Ordonner les éléments d'une liste est une activité essentielle en informatique. Par exemple une fois qu'une liste est triée, il est très facile de chercher si elle contient tel ou tel élément.
Mais comment ranger des données afin de faciliter leur accès futur ? Nous allons ici découvrir différents algorithmes permettant de trier des données.

Partie I : Avec des cartes

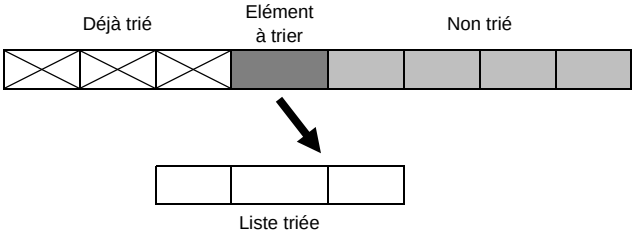
Commençons par étudier l'ordre intuitif dans lequel un joueur de cartes va ranger son jeu afin de limiter le temps de recherche pendant le déroulement de la partie.
Chaque groupe a à sa disposition un jeu de carte, composé des cartes 1 à 10 d'une seule couleur. Mélanger le jeu puis aligner les cartes sur la table, face vers le haut. Ordonner ensuite les cartes de 1 à 10.

I.1. Décrire **précisément**, étape par étape, comment vous avez fait pour ordonner les cartes.

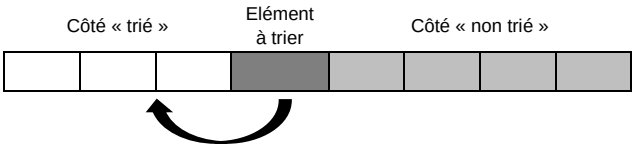
I.2. A l'aide des descriptions ci-dessous, déterminer quelle catégorie de tri vous avez utilisé pour ordonner vos cartes.

Méthodes de tris : insertion et sélection

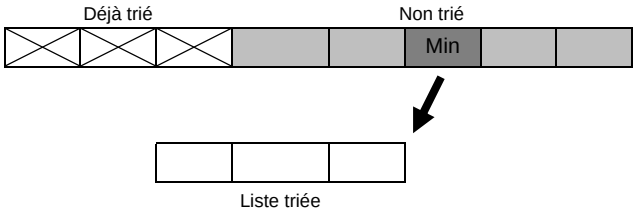
Le **tri par insertion** consiste à prendre chaque élément d'une liste l'un après l'autre, et à l'insérer au bon endroit dans l'ensemble des données déjà triées (qui sont soit dans une liste séparée, soit en ayant partagé la liste originale en un côté « trié » et un côté « pas encore trié »).



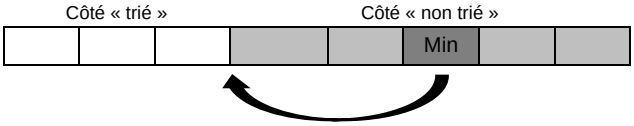
OU



Le **tri par sélection** consiste à traiter les éléments d'une liste directement dans l'ordre où on souhaite les ranger, en sélectionnant la valeur minimale de l'ensemble des données non triées et en la plaçant à la suite des données triées (qui sont soit dans une liste séparée, soit en ayant partagé la liste originale en un côté « trié » et un côté « pas encore trié »)..



OU



Partie II : Algorithmes de tri

A. Tri par insertion

Mélanger votre jeu, aligner les cartes sur la table puis les ordonner en suivant la méthode de tri par insertion (sans créer une nouvelle ligne de carte).

II-A.1. En vous aidant du tri des cartes, commenter le pseudo-code ci-dessous (que se passe-t-il au fur et à mesure ?)

Algorithme de tri par insertion

```
Entrée : L une liste d'entiers de taille N
i = 1
TANT QUE i < longueur de L
        Val = L[i]
        j = i
        TANT QUE j>0 et L[j-1]>Val
                L[j] = L[j-1]
                Soustraire 1 à j
        FIN TANT QUE
        L[j] = Val
        Ajouter 1 à i
FIN TANT QUE
RENVOYER L
```

.....
.....
.....
.....
.....
.....
.....
.....
.....

Testons l'algorithme. On considère la liste suivante : L = [8, 5, 4, 7]

II-A.2. Compléter un tableau ci-dessous, en suivant l'algorithme.

