Tris par insertion corrige

November 26, 2020

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 $[15, 4, 2, 9, 55, 16, 0, 1]$

[7]:
```
L = [15, 4, 2, 9, 55, 16, 0, 1]

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)
```

```
[15]
[15, 4]
[4, 15]
[4, 15, 2]
[4, 2, 15]
[2, 4, 15]
[2, 4, 15, 9]
[2, 4, 9, 15]
[2, 4, 9, 15, 55]
[2, 4, 9, 15, 55, 16]
[2, 4, 9, 15, 16, 55]
[2, 4, 9, 15, 16, 55, 0]
[2, 4, 9, 15, 16, 0, 55]
[2, 4, 9, 15, 0, 16, 55]
[2, 4, 9, 0, 15, 16, 55]
[2, 4, 0, 9, 15, 16, 55]
[2, 0, 4, 9, 15, 16, 55]
[0, 2, 4, 9, 15, 16, 55]
[0, 2, 4, 9, 15, 16, 55, 1]
[0, 2, 4, 9, 15, 16, 1, 55]
[0, 2, 4, 9, 15, 1, 16, 55]
[0, 2, 4, 9, 1, 15, 16, 55]
[0, 2, 4, 1, 9, 15, 16, 55]
[0, 2, 1, 4, 9, 15, 16, 55]
[0, 1, 2, 4, 9, 15, 16, 55]
```

[7]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

- Ecrire cette même liste où les éléments sont ordonnés dans le meilleur des cas et dérouler chaque étape de son tri

[8]:
```
L = [0, 1, 2, 4, 9, 15, 16, 55]

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)
```

```
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 4]
[0, 1, 2, 4, 9]
[0, 1, 2, 4, 9, 15]
[0, 1, 2, 4, 9, 15, 16]
[0, 1, 2, 4, 9, 15, 16, 55]
```

[8]: `[0, 1, 2, 4, 9, 15, 16, 55]`

- Ecrire cette même liste où les éléments sont ordonnés dans le pire des cas et dérouler chaque étape de son tri

[9]:
```
L = [55, 16, 15, 9, 4, 2, 1, 0]

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)
```

```
[55]
[55, 16]
[16, 55]
[16, 55, 15]
[16, 15, 55]
[15, 16, 55]
[15, 16, 55, 9]
[15, 16, 9, 55]
[15, 9, 16, 55]
[9, 15, 16, 55]
[9, 15, 16, 55, 4]
[9, 15, 16, 4, 55]
[9, 15, 4, 16, 55]
[9, 4, 15, 16, 55]
[4, 9, 15, 16, 55]
[4, 9, 15, 16, 55, 2]
[4, 9, 15, 16, 2, 55]
```

```
[4, 9, 15, 2, 16, 55]
[4, 9, 2, 15, 16, 55]
[4, 2, 9, 15, 16, 55]
[2, 4, 9, 15, 16, 55]
[2, 4, 9, 15, 16, 55, 1]
[2, 4, 9, 15, 16, 1, 55]
[2, 4, 9, 15, 1, 16, 55]
[2, 4, 9, 1, 15, 16, 55]
[2, 4, 1, 9, 15, 16, 55]
[2, 1, 4, 9, 15, 16, 55]
[1, 2, 4, 9, 15, 16, 55]
[1, 2, 4, 9, 15, 16, 55, 0]
[1, 2, 4, 9, 15, 16, 0, 55]
[1, 2, 4, 9, 15, 0, 16, 55]
[1, 2, 4, 9, 0, 15, 16, 55]
[1, 2, 4, 0, 9, 15, 16, 55]
[1, 2, 0, 4, 9, 15, 16, 55]
[1, 0, 2, 4, 9, 15, 16, 55]
[0, 1, 2, 4, 9, 15, 16, 55]
```

[9]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

- Commenter

On remarque que le nombre d'étape de tri est bien ordonné comme attendu:

```
- nbre d'étape cas quelconque = 24
- nbre d'étape meilleur des cas = 8
- nbre d'étape pire des cas = 36
```

ce qui est cohérent avec nbre(meilleur) < nbre(quelconque) < nbre(pire)
De plus les valeurs numérique sont cohérentes avec les calculs de complexité. Avec une taille de liste n = 8 à trier on a calculé

```
- nbre d'étape meilleur des cas = n = 8
- nbre d'étape pire des cas = n(n+1)/2 = 8*9/2 = 36
```

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'

[10]:
```
def tri_insertion_en_place(L):
    for i in range(1, len(L)):
        insertion(L, i)

L = [15, 4, 2, 9, 55, 16, 0, 1]
```

```
tri_insertion_en_place(L)
print(L)
```

```
[0, 1, 2, 4, 9, 15, 16, 55]
```

