Travail sur les listes

TRI (SORT en anglais)

list_examples.py

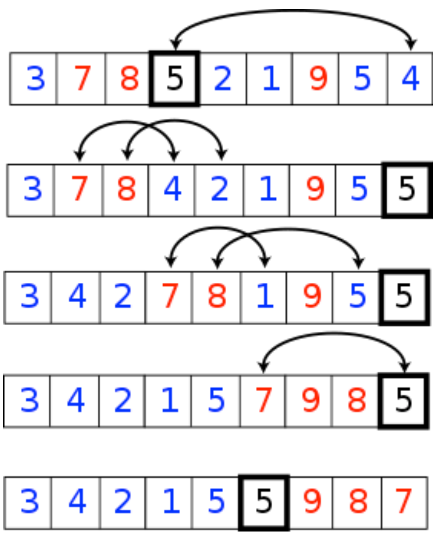
```python
list_numbers = [1, 2, 3, 4, 5, 6, 7, 8]

print(list_numbers[1:3])
print(list_numbers[:4])
print(list_numbers[5:])
print(list_numbers[:-5])
```

Run: list_examples

```
/Users/pankaj/Documents/PycharmProjects/Pyth
[2, 3]
[1, 2, 3, 4]
[6, 7, 8]
[1, 2, 3]

Process finished with exit code 0
```



1. SLICING avancé sur les listes

Nous avons vu les différentes opérations de base sur les listes, avec des méthodes python (`extend, append, pop, insert`…) mais aussi via le découpage en ***tranches = slicing***.

Ici, nous allons poursuivre dans le slicing en allant un peu plus loin vers le slicing avancé. Comment manipuler des sous-listes en une seule instruction (qui cache évidemment des opérations avancées…) ?

```
L1 = [1, 2, 3, 11, 12, 13]
L2 = L1
L1[0:3] = L1[-1:-4:-1] # recopie les 3 derniers éléments
    # de L1 et les place dans les 3 premières cases
print(L1)
print(L2) # L2 est changée car l'opération est faite "en place"
```

```
[13, 12, 11, 11, 12, 13]
[13, 12, 11, 11, 12, 13]
```

Comparer les deux exemples suivants :

- `L1[:] = L1[-1::-1]`, l'opération se fait **en place**
- `L1 = L1[-1::-1]`, un **nouvel objet** est créé et L1 le désigne

```
# inversion de l'ordre d'une liste, exemple 1
L1 = [k for k in range(10)]
L2 = L1
L1[:] = L1[-1::-1] # la sous-liste de tous les éléments de L1
# devient la sous-liste des éléments dans l'ordre inverse
print(L1)
print(L2) # L2 est modifiée : l'opération a lieu "en place"
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Inversion de l'ordre d'une liste :

```
# inversion de l'ordre d'une liste, exemple 2
L1 = [k for k in range(10)]
L2 = L1
L1 = L1[-1::-1] # la sous-liste de tous les éléments de L1
# devient la sous-liste des éléments dans l'ordre inverse
print(L1)
print(L2) # L2 n'est modifiée  !!
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Indexation par un slicing :

```
# Dernier exemple : indexation par un sciling
L1 = ['A', 1, 'B', 2, 'C', 3, 'D', 4]
L2 = L1
L1[0::2] = L1[1::2] # les termes d'indice impairs sont
                    # mis dans les termes d'indice pairs
print(L1)
print(L2) # L'opération sur L1 s'effectue 'en place'
```

```
[1, 1, 2, 2, 3, 3, 4, 4]
[1, 1, 2, 2, 3, 3, 4, 4]
```

2. Fonctions importantes sur les listes

Nous allons inroduire des fonction utilies et importantes afin de travailler sur les listes, à savoir :

- ✔ Mise à zéro des valeurs d'une liste inférieures à un seuil
- ✔ Recherche du miminum d'une liste de nombres
- ✔ Recherche du minimum et de l'indice d'un minimum

2.1. Mise à zéro des valeurs d'une listes inférieures à un seuil

```
def seuil(L, x): # L est une liste d'entiers, x est un entier
                 # la fonction seuil met à zéro tous les éléments de L qui sont␣
 ↪inférieurs à x
    n=len(L)
    for k in range(n):
        if L[k] < x:
            L[k] =0  # mise à zéro du k-ième élément de L
L0 = [2, 5, 3, 7, 1, 8]
seuil(L0, 5)
print(L0)
```

```
[0, 5, 0, 7, 0, 8]
```

Remarques :

- L'instruction L[k] = 0 met à zéro la k-ième valeur de la liste L, sans qu'une nouvelle liste soit recréée.
- La k-ième valeur de la liste est modifiée mais il n'y a pas création d'un nouvel objet liste.
- Dans la fonction seuil, L est une variable locale qui fait référence au même objet que L0.

