## PARTIE 3 : Tri par insertion

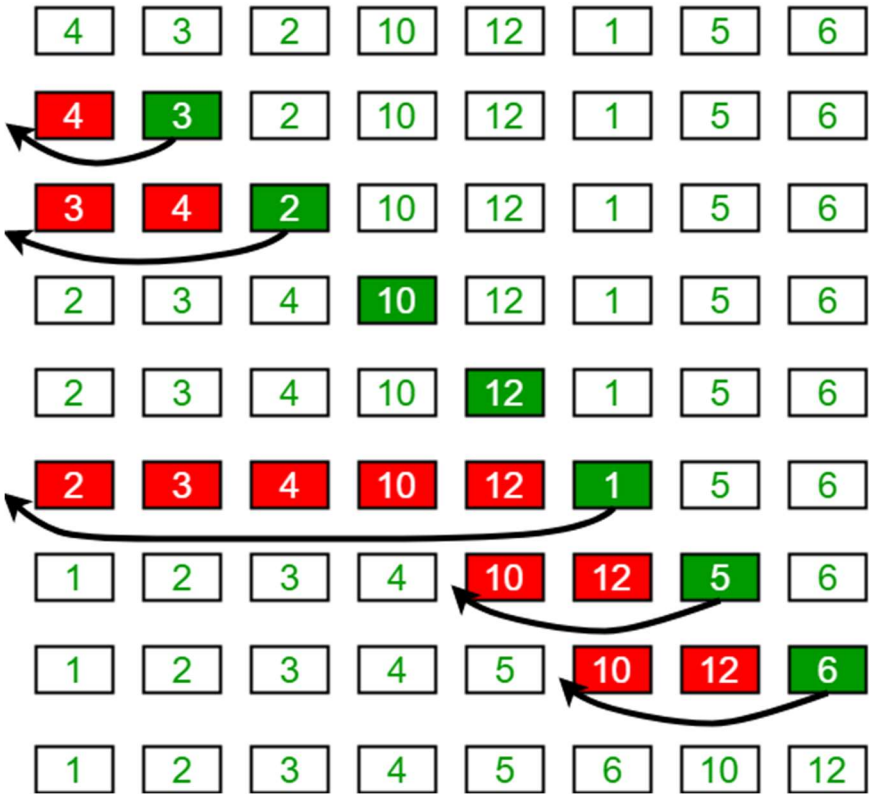
### 1. Principe

Il s'agit du principe du tri d'un jeu de carte : on range une carte piochée dans son jeu (déjà trié). Pour l'appliquer, on parcourt les éléments d'une liste et on insère successivement chaque élément dans la liste déjà triée.
Cet algorithme est assez lent, en revanche il est relativement efficace lorsqu'il s'agit de trier de courtes listes. Il s'agit d'un tri en place.

### 2. Exemple

Soit une liste non triée t = [4,3,2,10,12,1,5,6].

- On commence à trier par la gauche en supposant que la première valeur (4) est triée.
- On trie l'élément suivant (3) que l'on compare avec tous les éléments de la liste triée jusqu'à lui trouver sa place.
- Une fois sa place trouvée, la liste triée augmente d'un élément et on cherche à ranger l'élément suivant
- On recommence ces opérations jusqu'à avoir placé la dernière valeur.




Voici l'algorithme en pseudo code :

```
n←nombre d'éléments de t
pour i allant de 1 à n-1
  tant que i > 0 et t[i] < t[i-1]
    permuter t[i] et t[i-1]
    i←i – 1
  fin tant que
fin pour
```

✍ EXERCICE 1 : A l'aide du pseudo code ci-dessus créer une fonction triInsertion(t) qui tri par insertion la liste t.

✍ EXERCICE 1 bis :

```
1  # Algorithme de tri par insertion d'une liste t en place
2  def triInsertion(t):
3      n = len(t)
4      for i in range(1,n):
5          while i > 0 and t[i] < t[i -1]:      # t[i] mal place
6              t[i-1],t[i] = t[i],t[i-1]        # Permutation de t[i] et t[i-1]
7              i = i - 1                        # On descend dans les indices
                                                # pour continuer a tester si t[i] bien place
```

Nb : Il n'y a pas de return car cette fonction (ou plus exactement procédure) ne renvoie rien. Elle modifie t en place.

