Chapitre 1 : Les algorithmes de tris par insertion et par sélection

Dans ce chapitre on considère un tableau `T` d'entiers que l'on veut trier par ordre croissant.

Spécification d'un algorithme de tri

Entrée/Sortie : tableau T (de taille n constitué d'entiers)
Rôle : trier T par ordre croissant
Précondition : T non vide
Postcondition : T est trié c'est-à-dire que chaque élément de T est inférieur ou égal à l'élément suivant : $\forall i \in [0, Taille(T) - 2]$, $T[i] \leq T[i+1]$

Remarque : on ne doit pas parler dans la spécification de *comment* trier !

I. Tri par sélection

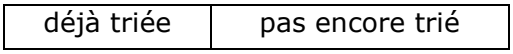
A faire **Activité 1 : Découverte de deux algorithmes de tri - Partie 1**

A. Algorithme

Voici l'algorithme du « tri par sélection » écrit en français :

- Rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 ;
- Rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1 ;
- Continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

Le tri par sélection parcourt ainsi le tableau de la gauche vers la droite, en maintenant sur la gauche une partie déjà triée et à sa place définitive :



Tri par sélection en Python

```
def echange(T, i, j):
    """échange T[i] et T[j]"""
    temp = T[i]
    T[i] = T[j]
    T[j] = temp

def tri_par_selection(T):
    """trie le tableau T dans l'ordre croissant"""
    for i in range(len(T)):
        ind_min = i
        for j in range(i+1, len(T)):
            if T[j] < T[ind_min]:
                ind_min = j
        echange(T, i, ind_min)
```

B. Efficacité de l'algorithme

On rappelle que l'efficacité d'un algorithme correspond à son coût en terme d'opérations élémentaires *dans le pire cas*. Pour simplifier cette étude, nous n'allons étudier ici que le nombre de comparaisons du tri par sélection.

- Au premier passage dans la boucle, il y a n-1 comparaisons dans la recherche du plus petit élément ;
- Au deuxième passage, il y en a n-2 comparaisons ;
- Ainsi de suite ;
- Au dernier passage, il n'y a plus de comparaison car il n'y a plus qu'une valeur à examiner.

Il y a donc en tout $(n-1)+(n-2)+\cdots+1+0$ comparaisons.

Propriété : Pour tout entier naturel n, on a :

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$$

Démonstration :

Bilan : Il y a dans tous les cas (donc dans le pire cas), $\frac{n(n-1)}{2}$ comparaisons à faire.
Comme $\frac{n(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}$, on peut alors dire que le coût de l'algorithme est de l'ordre de n^2.

A retenir

La complexité du **tri par sélection** est donc de l'ordre de n^2. On dit que son **coût est quadratique** dans le pire cas, ce qui se note $O(n^2)$ (se lit « grand O de n^2 »).
Cela signifie que si on double la taille initiale du tableau T, alors le temps est de calcul (théorique) n'est pas doublé mais est multiplié par 4.

II. Tri par insertion

Un autre algorithme de tri est le tri par insertion. Il s'agit de l'algorithme « naturel » que l'on applique pour trier des cartes par ordre croissant.

A faire **Activité 1 : Découverte de deux algorithmes de tri - Partie 2**

A. Algorithme

Voici l'algorithme du « tri par insertion » écrit en français :

- Prendre le deuxième élément du tableau et l'insérer à sa place parmi les éléments qui le précède

- Prendre le troisième élément du tableau et l'insérer à sa place parmi les éléments qui le précède
- Continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

Le tri par sélection parcourt donc également le tableau de la gauche vers la droite, en maintenant une partie déjà triée sur la gauche :

déjà triée	pas encore trié

Au lieu de chercher la plus petite valeur dans la partie de droite, le tri par insertion va insérer la première valeur non encore triée au bon endroit dans la partie de gauche déjà triée. (La partie de gauche est donc amenée à évoluer avec les insertions successives).

Tri par insertion en Python

```
def tri_par_insertion(T):
    """trie le tableau T dans l'ordre croissant"""
    for i in range(1,len(T)):
        x = T[i]
        j = i
        while j>0 and x < T[j-1]:
            T[j] = T[j-1]
            j = j - 1
        T[j] = x
```

B. Efficacité de l'algorithme

Pour simplifier cette étude, nous n'allons étudier ici que le nombre de comparaisons de deux éléments du tableau (qui est le nombre de décalages à 1 près à chaque itération). Dans le *pire des cas*, les éléments du tableau de départ sont rangés dans l'ordre décroissant. En effet, dans ce cas, il faut alors faire « remonter » successivement chaque élément de la partie non triée *au début* de la partie triée, ce qui occasionne le maximum de décalages et donc de comparaisons.

Plus précisément :

- Au premier passage dans la boucle (i = 1), on doit insérer le 2ème élément en première position : il y a 1 comparaison à faire (T[1] est comparé à T[0]) ;
- Au deuxième passage (i = 2), on doit insérer le 3ème élément en première position : il y a 2 comparaisons à faire (T[2] est comparé successivement à T[1] et T[0]) ;
- Ainsi de suite ;
- Au dernier passage (i = n − 1), on doit insérer le dernier élément en première position : il y a n − 1 comparaisons à faire (T[n-1] est comparé successivement à T[n-2], T[n-3], …, T[0]).

Bilan : Il y a donc en tout : $1+2+\cdots+n-1=\frac{n(n-1)}{2}$ comparaisons à faire dans le pire cas. Le tri par insertion a donc également un coût quadratique dans le pire cas.