Donc, en modifiant l'objet référencé par L, on modifie l'objet référencé par L0. Il est donc inutile de mettre un **return** en fin de fonction.

2.2. Recherche du minimum d'une liste de nombres

```
def getMin(L): # renvoie le minimum d'une liste de nombres
    res = L[0] # on initialise avec la valeur du 1er élément
    for el in L[1:]: # sous liste créée en retirant le premier élément
        if (el < res): # el est un candidat pour être le minimum
            res = el
    return res # toute la liste a été parcourue
```

```
L0 = [2, 4, 2.1, 1.2, 54., 8, 1.4, 2.01]
print(getMin(L0))
```

```
1.2
```

2.3. Recherche du minimum et de l'indice d'une liste de nombres

```
def getMin2(L): # renvoie le tuple (min(L), iMin) de la liste L
    iRes = 0   # indice du minimum initialisé à zéro
    res = L[0] # valeur du minimum initialisé à la valeur du 1er élément
    n = len(L) # nombre d'éléments de L
    for k in range(1,n): # on parcourt les éléments à partir du 2ème
                         # c'est-à-dire celui dont l'indice vaut 1
        if L[k] < res:
            iRes = k   # mise à jour de l'indice
            res = L[k] # mise à jour du résultat
    return (res,iRes) # rq : les parenthèses sont optionnelles
L0 = [2, 4, 2.1, 1.2, 54., 8, 1.4, 2.01]
minimum,indice = getMin2(L0)
print('Le minimum de L0 vaut \t\t', minimum); # \t = caractère de tabulation
print("L'indice du minimum de L0 vaut \t", indice)
```

```
Le minimum de L0 vaut           1.2
L'indice du minimum de L0 vaut  3
```

3. Algorithmes de tris : introduction

Nous verrons que certains algorithmes sont ***plus efficaces si les données sont déjà triées***. Par exemple, la recherche d'un élément dans une liste quelconque possède un coût linéaire. Lorsque la liste est déjà triée, on peut utiliser une recherche dichotomique (que l'on verra plus tard) qui possède un coût logarithmique beaucoup plus faible en temps de calcul.

Hypothèse : les données à trier sont stockées dans une ***liste de nombres appelée TABLEAU***. On verra plus tard dans le semestre les tableaux mais pour l'instant un tableau est une liste ne contenant que des éléments de même typage (float ou int). Par exemples L = [1 , 3 , 56 , 34] est un tableau car il ne contient que des valeurs de même type, ici des int.

Les principes algorithmes de TRI que vous verrez cette année sont :

- ✓ ***Tri par SELECTION***
- ✓ ***Tri par INSERTION***

} Complexité quadratique $O(n^2)$

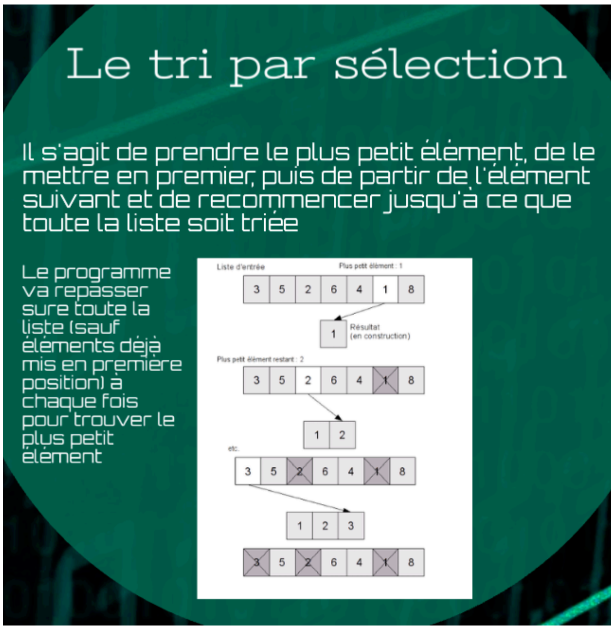
- ✓ ***Tri RAPIDE (QUICK)***
- ✓ ***Tri par FUSION (MERGE)***

