Tri par insertion

Principe : On considère une liste *lst* de nombres dont les éléments de rangs 0 à *i*-1 sont déjà triés, pour $i \geq 1$. On compare alors l'élément de rang *i*, noté *x* à celui de rang *i*-1. Si $x \geq$ lst[*i*-1], alors les éléments de rang 0 à *i* sont déjà triés. Sinon, on permute ces deux éléments, et on compare *x* à l'élément de rang *i*-2, puis *i*-3,… ainsi de suite jusqu'à ce que *x* soit à sa place. Les éléments de rang 0 à *i* sont alors triés.
On applique ce procédé en boucle afin de trier tous les éléments de la liste.

Une version Python

```
def tri_insertion(lst):
    for i in range(1, len(lst)):
        x = lst[i]                      # x est l'élément courant
        j = i
        while j > 0 and x < lst[j-1]:   # on compare x avec le précédent
            lst[j] = lst[j-1]           # on décale tous les éléments plus
            j -= 1                      # grand que x vers la droite
        lst[j] = x                      # on insère x à la bonne position
    return lst
```

Une version plus simple mais un peu moins performante

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

Tri rapide

Principe : On considère une liste *lst* de nombres. Si elle contient un seul élément, elle est déjà triée. Sinon, on choisit un élément *x* (le premier par exemple), et on partage *lst*\{*x*} en deux listes *linf* et *lsup*, *linf* contenant les éléments inférieurs ou égaux à *x*, *lsup* ceux supérieurs à *x*. La liste triée s'obtient en recollant les listes *linf*, [*x*] et *lsup*, après avoir trié les listes *linf* et *lsup* suivant le même principe.

Une version Python

```
def tri_rapide(lst):
    if len(lst) <= 1:
        return lst
    pivot = lst[0]
    linf = []
    lsup = []
    for x in lst[1:]:
        if x < pivot:
            linf.append(x)
        else:
            lsup.append(x)
    return tri_rapide(linf) + [pivot] + tri_rapide(lsup)
```

Remarque : on peut remplacer lst[0] et **for** x **in** lst[1:] par lst.pop() et **for** x **in** lst. Le pivot est alors le dernier élément.

Même version en utilisant les indices

```
def tri_rapide(lst):
    if len(lst) <= 1:
        return lst
    pivot = lst[0]
    linf = []
    lsup = []
    for i in range(1, len(lst)):
        if lst[i] < pivot:
            linf.append(lst[i])
        else:
            lsup.append(lst[i])
    return tri_rapide(linf) + [pivot] + tri_rapide(lsup)
```

Une version utilisant les listes en compréhension

```
def tri_rapide(lst):
    if len(lst) <= 1:
        return lst
    else:
        return ( tri_rapide([x for x in lst if x < lst[0]])
                 + [x for x in lst if x == lst[0]]
                 + tri_rapide([x for x in lst if x > lst[0]]) )
```

Tri fusion

Principe :

- Fusion : On considère deux listes de nombres triées *lst1* et *lst2*. On les fusionne en une liste *lst* également triée. On crée la liste *lst* élément par élément en ajoutant à chaque fois le plus petit élément encore disponible dans les listes *lst1* et *lst2*. Ainsi, le premier élément de *lst* sera le plus petit entre le premier élément de *lst1* et le premier élément de *lst2*. Si on le prend dans *lst1*, pour obtenir le deuxième élément de *lst*, on compare le deuxième élément de *lst1* avec le premier élément de *lst2* et ainsi de suite.
- Tri : Pour trier une liste *lst*, si elle contient un seul élément, elle est déjà triée, sinon on la divise en deux listes que l'on triera de la même manière avant de les fusionner.

