Il existe de très nombreux algorithmes de tri plus ou moins efficaces. Si l'on se restreint aux algorithmes pour lesquels les seules opérations élémentaires autorisées sont la comparaison de deux valeurs et l'échange de deux éléments du tableau, on peut démontrer qu'un tri nécessite au moins $O(n \ln n)$ opérations pour un tableau de longueur n. Un algorithme n'est considéré efficace que si sa complexité est de cet ordre de grandeur. Les algorithmes naïfs ont la plupart du temps un temps d'exécution en $O(n^2)$ opérations. Dans toute la suite, on note n le nombre d'éléments du tableau.

1 Algorithmes de tri élémentaires

1.1 tri par sélection

Le tri par sélection est sans doute l'algorithme de tri le plus naturel et le plus évident. Il consiste simplement à chercher parmi les éléments du tableau le plus petit (resp. le plus grand) élément et le mettre « à la bonne place » en début (resp. en fin) de tableau. On recommence ensuite l'opération avec les éléments restants dans le tableau. L'invariant de boucle utilisé est donc le suivant par exemple avec la sélection du maximum :

> $\mathcal{P}(i)$: Au début de la boucle d'indice i, les i dernières valeurs du tableau sont rangées par ordre croissant et supérieures aux $n-i$ premières valeurs (qui, elles, ne sont pas triées).

Pour les besoins du programme, on commence par écrire une fonction qui cherche la position du maximum d'un tableau dans un zone délimitée par deux indices `a` et `b`.

```
1 def max(t,a,b):
2     max=a
3     for i in range(a,b+1):
4         if t[max]<t[i]:
5             max = i
6     return max
```

Une fois cette fonction écrite, l'algorithme s'écrit de la manière suivante :

```
1 def tri_selection(t):
2     n=len(t)
3     for i in range(n):
4         p=max(t,0,n-1-i)
5         echange(t,p,n-1-i)
```

Complexité

Il est clair que la fonction `max` est de coût proportionnel à `b-a`. Puisque la boucle ligne 3 du programme `tri_selection` effectue un appel à `max` sur des intervalles de longueur n, puis $n-1$ et ainsi de suite jusqu'à 2, le coût total de l'algorithme est donné par

$$O\left(n + (n-1) + \cdots + 1\right) = O\left(n(n+1)/2\right) = O(n^2)$$

L'algorithme est donc de complexité quadratique, ce qui n'est pas catastrophique, mais pas extraordinaire à coté d'un algorithme de coût quasi-linéaire.

1.2 tri bulle

Le tri par bulle part du principe suivant : si l'on compare les éléments consécutifs du tableau `(t[i],t[i+1])` pour `i` croissant de 0 à $n-2$, en les permutant à chaque fois que `t[i]>t[i+1]`, le plus grand élément se retrouve nécessairement à la fin du tableau. Voici un exemple sur un tableau de taille 5. Les éléments comparés sont en rouge lorsqu'il faut les permuter, et en bleu sinon.

5	1	7	4	6

1	5	7	4	6

1	5	7	4	6

1	5	4	7	6

1	5	4	6	7

En répétant i fois l'opération, les i plus grands éléments se retrouve en bout de tableau, et il suffit donc de répéter l'opération n fois. A noter qu'à la i-ième étape, on peut se contenter d'effectuer seulement les $n-1-i$ premières comparaisons. Sur l'exemple, la fin de l'algorithme s'achève de la manière suivante :

1	5	4	6	7

1	5	4	6	7

1	4	5	6	7

et le reste des opérations ne change plus le tableau. On peut remarquer que pour cet algorithme, l'invariant de boucle reste le même que pour le tri par sélection. La fonction peut ainsi s'écrire de la manière suivante :

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

Complexité

Le coût de la boucle ligne 4 est un $O(n-1-i)$, celui de l'algorithme est donc à nouveau

$$O((n-1)+(n-2)+\cdots+1) \qquad \text{soit} \qquad O(n^2)$$

Asymptotiquement, il ne s'agit donc pas d'une amélioration par rapport au tri par sélection.