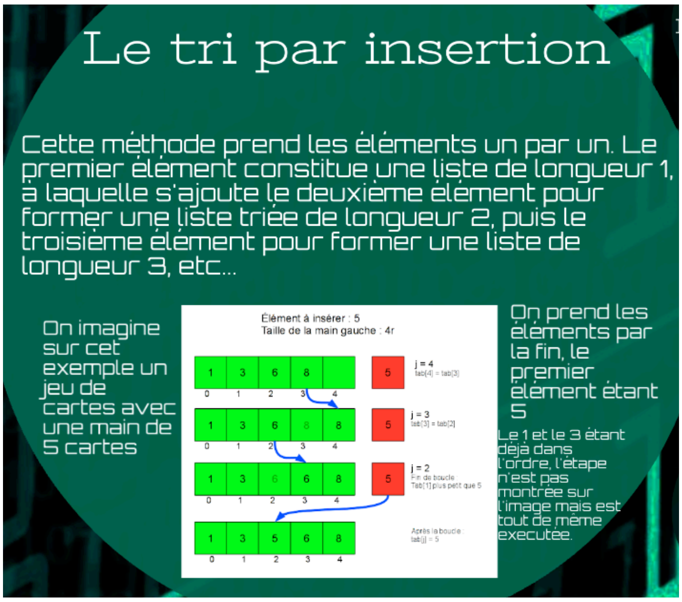
} Complexité pseudo linéaire $O(n\log n)$, bien plus rapide !!

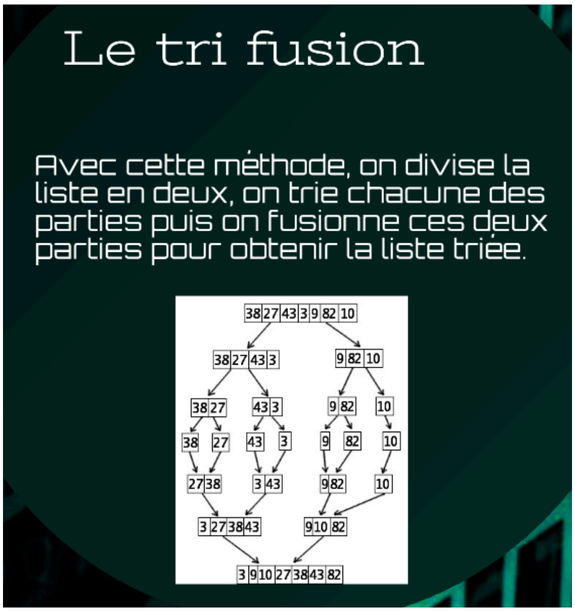
La notation $O(n^2)$ (big-oh en anglais) signifie que le temps de calcul est de l'ordre de grandeur du carré de la taille du tableau (n éléments). Nous allons dévelloper avec vous au semestre 1 les méthodes par ***sélection*** et ***insertion***, puis au semestre 2 vous verrez les 2 autres.

Travail sur les listes - TRI

Voici quelques informations de base sur ces méthodes de TRI.







Travail sur les listes - TRI

4. Algorithme de tri par SELECTION

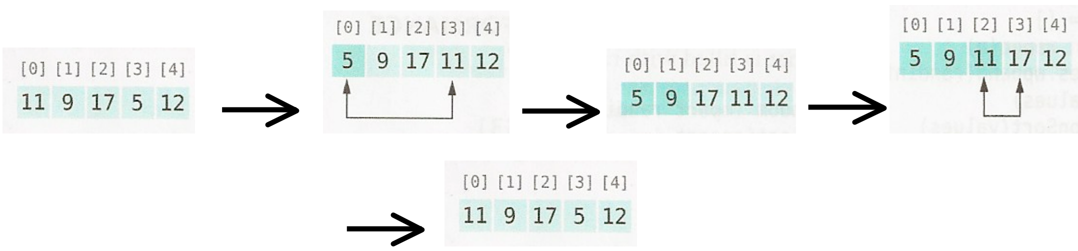
Principe : on souhaite trier par ordre croissant une liste de n éléments.

✔ On trie progressivement les éléments de la liste en sélectionnant le plus petit parmi ceux qui n'ont pas encore été triés
✔ On procède par itération : les éléments déjà triés sont rangés (par ordre croissant) parmi les k premiers de la liste
✔ Les éléments qui n'ont pas encore été triés restent à leur place en fin de liste

Algorithme en Français :

```
L'indice k parcourt les indices de tous les éléments SAUF DU DERNIER.
    # Ici, la liste des k-1 premiers éléments est supposée triée.
    Rechercher l'indice i du minimum parmi les éléments
    restants n-k+1 éléments restants.
    Permuter le k-ième élément avec le i-ème pour
    que le k-ième corresponde au minimum des restants.
```

