Algorithmes de tri

1ère partie

I. Position du problème

On désigne par **tri** l'opération consistant à ordonner un ensemble d'éléments en fonction de **clés** sur lesquelles est définie une relation d'ordre total.

On se limitera ici à des données stockées dans une liste `L` ; à chaque élément `x=L[i]` d'indice i est associée une clé notée `cle(x)` qui renvoie une valeur numérique servant à la comparaison. Si le tableau est déjà formé de valeurs numériques, il suffira de prendre `cle(x)` égale à `x`.

Dans certains cas, au lieu d'une procédure `cle` qui renvoie une valeur numérique, on peut disposer d'une procédure `est-plus-petit(i,j)` qui permet de comparer les éléments `L[i]` et `L[j]`.

On peut également avoir besoin d'une procédure `echange (i,j)` qui permet d'échanger les éléments d'indices i et j.

Une **inversion** dans le tableau `L` est un couple d'indices (i, j) tels que `i<j` et `cle(L[i])>cle(L[j])`.

Le tableau sera dit **trié** s'il ne contient aucune inversion (on a défini ici un tri par ordre croissant des clés ; les algorithmes de tri par ordre décroissant s'en déduisent facilement).

Un algorithme de tri sera dit **stable** s'il ne modifie pas l'ordre de deux éléments dont les clés sont égales. Cela peut être un critère important si l'on trie selon une certaine clé des éléments déjà triés selon une autre.

Si l'algorithme ne requiert qu'un espace en mémoire constant en plus de la liste d'entrée pour le trier, on dit que le tri s'effectue **en place**. Dans ce chapitre, le tri bulle, le tri par sélection, le tri par insertion et le tri de Shell s'effectuent en place. Dans les tris qui seront étudiés plus tard, on verra que le tri fusion ne trie pas en place.

> Pour simplifier les exemples, nous ne considèrerons plus que des tableaux contenant des valeurs numériques. Il n'y aura donc pas besoin d'implémenter une fonction `cle`. Ces tableaux seront comme en Python numérotés de 0 à $n-1$. De plus, puisque les algorithmes que l'on va écrire sont simples, on donne directement le code Python (du pseudo-code serait une simple paraphrase).

II. Complexité

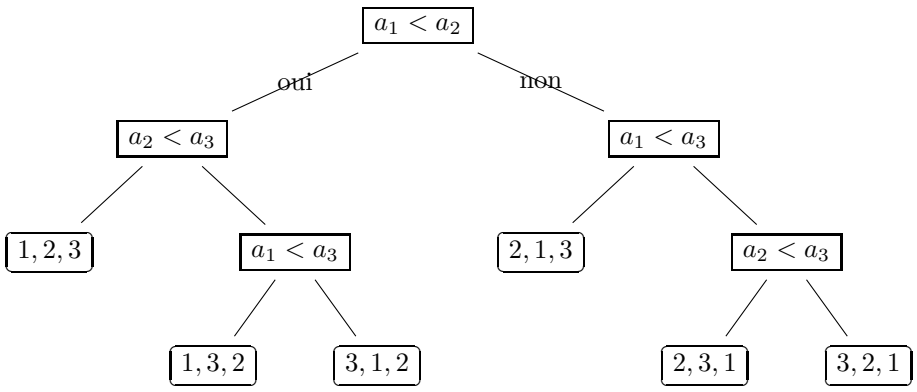
Pour estimer la complexité d'un algorithme de tri, on comptera deux types d'opérations distinctes :

- le nombre de comparaisons effectuées entre deux éléments de `L` ;
- le nombre d'affectations.

On supposera que ces deux opérations s'effectuent en temps constant.

Un algorithme de tri d'une liste de n éléments a une **complexité minimale** en $O(n)$: en effet, même si le tableau est déjà trié, il faudra au moins $n-1$ comparaisons pour s'en assurer !

La **complexité maximale** est, elle, d'au moins $O(n \ln n)$: en effet, si l'on considère l'arbre binaire de toutes les comparaisons que peut effectuer l'algorithme, cet arbre possède au moins $n!$ feuilles comme le montre l'exemple ci-dessous (pour $n = 3$) :



La hauteur h de cet arbre, qui est aussi le nombre minimal de comparaisons à faire dans le cas le plus défavorable, est donc le plus petit entier tel que $2^h \geqslant n!$ soit $h = \left\lfloor \dfrac{\ln n!}{\ln 2} \right\rfloor + 1$. Or par une banale méthode de comparaison série-intégrale, ou en utilisant la formule de Stirling, il est facile d'établir $\ln n! \underset{n\to+\infty}{\sim} n \ln n$, donc $h = O(n \ln n)$.

