# Tris par insertion

November 29, 2021

## 1 Algorithme de tri

Nous allons étudier et comparer trois types d'algorithme de tri.

Un alogorithme de tri consiste à modifier une liste $L = [c_0\ c_1\ c_2\ ...]$ en une liste contenant les mêmes éléments mais ordonné $L' = [c_\alpha\ c_\beta\ c_\gamma\ ...]$ avec $c_\alpha < c_\beta < c_\gamma$

Nous étudierons le cas où les éléments à trier sont des nombres l'objectif sera donc d'écrire un algorithme qui modifie une liste: [14 1 4 3] en une liste ordonnée [1 3 4 14]

Nous allons aborder un premier exemple d'algorithme effectuant cette tache: l'algorithme de tri par insertion. Nous le comparerons par la suite avec deux autres alogorithmes: le tri par fusion et le tri rapide.

## 2 Principe

Le principe du tri par insertion est de prendre chaque élément de la liste, puis de l'insérer à sa place dans une liste triée.

Expliquons son fonctionnement à l'aide de l'exemple $L = [14\ 1\ 4\ 3]$.

On prends d'abord le premier élément de la liste: 14

puis on place 14 dans la liste triée donc $L' = [14]$.

On prends ensuite le deuxième élément de la liste: 1

on place 1 à la suite de la liste $L' = [14\ 1]$, pour placer 1 à sa place on compare 1 avec 14, comme 1 < 14 alors on doit échanger les positions de 1 et 14, on modifie $L'$ pour $L' = [1\ 14]$

Le troisième élément est 4

on place 4 à la suite de la liste $L' = [1\ 14\ 4]$, on compare 4 et 14, comme 4 < 14 alors on doit échanger les positions de 4 et 14, on modifie $L'$ pour $L' = [1\ 4\ 14]$, on compare 4 et 1, comme 1 < 4 alors on a rien à faire.

Le quatrième élément est 3

on place 3 à la suite de la liste $L' = [1\ 4\ 14\ 3]$, on compare 3 et 14, comme 3 < 14 alors on doit échanger ces deux éléments on a alors $L' = [1\ 4\ 3\ 14]$, on compare 3 et 4, comme 3 < 4 on doit échanger ces deux éléments on a alors $L' = [1\ 3\ 4\ 14]$, on compare 3 et 1, comme 1 < 3 on a rien à faire.

On a parcouru tous les éléments de la liste, la liste est triée [1 3 4 14]

Chaque élément de la liste initiale est inséré dans la liste déjà triée en le comparant successivement aux éléments du plus grand au plus petit.

## 3 Réalisation

Ecrivons l'algorithme de tri en le décomposant en chaque fonction élémentaire.

La première fonction doit échanger les positions de deux éléments d'une liste.

La fonction ci-dessous prends en argument la liste L et les indices i et j et échange les éléments $L[i]$ et $L[j]$

[1]:
```
def echange(L, i, j):
    L[i], L[j] = L[j], L[i]
```

par exemple on peut échanger la position du 1 et du 4 avec les lignes de commandes suivantes

[2]:
```
L = [14, 1, 4, 3]

echange(L, 1, 2)

print(L)
```

```
[14, 4, 1, 3]
```

La deuxième fonction consiste à comparer successivement un élément à la position $i$ avec les éléments déjà ordonnés précédents.

On initialise d'abord la valeur de j à i, donc $L[j]$ est l'élément à insérer dans $L$.

Puis si l'élément $L[j]$ est plus grand que l'élément précédent $L[j-1]$ alors ils ne sont pas dans le bon ordre.

Donc on intervertit $L[j]$ et $L[j-1]$ de manière à ce que la liste soit ordonnée dans l'ordre croissant.

On actualise la valeur de j à j-1 car on vient de décaler l'élément initial de la position j à j-1.

On recommence la boucle si on remarque encore que l'élément $L[j]$ est inférieur à l'élément précédent.

La boucle s'arrête si j = 0, on a alors parcourus l'ensemble de la liste,

ou si $L[j-1] < L[j]$, on a alors l'élément $L[j]$ qui est correctement inséré dans la liste de manière à ce que la liste soit dans l'ordre croissant.