En quittant la boucle, les n-1 premiers éléments sont à leur place. Donc le n-ième l'est également. Pour ce tri, on utilise l'algorithme de recherche de l'indice du minimum dans une liste vu précédemment. Voici un exemple sur une liste d'entier du fonctionnement de l'algorithme :




```python
def triSelection(L):
    n=len(L)
    nbComp=0
    for k in range(n-1): # inutile de trier le denier élément : k s'arrête à n-2, c-à-d au n-1-ième -élément.
        # recherche du minimum dans la sous-liste L[k:], c'est-à-dire pour les indices k, k+1, k+2, ..., n
        iMin=k # initialisation de l'indice du min
        for i in range(k+1,n): # l'indice i prend les valeurs k+1, k+2, ..., n-1
            nbComp+=1
            if L[i]<L[iMin]:
                iMin=i # mise à jour de l'indice du mini
```

```python
        L[iMin],L[k]=L[k],L[iMin] # permutation des éléments d'indice iMin et k
    return nbComp
```

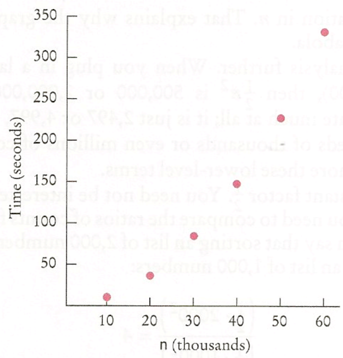
```python
L=[1,4,2,6,3,5,3,9,1,10,41,13,5]
n=len(L)
nb=triSelection(L)
print(L,nb,(n**2-n+1)/2)
```

→ [1, 1, 2, 3, 3, 4, 5, 5, 6, 9, 10, 13, 41] 78 78.5

Travail sur les listes - TRI

Remarques : Le tri s'effectue en place, l'instruction `return` n'est pas nécessaire, la liste L est modifiée car aucun nouvel objet list n'est créé. La dernière instruction aurait pu être écrite : `L[k],L[iMin] = L[iMin],L[k].`

Cet algorithme peut trier n'importe quelle tableau de nombres (entiers ou réels). Si la vitesse d'execution n'est pas un problème pour vous, vous pouvez vous arrêter ici, cette algorithme fera très bien le travail. Le problème est que ***cette algorithlme a de piètres qualités en terme de temps de calcul*** et cela devient critique quand il faut trier de tableaux de taille importantes. Le temps de calcul est proportionnel au carré du nombre d'éléments du tableau comme le montre le graphe ci-dessous. C'est pour cette raison que les informaticiens et les mathématiciens ont cherchés des algorithmes moins gourmants en temps de calcul.



n	Seconds
10,000	9
20,000	38
30,000	85
40,000	147
50,000	228
60,000	332

Figure 1 Time Taken by Selection Sort

5. Algorithme de tri par INSERTION

Le tri par insertion est utilisé dans les jeux de carte par beaucoup de personnes pour trier une main.

Dans cet algorithme, on suppose que le début de la liste est déjà triée juqu'à k (quand l'algorithme démarre, on prendra k égale à `0`).

`values[0] values[1] . . . values[k]`

On augmente le tableau en insérrant l'élément suivant, `value[k+1]`, du tableau à sa bonne place. Quand on atteind la fin du tableau, le processus de tri est achevé. Par exemple, étudions le tableau suivant :

11	9	16	5	7

Evidemment, le tableau initiale de longueur 1 est déjà trié. Nous ajoutons au tableau initiale `value[1]` qui à la valeur 9. Cet élement est inséré après l'élément 11. On obtient :

9	11	16	5	7

En continuant de la même façon, on obtient (le 16 est déjà à la bonne place) :

9	11	16	5	7

5	9	11	16	7

Voici la mise en œuvre de l'algorithme de tri par insertion :

```python
def insertionSort(tableau):

    # on traverse le tableau de 1 à len(tableau))
    for i in range(1, len(tableau)):

        cle = tableau[i]

        # on bouge les elements de tableau[0..i-1] qui sont
        # plus grand que la cle d'une position en avant
        # par rapport à leur position actuelle.
        j = i-1
        while j >=0 and cle < tableau[j] :
                tableau[j+1] = tableau[j]
                j -= 1

        # on insert l'element a la bonne place
        tableau[j+1] = cle

    return tableau


# on teste la fonction

essai = [12, 11, 13, 5, 6]

essaiTrie=insertionSort(essai)

print ("le tableau trie est:")
for i in range(len(essaiTrie)):
    print ("%d" %essaiTrie[i])
```

Dans le pire des cas, si le tableau initial n'est pas du tout trié, le temps de calcul est en $O\left(n^2\right)$ comme pour le tri par sélection. Mais si le tableau est en partie trié, voir dans le meilleur des cas complètement trié, le temps de calcul temps vers $O\left(n\right)$ ce qui est ; un avantage par rapport au tri par sélection. Ce dernier est toujours en $O\left(n^2\right)$ même si le tableau est en partie trié !