on le stocke dans le deuxième tableau.			14			17
	5	7	8	11		
On continu la fusion en comparant les premiers nombres restants dans chaque bloc (14 et 17).			14			17
	5	7	8	11		
14<17 on le stock dans le deuxième tableau et comme c'est le dernier restant on ajoute 17.						
	5	7	8	11	14	17

4.2 Implémentation

On souhaite trier un tableau a :

- Le caractère récursif du tri fusion fait que l'on s'intéresse seulement à la partie de a entre les indices g (inclus) et d (exclu).
- Afin de découper le groupe à trier en deux, on calcule l'indice médian :

$$m = \frac{g+d}{2}$$

.

- On trie alors récursivement les deux parties :
  - délimitées par g et m d'une part,
  - et m et d d'autre part.
  - **Si** $g \geq d-1$, il y a au plus un élément à trier donc rien à faire (initialisation de la récursivité).
- Il reste à effectuer la fusion. On utilise pour cela un second tableau a_0, alloué une et une seule fois au début du tri.
- On réalise la fonction fusion :

- qui prend en arguments deux tableaux, $a0$ et a, et les trois indices g, m et d.
- Les portions $a0[g \cdots m[$ et $a0[m \cdots d[$ sont supposées triées.
- On veut les fusionner dans $a[g \cdots d[$.
- On va parcourir chacune des listes déjà triées en utilisant un indice i pour L_1 et j pour L_2.
- On compare à chaque itération de la boucle **for** $a0[i]$ et $a0[j]$ et on place la plus petite des valeurs dans $a[k]$ et on incrémente l'indice correspondant i ou j selon celui que l'on a déplacé.
- On traite aussi le cas ou une des deux listes est vide.

Exemple 7 : *Algorithme complet du tri par fusion*

```
#Definition des fonctions

def tri_fusion(a,g,d):
    a0=a[:]
    if g>=d-1:
        return
    else:
        m=(g+d)//2
        tri_fusion(a,g,m)
        tri_fusion(a,m,d)
        a0[g:d]=a[g:d]
        fusion(a0,a,g,m,d)

def fusion(a0,a,g,m,d):
    i,j=g,m
    for k in range(g,d):
        if i<m and (j==d or a0[i]<=a0[j]):
            a[k]=a0[i]
            i=i+1
        else:
            a[k]=a0[j]
            j=j+1

#Programme principal
a=[5,14,11,8,17]
#a=[4,6,3,7,5]
a=[7,6,4,3,2,1]
tri_fusion(a,0,len(a))
```

S2

4.3 Complexité

Propriété Efficacité de l'algorithme
La colmplexité est $nlog(n)$

meilleur cas	pire cas
$nlogn$	$nlogn$

5 Dans Python : sort et sorted

Les listes Python ont une méthode native `list.sort()` qui modifie les listes elles-mêmes et renvoie **None**.

```
>>> a=[25,94,89,113,67]
>>> a.sort()
>>> a
25,67,89,94,113
```

Il y a également une fonction native `sorted()` qui construit une nouvelle liste triée depuis un itérable (liste de listes, tuple, dictionnaires).

```
>>> sorted([25,94,89,113,67])
25,67,89,94,113
```

On utilise l'indice de la "colonne" à trier en utilisant la fonction `lambda` associée à `key` :

```
>>> etudiants=[('Julie','MPS1',15),('Elio','MPS2',14),('Jules','MPS1',17),('Adam','MPS2',16)]
>>> sorted(etudiants, key=\textbf{lambda} etudiants : etudiants[2])
('Elio','MPS2',14),('Julie','MPS1',15),('Adam','MPS2',16),('Jules','MPS1',17)
>>> etudiants # etudiants est inchangé
(Julie','MPS1',15),('Elio','MPS2',14),('Jules','MPS1',17),('Adam','MPS2',16)
```