1.3 tri par insertion

Ce dernier tri porte également le surnom du « tri du joueur de carte ». Le principe est le suivant : si l'on considére un tableau trié, on peut facilement rajouter une valeur en l'insérant à la bonne position dans le tableau « à la bonne place » tout en décalant les éléments supérieurs d'une case vers la droite. Au début de l'algorithme, sachant que la partie gauche du tableau réduite à l'élément d'indice 0 est triée, il suffit donc « d'insérer » successivement les éléments `t[1], t[2], ..., t[n-1]` dans la partie triée des éléments du début de tableau. Sur l'exemple du tri bulle, le tri par sélection donne donc les étapes suivantes après chaque insertion (en rouge, l'élément à insérer, en bleu la partie triée) :

5	1	7	4	6

1	5	7	4	6

1	5	7	4	6

1	4	5	7	6

1	4	5	6	7

L'invariant de boucle peut être allégé par rapport aux deux algorithmes précédents car, s'il suppose comme les autres qu'une partie du tableau est trié, il ne requiert pas l'hypothèse que les éléments triés soient supérieurs à ce qui reste à explorer dans le tableau. On se contente donc de

$\mathcal{P}(i)$: Au début de la boucle d'indice i, les i premières valeurs du tableau sont rangées par ordre croissant.

La procédure d'insertion peut s'écrire simplement par échanges successifs : tant que l'élément à insérer est strictement supérieur à la valeur qui le précède dans le tableau, on permute les deux éléments jusqu'à arriver à un élément plus petit ou au début du tableau. En suivant ce principe, on se retrouve avec le code suivant :

```
1 def tri_insertion(t):
2     n=len(t)
3     for i in range(1,n):
4         p=i-1
5         while t[p+1]<t[p] and p>=0:
6             echange(t,p,p+1)
7             p-=1
```

Complexité

La boucle `while` ligne 5 peut potentiellement s'exécuter $i-1$ fois pour chaque valeur de i (si le tableau est trié par ordre décroissant par exemple). On retrouve donc dans le pire cas une complexité de l'ordre de $O(n^2)$ comme pour les deux précédents algorithmes.

2 Tris efficaces

2.1 Tri fusion

Le tri-fusion est une application immédiate de la méthode diviser pour régner. Il consiste à trier dans un premier temps les deux moitiés du tableau (de manière récursive) puis à fusionner les deux résultats. La complexité est améliorée par rapport aux algorithmes du tri par insertion ou par selection parce que la fusion peut s'effectuer en temps linéaire.
On écrit un programme qui prend en argument un tableau `t` et 3 entiers `d`, `m` et `f` qui délimitent les parties du tableaux triées que l'on doit fusionner. En d'autres termes,

$$t_d \leq t_{d+1} \leq \cdots \leq t_m \qquad \text{et} \qquad t_{m+1} \leq t_{m+2} \leq \cdots \leq t_f$$

On est amené à créer une liste pour recopier les éléments au fur et à mesure. A noter que l'on effectue la fusion sur place (il n'y a pas de retour, on se contente de modifier le tableau).

```
def fusion(L,d,m,f):
    L2=[];
    i=d; j=m+1
    while i<=m and j<=f:
        if L[i]<L[j]:
            L2.append(L[i]); i+=1
        else:
            L2.append(L[j]); j+=1
    while i<=m:
        L2.append(L[i]); i+=1
    while j<=f:                 # ces deux lignes sont facultatives
        L2.append(L[j]); j+=1   # en pratique
    for k in range(len(L2)):
        L[d+k]=L2[k]
```

Pour rédiger le tri-fusion, il n'y a plus qu'à utiliser une sous-fonction récursive dont le but est de ne trier qu'une partie du tableau. Elle opère de façon récursive en triant la partie gauche, puis la droite, puis en fusionnant les morceaux. On termine en l'appliquant au tableau tout entier.

```
def tri_fusion(L):
    def tri_entre(a,b):
        if a<b:
            m=(a+b)//2
            tri_entre(a,m)
            tri_entre(m+1,b)
            fusion(L,a,m,b)
    tri_entre(0,len(L)-1)
```