[3]:
```
def insertion(L, i):
    j = i
    while j > 0 and L[j] < L[j - 1]:
        echange(L, j - 1, j)
        j -= 1
```

par exemple on peut visualiser les différentes étapes de l'insertion de 3 dans la liste [1 4 14] avec les lignes de commandes suivantes

[4]:
```
def insertion_visualisation(L, i):
    print(L)
    j = i
    while j > 0 and L[j] < L[j - 1]:
        echange(L, j - 1, j)
        print(L)
        j -= 1

L = [1, 4, 14, 3]
```

```
insertion_visualisation(L, 3)
```

```
[1, 4, 14, 3]
[1, 4, 3, 14]
[1, 3, 4, 14]
```

Enfin on peut écrire le programme final qui crée une liste $L'$ puis va insérer successivement tous les éléments de la liste $L$ dans $L'$ mais de manière à ce que $L'$ soit toujours dans l'ordre croissant.

On commence donc par insérer le premier élément $L[0]$ dans la liste $L'$, étant donné qu'il n'y a qu'un seul élément elle est dans l'ordre croissant.

Puis on parcours tous les éléments suivant de la liste $L[i]$ avec $i$ allant de 1 à n-1 avec n la taille de la liste.

On place chaque nouvel élément à la fin de la liste $L'$ de manière à avoir une nouvelle liste $[L', L[i]]$.

Puis on utilise la fonction insertion précédente pour insérer correctement le dernier élément de $L'$ dans $L'$ de manière à ce que $L'$ soit dans l'ordre croissant.

[5]:
```
def tri_insertion(L):
    L_prime = []
    L_prime.append(L[0])
    for i in range(1, len(L)):
        L_prime.append(L[i])
        insertion(L_prime, i)
    return L_prime
```

par exemple on peut visualiser les différentes étapes du tri de la liste [14 1 4 3] avec les lignes de commandes suivantes

[6]:
```
L = [14, 1, 4, 3]

def tri_insertion_visualisation(L):
    L_prime = []
    L_prime.append(L[0])
    print(L_prime)
    for i in range(1, len(L)):
        L_prime.append(L[i])
        insertion_visualisation(L_prime, i)
    return L_prime

tri_insertion_visualisation(L)
```

```
[14]
[14, 1]
[1, 14]
[1, 14, 4]
[1, 4, 14]
[1, 4, 14, 3]
```

```
[1, 4, 3, 14]
[1, 3, 4, 14]
```

[6]:
```
[1, 3, 4, 14]
```

# 4 Complexité temporelle

## 4.1 Définition de la complexité dans le meilleur et pire des cas

Le temps que va mettre l'algorithme à trier une liste dépendra de l'ordre initial des éléments dans la liste. Sa complexité qui est le nombre d'opérations effectuées dépendra également de l'ordre initial des éléments. On définit alors une complexité "dans le meilleur des cas", c'est la complexité minimale, c'est-à-dire que l'ordre des éléments de la liste initiale permet à l'algorithme d'effectuer un minimum d'opération. On définit également une complexité "dans le pire des cas", c'est la complexité maximale, c'est-à-dire que l'ordre des éléments de la liste initiale oblige l'algorithme à effectuer un maximum d'opération.

## 4.2 Calcul de la complexité dans le meilleur des cas

Le meilleur des cas est le cas où la liste est déjà triée, en effet lorsque l'algorithme parcoure la liste L il n'a pas besoin de déplacer les éléments car ils sont déjà en place. Ainsi on a juste 1 opération de comparaison à faire par étape de la boucle et si la liste est de taille $n$ alors il y a $n$ étapes de boucle à effectuer donc la complexité vaut $C(n) = n \times 1 = O(n)$. La complexité est linéaire dans le meilleur des cas.