[11]:
```
def tri_insertion_en_place_visualisation(L):
    for i in range(1, len(L)):
        insertion_visualisation(L, i)

L = [15, 4, 2, 9, 55, 16, 0, 1]
tri_insertion_en_place_visualisation(L)
```

```
[15, 4, 2, 9, 55, 16, 0, 1]
[4, 15, 2, 9, 55, 16, 0, 1]
[4, 15, 2, 9, 55, 16, 0, 1]
[4, 2, 15, 9, 55, 16, 0, 1]
[2, 4, 15, 9, 55, 16, 0, 1]
[2, 4, 15, 9, 55, 16, 0, 1]
[2, 4, 9, 15, 55, 16, 0, 1]
[2, 4, 9, 15, 55, 16, 0, 1]
[2, 4, 9, 15, 55, 16, 0, 1]
[2, 4, 9, 15, 16, 55, 0, 1]
[2, 4, 9, 15, 16, 55, 0, 1]
[2, 4, 9, 15, 16, 0, 55, 1]
[2, 4, 9, 15, 0, 16, 55, 1]
[2, 4, 9, 0, 15, 16, 55, 1]
[2, 4, 0, 9, 15, 16, 55, 1]
[2, 0, 4, 9, 15, 16, 55, 1]
[0, 2, 4, 9, 15, 16, 55, 1]
[0, 2, 4, 9, 15, 16, 55, 1]
[0, 2, 4, 9, 15, 16, 1, 55]
[0, 2, 4, 9, 15, 1, 16, 55]
[0, 2, 4, 9, 1, 15, 16, 55]
[0, 2, 4, 1, 9, 15, 16, 55]
[0, 2, 1, 4, 9, 15, 16, 55]
[0, 1, 2, 4, 9, 15, 16, 55]
```

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.

[12]:
```
def creer_pile():
    return []

def taille(p):
```

```
    return len(p)

def est_vide(p):
    return taille(p) == 0

def empiler(p, v):
    p.append(v)

def depiler(p):
    assert taille(p) > 0
    return p.pop()
```

[13]:
```
def renverser(p):
    p1 = creer_pile()
    p2 = creer_pile()
    while not(est_vide(p)):
        empiler(p1,depiler(p))
    while not(est_vide(p1)):
        empiler(p2,depiler(p1))
    while not(est_vide(p2)):
        empiler(p,depiler(p2))

def copie(p1):
    w = depiler(p1)
    empiler(p1,w)
    return w

def insertion_pile(p1, v):
    p2 = creer_pile()
    w = copie(p1)
    while not(est_vide(p1)) and (v < w):
        empiler(p2,depiler(p1))
        if not(est_vide(p1)):
            w = copie(p1)

    empiler(p1,v)
    while not(est_vide(p2)):
        empiler(p1,depiler(p2))

def tri_insertion_pile(p):
    renverser(p)
    p1 = creer_pile()
    empiler(p1,depiler(p))
    while not(est_vide(p)):
        insertion_pile(p1,depiler(p))
    return p1
```

[14]:
```
p = [15, 4, 2, 9, 55, 16, 0, 1]
tri_insertion_pile(p)
```

[14]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

5.4 exercice : complexité spatiale

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

Un algorithme de tri par insertion en place, n'a pas besoin de créer de liste annexe, sa complexité spatiale est donc la taille de la liste en argument n = O(n). C'est une complexité linéaire.

Un alogorithme de tri par insertion peut aussi être implémenté en créant une liste annexe que l'on trie au fur et à mesure. On remarque que pour une liste à trier en argument de taille n, la liste annexe se remplie progressivement et est de taille m avec m allant de 0 à n. L'espace mémoire total utilisé est donc compris entre n+0 à n+n = 2n soit un O(n). C'est aussi une complexité linéaire.

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 ?

