1 Tri par insertion

1.1 Présentation de l'algorithme

On désire trier un tableau de n éléments par ordre croissant de valeurs.
On suppose que les i premiers élément sont déjà triés. On insère alors le 1er élément non trié à sa place parmi ceux déjà triés. Pour cela on décale vers la droite les éléments dont la valeur est plus grande que celle de l'élément à insérer.
Remarque : c'est souvent la manière dont les gens trient un jeu de cartes.

```
def tri_insertion(t):
    """ tri le tableau t """
    # précondition : t est un tableau de valeurs comparables
    # postcondition : le tableau est trié "en place"
    for i in range(1, len(t)): # l'élément d'indice 0 est trié par défaut
        valeur = t[i] # valeur du 1er élément non trié
        j = i - 1 # indice de l'élément trié de plus grande valeur
        while j >= 0 and t[j] > valeur:
        # on décale vers la droite les élément triés
        # dont la valeur est plus grande que celle de l'élément à trier
            t[j+1] = t[j]
            j = j - 1 # on passe à l'élément suivant de plus grande valeur
        t[j+1] = valeur  # insertion de l'élément à trier à sa place
```

1.2 Preuve de correction

Un algorithme est **correct** s'il fait ce qu'on attend de lui ! L'algorithme de tri est donc correct s'il trie le tableau.
Pour prouver la correction, il faut d'abord prouver sa **terminaison**, càd s'assurer que les boucles ont bien une fin ; et enfin exhiber un **invariant de boucle**, càd une **propriété** (en rapport avec le but de l'algorithme) **qui est vraie avant et après chaque itération de boucle**.

Preuve de la terminaison :

1. Une boucle *for* termine toujours puisqu'elle est bornée (il y a une valeur maximum de i qui est incrémenté à chaque itération).
2. La boucle *while* termine aussi ici puisqu'à chaque itération on décrémente la valeur de j (positive au début) et qu'on quitte la boucle *while* si j devient négatif.

Invariant de boucle (boucle *for*) : *Le tableau est trié jusqu'à l'indice* $i-1$ *(inclus).*

1. On vérifie aisément que cette propriété est vraie au début pour $i = 1$ car le tableau jusqu'à l'indice 0 est forcément trié : c'est une unique valeur !
2. Supposons maintenant que l'invariant est vrai pour i quelconque et montrons qu'il reste vrai à la fin d'une itération.
   Au cours d'une itération de la boucle *for*, toutes les valeurs du tableau sont décalées d'un cran vers la droite jusqu'à l'indice i et on insère la valeur à trier au bon endroit (à

un indice inférieur à i). Donc à la fin de l'itération, le tableau est devenu trié jusqu'à l'indice i inclus.
Et finalement, avant d'aborder l'itération suivante, le tableau est bien trié jusqu'à l'indice $(i-1)$ du "point de vue" de cette nouvelle itération.

3. À l'abord de la dernière itération de la boucle *for*, $i = len(t) - 1$, et donc le tableau est trié jusqu'à l'indice $len(t) - 2$; au cours de cette dernière itération le tableau est alors trié jusqu'à l'indice $len(t) - 1$, càd le dernier indice du tableau, et donc le tableau est bien trié dans son intégralité ! (preuve de correction terminée).

1.3 Complexité

Le coût de cet algorithme dépend de l'état du tableau initial : s'il est trié par ordre décroissant, c'est le cas le pire et il faut à chaque itération décaler le nouvel élément (d'indice i) jusqu'à la position d'indice 0 et donc faire i décalages des autres éléments.
Il peut donc être nécessaire d'effectuer $1 + 2 + ... + (n-1) = \dfrac{n(n-1)}{2}$ opérations pour un tableau de taille n.
Pour un tableau de **grande** taille n, ce nombre est de l'ordre de n^2.
Le coût de cet algorithme est donc quadratique (varie comme n^2 si n devient grand).