2.2 Tri rapide

Le principe du tri rapide (`quick_sort`) consiste essentiellement en une étape de séparation qui consiste à choisir un élément α au hasard du tableau, puis à organiser le tableau de manière à ce que tous les éléments inférieurs à α (resp. supérieurs) soient déplacés à gauche (resp. à droite) de α. L'élément ainsi choisi est appelé le pivot. Pour simplifier, on choisit ici comme pivot le premier élément du tableau.

t_0	t_1	$\cdots$	t_{n-1}	$\longrightarrow$	$\leq t_0$	t_0	$\geq t_0$

Une fois cette étape effectuée, il n'y a plus qu'à trier de manière récursive chacune des parties gauche et droite, l'étape de fusion étant ici inexistante ! Il faut toutefois noter que les parties gauche et droite peuvent être de longueur arbitraire, notamment l'une peut être vide.
Pour écrire le programme, il n'y a donc qu'à rédiger la procédure de séparation (qui s'effectue à nouveau sur place). Cette fonction prend deux arguments `d` et `f` indiquant le début et la fin de la partie à séparer. Le pivot est ici systématiquement pris comme le premier élément de la partie mais on pourrait envisager de le choisir de manière aléatoire. Il est cependant nécessaire de renvoyer la place du pivot à la fin du travail pour savoir avec quels nouveaux indices il convient de travailler lors de l'appel récursif. Pendant l'exécution, on maintient l'invariant de boucle suivant : on a deux références `i` et `m` initialisée à `d`.

$$\forall j \in [\![d+1; m]\!] \quad a_j < a_d \qquad \text{et} \qquad \forall j \in [\![m+1; i]\!] \quad a_j \geq a_d$$

a_d	$< a_d$	$\geq a_d$	
d	m	i	f

La procédure de séparation s'écrit un style impératif, de la manière suivante :

Tant que $i < f$, à chaque tour de boucle :

- soit $a_{i+1} \geq a_d$ et on ne fait rien ;
- soit $a_{i+1} < a_d$ et on l'inverse avec a_{m+1}, puis on augmente m de 1.

Après quoi on incrémente i dans tous les cas.

Une fois la boucle finie, on est dans la configuration

a_d	$< a_l$	$\geq a_l$
d	m	f

et il suffit d'inverser a_d et a_m puis de renvoyer m.

```
def separation(L,d,f):
    m=d; i=d
    while i<f:
        if L[i+1]<L[d]:
            echange(L,i+1,m+1)
            m+=1
        i+=1
    echange(L,d,m)
    return m
```

Le programme `quicksort` s'écrit finalement de la manière suivante :

```
def quicksort(L):
    def tri_entre(a,b):
        if a<b:
            m=separation(L,a,b)
            tri_entre(a,m-1)
            tri_entre(m+1,b)
    tri_entre(0,len(L)-1)
```

3 Stabilité des tris

On conclut ce paragraphe avec la notion de tri stable. Imaginons que la liste à trier soit une liste d'éléments contenant plusieurs informations. A titre d'exemple, considérons la liste

```
L = [("toto", 27, 70, 1.80), ("titi",12,50,1.40), ("pepe",80,60,1.70)]
```

Chaque élément est un 4-uple représentant un individu, son âge, son poids et sa taille (titi est peu être en léger surpoids). Lorsque l'on souhaite trier une telle liste, on peut le faire soit par rapport à n'importe lequel des 4 données (nom, age, poids, taille).
Imaginons maintenant par exemple la liste des couples (élèves, note de maths) de la PC* triée par ordre alphabétique. On désire maintenant la trier par ordre croissant par note de maths décroissante. Il s'avère que cette fois, deux élèves peuvent avoir la même note. On désire que le tri conserve dans ce cas l'ordre alphabétique pour chaque groupe d'élève ayant la même note. Un algorithme de tri est dit **stable** si, à partir de la liste triée par ordre alphabétique, le tri du tableau s'effectue en conservant l'ordre pré-établit selon le premier critère.

Question : Parmi les trois tris précédents, lesquels sont stables ?