III. Le tri bulle

Le principe du tri bulle est le suivant : on part de la fin de la liste et l'on regarde les éléments consécutivement, en les échangeant s'ils ne sont pas dans l'ordre. A la fin d'un premier passage sur la liste, l'élément le plus petit se retrouve en tête de liste, puis on applique de nouveau le procédé sur la partie de la liste entre le second et le dernier élément etc.... C'est de là que provient le nom : l'algorithme imite la remontée de bulles dans un verre de champagne. Si la liste est de taille n, après n passages, les n éléments seront remontés et classées dans l'ordre.

```
def TriBulle(L):
# le paramètre est ici passé par adresse: la variable L est modifiée
# et la liste est triée en place. Pas besoin de return
    n = len(L)
    for i in range(n-1):
    # Invariant1(i): L[0:i] est triée et ses éléments sont inférieurs aux autres éléments de la lis
        for j in range(n-1, i, -1):
        # Invariant2(j):  L[j] est le plus petit des éléments de L[j:n]
            if L[j-1] > L[j]:
                L[j-1], L[j] = L[j], L[j-1]
```

Terminaison : L'algorithme est constitué de deux boucles `for` imbriquées, il termine donc !

Preuve : facile.

Complexité :

L'étude de la complexité du tri bulle est relativement simple. On s'intéresse au nombre de comparaisons effectuées par un tri bulle.Lors de chaque boucle interne, une comparaison est faite. La boucle interne comprend, pour chaque $i, n-i-1$ itérations, et i varie entre 0 et $n-2$. Le nombre total de comparaisons sera donc :

$$\sum_{i=0}^{n-2}(n-i-1) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}\cdot$$

La complexité est donc $\mathcal{O}(n^2)$.
Le nombre ci-dessus est le nombre de comparaisons ; le nombre d'échanges effectué peut être inférieur. Plus précisément :

- dans le meilleur des cas, il n'y aura aucun échange (liste déjà triée).
- dans le pire des cas, il y aura autant d'échanges que de comparaisons (liste déjà triée, mais en sens inverse).
- on peut alors chercher la complexité en moyenne. Le nombre d'échanges faits est égal au nombre d'inversions, puisque chaque échange en supprime une.
  Il s'agit donc de calculer le nombre moyen d'inversions parmi toutes les permutations. Pour toute permutation $\sigma = (x_1, \dots, x_n)$, notons $\overline{\sigma}$ la permutation $(x_n, \dots, x_1)$, de sorte que si (i,j) est une inversion dans σ ce n'en est pas une dans $\overline{\sigma}$ et réciproquement. Comme il y a $\dfrac{n(n-1)}{2}$ couples (i,j) possibles la somme du nombre d'inversions de σ et de $\overline{\sigma}$ est $\dfrac{n(n-1)}{2}$, donc le nombre moyen d'inversions et $\dfrac{n(n-1)}{4}$: il s'agit de la complexité moyenne, qui est de toutes façons encore un $\mathcal{O}(n^2)$.

Autre version :

Certains proposent une version soi-disant « améliorée » de l'algorithme précédent :

```
def TriBulle2(L):
    n = len(L)
    inversion = True
    while inversion:
        inversion = False
        for i in range(n-1):
            if L[i] > L[i+1]:
                inversion = True
                L[i], L[i+1] = L[i+1], L[i]
        n-=1
```

Exercice :

a) Prouver que cet algorithme se termine.

b) Trouver l'invariant de la boucle `while` et le prouver.

c) Que pensez-vous de cette version par rapport à la précédente ?

Solution:

a) n décrit une suite d'entiers strictement décroissante à chaque passage dans la boucle `while`, et si $n \leqslant 1$, la boucle `for` ne fait rien donc `inversion` reste à `False` et la boucle s'arrête.

b) L'invariant est :

`L[n:len(L)]` est triée, ses éléments sont plus grands que les autres éléments de la liste,
et de plus, `inversion=False` ou `L[0:n]` est triée.

c) La complexité minimale de cette version est un $\mathcal{O}(n)$, puisque si la liste est déjà triée, le programme s'arrête après un seul parcours (plus d'inversion).

Cependant, dans l'immense majorité des cas il faudra faire autant de passages que dans la 1ère version, mais on aura fait des affectations supplémentaires (sur la variable `inversion`) d'où un temps un peu plus long !

