Leçon d'informatique : tris

S. Benlhajlahsen - PCSI$_1$

Sommaire

Extrait du programme :

Thèmes	Exemples d'activité, au choix du professeur et non exigibles des étudiants. Commentaires
Tris	Algorithmes quadratiques : tri par insertion, par sélection. Tri par partition-fusion. Tri rapide. Tri par comptage. *On fait observer différentes caractéristiques (par exemple, stable ou non, en place ou non, comparatif ou non, etc).*

I Présentation

On a souvent besoin d'utiliser un ensemble trié d'éléments :

- pour déterminer le rang d'un élément,
- pour calculer la médiane,
- pour utiliser la recherche rapide d'un élément,
- pour sélectionner selon un critère.
- ...

L'ordre provient de comparaisons entre éléments : leur ensemble est muni d'une relation d'ordre total.

- Cela peut être des nombres ou des chaînes de caractères, dans ce cas la relation d'ordre est définie dans Python par les relations `<`, `<=`, `>`, `>=`.
- Dans des cas plus généraux, on devra définir une fonction à résultat booléen : par exemple, `plusGrand(a, b)` qui devra renvoyer `True` si `a` est strictement supérieur à `b` pour la relation de comparaison et `False` si `a` est inférieur ou égal à `b`.

Remarque : Dans toute la suite, on ne manipulera que des listes de nombres (flottants ou entiers). La liste sera **triée** par ordre croissant : l'élément d'indice i devra être inférieur ou égal à l'élément d'indice $i+1$.

Tri externe ou en place : La construction d'un algorithme de tri dépend du type de renvoi.

1. On peut souhaiter construire une nouvelle liste, triée, on parle alors de tri **externe**. L'avantage est qu'alors la liste initiale n'est pas détruite. Dans ce cas, la fonction devra renvoyer une nouvelle liste.
2. On peut souhaiter modifier la liste initiale, on trie **en place**. L'avantage est qu'on ne crée par une seconde liste ce qui peut être indispensable lorsque la liste à trier est très grande[a]. Dans ce cas, la fontion ne présentera pas de `return`.

a. on évite d'encombrer la mémoire

À retenir : La plupart des tris que nous étudierons seront des tris en place. Les fonctions de tris seront alors des fonctions sans instruction `return`.

Permutation : Dans le cas des tris en place, un outil souvent utilisé est une fonction de permutation qui échange deux éléments dans une liste. Une écriture de base est donnée en figures 1 et 2.

```
def echange(liste, i, j):
    temp = liste[i]
    liste[i] = liste[j]
    liste[j] = temp

L = [0, 1, 2, 3, 4, 5]
echange(L, 0, 1)
print(L)     #affiche [1, 0, 2, 3, 4, 5]
```

FIGURE 1 – Fonction d'échange.

```
def echange(liste,i,j):
    liste[i], liste[j] = liste[j], liste[i]

L = [0, 1, 2, 3, 4, 5]
echange(L, 0, 1)
print(L)   #affiche [1, 0, 2, 3, 4, 5]
```

FIGURE 2 – Fonction d'échange.

Cas d'égalité : Le résultat d'un tri est défini sans ambiguïté lorsque qu'il n'existe pas deux éléments égaux. On peut imposer que les éléments égaux pour la relation d'ordre se retrouvent dans le même ordre à la fin. Par exemple :

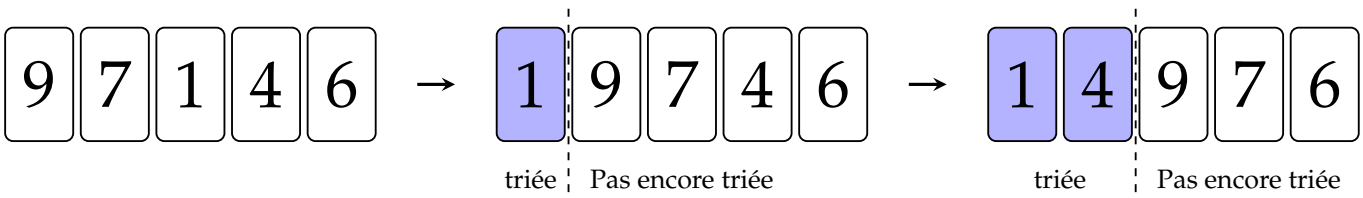
$$[10, 20, 4, 1, \underline{11}, 9, 11] \xrightarrow{\text{tri}} [1, 4, 9, 10, \underline{11}, 11, 20]$$

À retenir : Un tri est stable si des éléments égaux sont dans le même ordre dans le résultat final qu'à l'origine.

II Tri par sélection

Idée : Partant d'une liste non triée, on peut **sélectionner** le minimum de la liste que l'on place directement à gauche. Ensuite, dans la partie restante, on **sélectionne** de nouveau le minimum que l'on place en deuxième position : c'est le par sélection.

