Après le tri à bulle étudions un autre algorithme de tri : le tri par insertion. C'est celui qu'on utilise habituellement pour trier un paquet de carte. On démontre que c'est le plus raide pour trier une faible nombre de données.

2 Tri par insertion

2.1 Principe

Etudions un autre algorithme de tri : le Tri par insertion. C'est celui que l'on utilise habituellement dans la vie courante, par exemple, pour trier un paquet de carte :

On constitue (imaginairement) 2 tas :

- l'un dans la main droite, contenant toutes les cartes avant tri,
- l'autre dans la main gauche, contenant les cartes déjà triées,

Initialement la main gauche est vide.

Chaque étape consiste à :

- prendre la première carte du tas non trié
- L'insérer progressivement à sa bonne place dans le tas trié, en la faisant descendre d'une position tant que sa valeur reste inférieure à celle de la carte située en dessous-d'elle.

Après chaque étape le tas non trié contient une carte de moins, le tas trié une carte de plus.

A la fin du tri la main droite est vide. La main gauche contient toutes les cartes, triées.

2.2 Tri par insertion - exemple

- Code de couleurs :
  - En noir : Partie du tableau, non triée,
  - En bleu : Partie du tableau triée,
  - En rouge : Elément à insérer dans la partie triée.

3	2	1	5	6	4	Tableau à trier
3	2	1	5	6	4	Premier élément
3	2	1	5	6	4	Deuxième élément
2	3	1	5	6	4	Inséré dans le tableau trié
2	3	1	5	6	4	Troisième élément
2	1	3	5	6	4	Inséré dans le tableau trié
1	2	3	5	6	4	⋮
1	2	3	5	6	4	Quatrième élément
1	2	3	5	6	4	Inséré dans le tableau trié
1	2	3	5	6	4	Cinquième élément
1	2	3	5	6	4	Inséré dans le tableau trié
1	2	3	5	6	4	Dernier élément
1	2	3	5	4	6	Inséré dans le tableau trié
1	2	3	4	5	6	⋮
1	2	3	4	5	6	Tableau trié.

2.3 Tri par insertion - Algorithme

Le tri peut s'opérer directement sur le tableau passé en paramètre : on parle de **Tri en place**.

En pseudo-code, l'algorithme de Tri par insertion s'écrit :

(on prend pour convention que les éléments du tableau sont indicés à partir de 0, et donc jusqu'à `longueur(tableau) - 1`.)

```
Tri_Par_insertion(T)    # T est le tableau à trier
    N = longueur(T)     # N = longueur du tableau
    # Balayage du tableau du 2ème jusqu'au dernier élément :
    Pour i variant de 1 à N-1:
        x = T[i]      # C'est l'élément à insérer
        k = i         # k est son indice
        # Insertion dans la partie triée :
        Tant que k > 0 et T[k-1] > x:
            # Décalage d'un cran sur la gauche :
            T[k] = T[k-1]
            k = k-1
        T[k] = x
```

2.4 Tri par insertion - Code

En `python` :

```
def tri_insertion(T):  # T est le tableau à trier
    N = len(T)         # N = longueur du tableau
    # Balayage du tableau du 2eme jusqu'au dernier élément:
    for i in range(1,N):
        x = T[i]      # C'est l'élément à insérer
        k = i         # k est son indice
        # Insertion dans la partie triée :
        while k > 0 and T[k-1] > x:
            # Décalage d'un cran sur la gauche :
            T[k] = T[k-1]
            k = k-1
        T[k] = x
    return T     # Si l'on souhaite retourner le résultat
```

Correction de l'algorithme. Considérer comme invariant de boucle :

"Après le k-ième balayage les k premiers éléments du tableau sont ordonnés dans le sens croissant. "

(par récurrence sur k).

2.5 Tri par insertion : complexité

• Déterminons le nombre d'opérations élémentaires dans le pire et le meilleur des cas en fonction de la taille `N` du tableau à trier.

- 1 opération élémentaire pour retourner `N = len(T)`
- puis on répète `N-1` fois ce qui est dans la boucle `for` :
  - 1 opération pour `x = T[i]` (lecture dans un tableau et affectation)
  - 1 opération pour l'affectation `k = i`
  - 1 à 2 opérations pour tester la condition du `while`
  - au plus `N-1` décalages dans la boucle `while` :
    - 3 opérations (accès en lecture/écriture et décrémentation).