Conclusion :

Le tri bulle est certainement **le plus mauvais algorithme de tri !** Cela vient principalement du fait que chaque élément n'est comparé qu'à son voisin immédiat. Si par exemple l'élément le plus grand du tableau est à la première place mais que le reste du tableau est trié, on atteindra quand même la complexité maximale !

IV. Tri par selection

Le principe du tri par sélection est le suivant :

- sélectionner le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 ;
- sélectionner 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.

L'algorithme consiste donc en la recherche de $n-1$ minimums.

```
1  def TriSelection(L):
2      n = len(L)
3      for i in range(n-1):
4      # Invariant1(i): L[0:i] est triée et ses éléments sont inférieurs aux autres éléments de la lis
5          imin = i
6          for j in range(i+1,n):
7          # Invariant2(j): imin est l'indice du plus petit élément de L[i:j]
8              if L[j] < L[imin]:
9                  imin = j
10         if imin != i:
11             L[i], L[imin] = L[imin], L[i]
```

Terminaison : L'algorithme est constitué de deux boucles `for` imbriquées, il termine donc !

Preuve : Il faut déjà prouver l'invariant pour la boucle interne (celle sur `j`) :

- il est vrai pour `j=i+1` puisque le tableau `L[i:i+1]` se réduit à `L[i]` ;
- si Inv2(j) est vrai au début de la boucle (ligne 7), alors Inv2(j+1) est vrai à la fin de la boucle (après la ligne 9).

À la sortie de cette boucle où `j` varie de `i+1` à `n-1`, Inv2(n) est vrai c'est-à-dire que `imin` représente l'indice du plus petit élément de `L[i:n]`. Cela permet de démontrer ensuite l'invariant de la boucle externe (celle sur `i`), puisque l'on positionne ce plus petit élément à l'indice `i` par les instructions lignes 10 et 11.

À la sortie de cette boucle , Inv1(n-1) est vérifiée c'est-à-dire que la liste `L[0:n-1]` est triée et ses éléments sont inférieurs à `L[n]`, donc toute la liste est triée.

Complexité :

Le nombre de comparaisons effectuées est facile à calculer puisqu'il y a une comparaison par itération de la boucle interne. Cela en fait donc

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{n(n-1)}{2} .$$

La complexité de l'algorithme est donc $O(n^2)$. De ce point de vue, il n'est pas plus efficace que le tri bulle. Par contre, le tri par sélection n'effectue que peu d'échanges : on peut montrer que le nombre d'échanges est en moyenne un $O(n)$ (alors qu'il est un $O(n^2)$ dans le tri bulle) :

- $n-1$ échanges dans le pire cas, qui est atteint lorsque la liste est déjà triée à l'envers ;
- 0 échange dans le meilleur des cas (liste déjà triée) ;
- on s'intéresse au nombre moyen d'échanges : à l'étape i de la boucle externe, il reste à trier une liste de taille $n-i$; il n'y aura pas d'échange si et seulement si l'élément examiné t_i est déjà à sa place ; si on considère toutes les permutations comme équiprobables, la probabilité de cet évènement est $\dfrac{(n-i-1)!}{(n-i)!}$ soit $\dfrac{1}{n-i}$. La probabilité d'avoir un échange est donc $1-\dfrac{1}{n-i}$ et le nombre moyen d'échanges sera égal à $\displaystyle\sum_{i=0}^{n-1}\left(1-\frac{1}{n-i}\right) = n - \sum_{i=1}^{n}\frac{1}{i} = n - \ln n - \gamma + o(1)$.

Donc le nombre moyen d'échanges est un $O(n)$, ce qui est mieux que le tri bulle.

V. Tri par insertion

Le tri par insertion est un algorithme de tri naturel : on ajoute un élément à la fois en le mettant à sa place. C'est la méthode utilisée par les joueurs de cartes pour ranger leur main, ou par un enseignant pour ordonner des dossiers ou des copies...

Dans l'algorithme, on parcourt le tableau à trier du début à la fin. Au moment où on considère le i-ème élément, les éléments qui le précèdent sont déjà triés, et il s'agit d'insérer cet élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. En pratique, ces deux actions sont effectuées en une seule passe, qui consiste à faire remonter l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

```
1  def TriInsertion(L):
2      n = len(L)
3      for i in range(1, n):
4      # Invariant1(i): L[0:i] contient les i premiers éléments de L, triés.
5          x = L[i]
6          j = i-1
7          while (j >= 0) and (L[j] > x):
8          # Invariant2(j): les tableaux L[0:j+1] et L[j+1:i+1] sont triés et
9          # pour tout k entier de [j+2,i], on a  L[k] > x
10             L[j+1] = L[j]
11             j -= 1
12         L[j+1] = x
```

Terminaison : Il y a une boucle `while` dans une boucle `for`, il suffit donc de montrer que la boucle `while` se termine, et cela est évident puisque la suite des valeurs prises par `j` est strictement décroissante.