Remarque : Le tri par sélection est celui qui est l'un des plus instinctifs pour les joueurs de cartes. On prend une carte à la fois que l'on place à gauche de sa main.



À chaque étape, on a déterminé le minimum parmi les termes non triés et on l'a placé. On a donc besoin d'une fonction qui recherche l'indice du minimum à partir d'un rang.

Indice du minimum depuis `i` : En figure 3, on donne une implémentation de cette recherche.

Remarque : Comme on choisit la première apparition du minimum, cela assurera la stabilité du tri.

Tri par sélection : On peut résumer l'algorithme de la façon suivante. À chaque étape `i` :

- on détermine l'indice `k` du minimum depuis `i` (partie non triée) ;
- on échange les éléments en position `i` et `k`.

En figure 4, on donne une implémentation.

Terminaison : Les fonctions n'utilisent que des boucles `for`, on est donc assurer qu'elles terminent.

Correction/Preuve : On a construit la liste triée pas-à-pas. À la fin de la boucle, i a pris la valeur $n-1$ et la propriété implique que les $n-1+1$ premiers éléments de la liste triée sont à leur place : la liste est triée.
On doit donc s'assurer qu'on place le $i+1$-ième élément trié à la position i. Comme les i premiers éléments, les plus petits, sont déjà placés, on doit choisir le minimum de ceux qui restent.

```python
def indMinDepuis(liste, i):
    n = len(liste)
    ind_min = i
    mini = liste[i]
    for j in range(i+1,n):
        if liste[j]< mini:
            mini = liste[j]
            ind_min = j
    return ind_min

L = [20, 4, 10, 11, 3, 9]
print(indMinDepuis(L, 2)) # renvoie 4 car 3 est le min de [10, 11, 3, 9]
```

FIGURE 3 – Détermination de l'indice du minimum depuis `i`.

```python
def triSelection(liste):
    n = len(liste)
    for i in range(n):
        k = indMinDepuis(liste, i)
        echange(liste, i, k)

L = [10, 20, 4, 1, 11, 9]
triSelection(L)
print(L) # affiche [1, 4, 9, 10, 11, 20]
```

FIGURE 4 – Implémentation d'un tri par sélection.

On peut prouver que `indMinDepuis(liste, i)` détermine bien le minimum en prouvant que `mini` contient, au début de la boucle pour j, le minimum des valeurs de la liste entre i et $j-1$, c'est notre invariant de boucle.
Si on note a_k la valeur de `liste[k]` et $m_{i,k} = \min\{a_i, a_{i+1}, \ldots, a_k\}$, l'algorithme calcule $m_{i,j}$ en fonction de $m_{i,j-1}$ en utilisant la propriété $m_{i,j} = m_{i,j-1}$ si $a_j \geq m_{i,j-1}$ et $m_j = a_j$ sinon.

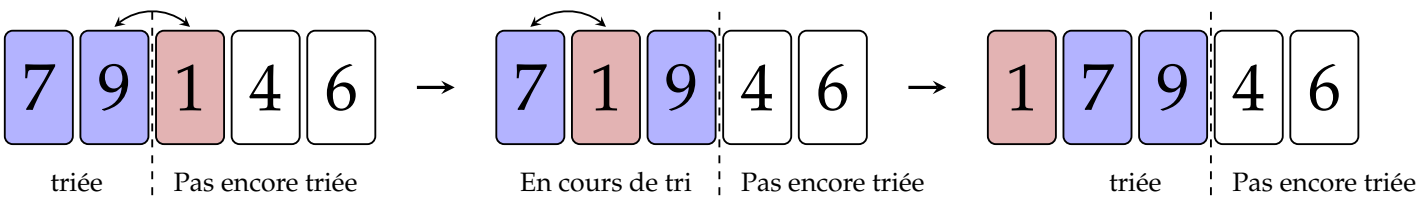
Complexité : On effectue une comparaison pour chaque j de $i+1$ à $n-1$ donc le nombre de comparaisons est $n-1-i$. On en déduit que le nombre de comparaisons de `triSelection` pour une liste de taille n est donc

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

La complexité est un $\mathcal{O}(n^2)$.

III Tri par insertion

Idée : On considère des cartes partiellement triée (voir ci-dessous). Les deux premieres cartes sont triées. On prend la troisième et on la compare à la deuxième et on échange éventuellement. On voit alors le déplacement progressif de la carte 1 vers la gauche. À la fin de cette étape, les trois premieres cartes sont triées.



Tri par insertion : L'algorithme peut s'énoncer sous la forme :

- Pour i allant de 0 à $n-1$ (n est la longueur de la liste)
- insérer le i-ième terme parmi les $i-1$ premiers en conservant le caractère trié.

