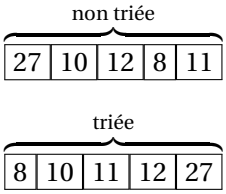
Cours - NSI
Algorithme de tri

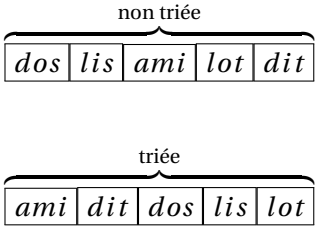
1 Notion de tri

Un algorithme de tri est, en informatique ou en mathématiques, un algorithme qui permet d'organiser une collection d'objets selon une relation d'ordre déterminée. Les objets à trier sont des éléments d'un ensemble muni d'un ordre total.
Classiquement on tri :

— Des nombres selon la relation d'ordre usuelle « est inférieur ou égal à » ("<=" en python).



— Des mots à l'aide de l'ordre lexicographique (alphabet).



En python l'opérateur logique "<=" marche avec les nombres et les chaînes de caractère. "<=" utilise le code ASCII pour les str. Il faudra donc faire attention à ce que les chaînes de caractère s'écrivent avec la même casse (Uniquement majuscule ou uniquement minuscule). On utilise les méthodes lower() ou capitalize().

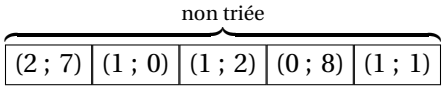
```
# Dans la console Python
>>>  2<=3
True
>>> "ami"<="dos"
True
>>>"aMi"<="DOS"
False
>>>"aMi".lower()<="DOS".lower()
True
>>>"aMi".capitalize()<="DOS".capitalize()
True
```

1.1 Tri en place

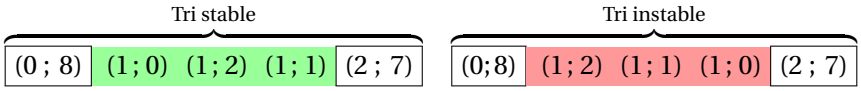
Un tri est dit en place s'il n'utilise qu'un nombre très limité de variables et qu'il modifie directement la structure qu'il est en train de trier. Ceci nécessite l'utilisation d'une structure de donnée adaptée (un tableau par exemple). Ce caractère peut être très important si on ne dispose pas de beaucoup de mémoire.

1.2 Tri stable

Un tri est dit stable s'il préserve l'ordonnancement initial des éléments que l'ordre considère comme égaux. Pour définir cette notion, il est nécessaire que la collection à trier soit ordonnancée d'une certaine manière (ce qui est souvent le cas pour beaucoup de structures de données, par exemple pour les listes ou les tableaux).
Par exemple supposons que l'on veut trier une liste de couples suivant la première coordonnée.



Les couples (1 ; 0), (1 ; 2) et (1 ; 1) sont supposés égaux (car de même première coordonnée).



Dans le cas d'un tri stable (1 ; 0), (1 ; 2) et (1 ; 1) sont dans le même ordre initial.
Cette notion est importante si on considère par exemple des tris successifs sur des champs différents.

1.3 Complexité algorithmique

— La **complexité temporelle** (en **moyenne** ou **dans le pire des cas**) mesure le **nombre d'opérations élémentaires** effectuées pour trier une collection d'éléments. C'est un critère majeur pour comparer les algorithmes de tri, puisque c'est une estimation directe du temps d'exécution de l'algorithme. Dans le cas des algorithmes de tri par comparaison, la complexité en temps est le plus souvent assimilable au nombre de comparaisons effectuées, la comparaison et l'échange éventuel de deux valeurs s'effectuant en temps constant.