On notera que le test dans la boucle `while` est toujours correct : en effet, si `j=-1`, la condition `j >= 0` n'est pas vérifiée, donc il n'y a pas besoin d'évaluer `L[j]` qui irait chercher le dernier élément de la liste !

Preuve : On démontre déjà que la boucle `while` a bien l'invariant spécifié.
En effet, lorsque `j=i-1`, l'assertion est vraie puisque le tableau `L[0:i]` est trié (invariant1), le tableau `L[i:i+1]` aussi (évident)et que l'intervalle $[\![i+1\,;i]\!]$ est vide.
Et si Inv2(j) est vrai au début de la boucle (ligne 8), il le reste après l'exécution des lignes 10 et 11 : en effet, puisque j est devenu $j-1$, on a ajouté aux valeurs $k \in [\![j+2\,;i-1]\!]$ la valeur $k=j+1$, pour laquelle justement la propriété `L[k]>x` est vraie puisque `L[j+1]` a été remplacé par `L[j]` qui est $> x$ (on est dans la boucle).

En sortie de la boucle `while`, on a donc `j<0` (en fait, $j=-1$) OU (exclusif) `L[j] <= x`.

- Si $j=-1$, Inv2(j) s'écrit : $\forall k \in [\![1\,;i]\!]$, $L[k] > x$ et le tableau `L[0:i+1]` est trié, de sorte qu'après l'exécution de la ligne 12, le tableau `L[0:i+1]` est trié, c'est-à-dire Inv1(i+1) est vraie ;
- Sinon, on a `L[j] <= x =L[j+1]` donc le fait que les tableaux `L[0:j+1]` et `L[j+1:i+1]` sont triés implique que le tableau `L[0:i+1]` est trié, c'est-à-dire Inv1(i+1) est vraie.

L'invariant Inv1(i) étant évidemment vrai pour i=1, il le reste donc en sortie de la boucle `for`, c'est-à-dire que le tableau `L[0:n]` est trié.

Complexité :

Par la même méthode que pour les algorithmes précédents, on montre que le nombre maximum d'exécutions de la boucle est $\frac{n(n-1)}{2}$ (liste triée à l'envers) et que sa valeur moyenne est $\frac{n(n-1)}{4}$ (c'est encore le nombre moyen d'inversions). C'est donc encore un algorithme en $\mathcal{O}(n^2)$.

Cependant il a le gros avantage par rapport aux algorithmes précédents de ne pas faire d'échanges, mais seulement des décalages (donc une opération de moins à chaque fois).

De plus, dans le meilleur des cas (liste déjà triée), la complexité est en $\mathcal{O}(n)$ et surtout, la complexité reste en $\mathcal{O}(n)$ si le tableau est presque trié (par exemple, chaque élément est à une distance bornée de la position où il devrait être, ou bien si tous les éléments sauf un nombre borné sont à leur place). Dans cette situation particulière, le tri par insertion surpasse d'autres méthodes de tri : par exemple, le tri fusion et le tri rapide (avec choix aléatoire du pivot) sont tous les deux en $\mathcal{O}(n \ln n)$ même sur une liste déjà triée.

Conclusion

Le tri par insertion est considéré comme le tri le plus efficace sur des entrées de petite taille. Il est aussi très rapide lorsque les données sont déjà presque triées. Pour ces raisons, il est utilisé en pratique en combinaison avec d'autres méthodes comme le tri rapide.

Amélioration du programme Python

On peut sensiblement améliorer la rapidité du programme précédent en utilisant le *slicing* propre aux listes Python. En effet, à la i-ème étape, lorsqu'on insère `L[i]` dans la sous-liste `L[0:i]` supposée triée, on décale les éléments un par un. Il est plus rapide de chercher d'abord l'emplacement où doit aller `L[i]`, puis de décaler alors d'un seul coup tous les éléments à droite de cet emplacement, avant d'y mettre `L[i]`. Cela donne le programme suivant.

```
def TriInsertionOptimise(L):
    n = len(L)
    for i in range(1,n):
        x = L[i]
        j = i-1
        while (j >= 0) and (liste[j] > x):
            j -= 1
        L[j+2:i+1] = L[j+1:i]
        L[j+1] = x
```