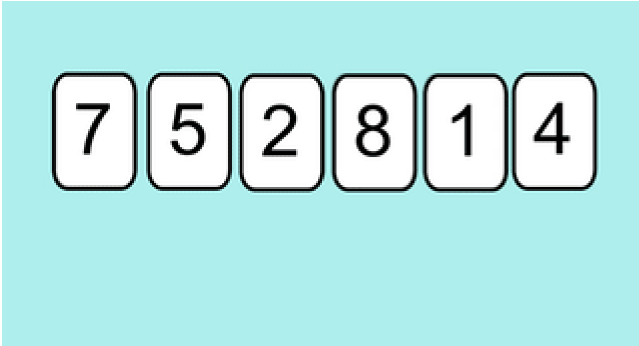
Fiche : Le tri par insertion

1. Tri par insertion

Observez l'animation ci-dessous et comparer avec la version initiale.



1.1. Codage de l'algorithme ❤️

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

1.2. Vérification :

```
>>> a = [7, 5, 2, 8, 1, 4]
>>> tri(a)
>>> print(a)
[1, 2, 4, 5, 7, 8]
```

2. Complexité de l'algorithme

On va étudier une moyenne sur 5 valeurs de deux appels sur la fonction `tri_insertion()`. On se place dans le pire des cas : on se place dans le pire des cas : une liste triée dans l'ordre décroissant.

Rajouter ce script au code précédent :

```
import time
somme_des_durees = 0
for i in range(5):
    a = [k for k in range(1_000,0,-1)]
    start_time=time.time()
    tri_insertion(a)
    somme_des_durees = somme_des_durees + time.time() - start_time
moyenne = somme_des_durees/5
print("Temps d execution pour 1_000: %s secondes ---" % (moyenne))

somme_des_durees = 0
for i in range(5):
    b = [k for k in range(10_000,0,-1)]
    start_time=time.time()
    tri_insertion(b)
    somme_des_durees = somme_des_durees + time.time() - start_time
moyenne = somme_des_durees/5
print("Temps d execution pour 10_000: %s secondes ---" % (moyenne))
```

```
Temps d execution pour 1_000: 0.05830273628234863 secondes ---
Temps d execution pour 10_000: 5.961895084381103 secondes ---
```

En comparant les temps de tri des listes a et b, que pouvez-vous supposer sur la complexité du tri par insertion ?
Une liste à trier 10 fois plus longue prend 100 fois plus de temps : l'algorithme semble de complexité **quadratique**.

2.1. Démonstration

Dénombrons le nombre d'opérations dans le pire des cas, pour une liste de taille *n*.

- boucle `for` : elle s'exécute *n-1* fois.
- boucle `while` : dans le pire des cas, elle exécute d'abord 1 opération, puis 2, puis 3... jusqu'à *n-1*. Or

$$1 + 2 + 3 + \cdots + n - 1 = \frac{n \times (n-1)}{2}$$

Si la liste est déjà triée, on ne rentre jamais dans la boucle `while` : le nombre d'opérations est dans ce cas égal à *n-1*, ce qui caractérise une complexité linéaire.

2.2. Résumé de la complexité

- dans le meilleur des cas (liste déjà triée) : complexité **linéaire**
- dans le pire des cas (liste triée dans l'ordre décroissant) : complexité **quadratique**. C'est cette complexité que nous retiendrons : **le tri par insertion est de complexité quadratique ♥**.

3. Preuve de la terminaison de l'algorithme

Est-on sûr que notre algorithme va s'arrêter ?
Le programme est constitué d'une boucle `while` imbriquée dans une boucle `for`. Seule la boucle `while` peut provoquer une non-terminaison de l'algorithme. Observons donc ses conditions de sortie :

```
while j>0 and l[j-1] > val :
```

La condition `l[j-1] > val` ne peut pas être rendue fausse avec certitude. Par contre, la condition `j>0` sera fausse dès que la variable `j` deviendra nulle. Or la ligne `j = j - 1` nous assure que la variable `j` diminuera à chaque tour de boucle. La condition `j>0` deviendra alors forcément fausse au bout d'un certain temps.
Nous avons donc prouvé la **terminaison** de l'algorithme.
On appelle la valeur `j` un **variant de boucle**. C'est une notion théorique (ici illustrée de manière simple par `j` qui permet de prouver la bonne sortie d'une boucle et donc la terminaison d'un algorithme.

4. Preuve de la correction de l'algorithme

Les preuves de correction sont des preuves théoriques. La preuve ici s'appuie sur le concept mathématique de **récurrence**.
Principe du raisonnement par récurrence : une propriété *P(n)* est vraie si :

- *P(0)* (par exemple) est vraie
- Pour tout entier naturel *n*, si *P(n)* est vraie alors *P(n+1)* est vraie.

Ici, la propriété serait : « Quand *i* varie entre 0 et `longueur(liste) -1`, la sous-liste de longueur *i* est triée dans l'ordre croissant.» On appelle cette propriété un **invariant de boucle** (sous-entendu : elle est vraie pour chaque boucle)

- quand *i* vaut 0, on place le minimum de la liste en `l[0]`, la sous-liste `l[0]` est donc triée.
- si la sous-liste de *k* éléments est triée, l'algorithme rajoute en dernière position de la liste le minimum de la sous-liste restante, dont tous les éléments sont supérieurs au maximum de la sous-liste de *k* éléments. La sous-liste de *k+1* éléments est donc aussi triée.