Soit l'algorithme de tri par insertion ci-dessus, effectuer la trace de l'algorithme de tri par insertion dans le cas triInsertion ([5,2,1,8]

| # ligne | i | t[i] | t[i-1] | Condition while | t[0] | t[1] | t[2] | t[3] |
|---|---|---|---|---|---|---|---|---|
| 4 | 1 | | | | 5 | 2 | 1 | 8 |
| 5 | 1 | 2 | 5 | True | 5 | 2 | 1 | 8 |
| 6 | 1 | 5 | 2 | | 2 | 5 | 1 | 8 |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |

## 3. Complexité temporelle

On s'intéresse au nombre de comparaisons effectuées :

• dans le meilleur des cas, la liste est déjà triée par ordre croissant. La boucle for effectue n − 1 itérations : on effectue n − 1 comparaisons. Le coût en temps est donc linéaire. On est en O(n).

• dans le pire des cas, la liste est triée par ordre décroissant. La boucle for est toujours effectuée n − 1 fois.

A chaque valeur de i, la boucle while effectue i comparaisons. Le nombre de comparaisons au total est :

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

La complexité du tri par insertion est donc de l'ordre de $n^2$. On écrira **$O(n^2)$**, on parle aussi de **coût en temps quadratique**.

## 4. Notion d'invariant de boucle

Un invariant est une hypothèse pouvant être vraie ou fausse. Les invariants de boucles servent à prouver la validité d'un algorithme comportant une ou plusieurs boucles for ou while. Un invariant de boucle est une proposition qui :

• est vraie avant d'entrer dans la boucle (initialisation)

• reste vraie après une itération si elle était vraie avant (conservation)

• donne le résultat attendu en fin de boucle (terminaison)

Si pour un algorithme donné, il existe un invariant qui vérifie ces trois étapes, alors on dit que l'algorithme est valide.

## 5. Validité du tri par insertion

On choisit comme invariant de boucle la proposition H :

« la liste t[0:i+1] est triée par ordre croissant à l'issue de l'itération i »

**Initialisation :** Avant d'entrer dans la boucle, i = 0, la liste est constituée du terme de rang 0 : [t[0]]. Cette liste constituée d'un seul élément est forcément triée. H est donc vraie.

**Conservation :** On se place à la fin de l'itération i. La liste t[0;i+1]=[t[0],t[1],...,t[i]] est supposée triée dans l'ordre croissant si H est vraie. On effectue un nouveau tour de boucle donc on se place à l'itération i + 1. Au rang i + 1, la boucle effectue :

```
while i+1 > 0 and t[i+1] < t[i]:          # t[i+1] mal place
        t[i],t[i+1] = t[i+1],t[i]         # Permutation de t[i+1] et t
        ...
```

Alors la liste t[0:i+2]=[t[0],t[1],...,t[i],t[i+1]] devient triée dans l'ordre croissant. L'hypothèse H reste vraie.

**Terminaison :** En sortie de boucle, i a pris la dernière valeur n − 1 et la boucle permet d'effectuer le tri au rang n − 1 :

```
while n-1 > 0 and t[n-1] < t[n -2]:
        t[n-2],t[n -1] = t[n-1],t[n -2]
        ...
```

La fonction ne contient pas de return mais le tri est effectué directement sur la liste t qui est modifiée à chaque itération. La liste t est donc triée dans l'ordre croissant.

## 6. Exercice

✍ EXERCICE 2 : Écrire une fonction validation(liste :list)–>bool qui vérifie si une liste est bien triée.

```
assert validation([1 ,2 ,3]) == True
assert validation([1 ,3 ,3]) == True
```

✍ EXERCICE 3 : Ajouter un assert dans la fonction triInsertion() pour vérifier la validité du programme.