Numérique et sciences informatiques
Classe de première

D Algorithmique
2 Deux algorithmes de tri

On dispose d'un tableau de nombres. On souhaite « trier ce tableau » : c'est-à-dire obtenir un tableau qui comporte les mêmes éléments mais classés par ordre croissant. Par exemple trier le tableau suivant par ordre croissant

`[48, 15, 12, 84, 3, 57, 142, 35, 48, 14, 15, 57, 48]`

c'est obtenir le tableau :

`[3, 12, 14, 15, 15, 35, 48, 48, 48, 57, 57, 84, 142]`

On voit notamment que le tri *conserve le nombre d'occurrences d'un même nombre* : le tableau initial comportait trois fois de nombre 48 et donc que le tableau trié comporte lui aussi trois fois de nombre 48.

Travailler sur un tableau trié facilite grandement la manipulation du tableau : on peut par exemple trouver très facilement la valeur minimum ou la valeur maximum, savoir si une valeur se trouve dans le tableau et, si oui, en combien d'occurrences.

Il existe de très nombreux algorithmes de tri. Nous allons voir deux algorithmes de tri très classiques qui sont relativement simples à comprendre et à programmer mais qui ne sont pas parmi les plus efficaces.

I L'algorithme de tri par sélection

I.1 Le principe de l'algorithme de tri par sélection

L'algorithme de tri par sélection consiste à :

* rechercher l'élément minimum du tableau et d'échanger sa place avec le 1er élément du tableau ;
* rechercher l'élément minimum de la partie du tableau située à partir de la 2ème place et d'échanger sa place avec le 2ème élément du tableau ;
* rechercher l'élément minimum de la partie du tableau située à partir de la 3ème place et d'échanger sa place avec le 3ème élément du tableau ;

etc.

Au fur et à mesure de l'avancement de l'algorithme, la partie gauche du tableau contiendra les plus petits éléments du tableau classés par ordre croissant

On poursuit cette procédure ainsi jusqu'à ce que l'on soit arrivé au dernier élément du tableau qui se trouve alors être l'élément maximum du tableau. Le tableau est alors trié par ordre croissant.

I.2 Un exemple

On considère le tableau suivant :

partie triée | **partie non triée**

`[9, 1, 2, 10, 6, 2, 5, 8]`

Les étapes successives du tri par sélection seront alors les suivantes :

• On trouve l'élément minimum du tableau, de valeur 1 (en 2ème position) et l'on échange sa place avec l'élément en 1ère position (de valeur 9). On obtient :

partie triée | **partie non triée**

[1, 9, 2, 10, 6, 2, 5, 8]

• On trouve l'élément minimum de la partie du tableau située à partir de la 2ème position, de valeur 2 (en 3ème position) et l'on échange sa place avec l'élément en 2ème position (de valeur 9). On obtient :

partie triée | partie non triée
[1, 2, 9, 10, 6, 2, 5, 8]

• On trouve l'élément minimum de la partie du tableau située à partir de la 3ème position, de valeur 2 (en 6ème position) et l'on échange sa place avec l'élément en 3ème position (de valeur 9). On obtient :

partie triée | partie non triée
[1, 2, 2, 10, 6, 9, 5, 8]

• On trouve l'élément minimum de la partie du tableau située à partir de la 4ème position, de valeur 5 (en 7ème position) et l'on échange sa place avec l'élément en 4ème position (de valeur 10). On obtient :

partie triée | partie non triée
[1, 2, 2, 5, 6, 9, 10, 8]

La suite de l'algorithme donne les étapes suivantes :

[1, 2, 2, 5, 6, 9, 10, 8]
[1, 2, 2, 5, 6, 8, 10, 9]

On obtient finalement le tableau trié par ordre croissant :

[1, 2, 2, 5, 6, 8, 9, 10]

I.3 Programmer l'algorithme de tri par sélection en Python

Pour programmer cet algorithme de tri, on doit programmer :

* une fonction `echanger(tab, i, j)` qui place l'élément d'indice i à l'indice j et l'élément d'indice j à l'indice i dans le tableau `tab`.
* une fonction `minTab(tab, g)` qui renvoie *l'indice de* l'élément minimum de la partie du tableau située à partir de l'indice `g` compris (c'est-à-dire à droite de cet indice) ;
* la fonction `tri_selection(tab)` qui met en œuvre le tri par sélection du tableau `tab` en utilisant les fonctions `minTab(...)` et `echanger(...)`.