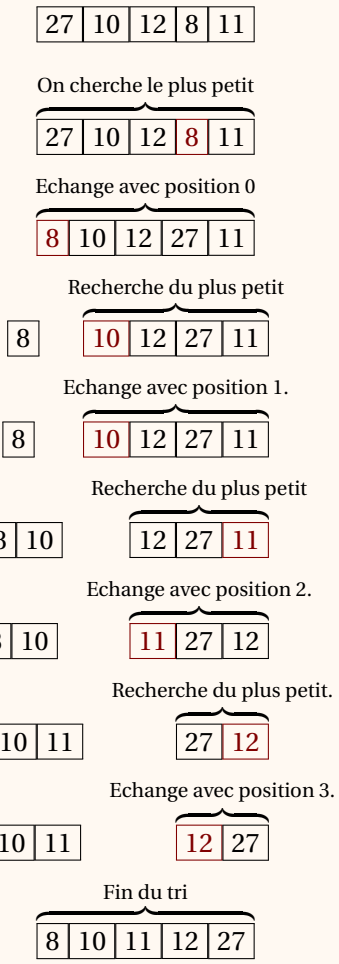
— La **complexité spatiale** (en **moyenne** ou **dans le pire des cas**) représente, quant à elle, **la quantité de mémoire** dont va avoir besoin l'algorithme pour s'exécuter. Celle-ci peut dépendre, comme le temps d'exécution, de la taille de l'entrée. Il est fréquent que les complexités spatiales en moyenne et dans le pire des cas soient identiques. C'est souvent implicitement le cas lorsqu'une complexité est donnée sans indication supplémentaire.

2 Tri par sélection

Le **Tri par sélection** est un tri sur place ; instable par défaut (peut être rendu stable).
On commence par un exemple (A faire avec des cartons numérotés au tableau) :

Exemple

On va trier la liste à 5 éléments dans l'ordre croissant :



Exercice 1

Écrire une fonction **mini(liste)** qui renvoie la valeur minimale d'une liste de valeurs et l'indice de cette valeur dans la liste.

Code Python

```
# Dans l'éditeur PYTHON
def mini(A):
    '''In : une liste A de nombres
      Out : un tuple, le minimun de A et son indice'''
    min=A[0]
    indice=0
    ...
    return (min,indice)
```

2.1 Algorithme

On donne l'algorithme de tri par sélection en pseudo code.

Algorithm 1 Tri_Par_Selection(A) :

ENTRÉE : A est une liste avec une relation d'ordre comme les entiers, les flottants ou les chaînes de caractères.

Pour $i = 0$ à $longueur(A) - 1$ **faire**
  $m \leftarrow i$
  Pour $j = i + 1$ à $longueur(A) - 1$ **faire**
    Si $A[j] < A[m]$ **alors**
      $m \leftarrow j$
    Fin Si
  Fin Pour
  Echanger $A[i]$ et $A[m]$
Fin Pour

SORTIE : Une liste triée A

Exercice 2

Écrire cet algorithme sous Python. Vous pouvez utilisez une fonction d'échange permettant d'échanger les éléments A[i] et A[j], ... ou pas (cf exercice 3)

Exercice 3

Une des particularités de Python est qu'il permet de s'affranchir de la variable dite tampon *tmp* dans la fonction d'échange. On peut très facilement échanger les valeurs de a et b par l'instruction :

$$a, b = b, a$$

Modifier donc votre fonction de tri avec cette astuce propre à Python si vous ne l'avez pas déjà fait.

Code Python

```
# Dans l'éditeur PYTHON
def tri_Par_Selection(A):
    """
    In: A une liste de int ou float ou str
    Out: A une liste triée de int ou float ou str
    tri croissant de A
    """
    ...
    return A
```

Exercice 4

Bonus : Plus délicat ...

Écrire une nouvelle fonction de tri par sélection qui utilise la fonction mini de l'exercice 1.

2.2 Complexité

Méthode

Calculons la complexité du tri par sélection

- Étape 1 : n opérations. On parcourt le tableau entièrement à la recherche de la plus petite valeur ;
- Étape 2 : $n-1$ opérations. On parcourt le tableau entièrement sauf la 1ère valeur.
- Étape 3 : $n-2$ opérations. On parcourt le tableau entièrement sauf les deux 1ère valeurs.
- ...
- Étape $n-1$: 2 opérations. On parcourt les deux dernières valeurs du tableau.
- Étape n : 1 opération. On parcourt la dernière valeur du tableau.

Conclusion : Il y a

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

La fonction de complexité du tri par sélection est un polynôme du second de degré. On dit que la complexité est en $O(n^2)$ qui se lit "grand O de n^2".

