Fiche d'identité de l'algorithme de **TRI par INSERTION**

Principe : on parcourt les éléments de la liste et on insère successivement chaque élément dans la liste déjà triée → tri en place
Exemple : le tri du jeu de cartes, on range une carte piochée dans son jeu déjà classé

Pseudo Code

On écrit deux boucles qui s'appliquent à une liste t de n éléments :
*une boucle for qui étudie les indices i de la liste t de 1 à n-1
*dans cette boucle est imbriquée une boucle while : on compare deux éléments voisins de la liste. Tant que t[i]<t[i-1], on les permute

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

NB : on aurait pu utiliser une variable tampon pour la permutation
tmp ← t[i]
t[i] ← t[j]
t[j] ← tmp

Python

On parle de procédure car il n'y a pas de renvoi (pas de return) : on trie la liste en place

```
def tri_insertion(t):
    n=len(t)
    for i in range (1,n):
        while ((i>0) and (t[i]<t[i-1])):  #t[i] mal placé
            t[i-1],t[i] = t[i],t[i-1]  # permute t[i] et t[i-1]
            i = i-1   # on descend les indices et on passe à l'élément suivant
```

Algorithme valide (CORRECTION et TERMINAISON)
On choisit comme **invariant de boucle** (procédé itératif) H : « la liste t[0 : i+1] est triée par ordre croissant à l'issue de l'itération i »
Initialisation (**est ce vrai avant d'entrer dans la boucle** ?)
si i=0, au rang 0 on a [t[0]] ou encore t[0:0+1] : constituée d'un seul terme, elle est donc triée : H est vraie
Conservation = hérédité (**reste vraie après une itération, si elle était vraie avant**)
A la fin de l'itération i, 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 à l'itération i+1. La boucle effectue le tri

tant que i+1 >0 et t[i+1]<t[i] //teste si t[i+1] est mal placé
    permuter t[i+1] et t[i]

Alors la liste t[0 : i+2]=[t[0],t[1],...,t[i], t[i+1]] devient triée dans l'ordre croissant. H reste vraie
Terminaison (**donne le résultat attendu en fin de boucle**)
En sortie de boucle, i a pris sa dernière valeur i= n-1 et la boucle permet d'effectuer le tri au rang n-1

tant que n-1 >0 et t[n-1]<t[n-2] //teste si t[n-1] est mal placé
    permuter t[n-1] et t[n-2]

Malgré l'absence de « renvoi »(return en python) le tri est effectué sur la liste t, modifiée à chaque tour de boucle. La liste t est triée dans l'ordre croissant et la fonction *tri* a rempli son objectif

Complexité en temps : algorithme lent mais relativement efficace sur de courtes listes (on notera également une propriété importante du tri par insertion : contrairement à celle d'autres méthodes, son efficacité est meilleure si le tableau initial possède un certain ordre. L'algorithme tirera en effet parti de tout ordre partiel présent dans le tableau. Jointe à la simplicité de l'algorithme, cette propriété le **désigne tout naturellement pour "finir le travail" de méthodes plus ambitieuses** comme le tri rapide, variante de « diviser pour régner » utilisée par la dichotomie)

Dans le meilleur des cas, avec des données déjà triées, l'algorithme effectuera seulement *n-1* comparaisons. Sa complexité dans le meilleur des cas est donc en $\Theta(n)$: c'est un coût linéaire
Dans le pire des cas, avec des données triées à l'envers, les parcours successifs du tableau imposent d'effectuer *(n-1)+(n-2)+(n-3)..+1* comparaisons et échanges, soit $\mathbf{(n^2-n)/2.}$ On a donc une complexité dans le pire des cas du tri par insertion en $\Theta(n^2)$: c'est un coût quadratique
Si tous les éléments de la série à trier sont distincts et que toutes leurs permutations sont équiprobables, la complexité en moyenne de l'algorithme est de l'ordre de $(n^2-n)/4$ comparaisons et échanges. La complexité en moyenne du tri par insertion est donc également en $\Theta(n^2)$

Formalisation

Lien vers la vidéo : https://www.youtube.com/watch?v=ejpFmtYM8Cw

1°) Faire fonctionner à la main le tri par insertion sur un nouveau jeu de 4 cartes

puis sur une liste, [9,3,1,6] par exemple qui pourra être aussi illustrée si besoin par les 4 cartes de valeur correspondantes → Implémenter alors en pseudo code puis en Python cet algorithme

[9,3,1,6] → [3,9,1,6] → [1,3,9,6] → [1,3,6,9]

Vous pourrez aussi utiliser le mode pas à pas de Thonny

2°)

Quel est le meilleur des cas ? Combien y-a-il alors d'itérations dans la boucle *pour*? Combien y-a-il alors de comparaisons dans la boucle *tant que* ?

La liste est triée par ordre croissant : on effectue n-1 itérations et n-1 comparaisons : on est de l'ordre de n opérations (coût linéaire)

Quel est le pire des cas ? Combien y-a-il alors d'itérations dans la boucle *pour*? Combien y-a-il alors de comparaisons dans la boucle *tant que* ?

La liste est triée par ordre décroissant : on effectue n-1 itérations et i comparaisons à chaque tour de boucle. $1+2+....+(n-2)+(n-1) = (n-1)n/2 = (n^2-n)/2$ comparaisons (coût quadratique $O(n^2)$)

3°) Definir et trouver un *Invariant de boucle*

Pseudo code + code Python + Correction du code + terminaison du code + complexité en temps → voir fiche ID tri Insertion

Exercices

Faire fonctionner votre tri par insertion sur des listes de votre choix

Exercice 1 – *ANALYSER et CONCEVOIR*

On souhaite trier la liste de listes *Pers* selon le numéro situé en 2ème position dans les sous-listes que contient *Pers* :

```
Pers =[['Portillon',4],['Sam',3],['Julie',1],['Tom',2],['Charlie',5]]
```

Proposer une fonction mettant en œuvre un algorithme de tri par insertion permettant de trier la liste *Pers*

Exercice 2 – *CONCEVOIR et COMPARER*

Afin de comparer les coûts en temps des méthodes de tri, proposer une fonction qui génère aléatoirement un tableau de taille n choisi par l'utilisateur d'entiers positifs.

Compléter à l'aide la fonction déjà utilisée lors de l'introduction du cours d'algorithmique

```
from time import time
debut = time()

# Code dont on mesure le temps

fin = time()
print("Temps passé : ",fin - debut)
```

le tableau ce dessous et comparer avec vos camarades (selon la taille et la machine utilisée)

Coefficients multiplicateurs	Taille n du tableau	Temps Tri insertion	Coefficients multiplicateurs
	10	0	
10 fois plus	100	0	??
10 fois plus	1000	0,2s	??
5 fois plus	5000	5,2s	$25=5^2$ fois plus
2 fois plus	10000	20s	$4=2^2$ fois plus

Si la taille est multipliée par k, le temps de tri semble multiplié par k^2