• La complexité de l'algorithme est donc :

- Dans le pire des cas majorée par l'ordre de :

$$\underbrace{(N-1)}_{\texttt{for}} \times \underbrace{(N-1)}_{\texttt{while}} = \Theta(N^2) \qquad \text{(quadratique)}$$

- Dans le meilleur des cas minorée par l'ordre de :

$$\underbrace{(N-1)}_{\texttt{for}} \times \underbrace{1}_{\texttt{while}} = \Theta(N) \qquad \text{(linéaire)}$$

• Montrons que ces bornes sont atteintes :

- Dans le pire des cas : le cas d'un tableau ordonné dans le sens décroissant :

  Par exemple : | N | $(N-1)$ | $\cdots$ | 2 | 1 |

  Lors de la i-ème insertion l'élément doit être inséré en tout début de tableau : la boucle `while` s'exécute $i-1$ fois ; de l'ordre exactement de i opérations pour l'insertion du i-ème élément. La complexité est exactement d'ordre :

$$\sum_{i=1}^{N-1} i = \Theta(N^2) \qquad \text{(quadratique)}$$

- Dans le meilleur des cas : le cas d'un tableau ordonné dans le sens croissant :

  Par exemple : | 1 | 2 | $\cdots$ | $(N-1)$ | N |

  Lors de la i-ème insertion l'élément est déjà à la bonne place : la boucle `while` ne s'exécute pas ; de l'ordre d'1 opérations pour chaque insertion. La complexité est exactement d'ordre :

$$(N-1) \times 1 = \Theta(N) \qquad \text{(linéaire)}$$

Ainsi :

1. Le tri par insertion est un tri en place : sa complexité en espace mémoire est en $O(1)$.
2. Le tri par insertion a une complexité (temporellle) :
   (a) linéaire dans le meilleur des cas (*i.e.* le cas d'un tableau déjà trié).
   (b) quadratique dans le pires des cas (ce n'est pas un très bon algorithme de tri pour un grand nombre de données).
3. Cependant on démontre que dans le cas d'un petit nombre d'éléments à trier (inférieur à ...) c'est le plus rapide en moyenne.
4. Il est également très efficace lorsque le tableau est déjà presque trié. Par exemple la complexité est linéaire si de plus tous les éléments sont à une distance uniformément bornée (ne dépendant pas de N) de leur position finale. Ou lorsque le nombre d'éléments qui ne sont à la bonne place est uniformément borné.

Conclusion : continuons à l'utiliser pour trier un paquet de carte (c'est le meilleur possible), mais nous allons voir que lorsque le nombre d'éléments à trier devient grand, ce n'est plus celui à utiliser sur de grands tableaux très "désordonnés".

2.6 Tri par insertion - Exemples

Modifions le code pour afficher les états intermédiaires et les insertions :

```
def tri_insertion(T):  # T est le tableau à trier
    N = len(T)         # N = longueur du tableau
    print(T)     # Affichage initial
    # Balayage du tableau du 2eme jusqu'au dernier élément:
    for i in range(1,N):
        x = T[i]      # C'est l'élément à insérer
        k = i         # k est son indice
        print("Insertion de", T[i], "d'indice", i)  # Annonce d'insertion
        # Insertion dans la partie triée :
        while k > 0 and T[k-1] > x:
            # Décalage d'un cran sur la gauche :
            T[k] = T[k-1]
            T[k-1] = x
            k = k-1
            print(T)     # Affichage insertion
    return T     # Si l'on souhaite retourner le résultat
```

```
In [1]: tri_insertion([3,2,5,1,6,4])
[3, 2, 5, 1, 6, 4]
Insertion de 2 d'indice 1
[2, 3, 5, 1, 6, 4]
Insertion de 5 d'indice 2
Insertion de 1 d'indice 3
[2, 3, 1, 5, 6, 4]
[2, 1, 3, 5, 6, 4]
[1, 2, 3, 5, 6, 4]
Insertion de 6 d'indice 4
Insertion de 4 d'indice 5
[1, 2, 3, 5, 4, 6]
[1, 2, 3, 4, 5, 6]
Out[1]: [1, 2, 3, 4, 5, 6]
```

Cas d'une liste ordonnée dans le sens décroissant :

```
In [2]: tri_insertion([6,5,4,3,2,1])
[6, 5, 4, 3, 2, 1]
Insertion de 5 d'indice 1
[5, 6, 4, 3, 2, 1]
Insertion de 4 d'indice 2
[5, 4, 6, 3, 2, 1]
[4, 5, 6, 3, 2, 1]
Insertion de 3 d'indice 3
[4, 5, 3, 6, 2, 1]
[4, 3, 5, 6, 2, 1]
[3, 4, 5, 6, 2, 1]
Insertion de 2 d'indice 4
[3, 4, 5, 2, 6, 1]
[3, 4, 2, 5, 6, 1]
[3, 2, 4, 5, 6, 1]
[2, 3, 4, 5, 6, 1]
Insertion de 1 d'indice 5
[2, 3, 4, 5, 1, 6]
[2, 3, 4, 1, 5, 6]
[2, 3, 1, 4, 5, 6]
[2, 1, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
Out[2]: [1, 2, 3, 4, 5, 6]
```

- Sur une liste déjà triée :

```
In[3]: tri_insertion([1,2,3,4,5,6])
[1, 2, 3, 4, 5, 6]
Insertion de 2 d'indice 1
Insertion de 3 d'indice 2
Insertion de 4 d'indice 3
Insertion de 5 d'indice 4
Insertion de 6 d'indice 5
Out[3]: [1, 2, 3, 4, 5, 6]
```