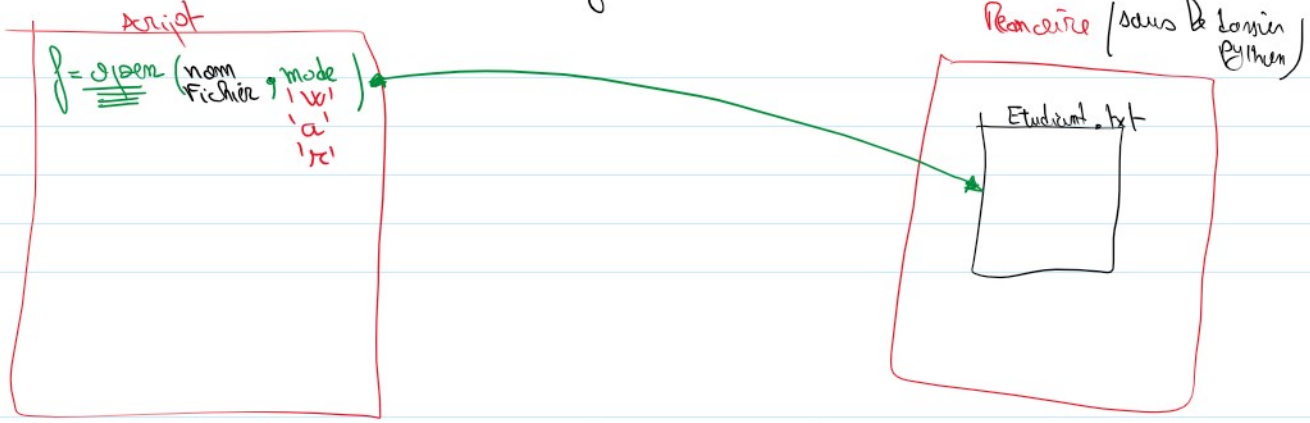
dimanche 16 avril 2023 14:16

Les fichiers

= un document enregistré dans la mémoire



3 modes
- write (Ecriture) : 'w' (si le fichier ∄ → il sera créé. / si le fichier ∃ → le contenu sera effacé.
- append (ajout à la fin) : 'a' (si le fichier ∄ → il sera créé / si le fichier ∃ → On continue l'écriture à la fin
- read (Lecture) : 'r' (le fichier doit exister sinon → Erreur

2) Ecriture

mode write / append

f.write (ligne) — chaine qui se termine par '\n' caractère retour à la ligne.

f.writelines (L) — liste de lignes

```
f = open ('Etud.txt', 'w')
f.write ('Med 13.2')
f.write ('Ali 11.6')
```
→ Etudiant.txt : Med 13.2Ali 11.6.....

L = ['Med 13.2', 'Ali 11.6']
f.writelines(L)

```
f = open
f.write ('Med 13.2\n')
f.write ('Ali 11.6\n')
```
→ Med 13.2
Ali 11.6

L = ['——\n', '——\n']

3 – Lecture

L = f.readlines()

liste de chaines (chaine ↔ une ligne)

= ['Med 13.6 \n', 'Ali 11.6 \n']

for ligne in f:

it1 ligne = 'Med 13.6\n'
it2 ligne = 'Ali 11.6\n'

4 – Fermeture

f.close() (on n'a pas le droit d'utiliser le fichier dans 2 modes

4 – Fermeture

f.close() (on n'a pas le droit d'utiliser le fichier dans 2 modes simultanément)

1)
```
def InfoTelechargement():
    f = open('Pagejdf.txt', 'r')
    L = []
    Lf = f.readlines()
    f.close()
    for ligne in L:
        'adIP heure date nomdoc\n'
        L1 = ligne.strip().split()
            supp. le dernier car (\n)
        ou
        L1 = ligne[:-1].split()   # [adIP, h, date, nomdoc]
        L.append(L1)
    return(L)
```

```
f = open('Pagejdf.txt', 'r')
for ligne in f:
    .
    .
    .
f.close()
return(L)
```

Rq */ ch = "Red * Salem * 19 * 11.6" (nom, pre, age, moy)

L = ch.split('*')
= ['Red', 'Salem', '19', '11.6']
liste de chaines

* age = int(L[2])
* moy = float(L[3])