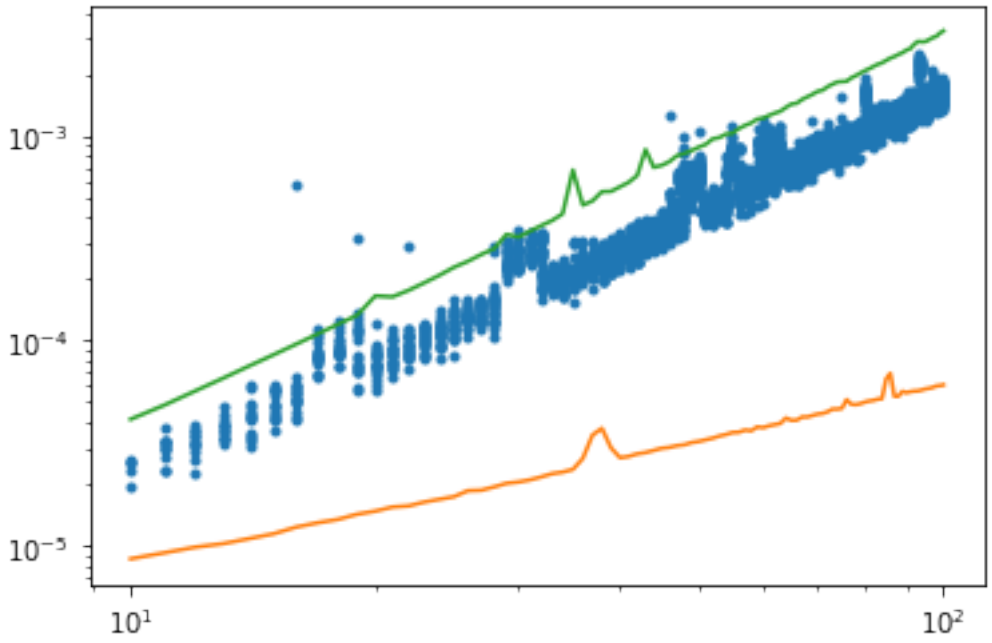
[15]:
```
import time
import matplotlib.pyplot as plt
import numpy as np
import random as rd

n_max = 10000

sqrt_n_max = int(np.sqrt(n_max))

n_max = sqrt_n_max*sqrt_n_max

duree = []
taille = []
L = []
for i in range(sqrt_n_max):
    for j in range(i+1):
        taille.append(i+1)
        L.append(np.arange(i+1))
        rd.shuffle(L[len(L)-1])

for i in range(len(L)):
    depart = time.clock()
    tri_insertion(L[i])
    arrive = time.clock()
    duree.append(arrive-depart)
```

```
duree_meilleur = []
taille_meilleur = []
L_meilleur = []
for i in range(sqrt_n_max):
    taille_meilleur.append(i+1)
    L_meilleur.append(np.arange(i+1))

for i in range(len(L_meilleur)):
    depart = time.clock()
    tri_insertion(L_meilleur[i])
    arrive = time.clock()
    duree_meilleur.append(arrive-depart)

duree_pire = []
taille_pire = []
L_pire = []
for i in range(sqrt_n_max):
    taille_pire.append(i+1)
    l = np.arange(i+1)
    L_pire.append(np.flip(l))

for i in range(len(L_pire)):
    depart = time.clock()
    tri_insertion(L_pire[i])
    arrive = time.clock()
    duree_pire.append(arrive-depart)
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:22: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:24: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:35: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:37: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:49: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
```

```
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:51: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```

[16]:
```
plt.loglog(taille[50:],duree[50:],'.')
plt.loglog(taille_meilleur[9:],duree_meilleur[9:],'-')
plt.loglog(taille_pire[9:],duree_pire[9:],'-')
plt.show()
```



On trace dans le graphe ci-dessus la durée d'éxécution du tri par insertion $C(n)$ en fonction de la taille de la liste à trier n.

Les points représentent la durée pour diverses listes triés aléatoirement.

Les traits représentent la durée dans le meilleur et le pire des cas.

Les axes des abscisses et des ordonnées sont représentés en échelle logarithmique car les complexités attendues sont polynomiales. En effet pour une équation de la forme $y = ax^k$ le graphe de $\log(y)$ en fonction de $\log(x)$ donne une droite de pente k car $(y) = \log(ax^k) = \log(a) + k\log(x)$. Pour une complexité linéaire en O(n), on attends donc une droite de pente 1, et pour une complexité quadratique en O(n^2) on attends une droite de pente 2.

On observe bien que les cas quelconques des points bleus, sont compris entre le meilleur et le pire des cas.

On observe bien une droite de pente 2 dans le pire des cas, donc une complexité quadratique, et une droite de pente 1 dans le meilleur des cas donc une complexité linéaire.

On observe enfin que pour une liste prise aléatoirement, on peut appliquer le plus souvent une complexité quadratique proche du pire des cas pour un tri par insertion.

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.

[17]:
```
def indice_max(L, i):
    pos = 0 # pos est la position du maximum
    for j in range(1, i + 1):
        if L[j] > L[pos]:
            pos = j
    return pos