2 Tri par sélection

2.1 Présentation de l'algorithme

On désire à nouveau trier un tableau par ordre croissant.
On commence par rechercher l'élément de plus petite valeur et on permute cet élément avec celui en 1ère position du tableau. Par la suite, on ne modifiera plus cet élément. On reprend alors la recherche de l'élément de plus petite valeur parmi ceux non triés et on le permute avec le 2ème élément du tableau. Et ainsi de suite.

On peut reformuler cet algorithme ainsi :
On suppose que les i premiers éléments sont déjà triés. On recherche alors l'élément de plus petite valeur parmi ceux non triés et on le permute avec le 1er élément non trié.

```
def tri_selection(t):
  """ tri le tableau t """
  # précondition : t est un tableau de valeurs comparables
  for i in range(len(t)-1): # i = indice du 1er élément non trié
    i_mini = i # indice de l'élément de valeur mini parmi les non triés
    # recherche de l'indice de l'élément non trié de valeur mini
    for j in range(i+1, len(t)):
      if t[j] < t[i_mini] :
        i_mini = j
    # permutation de l'élément non trié de valeur mini
    # avec le 1er élément non trié
    t[i_mini], t[i] = t[i], t[i_mini]
```

2.2 Preuve de correction

Cherchons de nouveau à prouver la terminaison de l'algorithme et un invariant de boucle.

Preuve de la terminaison :
L'algorithme termine forcément puisqu'il ne fait intervenir que 2 boucles *for*, qui terminent toujours puisqu'elles sont bornées.

Invariant de boucle (boucle *for* principale) : *Le tableau est trié jusqu'à l'indice i (inclus) à la fin d'une itération de la boucle for principale, et tous les autres éléments sont plus grands.*

1. On vérifie aisément que cette propriété est vraie au début pour $i = 0$ car à la fin de la boucle *for* principale, on a placé le plus petit élément du tableau à cet indice 0. Le tableau est donc trié pour cet unique élément et tous les autres éléments sont plus grands.
2. Supposons maintenant que l'invariant est vrai pour i quelconque et montrons qu'il reste vrai à la fin d'une nouvelle itération.
   Au cours d'une itération de la boucle *for* principale, la boucle *for* secondaire permet de trouver le plus élément restant non trié, et ensuite cet élément est placé à l'indice i.
   Le tableau devient donc bien trié jusqu'à l'indice i avec les autres éléments restants plus grands.
3. À la fin de la boucle for principale, $i = len(t) - 2$, et donc le tableau devient trié jusqu'à cet indice $len(t) - 2$, càd l'avant dernier indice du tableau ; en outre le seul dernier élément est plus grand, et donc le tableau est bien trié dans son intégralité ! (preuve de correction terminée).

2.3 Complexité

Dans cet algorithme pour un tableau de taille n, dans la boucle for principale, i varie de 0 à $(n-2)$, et pour chacune de ces valeurs de i, dans la boucle for secondaire, j varie de $(i+1)$ à $(n-1)$ [soit $(n-1-i)$ valeurs] à la recherche du minimum en effectuant des comparaisons.
Il y a donc $(n-1)$ comparaisons pour $i = 0$ puis de moins en moins jusqu'à 1 seule comparaison pour $i = n-2$.
Le nombre total de comparaisons est donc : $(n-1) + (n-2) + ... + 1 = \dfrac{n(n-1)}{2}$.

À nouveau, pour un tableau de **grande** taille n, ce nombre est de l'ordre de n^2.
Le **coût** de cet algorithme est donc aussi **quadratique** (varie comme n^2 si n devient grand).

3 D'autres tris

Une complexité quadratique est prohibitive pour des tableaux de trop grande taille.
Heureusement, il existe d'autres algorithmes de tris beaucoup plus performants avec un coût moindre. Les meilleurs algorithmes ont une complexité en $n\ln(n)$.
Ils sont par exemple implémentés dans les fonction de tri intégrées à Python.

Voir une animation : https ://www.toptal.com/developers/sorting-algorithms