*/ ch = 'ab ef' ; L = list(ch) = ['a', 'b', ' ', 'e', 'f']

2)
```
def Document(L):
    LDoc = []
    for e in L:
        = [adIP, h, date, nomdoc]
        LDoc.append(e[3])
            (différents nom)
                ↓
            sans répétitions.
```

```
LDoc = []
for e in L:
    if e[3] not in LDoc:
        LDoc.append(e[3])
return(LDoc)
```

3

* autrement

```
E = set()
for e in L:
    E.add(e[3])
return(list(E))
```

La Récursivité

La Recursivité

une fct recursive = une fct qui fait appel à elle même

Syntaxe

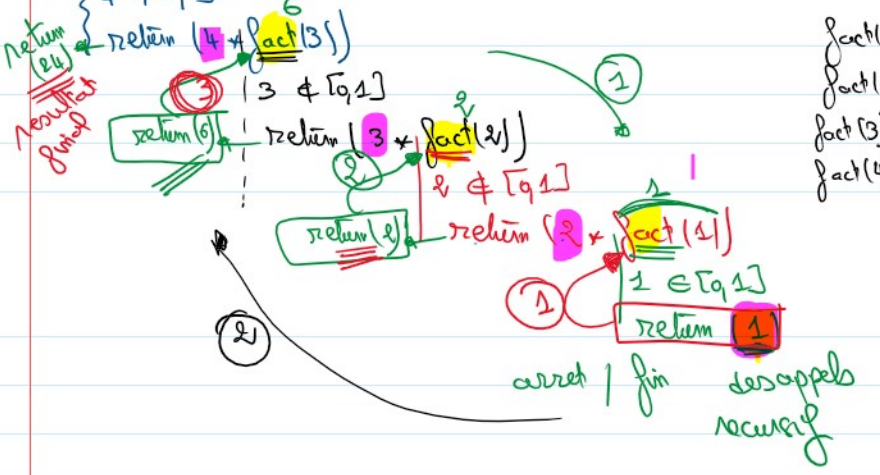
```
def nom_fct (parametres):
    if cond_d'arret (des appels recursifs): return (resultat de la cond d'arret)   # Cas particulier
    else:                                                                          # Cas general
        doit contenir un appel recursif
```

Exp Cas particulier

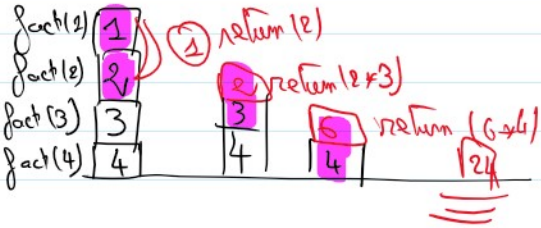
$$\begin{cases} n = 0,1 \longrightarrow fact(n) = 1 \\ n > 1 \longrightarrow fact(n) = n * fact(n-1) \longrightarrow \text{Cas general} \end{cases}$$

```
def fact(n):
    if n in [0,1]: return (1)
    else: return (n * fact(n-1))
```

Appel à fact(4)



Rq

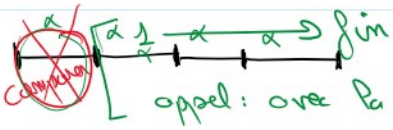


2- recursivité + Conteneurs

2 methodes { * le slicing
+ un comp incrementé dans chaque appel recursif

* **slicing** (Exp Recherche : verifier si $x \in L$)

Principe : Diminuer la taille des conts dans chaque appel recursif




```
def Rech (L, x):
    if L == []: return (False)
```

L = [3, -1, 4, 6] x = 4

Appel Rech (L, x)
L ≠ []
L[0] = 3 ≠ x
return (Rech (L[1:], x))

```
if L == []: return (False)
elif L[0] == x: return (True)
else: return (Rech (L[1:], x))
```

L ≠ []
L[0] = 3 ≠ x
return (Rech (L[1:], x))
L = [-1, 4, 6]
L ≠ []
L[0] = -1 ≠ x
return (Rech (L[1:], x))
L = [4, 6]
L ≠ []
L[0] = 4 == x return (True)
Avec