Elle mettra donc en œuvre l'algorithme suivant :

```
lg <- longueur du tableau tab
Pour i de 0 à lg-2 :
  m <- minTab(tab, i)
  echanger(tab, i, m)
Fin pour
```

Remarque : L'algorithme fonctionne même si la boucle `Pour` va seulement de 0 à `lg-2` car, bien sûr, « échanger le dernier élément du tableau avec lui-même » ne modifie pas le tableau.

• On obtient le code suivant pour les fonctions `minTab`, `echanger` et `tri_selection` :

```
def minTab(tab, g=0) :
    """donne l'indice du plus petit element du
tableau tab situé à droite de l'indice g """
    indMin, mini = g, tab[g]
    for i in range(g+1, len(tab)) :
        if tab[i] < mini :
            indMin = i
            mini = tab[i]
```

```
    return indMin

def echanger(tab, a, b) :
    """echange en place de l'élément d'indice a
       avec l'élément d'indice b dans le tableau
tab"""
    tab[a], tab[b] = tab[b], tab[a]

def tri_selection(tab) :
    """tri en place par sélection"""
    for i in range(len(tab)-1) :
        m = minTab(tab, i)
        echanger(tab, i, m)
```

II L'algorithme de tri par insertion

II.1 Le principe de l'algorithme de tri par insertion

L'algorithme de tri par insertion d'un tableau consiste à :

* «insérer » le 2ème élément du tableau juste avant ou juste après le 1er élément selon qu'il soit plus petit ou plus grand que ce 1er élément ;
* «insérer » le 3ème élément du tableau au bon endroit par rapport aux deux premiers éléments du tableau de telle façon que les trois premiers éléments soient classés par ordre croissant ;
* «insérer » le 4ème élément d'un tableau au bon endroit par rapport aux trois premiers éléments du tableau de telle façon que les quatre premiers éléments soient classés par ordre croissant ;

etc.

Au fur et à mesure de l'avancement de l'algorithme, la partie gauche du tableau sera triée par ordre croissant

On poursuit cette procédure ainsi jusqu'à ce que l'on soit arrivé au dernier élément du tableau qui doit lui aussi être inséré au bon endroit par rapport aux éléments qui se trouvent à situés avant. Le tableau est alors trié par ordre croissant.

II.2 Un exemple

On considère le tableau suivant :

partie triée | partie non triée

[10, 6, 10, 4, 8, 6, 2, 4]

* On « insère » le 2ème élément (de valeur 6) avant le 1er élément (de valeur 10). On obtient donc :

[6, 10, 10, 4, 8, 6, 2, 4]

* On laisse le 3ème élément (de valeur 10) à sa place puisqu'il n'est pas inférieur à l'élément précédent (de valeur 10 aussi). On obtient donc :

[6, 10, 10, 4, 8, 6, 2, 4]

* On « insère » le 4ème élément (de valeur 4) au début du tableau puisqu'il est inférieur à tous les éléments précédents. On obtient donc :

[4, 6, 10, 10, 8, 6, 2, 4]

* Les étapes suivantes sont les suivantes :

[4, 6, 8, 10, 10, 6, 2, 4]
[4, 6, 6, 8, 10, 10, 2, 4]
[2, 4, 6, 6, 8, 10, 10, 4]

```
[2, 4, 4, 6, 6, 8, 10, 10]
```

Le dernier tableau obtenu est bien le tableau trié par ordre croissant.

II.3 Programmer l'algorithme de tri par insertion

Pour programmer le tri par insertion, on doit programmer :

• Une fonction `inserer(val, tab, d)` qui insère la valeur `val` dans la partie du tableau `tab` située entre l'indice 0 et l'indice `d` compris. On suppose que

* cette partie de tableau est triée par ordre croissant ;
* la longueur du tableau `tab` est au moins de `d+2` de façon à permettre le décalage d'un cran et l'insertion.

• La fonction `tri_insertion(tab)` dont le corps est la boucle que l'on peut écrire en pseudo-code de la façon suivante :

```
lg <- longueur du tableau tab
Pour i de 0 à lg-2 :
     inserer(tab[i+1], tab, i)
fin pour
```

• La principale difficulté est donc d'écrire le corps de la fonction `inserer(tab, val, d)`. Il s'agit de suivre l'algorithme suivant :

```
i <- d
Tant que i >= 0 et val<tab[i] :
  tab[i+1] <- tab[i]
  i <- i-1
Fin Tant que
tab[i+1] <- val
```

• On obtient le code suivant pour les fonctions `inserer(...)` et `tri_insertion(...)` :

```
def inserer(tab, val, d) :
    """insere la valeur 'val' dans la partie du
tableau 'tab' situee entre l'indice 0 et l'indice
'd' compris. L'element place a l'indice 'd'+1 sera
perdu. On suppose que :
    * cette partie de tableau est triee ;
    * la longueur du tableau 'tab' est au moins de
'd'+2."""
    i = d
    while i>=0 and val < tab[i] :
        tab[i+1] = tab[i]
        i -= 1
    tab[i+1] = val

def tri_insertion(tab) :
    for i in range(len(tab)-1):
        inserer(tab, tab[i+1], i)
```

III Étude de l'efficacité des deux algorithmes

Il existe de nombreux algorithmes de tri. Il est donc important de s'interroger si nos algorithmes de tri réalisent efficacement la tâche demandée, c'est-à-dire en un temps aussi court que possible.