		A ne compléter QUE si (i < longueur de L) est Vrai		A ne compléter QUE si (j>0 et L[j-1]>Val) est Vrai			
i	i < longueur L	Val	j	j>0	L[j-1]	L[j-1]>Val	**La liste devient**
1	Vrai	5	1	Vrai	8	Vrai	[8, 8, 4, 7]
1	Vrai	5	0	Faux			[5, 8, 4, 7]

II-A.3. En étudiant le nombre de comparaisons à faire dans chaque boucle TANT QUE pour le pire des cas (que vous définirez), montrer que la complexité de l'algorithme de tri par insertion pour une liste de taille N est de l'ordre de N^2.

II-A.4. Preuve de terminaison : Soit *N* la longueur de la liste. Montrer que N-i est un variant de boucle pour le premier TANT QUE, et que j est un variant de boucle pour le second TANT QUE imbriqué.

II-A.5. Preuve de correction : Montrer que « les *i* premiers éléments de L sont classés par ordre croissant » est un invariant de boucle, en suivant les étapes suivantes.

a. Cette propriété est-elle vérifiée pour $i = 0$?
b. Que se passe-t-il à l'itération suivante ($i = 1$) ? La propriété est-elle alors toujours vérifiée ?
c. Si la propriété est vérifiée pour n'importe quelle itération i, l'est-elle toujours à l'itération i+1 ?
d. Conclure.

B. Tri par sélection

Mélanger votre jeu, aligner les cartes sur la table puis les ordonner en suivant la méthode de tri par sélection (sans créer une nouvelle ligne de carte).

II-B.1. En vous aidant du tri des cartes, compléter le pseudo-code ci-dessous.

Algorithme de tri par sélection

```
Entrée : L une liste d'entiers de taille N
i = 0
TANT QUE i < longueur de L
        iMin = i
        j = i+1
        TANT QUE j < longueur de L
                SI L[j] ≤ ..................................
                        iMin = j
                Ajouter 1 à j
        FIN TANT QUE
        Val = ..................................................
        L[i] = .................................................
        L[iMin] = ..............................................
        Ajouter 1 à i
FIN TANT QUE
RENVOYER L
```

II-B.2. En étudiant le nombre de comparaisons à faire dans chaque boucle TANT QUE pour le pire des cas (que vous définirez), montrer que la complexité de l'algorithme de tri par sélection pour une liste de taille N est de l'ordre de N^2.

II-B.3. Preuve de terminaison : Soit *N* la longueur de la liste. Montrer que N-i est un variant de boucle pour le premier TANT QUE, et que N-j est un variant de boucle pour le second TANT QUE imbriqué.

C. A vos ordinateurs !

II-C.1. A partir du pseudo-code donné, écrire une fonction Python de tri par insertion, nommée *TriIns*, qui prend en argument une liste *Liste* et qui renvoie la liste ordonnée.
Tester le code avec la liste *[8, 5, 4, 7]*. Avec des affichages print bien choisis, afficher le processus de tri dans la console, comme dans l'exemple ci-dessous.

```
>>> TriIns([8, 5, 4, 7])
Liste à trier:  [8, 5, 4, 7]
[8] 5
[5, 8] 4
[4, 5, 8] 7
[4, 5, 7, 8]
```

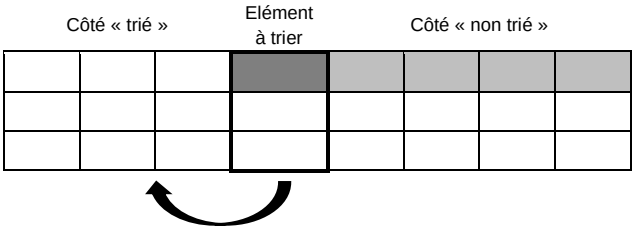
II-C.2. A partir du pseudo-code donné, écrire une fonction Python de tri par sélection, nommée *TriSel*, qui prend en argument une liste *Liste* et qui renvoie la liste ordonnée.
Tester le code avec la liste *[8, 5, 4, 7]*. Avec des affichages print bien choisis, afficher le processus de tri dans la console, comme dans l'exemple ci-contre.

```
>>> TriSel([8, 5, 4, 7])
Liste à trier:  [8, 5, 4, 7]
[4] [5, 8, 7]
[4, 5] [8, 7]
[4, 5, 7] [8]
[4, 5, 7, 8] []
[4, 5, 7, 8]
```

Partie III : Tri d'un tableau

On rappelle qu'un tableau à 2 dimensions correspond à une liste principale *M* (les lignes) contenant des listes secondaires L_n (les colonnes) : M = [L_0, L_1, L_3…]
Chaque élément de la matrice peut alors être retrouvé grâce à ses indices : `element = M[indiceM][indiceL]`

Nous allons maintenant étudier le cas du tri par insertion d'un tableau à 2 dimensions selon une colonne. Dans le cas du tri d'un tableau suivant une colonne, c'est toute la colonne qui se « déplace » lors du tri.