On peut remarquer qu'on a, avant d'écrire le programme, un invariant de boucle. Pour chaque entier i on a

- les i premiers éléments de la liste (d'indices 0 à $i-1$) sont les i premiers éléments de la liste initiale placés dans l'ordre
- les derniers éléments (d'indices i à $n-1$) sont les éléments d'origine de la liste, à leur place.

On peut donc écrire le programme de la figure

```
1 def triInsertion(liste):
2     n = len(liste)
3     for i in range(n):
4         inserer(liste, i)
5
6 L = [10, 20, 4, 1, 11, 9]
7 triInsertion(L)
8 print(L)   # affiche [1, 4, 9, 10, 11, 20]
```

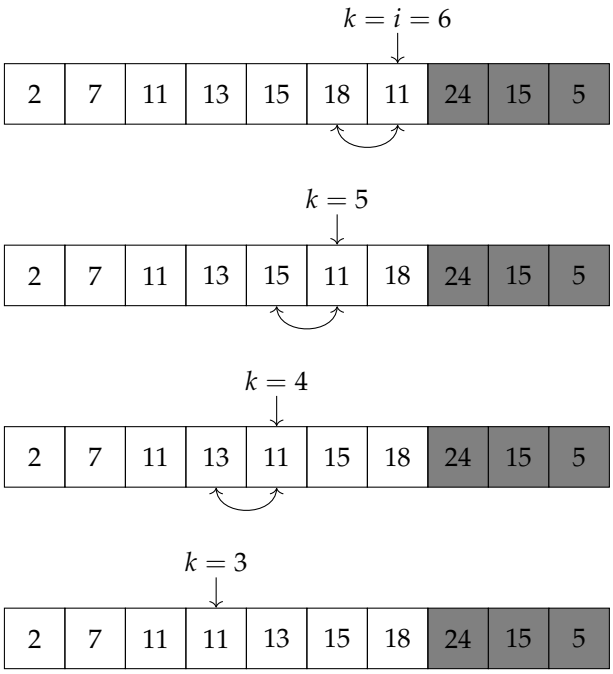
FIGURE 5 – Implémentation d'un tri par insertion.

Insertion : Pour réaliser `inserer(liste, i)` on peut itérer le procédé suivant en commençant par $k = i$.

1. Si $k = 0$ on a fini.
2. Si on a $k > 0$ et si `liste[k-1] > liste[k]` on échange les éléments des positions k et $k-1$ puis on diminue k de 1 (on décrémente k).
3. Si on a $k > 0$ et si `liste[k-1] <= liste[k]` on a fini.

Le fait de ne pas échanger en cas d'égalité donnera un tri stable.

Exemple : L'exécution de `inserer(liste, 6)` avec `liste = [2, 7, 11, 13, 15, 18, 11, 24, 15, 5]`.



Terminaison : La fonction `inserer` termine car l'entier `k` décroît strictement à chaque passage de la boucle donc finit par être nul si n'est pas sorti de la boucle auparavant. La fonction `triInsertion` fait appel $n-1$ fois à la fonction `inserer` qui termine donc le tri termine.

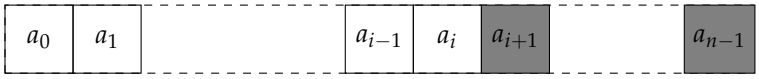
```python
def inserer(liste, i):
    k = i
    while k > 0:
        if liste[k-1]> liste[k]:
            echange(liste, k, k-1)
            k = k - 1
        else: # facultatif mais limite les opérations inutiles
            break
L = [10, 20, 4, 1, 11, 9]
inserer(L, 2)
print(L) # affiche [4, 10, 20, 1, 11, 9]
```

FIGURE 6 – Fonction d'insertion.

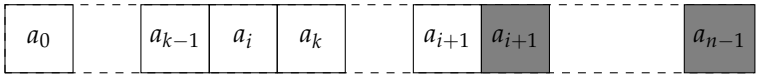
Correction/Preuve : On veut que la liste soit triée : pour cela on trie des portions de plus en plus grandes. On introduit une propriété vérifiée lors du passage de la boucle.

Les i premiers éléments de la liste sont triés, les autres sont inchangés.

À la fin de la boucle, i a pris la valeur $n-1$ et la propriété implique que les $n-1+1$ premiers éléments sont triés : la liste est triée. On doit donc prouver que `inserer(liste, i)` transforme la propriété de i à $i+1$. On note a_j la valeur de `liste[j]`.