• Bien entendu, la vitesse d'exécution d'un programme dépend de la rapidité de la machine qui exécute ce programme et de la taille des données qu'il doit traiter. Par exemple, un programme réalisant notre algorithme de tri par sélection s'exécutera plus rapidement si il est lancé sur une machine puissante et sur un petit tableau que s'il est lancé sur une machine lente et sur un gros tableau.

• Nous souhaitons ici nous intéresser à l'efficacité *de l'algorithme lui-même* qui doit être conçu comme indépendant de la machine sur lequel il est exécuté. Pour cela, nous devons nous faire une idée précise de la relation de dépendance entre :

* d'une part, le nombre d'opération effectués
* d'autre part, le nombre de données à traiter (par exemple le nombre d'éléments qui se trouvent dans notre tableau à trier).

III.1 Le tri par sélection

Supposons que le tableau à trier comporte n éléments. On effectue alors les $n-1$ étapes suivantes :

* rechercher le plus petit élément parmi les n éléments du tableau puis effectuer un échange de deux valeurs ;
* rechercher le plus petit élément parmi les $n-1$ derniers éléments du tableau puis effectuer un échange de deux valeurs ;

...

* rechercher le plus petit élément parmi les deux derniers éléments du tableau puis effectuer un échange de deux valeurs.

• A chaque étape, la recherche du plus petit élément parmi les k derniers éléments du tableau (par la fonction `minTab`) demande $k-1$ comparaisons (à laquelle s'ajoutent deux affectations).

• Ainsi le nombre d'instructions exécutées par l'algorithme de tri par sélection est proportionnel à :

$$(n-1)+(n-2)+\cdots+1$$

Or, on sait par le cours de mathématiques que

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

• De plus, si n est grand, le terme $\frac{1}{2}n$ sera négligeable devant $\frac{1}{2}n^2$. On peut donc retenir que le nombre d'opérations effectuées par l'algorithme de tri par sélection est **proportionnel à n^2**. On dit qu'il est d'une **complexité quadratique** et ou bien qu'il est **en $O(n^2)$**.

III.2 Le tri par insertion

Supposons toujours que le tableau à trier comporte n éléments.

• On effectue alors les $n-1$ étapes suivantes :

* insérer correctement le 2ème élément du tableau par rapport au premier élément du tableau ;

* insérer correctement le 3ème élément du tableau par rapport aux deux premiers élément du tableau classés par ordre croissant ;

...

* insérer correctement le $n^{\text{ème}}$ élément du tableau par rapport aux $n-1$ premiers éléments du tableau classés par ordre croissant.

• Chaque opération d'insertion d'un élément parmi k éléments classés par ordre croissants demande *dans le pire des cas* k opérations de comparaison. On peut donc en déduire que *dans le cas le plus défavorable* (qui est celui où les éléments du tableau initial se présentent par ordre décroissant) on effectuera le nombre d'opérations de l'ordre de :

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

• On peut donc retenir que *dans le pire des cas* le nombre d'instructions exécutées par l'algorithme de tri par insertion est **proportionnel à n^2**.
On dira donc encore que l'algorithme est d'une **complexité quadratique** ou **en $O(n^2)$**.
Toutefois il s'agit cette fois de la complexité *dans le pire des cas* et il est possible que, dans des situations favorables (si les éléments du tableau se présentent dans un ordre presque croissant) le nombre d'opérations soit nettement inférieur.

III.3 Comprendre la complexité quadratique

Il faut retenir le principe suivant pour un algorithme de complexité quadratique : multiplier par k la taille des données à traiter revient à multiplier par k^2 le nombre d'instructions à exécuter et donc le temps de traitement.
Ainsi,

* si on double la taille des données on doit s'attendre à multiplier par 4 (soit 2^2) le temps de traitement ;
* si on multiplie par 10 la taille des données, on doit s'attendre à multiplier par 100 (soit 10^2) le temps de traitement.