III.1. On cherche à modifier l'algorithme de tri par insertion pour gérer un tableau au lieu d'une liste.

a. Dans l'algorithme pour une liste, la variable *Val* permet de sauvegarder la valeur de l'élément d'indice *i*. Comment doit-on modifier cette variable pour qu'elle sauvegarde la colonne *i* du tableau ? Quel est alors le type de la variable ?

b. Comment doit-on alors modifier la commande « `Liste[j-1]>Val` » pour comparer les valeurs de la première ligne de *Table* à la première valeur de la colonne *Val* ?

c. En utilisant le terme « toute la colonne … de *Table* » à la place de *Liste[…]* quand nécessaire, modifier le pseudo-code de tri par insertion pour gérer un tableau au lieu d'une liste.

Appeler le professeur pour vérifier le pseudo-code

III.2. Nous allons maintenant modifier le code : On copie le code de la fonction *TriIns(Liste)* et on nomme la fonction *TriInsTable(Table)*. C'est ce code que nous allons modifier, étape par étape. On peut commencer par enlever les `print` intermédiaires dans le code, juste garder l'affichage du tableau à trier.

a. A l'aide de la fonction len(), comment obtient-on le nombre *c* de colonnes ? Et le nombre *n* de ligne (pour la première colonne par exemple) ?
Créer les variables *c* et *n* en conséquence.

b. Comment faire pour créer une variable *Val* de type liste contenant autant de *0* qu'il y a de ligne dans le tableau *Table* ?
Créer la variable *Val* en conséquence.

c. Comment faire pour sauvegarder toute une colonne dans *Var* ou pour remplacer une colonne du tableau par *Var* ?
Modifier le code en conséquence.

Pour tester le code, on utilise un générateur de tableau de taille aléatoire (entre 3 et 10) et d'éléments aléatoirement choisis entre -100 et 100 :

```
from random import randint
col = randint(3,10)
ligne = randint(3,10)
Table = []
for i in range(ligne):
    ValCol = []
    for j in range(col):
        Val = randint(-100,100) # on génère un entier aléatoirement
        ValCol = ValCol + [Val] # on l'ajoute à la liste ValCol
    Table = Table + [ValCol] # on ajoute ValCol au tableau Table
```

III.3. Tester votre code et corriger les erreurs si nécessaire.

Ce qu'il faut retenir

❖ Tri par insertion

La méthode de tri par insertion consiste à prendre chaque élément d'une liste l'un après l'autre, et à l'insérer au bon endroit dans l'ensemble des données déjà triées

Algorithme de tri par insertion

```
Entrée : L une liste d'entiers de taille N
i = 1
TANT QUE i < longueur de L
    Val = L[i]
    j = i
    TANT QUE j>0 et L[j-1]>Val
        L[j] = L[j-1]
        Soustraire 1 à j
    FIN TANT QUE
    L[j] = Val
    Ajouter 1 à i
FIN TANT QUE
```

La complexité de l'algorithme de tri par insertion est quadratique ($\propto$ *Taille*^{2})

❖ Tri par sélection

La méthode de tri par sélection consiste à traiter les éléments d'une liste directement dans l'ordre où on souhaite les ranger, en sélectionnant la valeur minimale de l'ensemble des données non triées et en la plaçant à la suite des données triées

Algorithme de tri par sélection

```
Entrée : L une liste d'entiers de taille N
i = 0
TANT QUE i < longueur de L
    iMin = i
    j = i+1
    TANT QUE j < longueur de L
        SI L[j]≤L[iMin]
            iMin = j
        Ajouter 1 à j
    FIN TANT QUE
    Val = L[i]
    L[i] = L[iMin]
    L[iMin] = Val
    Ajouter 1 à i
FIN TANT QUE
```

La complexité de l'algorithme de tri par insertion est quadratique ($\propto$ *Taille*^{2})