Sans indication particulière, le tri se fait sur la première valeur puis sur la suivante dans le cas où les premières valeurs sont identiques :

```
>>> couple=[(3,3),(3,6),(3,1)]
>>> sorted(couple)
(3, 1), (3, 3), (3, 6)
>>> le tri est dit stable
```

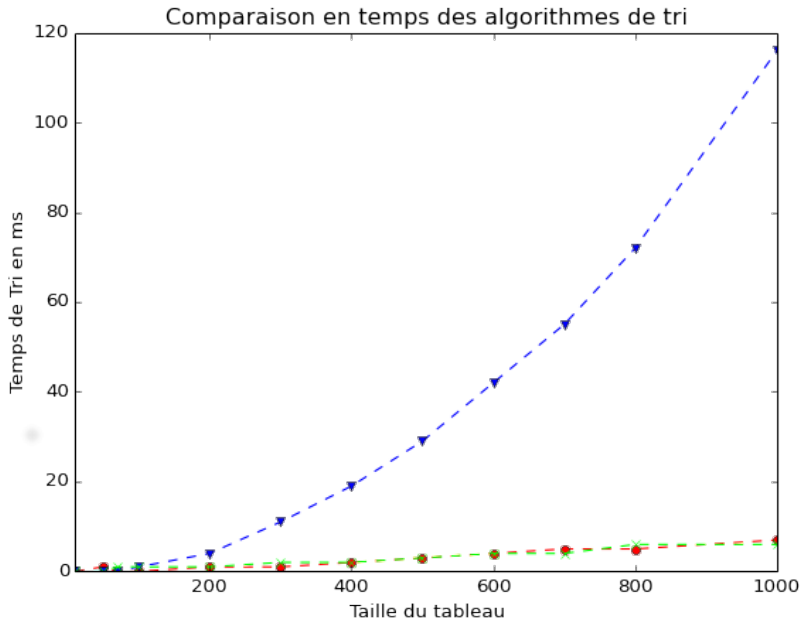
6 Conclusion et méthodes dans Python

Propriété DISTINGUER PAR LEURS COMPLEXITÉS DEUX ALGORITHMES RÉSOLVANT UN MÊME PROBLÈME :
La première étape consiste à vérifier que les algorithmes résolvent exactement le même problème.

- On compare la complexité en temps dans le pire des cas (préférable a l'utilisation du module Time comme ci dessous car indépendant de la machine).
- Il faut vérifier que le pire des cas peut arriver en situation réelle
- Penser à comparer la complexité en espace qui peut permettre de départager des algorithmes équivalents

Exemple 8 : *Illustration de la comparaison*

Pour comparer les temps globaux sur une même machine (Ici processeur Intel Core i7-4510U Haswell (2 GHz, TDP 15W),Mémoire vive 4 Go) on génère des tableaux de dimensions différentes avec des nombres aléatoires. On utilise le module time pour déterminer les temps. Ici c'est un temps global dépendant fortement de la machine utilisée. On fait varier la taille du tableau et on compare les temps mis par chacun des algorithmes (en rouge tri rapide, en bleu tri insertion, en vert tri fusion).



Références

[1] Irène Charon Olivier Hudry, Algorithmes de tri, cours de Télécom ParisTech, Avril 2014.

[2] B. Wack, S. Conchon, J. Courant, M. de Falco, G. Dowek, J.-C. Filliâtre, S. Gonnord, Informatique pour tous en classes préparatoires aux grandes écoles. Manuel d'algorithmique et programmation structurée avec Python. Nouveaux programmes 2013. Voies MP, PC, PSI, PT, TPC et TSI, Eyrolles, 2013.

[3] Beynet Patrick, Cours d'informatique publié sur le site de l'UPSTI.