* Compteur incrementé dans chaque appel

valeur / parametre par defaut

```
def Rech (L, x, i=0):
    if i == len(L): return (False)
    elif L[i] == x: return (True)
    else: return (Rech (L, x, i+1))
```

len(L) = 3
i=0 i=1 i=2 i=3 fin du parcours

L = [3, -1, 4, 6] x = 4

Appel à Rech (L, x) # par defaut i=0
i ≠ len(L)
L[i] = L[0] = 3 ≠ x
return (Rech (L, x, i+1))
1 ← remplacer la val par defaut

i = 1
i ≠ len(L)
L[1] = -1 ≠ x
return (Rech (L, x, i+1))
2 ← " " "

i ≠ len(L)
L[2] = 4 == x
return (True)

Rq ch = 'abcdcba'

'abcdcba'
'bcdcb'
'cdc'
'd'

len = 1, 0 ⟶ chaine Palindrome

*/ Boucle while:

```
while True:
    ))) Traite du cas general de la fct recursive.
```

))) Trait du cas general de ta fct recursive.

if condArret: break

↑ cond d'arret des appels recursif

```
def pgcd(a,b):
    while True:
        if a>b: a-=b
        elif b>a: b-=a
        else: break  # a==b
    return (a)
```

recursive →

```
pgcd(a,b):
    if a==b: return (a,b)
    else:
        if a>b: a-=b
        else: b-=a
        return ( pgcd(a,b))
```

déjà modifiés.

Suite Ex

3)
```
def DocumentRec ( L, LDoc=[] ):
    if L==[]: return (LDoc)
    else: # Trait avec L[0] = [addIP, h, d, nomd]
        if L[0][3] not in LDoc:
            LDoc.append ( L[0][3])
        return ( DocumentRec( L[1:], LDoc))
```

ou

```
def DocRec (L, E=set() ):
    if L==[] : return ( list(E) )
    else:
        E.add (L[0][3])
        return ( DocRec ( L[1:], E)
```

4) L = [[ad1, h1, d1, nomD1], [ad2, h2, d2, nomD2], ...]

LDoc = [nomDoc sans repetition]

Dstat { cle = nomD
val = nb de Tel.

Rq ch = 'abbooob' ⟶ dict (cle = car
val = nb d'occ.

Rq ch = 'abbacb' ⟶ dict (clé = car
val = nb d'occ

* D = dict()
for c in ch:
if c not in D:
D[c] = ch.count(c)

nb = 0
for c in ch:
if c == c1: nb += 1

1 seul parcours de ch :
D = {}
• c = 'a' ∉ D ⟶ Ajout D['a'] = 1
D = {'a': 1}
• c = 'b' ∉ D ⟶ D['b'] = 1 D = {'a': 1, 'b': 1}
• c = 'b' ∈ D ⟶ D['b'] += 1
D = {'b': 1, 'b': 2}

D = dict()
for c in ch:
if c not in D: D[c] = 1
else: D[c] += 1

ch = 'abbacb' ⟶ { clé = car
val = liste des indices

D = dict()
for c in ch:
if c not in D:
L = []
for i in range(len(ch)):
if ch[i] == c:
L.append(i)
D[c] = L

1 seul parcours
• i = 0 ch[0] = 'a' ∉ D ⟶ D['a'] = [0]
D = {'a': [0]}
• i = 1 ch[1] = 'b' ∉ D ⟶ D['b'] = [1]
D = {'a': [0], 'b': [1]}
• i = 2 ch[2] = 'b' ∈ D
D['b'].append(2)
D = {'a': [0], 'b': [1, 2]}

D = dict()
for i in range(len(ch)):
if ch[i] not in D:
D[ch[i]] = [i]
else:
D[ch[i]].append(i)
liste

Complexité

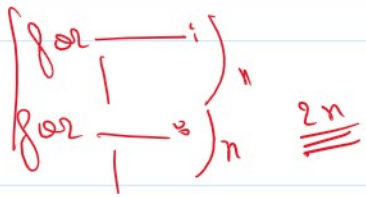
*D = dict()
for c in ch: n
if c not in D:
L = []
n*
for i in range(len(ch)):
if ch[i] == c:
L.append(i)
*D[c] = L

* Calculer la complexité
T(n) = 1aff + n*[1cond + 1aff + n*[1cond + 1aff] + 1aff]
↑ (nb total des instructions)
= 1 + 3n + 2n²

