

# NSI 1ère - Algorithmique

Parcours séquentiel

---

qkzk

# Algorithmes sur les tableaux

---

Lorsqu'on dispose de plusieurs données similaires (notes à un devoirs, dépenses journalières etc.) il est commode de les ranger dans un tableau.

On est alors amené à effectuer de nombreuses opérations sur ces tableaux.

```
notes = [12, 10, 8, 6, 20]
```

Est-ce qu'un élève a eu 20 ?

La réponse est oui, 20 est le dernier élément du tableau. Pour une machine, répondre à cette question n'est pas immédiat.

Python propose l'instruction `in` qui teste l'appartenance :

```
>>> 20 in tableau
```

```
True
```

```
>>> 13 in tableau
```

```
False
```

Mais comment fait-il ?

## Autre outil natif pour la moyenne

De même, pour calculer la moyenne à ce devoir, on pourrait utiliser :

```
>>> moyenne = sum(tableau) / len(tableau)
```

```
>>> moyenne
```

```
11.2
```

Mais comment ?

Et cette approche pose un problème majeur : **ces instructions n'existent pas dans tous les langages !**

- Nous allons étudier de simples algorithmes qui utilisent le caractère séquentiel d'un tableau.
- Systématiquement, nos algorithmes vont commencer par une boucle qui parcourt les éléments du tableau.
- À chaque fois il faudra adapter ce que nous faisons dans la boucle à notre contexte.

## Parcours séquentiel d'un tableau

---



**Attendus :** Écrire un algorithme de recherche d'une occurrence sur des valeurs de type quelconque. Écrire un algorithme de recherche d'un extremum, de calcul d'une moyenne.

**Commentaire :** On montre que le coût est linéaire.

**Un tableau est une série d'objets, généralement situés côte à côte dans la mémoire.**

On suppose pouvoir parcourir le tableau, élément par élément.

# Les tableaux en Python

En Python, pour illustrer les tableaux, on utilise les objets `list`.

```
>>> T = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> type(T)
```

```
<class 'list'>
```

Il existe quatre méthodes simples pour construire les tableaux en Python

## Construction directe

- En les définissant directement

```
equipe = ["Diego", "Franz", "Michel", "Johann",  
...      "Lionel", "Christiano"]
```

- En ajoutant, élément par élément avec append,

```
longueurs = []  
for joueur in equipe:  
    longueurs.append(len(joueur))
```

On obtient ici :

```
[5, 5, 6, 6, 6, 10]
```

- par compréhension,

```
cubes = [x ** 3 for x in [1, 2, 3, 4]]
```

- cas des nombres espacés régulièrement : `range`

La fonction `range` renvoie un itérable composé de nombres séparés régulièrement.

`range` accepte de 1 à 3 arguments.

```
range(debut, fin, pas)
```

Les nombres entre `debut` (inclu) et `fin` (exclu) séparés de `pas` à chaque fois

```
range(3, 19, 4)
```

Les nombres entre 3 (inclu) et 19 (exclu) séparés de 4 à chaque fois



On va obtenir un itérable correspondant à [3, 7, 11, 15]  
19 n'y figure pas car la borne de droite est toujours exclue.

## range avec ou sans bornes

- S'il n'y a que deux nombres, ce sont les bornes de début et de fin. Le pas est 1.
- S'il n'y a qu'un nombre, c'est la borne de fin. La borne de début est 0.

`range(6)` retourne l'itérable correspondant à la liste des entiers  
[0, 1, 2, 3, 4, 5]

Cela demande forcément une boucle.

Rappelons qu'il existe deux types de boucles :

1. **boucles avec for** qui parcourt directement un objet *itérable* (list, tuple, dict, etc.).

```
for joueur in tableau:  
    faire_quelque_chose_avec(joueur)
```

2. **boucles avec while** qui utilise une condition d'arrêt. En comptant les éléments

```
compteur = 0
while compteur < 5:
    print(compteur) # on faire autre chose...
    compteur = compteur + 1
```

Une boucle avec while qui s'arrête doit comporter tous ces éléments.

Il est courant de créer des boucles qui ne s'arrêtent pas :

- serveur qui attend des messages

```
while True:  
    serveur_ecoute_message(message)
```

- dans un jeu vidéo :

```
while True:  
    ecouter_les_actions_joueur()  
    calculer_nouvel_etat_jeu()  
    dessiner_les_graphismes()
```

# Algorithmes sur les tableaux

---

## Recherche d'un élément dans un tableau

**Contexte** : on dispose d'un tableau,

par exemple  $T = [0, 1, 2, \dots, 10]$ .

On veut savoir si un nombre  $x$  figure dans le tableau.



## Algorithme :

```
fonction (tableau T, objet x) ---> booléen:  
  Pour chaque élément e de T,  
    Si e = x, alors on retourne Vrai  
  Si la boucle se termine, on retourne Faux.
```

## Exemple

T = [2, 5, -4, 12]

**A-t-on 9 dans le tableau ?**

élément	2	5	-4	12
élément == 9 ?	Faux	Faux	Faux	Faux

Le parcours de la boucle se termine et l'algorithme retourne Faux.