7 QCM

Question 1 *On dispose d'une liste de triplets : t = [(1,12,250),(1,12,251), (2,12,250),(2,13,250),(2,11,250),(2,12,249)]. On trie cette liste par ordre croissant des valeurs du second élément des triplets. En cas d'égalité, on trie par ordre croissant du troisième champ. Si les champs 2 et 3 sont égaux, on trie par ordre croissant du premier champ. Après ce tri, quel est le contenu de la liste ?*

1. `[(1,12,249),(1,12,250),(1,12,251),(2,11,250),(2,12,250),(2,13,250)]`.
2. `[(2,11,250),(1,12,249),(1,12,250),(2,12,250),(1,12,251),(2,13,250)]`.
3. `[(2,11,250),(1,12,249),(1,12,250),(1,12,251),(2,12,250),(2,13,250)]`.
4. `[(1,12,249),(2,11,250),(1,12,250),(2,12,250),(2,13,250),(1,12,251)]`.

Question 2 *Quelle valeur retourne la fonction "mystere" suivante ?*

```
def mystere(liste):
    valeur_de_retour = True
    indice = 0
    while indice < len(liste) - 1 :
        if liste[indice] > liste[indice + 1]:
            valeur_de_retour = False
        indice = indice + 1
    return valeur_de_retour
```

1. Une valeur booléenne indiquant si la liste liste passée en paramètre est triée.
2. La valeur du plus grand élément de la liste passée en paramètre.
3. La valeur du plus petit élément de la liste passée en paramètre.
4. Une valeur booléenne indiquant si la liste passée en paramètre contient plusieurs fois le même élément.

Question 3 *Combien d'échanges effectue la fonction Python suivante pour trier un tableau de 10 éléments au pire des cas ?*

```
def tri(tab) :
    for i in range (1, len(tab)) :
        for j in range (len(tab) - i) :
            if tab[j] > tab[j+1] :
                tab[j], tab[j+1] = tab[j+1], tab[j]
```

1. 45.
2. 100.
3. 10.
4. 55.

Question 4 *Que vaut l'expression* `f([7, 3, 1, 8, 19, 9, 3, 5], 0)` *?*

```
def f(t,i) :
    im = i
    m = t[i]
    for k in range(i+1, len(t)) :
        if t[k] < m :
            im, m = k, t[k]
    return im
```

1. 1.
2. 2.
3. 3.
4. 4.

Question 5 *Laquelle de ces listes de chaînes de caractères est triée en ordre croissant ?*

1. ['Chat' , 'Cheval' , 'Chien' , 'Cochon'].
2. ['Cochon' , 'Chat' , 'Cheval' , 'Chien'].
3. ['Cheval' , 'Chien‘ , 'Chat' , 'Cochon'].
4. ['Chat' , 'Cochon' , 'Cheval' , 'Chien'].

Question 6 *Laquelle de ces listes de chaînes de caractères est triée en ordre croissant ?*

1. ['12','142','21','8'].
2. ['8','12','142','21'].
3. ['8','12','21','142'].
4. ['12','21','8','142'].

Question 7 *Quelle est la valeur de la variable* `table` *après exécution du programme Python suivant ?*

```
table = [12, 43, 6, 22, 37]
for i in range(len(table) - 1):
    if table [i] > table [i+1] :
        table [i] ,table [i+1] = table [i+1] ,table [i]
```

1. [12, 6, 22, 37, 43].
2. [6, 12, 22, 37, 43].
3. [43, 12, 22, 37, 6].
4. [43, 37, 22, 12, 6].

Question 8 *Un algorithme cherche la valeur maximale d'une liste non triée de taille n. Combien de temps mettra cet algorithme sur une liste de taille $2n$?*

1. Le même temps que sur la liste de taille n si le maximum est dans la première moitié de la liste.
2. On a ajouté n valeurs, l'algorithme mettra donc n fois plus de temps que sur la liste de taille n.
3. Le temps sera simplement doublé par rapport au temps mis sur la liste de taille n.
4. On ne peut pas savoir, tout dépend de l'endroit où est le maximum.

Question 9 *Quel est le coût en temps dans le pire des cas du tri par insertion ?*

1. $\mathcal{O}(n)$.
2. $\mathcal{O}\left(n^2\right)$.
3. $\mathcal{O}\left(2^n\right)$.
4. $\mathcal{O}\left(\log n\right)$.