* La complexité
O(T(n)) = O(n²)

for — i

* Expliquer brièvement

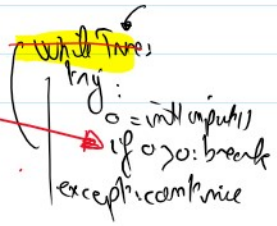


* Expliquer brièvement
On a effectué 2 boucles for imbriquées donc la complexité est en $O(n^2)$

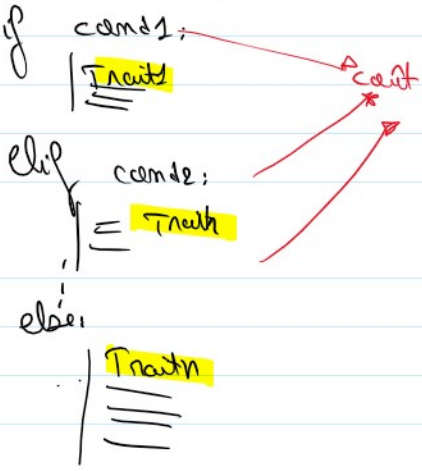
solution
```
D = dict()
for i in range(len(ch)):
    if ch[i] not in D:
        D[ch[i]] = [i]
    else:
        D[ch[i]].append(i)
```

Calcul
$T(n) = 1\text{aff} + n * [1\text{ cond} + 1\text{ aff}]$
$= 2n + 1$
Complexité
$O(T(n)) = O(n)$

Rq Calcul complexité
- les op simple (lecture, écriture, affectation) → coût 1 inst



- Structure cond



- boucle for / while → n * (instructions de la boucle)

Suite Ev { L = [[adIP1, h1, d1, nomD1], [adr, he, de, nomDe], ...]
elt1 elte
LDoc = [les noms des doc]

{ DStat (cle = nomD
val = nb de telechargement

methode 1
```
def Statistique(L, LDoc):
    DStat = dict()
    for nomD in LDoc:
        nb = 0
```

```
for nomD in LDoc:
    nb = 0
    for e in L:
        if e[3] == nomD:
            nb += 1
    DStat[nomD] = nb
```

DStat[nomD] = L.count(nomD) =0

ou

```
def Statistique(L, LDoc):
    L1 = [e[3] for e in L]
    DStat = dict()
    for nomD in LDoc:
        DStat[nomD] = L1.count(nomD)
    return(DStat)
```

2 boucles for imbriquées

count = n

ou

```
def ____:
    DStat = dict()
    for nomD in LDoc:
        L1 = [e for e in L if e[3] == nomD]
        DStat[nomD] = len(L1)
```

extraire seulement les elts qui vérifient les conds

méthodes

```
def Statistique(L):
    DStat = dict()
    for e in L:                      # e = [adIP, h, d, nomD]
        if e[3] not in DStat: DStat[e[3]] = 1
        else: DStat[e[3]] += 1
    return(DStat)
```

RA : Statistique
clé = nomD
val = liste des adIP qui ont téléchargé ce doc

[e[0]]

.append(e[0])

Version Récursive :

```
def StatRec(L, DStat = dict()):
    if L == []: return(DStat)
    else: * traite avec L[0] = [ad, h, d, nomD]
        if L[0][3] not in DStat: DStat[L[0][3]] = 1
        else: DStat[L[0][3]] += 1
        return(StatRec(L[1:], DStat))
```

5) DStat { clé = nomD
val = nb de téléchargement

→ LDoc1 : les noms des docs par ordre ↘ des nb téléchargt.

| val = nb de telechargement — par ordre ↘ des nb telcharg.

fct predefinie

sort (liste)

L = [0, 2, -1]

*/ L.sort()
print(L)
[-1, 0, 2]

*/ L.sort(reverse = True)
→ [2, 0, -1]

sorted (list / dict)

list

*/ L1 = sorted (L)
print(L1) → [-1, 0, 2]
print(L) → [0, 2, -1]

*/ L1 = sorted (L, reverse = True)
= [2, 0, -1]
L → [0, 2, -1]

*/ L = [[nom, Moy], ...]
L.sort() erreur

une fct à appliquer sur les elts avant de faire la comparaison

L1 = sorted (L , key = lambda e : e[1])

= [[nom1, 8.9], [nom2, 9.3], [nom3, 10.4] ....]