**A-t-on -4 dans le tableau ?**

élément	2	5	-4	12
élément == 9 ?	Faux	Faux	Vrai	x

L'algorithme retourne Vrai. La dernière case du tableau n'est jamais visitée !

# Coût du parcours séquentiel

De manière évidente,

1. en général on parcourt tout le tableau,
2. chaque étape est “identique” aux autres,

Le **coût** d'un algorithme correspond au nombre d'opérations à effectuer.

Il est très difficile de le calculer exactement, beaucoup plus facile de l'estimer.

## Coût du parcours séquentiel

Dans notre cas, on a, grosso modo, autant d'opérations qu'il y a d'éléments dans le tableau.

Le coût est *proportionnel à la taille du tableau*. On dit qu'il est *linéaire*.

On note : le parcours séquentiel est un algorithme en  $O(n)$

$n$  ici désigne la taille du tableau.

## En Python :

```
def recherche_sequentielle(tableau, x):  
    for elt in tableau:  
        if x == e:  
            return True  
    # on n'arrive ici que si l'élément n'est pas dans le tab  
    return False
```

Parcours séquentiel : seule manière de répondre à

**x figure-t-il dans le tableau T ?**

SAUF si on a des informations particulières sur le tableau !

Si le tableau est **trié** alors il existe des algorithmes plus rapides.

Nous en étudierons un important : **la recherche dichotomique**.

# Recherche d'extremum

**Contexte** : On cherche la valeur extrême d'un tableau de nombres T.

Le principe est simple.

- Si on n'a qu'un élément, `maximum` est cet élément.
- Sinon... on les compare et, à chaque fois qu'une valeur `e` est supérieure à `maximum`, on l'affecte à `maximum`.
- On retourne `maximum`

Pour le tableau `T = [2, 5, 9, 7]` cela donne :

élément e	2	5	9	7
<code>maximum</code>	2	5	9	9

Le `maximum` vaut 9.

fonction maximum(tableau T, nombre x) ---> nombre:

On affecte à max la valeur de l'élément d'indice 0 du tableau

Pour chaque élément e du tableau:

    si e > max:

        max = e

retourner max



```
def maximum(tableau):  
    m = tableau[0]  
    for elt in tableau:  
        if elt > m:  
            m = elt  
    return m
```

## Opérations natives : min et max

```
>>> tableau = [4, 5, 2, -1, 3]
```

```
>>> max(tableau)
```

```
5
```

```
>>> min(tableau)
```

```
-1
```

# Moyenne des éléments d'un tableau

**Contexte** : On calcule la moyenne d'un tableau de nombres

Le principe repose sur **le cumul d'une valeur**.

- On cumule un compteur initialisé à 0, il augmente de 1 à chaque étape.
- On cumule la somme des valeurs comme on le ferait à la main.

Pour le tableau  $T = [2, 5, 9, 8]$  cela donne :

élément e	2	5	9	8
nb_elements	1	2	3	4
somme	2	7	16	24

Le somme vaut 24 et il y a 4 éléments : la moyenne est  $24/4 = 6$

```
fonction moyenne(tableau T, nombre x) ---> nombre:  
  On affecte à Somme la valeur 0  
  On affecte à Effectif la valeur 0  
  Pour chaque élément e du tableau:  
    Somme = Somme + e  
    Effectif = Effectif + 1  
  retourner Somme / Effectif
```

```
def moyenne(tableau):  
    somme = 0  
    effectif = 0  
    for elt in tableau:  
        effectif += 1  
        somme += elt  
    return somme / effectif
```

Version courte qui n'illustre pas le programme :

```
def moyenne2(tableau):  
    return sum(tableau) / len(tableau)
```

Difficile de comprendre ce qui se passe !

## Retourner un tableau

```
>>> T = [1, 2, 3]
>>> retourner(tableau)
>>> [3, 2, 1]
```

Comment faire ?

- On crée un tableau vide  $R$  pour nos éléments retournés
- On parcourt le tableau  $T$ ,
  - pour chaque élément rencontré, on *l'insère au début de  $R$*
- On retourne  $R$



```
def retourner(tableau):  
    tab_retourne = []  
    for element in tableau:  
        tab_retourne = [element] + tableau_retourne  
        # variante : tab_retourne.insert(0, element)  
    return tab_retourne
```

- faites bien attention aux crochets [element] + tableau\_retourne

On *ajoute deux tableaux*: Python les met bout à bout !

## opération native en python

```
>>> tableau = [1, 2, 3]
>>> tableau.reverse()
>>> # ne retourne rien mais modifie le tableau initial
>>> tableau
[3, 2, 1]
```

# Tableaux à deux dimensions

---

## Tableaux à deux dimensions

On rencontre souvent des données qui sont présentées sous la forme d'un tableau à deux dimensions :

1234

5678

Ces tableaux sont appelés des **matrices**. Celle-ci a 2 lignes et 4 colonnes

# Affecter une matrice à une variable

Chaque ligne de la matrice est dans un tableau.