## 4.3 Calcul de la complexité dans le pire des cas

Le pire des cas se trouve pour une liste dans l'ordre strictement décroissant, en effet l'algorithme doit replacer au début chaque élément de la liste. Lorsque l'algorithme regarde l'élément $i$ de la liste il doit échanger sa position avec l'élément $i-1$, puis avec l'élément $i-2$, ... , jusqu'à le mettre en position 1, il doit donc effectuer $i$ opérations, pour $i$ le numéro de l'élément allant de 1 à $n$. La complexité est donc $C(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$

# 5 Exercices

## 5.1 exercice : meilleur et pire des cas

- Dérouler chaque étape de l'algorithme de tri par insertion sur la liste $[15, 4, 2, 9, 55, 16, 0, 1]$
- Ecrire cette même liste où les éléments sont ordonnés dans le meilleur des cas et dérouler chaque étape de son tri
- Ecrire cette même liste où les éléments sont ordonnés dans le pire des cas et dérouler chaque étape de son tri
- Commenter

### 5.2 exercice : tri en place

- ré-écrire la fonction tri par insertion en utilisant un tri en place, c'est-à-dire en modifiant directement la liste d'entrée $L$ sans utiliser de fonction auxiliaire $L'$ et donc sans commande return $L'$

### 5.3 exercice : utilisation de pile

- ré-écrire la fonction tri par insertion en utilisant uniquement des piles comme strucutre de données. L'argument de la fonction est donc une pile et la fonction ne peut utiliser que les fonctions empiler et dépiler sur cette pile.

### 5.4 exercice : complexité spatiale

- calculer la complexité spatiale d'un algorithme de tri par insertion

### 5.5 exercice : calcul numérique de la complexité temporelle

- à l'aide de tirage au sort successif de liste de différentes tailles, mesurer numériquement le temps mis pour l'exécution du tri par insertion sur ces listes et tracer le temps d'éxécution en fonction de la taille des listes à trier.
- est-ce cohérent avec les calculs de complexité effectués précédemment ?

### 5.6 exercice : tri par sélection

On considère dans cet exercice des listes L dont les éléments sont comparables par le biais de <.

- Écrire une fonction indice_max($L$,$i$) qui, quand $0 \leq i \leq len(L)$, calcule la position du maximum de la sous-liste $L[: i+1]$.
- En déduire le code d'une fonction tri_selection($L$) qui trie une liste $L$ de longueur $n$ en plaçant le maximum de $L$ en position $n1$, puis le maximum de $L[: n1]$ en position $n2$, et ainsi de suite.
- Étudier la complexité temporelle et spatialle de cet algorithme de tri.

### 5.7 exercice : tri à bulles

L'algorithme de tri à bulles reprend l'idée du tri par sélection : on place en position $n1$ le maximum de la liste $L = L[: n]$, puis en position $n2$ le maximum de la sous-liste $L[: n1]$, et ainsi de suite jusqu'à placer en position 1 le maximum de la sous-liste $L[: 2]$. La différence est que l'on va cette fois-ci arrêter le calcul dès que la liste est triée.

- Écrire le code d'une fonction remonter($L$, $i$) qui remonte le maximum de la sous-liste $L[: i+1]$ jusqu'à la position $i$ ; on procédera pour cela à des échanges successifs éventuels des contenus des cases 0 et 1, 1 et 2, . . . , i  1 et i, comme si une bulle remontait le long de la liste. Cette fonction renverra un booléen égal à True si aucun échange n'a été fait (i.e. si la sous-liste $L[: i+1]$ était croissante) et à False sinon.

- En déduire le code d'une fonction tri_bulles($L$) qui trie la liste $L$, en arrêtant le calcul dès que la liste est triée.
- Étudier la complexité temporelle et spatialle de cet algorithme de tri.

### 5.8 exercice : tri crêpes

On empile un tas de crêpes de diamètres différents. On ne s'autorise qu'à donner un coup de spatule à l'intérieur du tas de crêpes ce qui a pour effet de retourner tout ou partie de la pile (à partir du sommet).

- Écrire une fonction retourne(p, k) qui retourne les k premiers éléments de la pile p (en partant du sommet), c'est-à-dire qui donne un coup de spatule sur le tas de crêpes au dessous de la $k^{ime}$ crêpe.
- Écrire une fonction taille(p) qui retourne le nombre d'éléments de pile p.
- Écrire une fonction trouve_max(p, n) qui renvoie le numéro de la crêpe de diamètre maximal parmi les n premiers éléments de la pile p (= tas de crêpes).
- Concevoir un algorithme (simple) à base de coup de spatules sur le tas de crêpes pour trier le tas par diamètre croissant (la crêpe la plus petite est au sommet).