*/ L = ['efgh', 'abc', 'abcdij']

L1 = sorted (L, key = lambda e : len(e))

= ['abc', 'efgh', 'abcdij']

dict

sorted (par defaut = Tri des clés)

D = { cle = car
val = nb d'occ

D = { 'c' : 3, 'a' : 2, 'e' : 1 }

L = sorted(D) = ['a', 'c', 'e']

L = sorted (D, key = lambda cle : D[cle]), reverse = True

= ['e', 'a', 'c']

*/def Meilleur (DStat):
LDoc1 = sorted (DStat , key = lambda cle = DStat[cle] , reverse = True)
return (LDoc1)

return (LDoc1)

* autrement

Dstat = {'Ev1': 300, 'corrEv1': 300, 'Ds1': 200, 'corrDs1': 150, ...}

L = list(Dstat.values()) = [300, 300, 200, 150, 200, ...]

E = set(Dstat.values()) = {300, 150, 200 ...}

```
def Meilleur(Dstat):
    E = set(Dstat.values())
    L = set(E)
    LDoc1 = []
    while L != []:
        m = max(L)
        for nomD, nb in Dstat.items():
            if nb == m: LDoc1.append(nomD)
        L.remove(m)
    return(LDoc1)
```

6) def Affiche(LDoc1):
print("Le meilleur doc est " + " ensuite " + " et enfin " + ".".format(LDoc1[0], ', '.join(LDoc1[1:-1]), LDoc1[-1]))

*) L = ['ab', 'cd', 'ef']

ch = ''.join(L) = 'ab cd ef'

ch1 = '*'.join(L) = 'ab * cd * ef'

ch2 = ''.join(L) = 'abcdef'

7) def Heure(H): 'hh:mm:ss'
return(int(H[:2]))

```
i = H.index(':')
return(int(H[:i]))
```

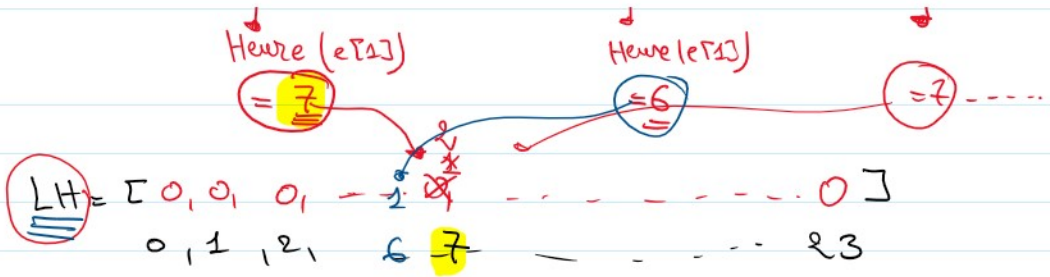
```
L = H.split(':')
return(int(L[0]))
```

8) L = [[ad1, h1, d1, nomD], [ad2, h2, d2, nomD2], ...]

Heure(e[1]) = 7

Heure(e[1]) = 6

= 7 ....




```
def DocHeure(L):
    LH = [0] * 24
    for e in L:
        h = Heure(e[1])
        LH[h] += 1
    return(LH)
```

} complexité en O(n)

9) def EnregistrerStatistique(DStat, LDoc1, LH): (les ind ⟷ les heures, elts ⟷ nb telchargt

```
    f = open('Statistique.txt','w')
    for nomD in LDoc1:
        nb = DStat[nomD]
        ligne = "{} : {}\n".format(nomD, nb)
        f.write(ligne)
    for h in range(24):
        ligne = "{} : {}\n".format(h, LH[h])
        f.write(ligne)
    f.close()
```

10)

```
L = InfoTelechargement()
```

* methode 1

```
LDoc = Document(L)
DStat = Statistique(L, LDoc)
LDoc1 = Meilleur(DStat)
```

* methode 2

```
DStat = Statistique(L)
LDoc1 = Meilleur(DStat)
```

```
Affiche(LDoc1)
LH = DocHeure(L)
print(LH)
EnregistrerStatistique(DStat, LDoc1, LH)
```