L = [15, 4, 2, 9, 55, 16, 0, 1]
indice_max(L, 3)
```

[17]:
```
0
```

[18]:
```
def echanger(L, i, j):
    L[i], L[j] = L[j], L[i]

def tri_selection(L):
    for i in range(len(L) - 1, 0, -1):
        echanger(L, i, indice_max(L, i))
    return L

L = [15, 4, 2, 9, 55, 16, 0, 1]
tri_selection(L)
```

[18]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

[19]:
```
def tri_selection_visualisation(L):
    for i in range(len(L) - 1, 0, -1):
        print(L)
        print('echange des indices', indice_max(L, i), 'avec', i)
        echanger(L, i, indice_max(L, i))
    return L

L = [15, 4, 2, 9, 55, 16, 0, 1]
tri_selection_visualisation(L)
```

```
[15, 4, 2, 9, 55, 16, 0, 1]
echange des indices 4 avec 7
[15, 4, 2, 9, 1, 16, 0, 55]
```

```
echange des indices 5 avec 6
[15, 4, 2, 9, 1, 0, 16, 55]
echange des indices 0 avec 5
[0, 4, 2, 9, 1, 15, 16, 55]
echange des indices 3 avec 4
[0, 4, 2, 1, 9, 15, 16, 55]
echange des indices 1 avec 3
[0, 1, 2, 4, 9, 15, 16, 55]
echange des indices 2 avec 2
[0, 1, 2, 4, 9, 15, 16, 55]
echange des indices 1 avec 1
```

[19]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

La complexité temporelle de indice_max(L,i) est i, car on fait une opération par étape de boucle et il y a i étapes de boucle.

Le tri par sélection a une complexité temporelle de $(n-1)+(n-2)++1 = O(n^2)$, c'est une complexité temporelle quadratique.

Le tri par sélection trie la liste en place, il manipule directement la liste en entrée L, l'espace mémoire utilisé est donc de même taille que L donc $O(n)$, c'est une complexité spatiale linéaire.

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.

[20]:
```python
def echange(L, i, j):
    L[i], L[j] = L[j], L[i]

def remonter(L, i):
    b = True
    for j in range(i):
        if L[j] > L[j + 1]:
            b = False
            echange(L, j, j + 1)
    return b
```

```
L = [15, 4, 2, 9, 55, 16, 0, 1]
remonter(L, 3)
print(L)
```

```
[4, 2, 9, 15, 55, 16, 0, 1]
```

[21]:
```
def tri_bulle(L):
    i = len(L) - 1
    b = False
    while not(b) and i>0 :
        b = remonter(L, i)
        i -= 1
    return L

L = [15, 4, 2, 9, 55, 16, 0, 1]
tri_bulle(L)
```

[21]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

[22]:
```
def echange_visualisation(L, i, j):
    L[i], L[j] = L[j], L[i]
    print(L, 'echange des indices', i, 'avec', j)

def remonter_visualisation(L, i):
    b = True
    for j in range(i):
        if L[j] > L[j + 1]:
            b = False
            echange_visualisation(L, j, j + 1)
    return b

def tri_bulle_visualisation(L):
    i = len(L) - 1
    b = False
    print(L)
    while not(b) and i>0 :
        b = remonter_visualisation(L, i)
        i -= 1
    return L

L = [15, 4, 2, 9, 55, 16, 0, 1]
tri_bulle_visualisation(L)
```

```
[15, 4, 2, 9, 55, 16, 0, 1]
[4, 15, 2, 9, 55, 16, 0, 1] echange des indices 0 avec 1
[4, 2, 15, 9, 55, 16, 0, 1] echange des indices 1 avec 2
[4, 2, 9, 15, 55, 16, 0, 1] echange des indices 2 avec 3
```