On peut vérifier cette propriété en mesurant les temps de calculs avec le code suivant :

Exercice 5

1. La fonction **random.randrange(0, 100)** du module **random** renvoie un entier aléatoire entre 0 et 100. Écrire une fonction **alea(i)** qui renvoie une liste de i entiers compris entre 0 et 100.
2. Compléter la fonction **TempsAlgo(i)** qui en entrée prend entier i, la taille d'un tableau et renvoie le temps mis par l'algorithme de tri par sélection.
   ***Aide** : la fonction **time.time()** du module **time** : donne l'heure en seconde. Appelez-la avant puis après de faire le tri du tableau.*
3. A partir de quelle taille de tableau le temps semble-t-il dépasser 1 seconde? 2 secondes? 5 secondes?

Code Python

```
# Dans l'éditeur PYTHON
import matplotlib.pyplot as plt   #Pour faire des graphiques
import random                     #Pour faire de l'aléatoire
import time                       #Pour manipuler le temps

def alea(i):
    '''In : i un entier, la taille du tableau
      Out : une liste de i entiers de 0 à 100'''
    return ...

def TempsAlgo(i):
    '''In : i un entier, la taille du tableau
    Out : le temps mis par l'algorithme de tri par séléction'''
    A = alea(i) # On fait un tableau rempli de i valeurs aléatoires
    a = time.time() #times.times() donne l'heure en seconde.
    ...
    return ...
```

Exercice 6

Compléter maintenant l'algorithme ci-dessous afin d'obtenir un graphique du temps mis par l'algorithme de tri en fonction de la taille du tableau.
Pour utiliser **Matplotlib**, on a besoin de deux listes, l'une avec les tailles de tableaux à trier, l'autre avec les temps correspondant.

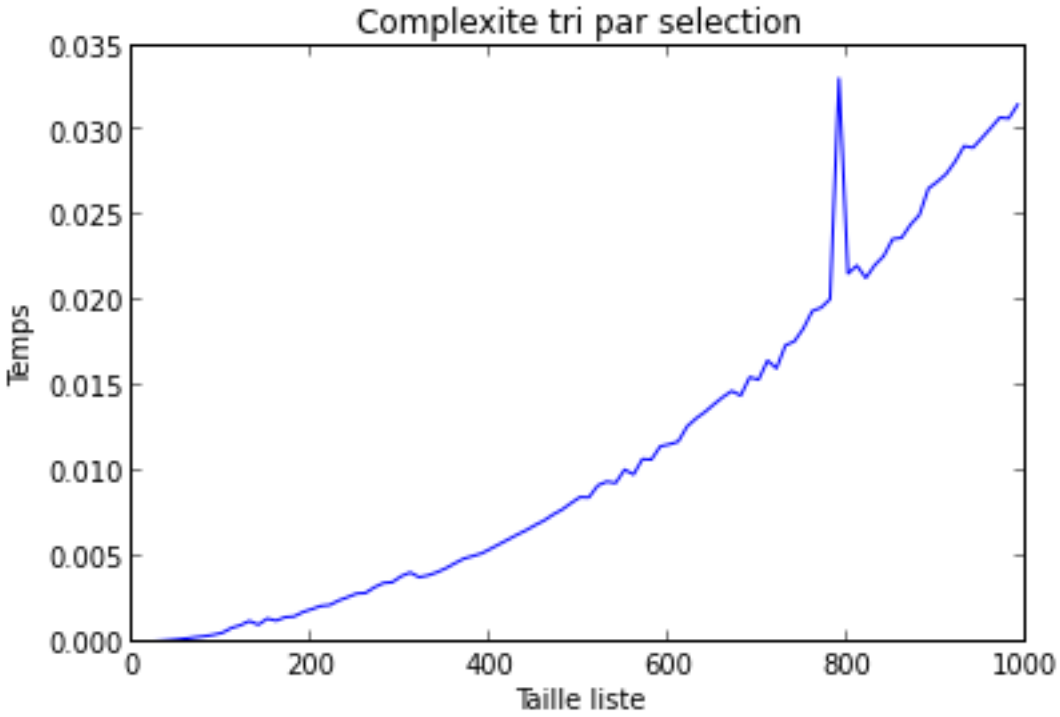
Code Python

```
# Dans l'éditeur PYTHON
import matplotlib.pyplot as plt   #Pour faire des graphiques
import random                     #Pour faire de l'aléatoire
import time                       #Pour manipuler le temps

taille=[] #Un tableau qui contient les tailles de tableau
temps=[]  #Un tableau qui contient les temps de calculs
for i in range(1000,10000,1000): #On fait varier les tailles de tableau
    ...

fig = plt.figure() # nécessaire sur repl.it
plt.title("Complexite tri par selection")
plt.plot(taille,temps)#On trace la figure les tailles en abscisse,
                      #les temps en ordonnées
plt.xlabel('Taille liste')
plt.ylabel('Temps')
fig.savefig('graph.png') # nécessaire sur repl.it, sinon simplement plt.show()
plt.show() #On montre le graphe.
```