Remarque : Dans le meilleur cas le tableau T donnée en entrée est déjà trié et dans ce cas, à chaque passage dans la boucle il n'y a qu'une seule comparaison à faire pour constater qu'il n'y aucun décalage à faire. Comme il y n-1 itérations de boucle, cela fait en tout n-1 comparaisons. Ainsi, dans le meilleur cas, le tri par insertion a un coût linéaire (car de l'ordre de n). Dans le cas où le tableau d'entrée est « presque trié », le

nombre de comparaisons à effectuer est également intéressant par rapport au tri par sélection qui effectue dans tous les cas $\frac{n(n-1)}{2}$ comparaisons.

A retenir

Le **tri par insertion** a également un **coût quadratique** dans le pire cas. Il possède en revanche un coût linéaire dans le meilleur cas (si le tableau d'entrée est trié).

III. Les tris fournis par Python

A. Des tris beaucoup plus efficaces

Il existe d'autres algorithmes de tris et certains sont (beaucoup) plus efficaces (leur coût est de l'ordre de n log(n) qui est inférieur à n²).
Python fournit notamment des fonctions permettant de trier de manière plus efficace un tableau. Elles se présentent de deux façons différentes, selon que l'on veuille obtenir une copie triée du tableau, sans le modifier (`sorted`), ou au contraire modifier le tableau pour le trier (`sort`).

La fonction `sorted` prend en argument un tableau et renvoie un *nouveau tableau*, trié, contenant les mêmes éléments.

```
>>> t = [12, 5, 3, 6, 8, 10]
>>> sorted(t)
[3, 5, 6, 8, 10, 12]
>>> t
[12, 5, 3, 6, 8, 10]
```

On voit que le tableau `t` n'a pas été modifié.

La construction `t.sort()`, en revanche, modifie le tableau `t` pour le trier sans rien renvoyer.

```
>>> t.sort()
>>> t
[3, 5, 6, 8, 10, 12]
```

Ces deux fonctions sont largement plus efficaces que les tris par sélection et par insertion. Trier un tableau d'un million d'éléments, par exemple, est instantané.

B. Application : trier des tables

On peut utiliser ces deux fonctions pour trier efficacement des tables de données.

A faire **Activité 2 : Notebook « Tri (et fusion) de tables »**

Pour trier une table de données avec la fonction sorted (idem avec sort), il faut lui donner un paramètre ce que l'on appelle une clé : il s'agit d'une fonction qui prend en paramètre un objet de la table (le dictionnaire correspond à une ligne de la table) et qui renvoie la valeur que l'on souhaite comparer.

Exemple : Considérons la table d'élèves qui est mémorisée dans le tableau `eleves`.

Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS

prénom	jour	mois	année	sexe	projet
Louan	13	4	2003	G	être heureux
Mael	29	3	2003	G	manger une glace
Alexis	20	10	2003	G	gagner au loto
...	...	...	...	...	...

```
eleves = [{'prénom': 'Louan',
'jour': '13', 'mois': '4', 'année':
'2003', 'sexe': 'G', 'projet':
'être heureux'}, {'prénom': 'Mael',
'jour': '29', 'mois': '3', 'année':
'2003', 'sexe': 'G', 'projet':
'manger une glace'}, {'prénom':
'Alexis', 'jour': '20', 'mois':
'10', 'année': '2003', 'sexe': 'G',
'projet': 'gagner au loto'}, ...]
```

Pour trier cette table par ordre alphabétique (des prénoms), on définit la fonction suivante dans laquelle la variable `x` désigne un dictionnaire de la table (= une ligne de la table donc un élève).

```
def prenom(x):
    return x["prénom"]
```

Il suffit ensuite de la passer en paramètre à la fonction `sorted` comme suit.

```
tri_eleves = sorted(eleves, key=prenom)
```

La nouvelle table `tri_eleves` ainsi créée contient les données de la table `eleves` triées par ordre alphabétique des prénoms.

```
[{'prénom': 'Alexis', 'jour': '20', 'mois': '10', 'année': '2003', 'sexe':
'G', 'projet': 'gagner au loto'}, {'prénom': 'Arthur', 'jour': '14',
'mois': '1', 'année': '2003', 'sexe': 'G', 'projet': 'devenir quelqu'un de
célèbre'}, ...]
```

Pour trier selon plusieurs critères, il suffit que la fonction renvoie un n-uplet avec les différents critères dans l'ordre souhaité. Par exemple, si on passe la fonction suivante en argument à la fonction `sorted`, alors le tri de la table `eleves` se fera d'abord par année de naissance puis, en cas d'égalité, par ordre alphabétique des prénoms.

```
def annee_puis_prenom(x):
    return int(x["année"]), x["prénom"]
```

On dit que la fonction `sorted` réalise l'*ordre lexicographique* : elle compare les deux premiers éléments puis (si nécessaire) les deux suivants, etc.

A retenir

Le **tri par sélection** et le **tri par insertion** sont deux algorithmes de tri élémentaires, qui peuvent être utilisés pour trier des tableaux. Ces deux algorithmes ont un coût quadratique dans le pire cas. Le tri par insertion a un meilleur comportement que le tri par sélection lorsque le tableau est presque trié. Les deux restent peu efficaces dès que les tableaux contiennent plusieurs milliers d'éléments. Il existe de meilleurs algorithmes de tri, plus complexes, dont celui offert par Python avec les fonctions `sort` et `sorted`. Il est possible de trier des tables de données avec n'importe quel algorithme de tri mais l'utilisation de ces deux fonctions est très simple : il suffit de leur passer en paramètre une *fonction clé* qui définit les critères de tri souhaité.

Ressources :

- Documents ressources du DIU EIL, Université de Nantes.
- *Numérique et Sciences Informatiques*, T. BALABONSKI, S. CONCHON, J.-C. FILLIATRE, K. NGUYEN, Ellipses.