```
[4, 2, 9, 15, 16, 55, 0, 1] echange des indices 4 avec 5
[4, 2, 9, 15, 16, 0, 55, 1] echange des indices 5 avec 6
[4, 2, 9, 15, 16, 0, 1, 55] echange des indices 6 avec 7
[2, 4, 9, 15, 16, 0, 1, 55] echange des indices 0 avec 1
[2, 4, 9, 15, 0, 16, 1, 55] echange des indices 4 avec 5
[2, 4, 9, 15, 0, 1, 16, 55] echange des indices 5 avec 6
[2, 4, 9, 0, 15, 1, 16, 55] echange des indices 3 avec 4
[2, 4, 9, 0, 1, 15, 16, 55] echange des indices 4 avec 5
[2, 4, 0, 9, 1, 15, 16, 55] echange des indices 2 avec 3
[2, 4, 0, 1, 9, 15, 16, 55] echange des indices 3 avec 4
[2, 0, 4, 1, 9, 15, 16, 55] echange des indices 1 avec 2
[2, 0, 1, 4, 9, 15, 16, 55] echange des indices 2 avec 3
[0, 2, 1, 4, 9, 15, 16, 55] echange des indices 0 avec 1
[0, 1, 2, 4, 9, 15, 16, 55] echange des indices 1 avec 2
```

[22]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

La complexité temporelle de remonter(L,i) est de i, car on fait une opération par étape de boucle et il y a i étape de boucle.

Dans le meilleur des cas, quand la liste est déjà trié, on a besoin d'appeler remonter une seule fois avec remonter(L,n-1), donc la complexité temporelle est de n-1 = O(n), c'est une complexité linéaire.

Dans le pire des cas, quand la liste est en ordre décroissant, on a besoin d'appeler remonter pour tous les indices i allant de (n - 1) à 1 donc la compléxité temporelle est de $(n-1)+(n-2)+ +1 = O(n^2)$, c'est une complexité temporelle quadratique.

Le tri à bulles trie la liste en place, il manipule directement la liste en entrée L, l'espace mémoire utilisé est donc de même taille que L donc $O(n)$, c'est une complexité spatiale linéaire.

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).

[23]:
```
def creer_pile():
    return []

def empiler(p, v):
```

```
    p.append(v)

def taille(p):
    return len(p)

def depiler(p):
    assert taille(p) > 0
    return p.pop()

def est_vide(p):
    return taille(p) == 0
```

[24]:
```
def retourne(p, k):
    p_aux = creer_pile()
    for i in range(k):
        empiler(p_aux,depiler(p))
    p_rev = creer_pile()
    for i in range(k):
        empiler(p_rev,depiler(p_aux))
    for i in range(k):
        empiler(p,depiler(p_rev))
    return p

L = [15, 4, 2, 9, 55, 16, 0, 1]
retourne(L, 4)
```

[24]:
```
[15, 4, 2, 9, 1, 0, 16, 55]
```

[25]:
```
def trouve_max(p, n):
    p_aux = creer_pile()
    M = depiler(p)
    empiler(p_aux,M)
    pos = 1
    for i in range(1, n):
        el = depiler(p)
        if el > M:
            M = el
            pos = i + 1
        empiler(p_aux,el)
    for i in range(n):
        empiler(p,depiler(p_aux))
    return pos

L = [15, 4, 2, 9, 55, 16, 0, 1]
trouve_max(L, 8)
```

[25]:
```
4
```

[26]:
```
def tricrepe(p):
    long = taille(p)
    for n in range(long, 1, -1):
        k = trouve_max(p, n)
        if k != n:
            retourne(p, k)
            retourne(p, n)
    return p

L = [15, 4, 2, 9, 55, 16, 0, 1]
tricrepe(L)
```

[26]:
```
[55, 16, 15, 9, 4, 2, 1, 0]
```

[27]:
```
def tricrepe_visualisation(p):
    long = taille(p)
    print(L)
    for n in range(long, 1, -1):
        k = trouve_max(p, n)
        print('pour n=',n,'indice du maximum=', k, '=> faire qqch', k != n)
        if k != n:
            retourne(p, k)
            print(L)
            retourne(p, n)
            print(L)
    return p

L = [15, 4, 2, 9, 55, 16, 0, 1]
tricrepe_visualisation(L)
```

```
[15, 4, 2, 9, 55, 16, 0, 1]
pour n= 8 indice du maximum= 4 => faire qqch True
[15, 4, 2, 9, 1, 0, 16, 55]
[55, 16, 0, 1, 9, 2, 4, 15]
pour n= 7 indice du maximum= 7 => faire qqch False
pour n= 6 indice du maximum= 1 => faire qqch True
[55, 16, 0, 1, 9, 2, 4, 15]
[55, 16, 15, 4, 2, 9, 1, 0]
pour n= 5 indice du maximum= 3 => faire qqch True
[55, 16, 15, 4, 2, 0, 1, 9]
[55, 16, 15, 9, 1, 0, 2, 4]
pour n= 4 indice du maximum= 1 => faire qqch True
[55, 16, 15, 9, 1, 0, 2, 4]
[55, 16, 15, 9, 4, 2, 0, 1]
pour n= 3 indice du maximum= 3 => faire qqch False
pour n= 2 indice du maximum= 1 => faire qqch True
[55, 16, 15, 9, 4, 2, 0, 1]
[55, 16, 15, 9, 4, 2, 1, 0]
```

[27]: `[55, 16, 15, 9, 4, 2, 1, 0]`