IV Prolongements sur les tris

IV.1 Tri en place et tri sans modification

b) Un nouvel objectif et l'idée d'une recopie du tableau

Les implémentations des algorithmes de tri que nous avons proposées ont comme propriété que les tableaux passés en argument sont transformés lors des appels des fonctions `tri_selection(tab)` et `tri_insertion(tab)`. En effet, après exécution de ces appels, le tableau `tab` conserve les mêmes éléments mais ceux-ci apparaissent dans un ordre croissant. On dit alors que l'on a effectué un **tri en place**.

• Il est parfois souhaitable d'implémenter une fonction de tri qui *renvoie un tableau trié* mais qui fasse en sorte que le tableau passé en argument ne soit pas modifié par l'appel de la fonction de tri. Pour obtenir ce résultat, il faut apporter quelques changements à notre implémentation.

• Pour éviter la modification du tableau passé en argument, nous allons devoir dans un premier temps réaliser une copie de ce tableau grâce à l'instruction suivante :

```
tab_res = tab.copy()
```

• Les actions permettant de trier le tableau seront effectuées sur le tableau `tab_res` et une fois le trié achevé la fonction renverra le résultat par l'instruction suivante :

```
    return tab_res
```

• Bien entendu, puisque cette nouvelle fonction de tri renverra un tableau trié sans modification du premier tableau, il faudra l'appeler de la façon suivante :

```
tabTrie = tri_selection2(tab)
```

• Rappelons la différence entre l'instruction d'affectation classique :

```
tab2 = tab
```

et l'instruction :

```
tab3 = tab.copy()
```

* Dans le premier cas, les variables `tab` et `tab2` renverront *à l'adresse mémoire du même tableau physique* et donc toute modification appliquée à ce tableau par l'intermédiaire de la variable `tab2` (que ce soit dans le corps d'une fonction ou dans le programme principal) modifiera aussi les valeurs obtenues par l'intermédiaire du tableau `tab` (et vice versa).

* En revanche, l'instruction

```
tab3 = tab.copy()
```

réalise une véritable recopie du tableau pointé par la variable `tab3` *dans un tableau situé à un autre emplacement mémoire* que le premier tableau. Ainsi les modification appliquées au tableau par l'intermédiaire de la variable `tab3` n'auront aucun effet sur le tableau pointé par la variable `tab`.

b) Application au tri par sélection

• Nous pouvons alors créer une fonction `tri_selection2(tab)` de la façon suivante :

```
def tri_selection2(tab) :
    """tri par sélection sans modification du
tableau 'tab'"""
    tab_res = tab.copy()
    for i in range(len(tab_res)-1) :
        m = minTab(tab_res, i)
        echanger(tab_res, i, m)
    return tab_res
```

Remarquons que nous utilisons dans la fonction `tri_selection2(...)` les mêmes fonctions `echanger(...)` et `minTab(...)` qui n'ont pas été modifiées.

c) Application au tri par insertion

• Nous pouvons alors créer une fonction `tri_insertion2(tab)` de la façon suivante :

```
def tri_insertion2(tab) :
    tab_res = tab.copy()
    for i in range(len(tab_res)-1):
        inserer(tab_res, tab_res[i+1], i)
    return tab_res
```

Là aussi, nous réutilisons la fonction `inserer(...)` sans modification.

IV.2 Les fonctions de tri proposées par Python

Il existe bien entendu des fonctions de tri de tableau déjà implémentées de façon native dans le langage Python.

• Pour un tri sans modification on peut utiliser la fonction `sorted(...)` de la façon suivante :

```
tabTrie = sorted(tab)
```

Pour obtenir un tri *par ordre décroissant*, il suffit d'écrire :

```
tabTrie = sorted(tab, reverse=True)
```

Le tableau `tab` ne sera pas modifié et la variable `tabTrie` recevra une version triée de ce tableau.

• Pour un tri en place on peut utiliser la méthode `sort()` de la façon suivante :

```
tab.sort()
```

Pour obtenir un tri par ordre décroissant, il suffit d'écrire :

```
tab.sort(reverse=True)
```

Exemples :

```
>>> tab=[7, 89, 5, 51, 65, 84, 71, 97, 21, 14, 3, 46, 32]
>>> tab2=sorted(tab)
>>> tab
[7, 89, 5, 51, 65, 84, 71, 97, 21, 14, 3, 46, 32]
>>> tab2
[3, 5, 7, 14, 21, 32, 46, 51, 65, 71, 84, 89, 97]
>>> tab.sort()
>>> tab
[3, 5, 7, 14, 21, 32, 46, 51, 65, 71, 84, 89, 97]
```

• Il faut enfin savoir que la fonction `sorted(...)` et la méthode `sort(...)` mettent en œuvre des algorithmes *beaucoup plus efficaces* que les algorithmes de tri par sélection ou par insertion que nous avons implémentés.