Une version Python

```
def fusionne(lst1, lst2):
    lst = []
    i1 = i2 = 0
    while i1 < len(lst1) and i2 < len(lst2):
        if lst1[i1] < lst2[i2]:
            lst.append(lst1[i1])
            i1 += 1
        else:
            lst.append(lst2[i2])
            i2 += 1
    lst = lst + lst1[i1:] + lst2[i2:]
    return lst
```

```
def tri_fusion(lst):
    if len(lst) <= 1:
        return lst
    else:
        n = len(lst)//2
        return fusionne(tri_fusion(lst[:n]), tri_fusion(lst[n:]))
```

Remarque : dans l'instruction lst = lst + lst1[i:] + lst2[j:], l'une des deux listes lst1[i:] ou lst2[j:] est vide. On peut aussi tester quelle liste est vide et se contenter d'ajouter l'autre.

Complexité et preuves

On évalue la complexité temporelle de ces algorithmes en comptant le nombre d'affectations ou le nombre de comparaisons. Nous choisirons les comparaisons et nous intéresserons, pour chaque algorithme, à la complexité dans le meilleur des cas, dans le pire des cas et en moyenne.

Tri de tableaux

Contrairement à une liste, un tableau à une taille fixée à la création. On ne peut pas ajouter d'élément (pas de append). En Python, on utilise fréquemment des tableaux numpy (np.array) principalement pour leurs fonctions vectoriels et la rapidité des calculs.

Les implémentations précédentes du tri rapide et du tri fusion, bien qu'utilisant des listes, peuvent prendre des tableaux en arguments. Il peut être intéressant de coder ces tris en utilisant uniquement des tableaux pour faciliter par exemple leur transposition dans d'autres langages (C, Java…).

Tri rapide d'un tableau

La fonction tri_rapide_tab trie le tableau tab original et ne renvoie pas un autre tableau contrairement aux implémentations précédentes du tri rapide.
On peut, si besoin, effectuer une copie du tableau en début de code, travailler sur cette copie et la renvoyer.

L'appel de tri(tab, deb, fin) trie le tableau tab[deb:fin] (donc la tranche du tableau tab des indices deb inclus à fin exclu).
On utilise un tableau temporaire de flottants. Si on veut des éléments du même type que tab, on initialise avec :

```
tab_temp = np.empty(len(tab), dtype=type(tab[0]))
```

```
def tri_rapide_tab(tab):
    tab_temp = np.empty(len(tab))

    def tri(tab, deb, fin):
        if fin - deb < 2:
            return
        pivot = tab[deb]
        i = deb + 1          # indice de tab[deb:fin]
        j = deb              # tab_inf est la partie gauche de tab_temp
        k = fin – 1          # tab_sup est la partie droite de tab_temp
        while i < fin:
            if tab[i] < pivot:
                tab_temp[j] = tab[i]
                j += 1
            else:
                tab_temp[k] = tab[i]
                k -= 1
            i += 1
        tab[deb:fin] = tab_temp[deb:fin]
        tab[j] = pivot
        tri(tab, deb, j)
        tri(tab, j+1, fin)

    tri(tab, 0, len(tab))
```

Tri fusion d'un tableau

La fonction tri_fusion_tab trie le tableau passé en argument et ne renvoie pas de nouveau tableau. On peut travailler sur une copie du tableau si besoin.

```
def tri_fusion_tab(tab):
    tab_temp = np.empty(len(tab))

    def fusionne_tab(tab, deb, n, fin):
        # fusionne tab[deb:n] et tab[n:fin]
        i = deb
        j = deb
        k = n
        while i < fin:
            if j == n :
                tab_temp[i] = tab[k]
                k += 1
            elif k == fin:
                tab_temp[i] = tab[j]
                j += 1
            elif tab[j] < tab[k]:
                tab_temp[i] = tab[j]
                j += 1
            else:
                tab_temp[i] = tab[k]
                k += 1
            i += 1
        tab[deb:fin] = tab_temp[deb:fin]

    def tri_fusion_(tab, deb, fin):
        if fin - deb > 1:
            n = (deb + fin)//2
            tri_fusion_(tab, deb, n)
            tri_fusion_(tab, n, fin)
            fusionne_tab(tab, deb, n, fin)

    tri_fusion_(tab, 0, len(tab))
```

Remarque : si on sort la fonction fusionne de la fonction tri_fusion_tab, il faut lui transmettre le tableau tab_temp, variable locale de tri_fusion_tab.