On peut obtenir un graphe qui ressemble à celui ci :

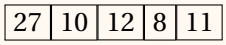


3 Tri par insertion

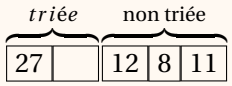
C'est le tri souvent utilisé par les joueur de carte. C'est un tri stable et en place. On divise le tableau en deux parties. A gauche une partie triée et à droite une partie non triée.
On commence par un exemple (A faire avec des cartons numérotés au tableau) :

Exemple

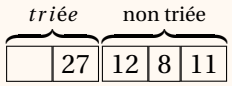
On va trier la liste à 5 éléments dans l'ordre croissant :



On prend l'élément le plus à gauche de la liste non triée.
On va insérer $\boxed{10}$ dans la liste triée :

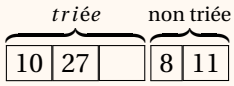


Comme $\boxed{10} < \boxed{27}$, il faut décaler $\boxed{27}$:

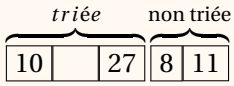


Et mettre $\boxed{10}$ au bon endroit :

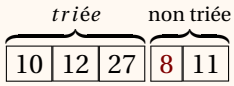
On prend l'élément le plus à gauche de la liste non triée.
On va insérer $\boxed{12}$ dans la liste triée :



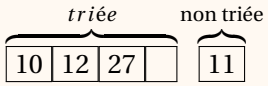
Comme $\boxed{12} < \boxed{27}$, il faut décaler $\boxed{27}$:



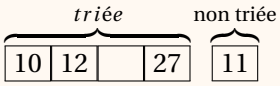
Comme $\boxed{10} < \boxed{12}$, on peut mettre $\boxed{12}$ au bon endroit :



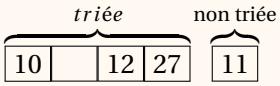
On prend l'élément le plus à gauche de la liste non triée.
On va insérer $\boxed{8}$ dans la liste triée :



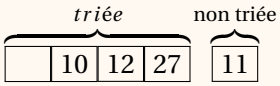
Comme $\boxed{8} < \boxed{27}$, il faut décaler $\boxed{27}$:



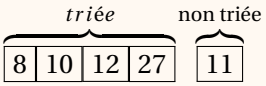
Comme $\boxed{8} < \boxed{12}$, il faut décaler $\boxed{12}$:



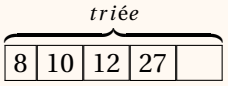
Comme $\boxed{8} < \boxed{10}$, il faut décaler $\boxed{10}$:

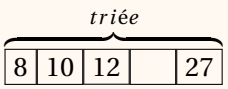


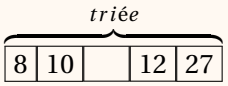
Et mettre $\boxed{8}$ au bon endroit :

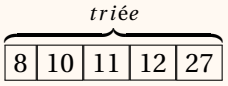


Il reste $\boxed{11}$ à insérer :









3.1 Algorithme

Finalement l'algorithme se réduit à la fonction suivante :

Algorithm 2 Tri_Par_Insertion(A) :

ENTRÉE : A est une liste avec une relation d'ordre comme les entiers, les flottants ou les chaînes de caractères.
  Pour $i = 1$ à $longueur(A) - 1$ **faire**
    Inserer $A[i]$ dans la liste $A[0...i[$
  Fin Pour
SORTIE : Une liste triée A

On voit donc que c'est la fonction d'insertion qui est la plus délicate :

Algorithm 3 insertion(A,i) :

ENTRÉE : A est une liste avec une relation d'ordre comme les entiers, les flottants ou les chaînes de caractères, i est entier qui est l'indice de l'élément à insérer dans A[0...i[
  $m = A[i]$
  Tant que $i > 0$ et $m < A[i-1]$ **faire**
    $A[i] = A[i-1]$
    $i \leftarrow i - 1$
  Fin Tant que
  $A[i] = m$
SORTIE : Une liste triée A

Exercice 7

Écrire le code Python correspondant.