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

## Remarquons

- qu'on crée un tableau extérieur `mat = [ ... ]`
- que chaque élément de ce tableau est **lui même un tableau** :  
[1, 2, 3, 4] etc.

On dit que c'est une structure *imbriquée*.

Cette notation est universelle.

## Atteindre un élément.

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

Disons qu'on veut atteindre le nombre 3. Il est dans la première ligne, 3ème colonne :

```
>>> mat[0][2]  
3
```

On utilise deux séries de [] pour les lignes puis pour les colonnes.

## Parcourir une matrice : boucles imbriquées

Parcourir une structure imbriquée **demande forcément deux boucles imbriquées.**

```
for ligne in tableau:  
    for element in ligne:  
        faire_quelque_chose_avec(element)
```

**BOUCLES IMBRIQUÉES : L'UNE DANS L'AUTRE.**

Si vous devez apprendre 1 phrase de cette partie c'est celle là.

## coût des boucles imbriquées

- Si j'ai 9 lignes et 7 colonnes... j'ai ...  $9 \times 7 = 63$  éléments.

Mon parcours va visiter chacun de ces 63 éléments.

Le coût est proportionnel à ce produit.

- Donc : chaque fois qu'on imbrique une boucle, on multiplie de le nombre d'étapes.

Si j'ai 4 boucles **imbriquées** avec respectivement 3, 5, 10 et 4 éléments à visiter,

je ferai  $3 \times 5 \times 10 \times 4 = 600$  opérations.



- indentation

```
t = [[1, 2, 3],  
     [4, 5, 6]]
```

```
for ligne in T:
```

```
    for element in ligne:  
        print(element)
```

cette syntaxe est fausse... La première boucle ne fait rien !

## erreurs fréquentes : pire

- même problème mais qui ne plante pas...

```
t = [[1, 2, 3],  
     [4, 5, 6]]
```

```
for ligne in T:  
    pass # on evite de planter !
```

```
for element in ligne:  
    print(element)
```

## erreurs fréquentes : pire

C'est encore pire ! Au moins la première fois on avait une erreur...

Cette fois que se passe-t-il ?

1. On visite bien chaque ligne ! Mais on ne fait rien : pass !
2. La deuxième boucle ne travaille que sur la **dernière** ligne

Que verra-t-on ?

4

5

6

Seulement les éléments de la dernière ligne

**IL FAUT IMBRIQUER LES BOUCLES. L'UNE DANS L'AUTRE !!!**

**IL FAUT IMBRIQUER LES BOUCLES. L'UNE DANS L'AUTRE !!!**

```
for ligne in matrice:  
    for element in ligne:  
        faire_qqchose(element)
```

**IL FAUT IMBRIQUER LES BOUCLES. L'UNE DANS L'AUTRE !!!**

## Parcourir une matrice pour afficher ses éléments

On utilise toujours deux boucles **imbriquées** pour parcourir une matrice

```
for ligne in mat:
    for cellule in ligne:
        print(cellule, end=', ')
```

Va nous afficher :

1, 2, 3, 4, 5, 6, 7, 8,

**Remarque :** `print(1, end=', ')`

Normalement à la fin d'un `print` python insère un retour à la ligne. Ici, on le remplace par les caractères `' , '` et nos éléments s'affichent en ligne.

## Calculer la somme des éléments “à la main”

```
mat = [[3, 9, 1, 2],  
       [4, 5, 0, 1]]
```

Calculons la somme de cette matrice à la main :

ligne 0	élément e	3	9	1	2
	effectif	1	2	3	4
	somme	3	12	13	15

On ne réinitialise pas effectif et somme entre les lignes !

ligne 1	élément e	4	5	0	1
	effectif	5	6	7	8
	somme	19	24	24	25

```
somme = 0
for ligne in mat:
    for element in ligne:
        somme += element
```

retourne 25

## D'autres représentations des matrices

On a vu (on on verra !) dans le TP sur les tableaux à 2 dimensions qu'il était parfois nécessaire de connaître la position d'une cellule dans la matrice, par exemple, pour construire un dégradé de pixels.

Dans l'exemple suivant, non seulement nous parcourons le tableau à l'aide d'indices, mais en plus les éléments de la matrice sont atteints autrement.



## Créer un dégradé

```
from PIL import Image
from IPython.display import display # pour Colab
img = Image.new('RGB', (255, 128))
# nouvelle image, 255 de large, 128 de haut
pixels = img.load() # on charge la matrice des pixels

for x in range(255):
    for y in range(128):
        # attention à la notation [x, y] !!!
        pixels[x, y] = (255 - x, 0, 0)

display(img) # afficher dans colab
# img.show() # afficher dans la console
```



**Figure 1:** Dégradé rouge -> noir