On peut remarquer que les transformations successives donnent des résultats



On peut exprimer ces positions par les propriétés :

```
liste[0] <= liste[1] <= ... <= liste[k-1]
liste[k] <= liste[k+1] <= ... <= liste[n-1]
                      liste[k-1] <= liste[k]
```

On prouve alors que les instructions `echange(liste, k, k-1)` conservent les propriétés, elles forment un invariant, lorsque l'on a `liste[k] < liste [k-1]`. Par contre, si on a `liste[k] >= liste [k-1]`, la portion de 0 à i est triée donc on a le résultat souhaité. De même, pour $k=0$ la portion est triée aussi. Ainsi la fonction `inserer` produit bien le résultat souhaité.

Complexité : On effectue une comparaison pour chaque k de i à k_0 avec $1 \leq k_0 \leq i$ donc le nombre de comparaisons est compris entre 1 et i pour `inserer(liste, i)`.
On en déduit que le nombre de comparaisons de `triInsertion` pour une liste de taille n, $C(n)$ vérifie

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

La complexité est un $\mathcal{O}(n^2)$.
D'après l'étude de la fonction `inserer` la complexité maximale est atteinte lorsque l'élément d'indice i est inférieur (strictement) à tous ceux qui le précèdent, pour chaque i : c'est le cas pour une liste strictement décroissante au départ. De même la complexité est minimale pour une liste strictement croissante au départ.
Le tri par insertion est préférable au tri par sélection, il peut être très rapide dans le cas de liste qui sont peu désordonnées. On l'emploie parfois dans des tris plus rapides lors des dernières étapes pour lesquelles la liste est triée par blocs. Si on compte aussi les échanges, on voit que le tri par sélection ne fait que n échanges alors que le tri par insertion en effectue en 0 et $\frac{n(n-1)}{2}$.

IV Vérification expérimentale de la complexité

La méthode `time()` du module `time` renvoie la date en seconde au moment de son exécution. La syntaxe ci-dessous permet de calculer la durée d'une suite du tri d'une liste.

```
from time import time
t1 = time()
triInsertion(L)
t2 = time()
duree  = t2-t1
```

On peut alors représenter le temps d'exécution du tri d'une liste de taille n construite aléatoirement (voir figures 7 et 8). En échelle logarithmique, on vérifie la complexité quadratique.

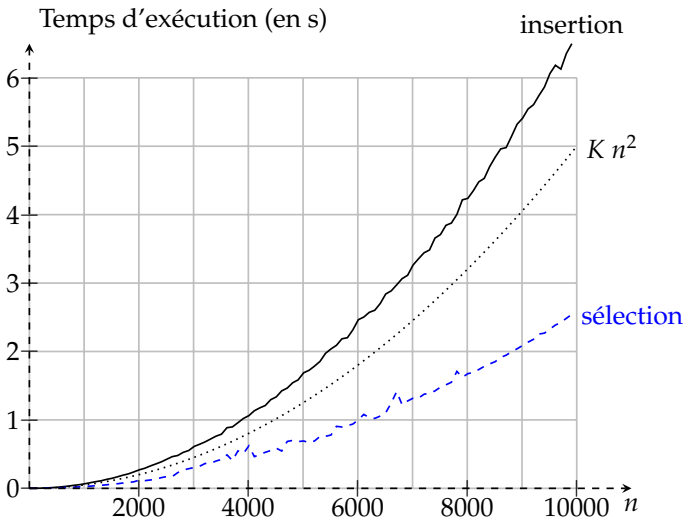


FIGURE 7 – Temps d'exécution en fonction de n en échelle linéaire. On a mis en trait noir le cas du tri par insertion et en tirets bleus le cas du tri par sélection. Enfin, on a rajouté une courbe représentative de $n \mapsto K\,n^2$ en pointillés.

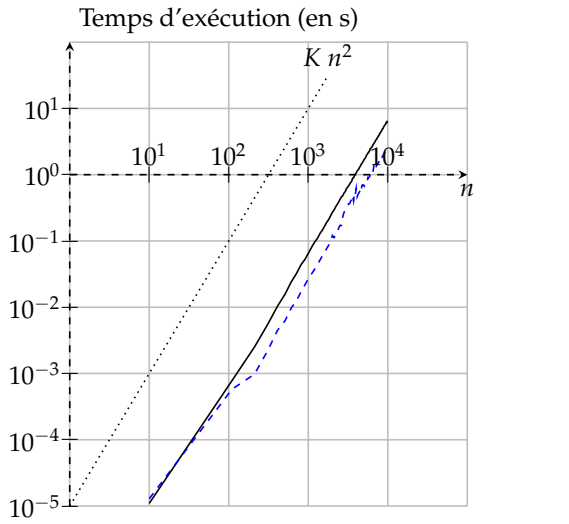


FIGURE 8 – Temps d'exécution en fonction de n en échelle logarithmique. On a mis en trait noir le cas du tri par insertion et en tirets bleus le cas du tri par sélection. Enfin, on a rajouté une courbe représentative de $n \mapsto K\,n^2$ en pointillés.