Code Python

```
# Dans l'éditeur PYTHON
def insertion(t,i):
    """
    In: t une liste de int, float ou str. i est un int
    Out:t une liste de int, float ou str.
    Insertion de t[i] dans t[0...i[
    """
    ...
    return t
def tri_par_insertion(t):
    """
    In: t une liste de int, float ou str.
    Out:t une liste de int, float ou str.
    Tri croissant de t par insertion
    """
    ...
    return t
```

3.2 Complexité

Méthode

Calculons la complexité du tri par Insertion Dans le tri par sélection, il n'y a que des boucle for donc le nombre d'opération est constant. Dans le tri par insertion il y a une boucle while, le nombre d'opération change suivant le tableau. On va donc se limiter à supposer qu'on est dans le pire cas : celui où l'on fait toutes les comparaisons. D'abord il y a $n-1$ étapes.

- Étape 1 : 1 opérations. On compare les éléments t[0] et t[1] ;
- Étape 2 : 2 opérations. On compare les éléments t[1] et t[2] puis t[0] et t[2].
- Étape 3 : 3 opérations. On compare les éléments t[2] et t[3] puis t[1] et t[3] puis t[0] et t[3].
- ...
- Étape n-1 : $n-1$ opérations. On compare tous les éléments de t[0] à t[n-2] avec t[n-1].

Conclusion : Il y a

$$1+2+3+\cdots+n-2+n-1=\frac{(n-1)n}{2}=\frac{1}{2}n^2-\frac{1}{2}n \quad \text{opérations}$$

La fonction de complexité du tri par sélection est un polynôme du second de degré. On dit que la complexité est en $O(n^2)$ qui se lit "grand O de n^2". A première vue c'est légèrement meilleur que le tri par sélection puisque

$$\frac{1}{2}n^2-\frac{1}{2}n<\frac{1}{2}n^2+\frac{1}{2}n$$

Mais l'informaticien voit que dans les deux cas c'est un polynôme du second degré, il considère que les deux algorithmes ont une complexité équivalente en $O(n^2)$.

Exercice 8

On peut vérifier la complexité en mesurant les temps de calculs.
Comme dans l'exercice 6, obtenir un graphique du temps mis par l'algorithme de tri par insertion en fonction de la taille du tableau.
Pour utiliser **Matplotlib**, on a besoin de deux tableau, l'un avec les tailles de tableaux, l'autre avec les temps correspondant.

Exercice 9

Que se passe passe-t-il lorsque le tri par insertion est appliqué à un tableau qui se présente en ordre décroissant ?

Exercice 10

Ecrire une fonction $est\_trie(tab)$ qui renvoie $True$ si le tableau tab est trié dans l'ordre croissant et $False$ sinon.

Exercice 11

Au lieu de modifier le contenu du tableau pour le trier, on peut aussi renvoyer le résultat du tri dans un nouveau tableau. Écrire une fonction $tri\_par\_insertion\_externe$ qui réalise cette idée avec le tri par insertion. On pourra se servir de la fonction $insertion$.

Exercice 12

Écrire une fonction $plus\_petits$ qui prend en paramètres un tableau t et un entier k supposé inférieur à la longueur de t. Cette fonction renvoie un tableau contenant dans l'ordre les k plus petits éléments de t. On pourra se servir de la fonction $insertion$.

Exercice 13

En se servant du tri par insertion, écrire ne fonction qui prend en argument un tableau d'entiers et renvoie la valeur la plus fréquente du tableau. Indication : Une fois le tableau trié, les valeurs égales se retrouvent cote à cote.

Exercice 14

Si on sait que les valeurs d'un tableau sont des entiers pas trop grands, on peut trier le tableau en comptant le nombre d'occurrences de chaque valeur, par ordre croissant. On appelle cela le tri par dénombrement. Supposons que les éléments sont des entiers positifs ou nuls et plus petits qu'une constante $maxv$ (par exemple 100). Écrire une fonction $tri\_par\_denombrement$.